

## III. Machine-Level Representation of Programs

---

- Computers execute **machine code** (sequences of bytes encoding the low-level operations that manipulate data, manage memory, read and write data on storage devices, and communicate over networks)
- **Compiler** generates machine code based on
  1. rules of the programming language
  2. instruction set of the target machine
  3. conventions followed by the OS
- **GCC** C compiler generates...
  1. C code -> **assembly code**
    - a textual representation of the machine code with the individual instructions in the program
  2. invokes **assembler** & **linker** to generate the **executable machine code** from the assembly code
- With modern **optimizing compilers**, the generated code is usually at least as efficient as what a skilled assembly-language programmer would write by hand
- But... Why do we have to learn machine codes? (read & understand)
  1. to understand the **optimization capabilities** of the compiler and analyze the underlying inefficiencies
  2. layer of abstraction hides information about the **run-time behavior** of a program that we need to understand
    - concurrent programs - how program data are shared or kept private by the different threads
  3. to understand how vulnerabilities arise and how to guard against them
    - how malwares store their run-time control information

### H3 §3.1 A Historical Perspective

: Intel processors and their key features

K = 1,000 ( $10^3$ ),    M = 1,000,000 ( $10^6$ ),    G = 1,000,000,000 ( $10^9$ )

1. 8086 (1978, 29K transistors)
  - 16-bit microprocessors

- contracted Microsoft to develop the MS-DOS OS
  - 32,768 bytes of memory + two floppy disks
  - limited to a 655,360-byte address space
  - introduced 8087 floating-point coprocessor (45K transistors)
    - executes the floating-point instructions
2. 80286 (1982, 134K transistors)
    - more addressing modes
    - formed the basis of the IBM PC-AT personal computer
      - original platform for MS Windows
  3. i386 (1985, 275K transistors)
    - expanded the architecture to **32 bits**
    - first machine that could fully support a Unix OS
  4. i486 (1989, 1.2M transistors)
    - improved performance and integrated the floating-point unit onto the processor chip
  5. Pentium (1993, 3.1M transistors)
    - improved performance with minor extensions to the instruction set
  6. PentiumPro (1995, 5.5M transistors)
    - radically new processor design
    - **P6** microarchitecture
    - added a class of " **conditional move** " instructions to the instruction set
  7. Pentium/MMX (1997, 4.5M transistors)
    - added a new class of instructions for manipulating **vectors of integers**
      - each datum can be 1, 2, or 4 bytes long
      - each vector totals 64 bits
  8. Pentium II (1997, 7M transistors)
    - continuation of the P6 microarchitecture
  9. Pentium III (1999, 8.2M transistors)
    - introduced SSE, a class of instructions for manipulating vectors of integer or floating-point data
      - vectors = 128 bits
    - later version went up to 24M transistors
      - due to the incorporation of the level-2 cache on chip
  10. Pentium 4 (2000, 42M transistors)
    - extended SSE to SSE2, adding new data types (double-precision floating point), along with 144 new instructions for new data types

- compilers could use SSE instructions to compile floating-point code

#### 11. Pentium 4 (2004, 125M transistors)

- added **hyperthreading** - method to run two programs simultaneously on a single processor
- EM64T ( **x86-64** ), a Intel's implementation of a 64-bit extension to IA32 was developed by AMD

#### 12. Core 2 (2006, 291M transistors)

- returned to a microarchitecture similar to P6
- first **multi-core** Intel microprocessor
  - multiple processors are implemented on a single chip
- did not support hyperthreading

#### 13. Core i7, Nehalem (2008, 781M transistors)

- incorporated both hyperthreading and multi-core
- supported two executing programs on each core and up to four cores on each chip

#### 14. Core i7, Sandy Bridge (2011, 1.17G transistors)

- introduced AVX (extension of the SSE) to support 256-bit vectors

#### 15. Core i7, Haswell (2013, 1.4G transistors)

- extended AVX to AVX2 (more instructions and formats)

- Each processor has been designed to be **backward compatible**
- **IA32** - Intel Architecture 32-bit
- **Intel64** = **x86-64** --> "x86" reflects the processor naming conventions up through the i486

### H3 §3.2 Program Encodings

- Suppose we write a C programs `p1.c` and `p2.c`. To compile,

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

- **gcc** - indicates the GCC C compiler
- **-Og** - indicates a level of optimization
  - invoking higher levels of optimization generates code that is heavily transformed
    - better in terms of program performance

- 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 (.i)
  2. **compiler** generates assembly-code version of source file (.s)
  3. **assembler** converts the assembly code into **binary object-code** (.o)
    - binary representations of all of the instructions
  4. **linker** merges object code(s) along with code implementing **library functions** (e.g. `printf`) and generates the final executable code file.

## H5 ¶3.2.1 Machine-Level Code

- Abstractions in machine-level programming
  1. **ISA** (*instruction set architecture*) - defines format & behavior of a machine-level program
    - defines the processor state, the format of the instructions, and the effect of each of these instructions will have on the state
    - describe the behavior of a program **as if** each\*\* instruction is executed in **sequence**
      - actually executing many instructions concurrently
  2. **Virtual addresses** - memory addresses used by a machine-level program
    - provides a memory model that appears to be a very large byte array
      - actually combination of multiple hardware memories and operating system software
- Machine code components (x86-64)
  1. **program counter** (PC) - `%rip`
    - indicates the address in memory of the next instruction to be executed
  2. integer **register file**
    - contains 16 names locations storing 64-bit values
    - can hold addresses ( $\approx$  C pointers) or integer data
    - some registers are used to keep track of critical parts of the program state
    - others to hold temporary data (arguments & local variables of a procedure & value to be returned by a function)
  3. **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

- **if** and **while** statements
4. **Vector registers**
- can each hold one or more integer or floating-point values
- 
- Machine code views the memory as simply a **large byte-addressable array**
  - **OS translates** virtual addresses into the physical addresses of values in the actual processor memory
  - Single machine instruction performs only a very **elementary operations**
    - adding two numbers stored in registers
    - transferring data between memory and a register
    - conditionally branching to a new instruction address
  - Compiler must generate **sequences** of such instructions to implement program constructs
    - arithmetic expression evaluation, loops, procedure calls, returns, etc.

## H5 ¶3.2.2 Code Examples

Suppose we have a C code named `mstore.c`

```
1  long mult2(long, long);
2
3  void multstore(long x, long y, long *dest) {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

To see the assembly code generated by the C compiler,

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

which causes GCC to run the compiler, generating an assembly file `mstore.s`

```

1  multstore:
2      pushq %rbx
3      movq  %rdx, %rbx
4      call mult2
5      movq  %rax, (%rbx)
6      popq  %rbx
7      ret

```

- each indented line corresponds to a single machine instruction
  - `pushq %rbx`
    - indicates that the contents of register `%rbx` should be pushed onto the program stack

```

1  linux gcc -Og -c mstore.c

```

- will generate an object-code file `mstore.o`
  - binary format, cannot be viewed directly
  - simply a sequence of bytes encoding a series of instructions
- to inspect the contents of machine-code files, use **disassemblers**
  - generates a format similar to assembly code from the machine code
  - with Linux system, **OBJDUMP**

```

1  linux> objdump -d mstore.o

```

will return..

```

1  0000000000000000 <multstore>:
2      0:      53                push  %rbx
3      1:      48 89 d3            mov   %rdx, %rbx
4      4:      e8 00 00 00 00      callq 9 <multstore+0x9>
5      9:      48 89 03            mov  %rax, (%rbx)
6      c:      5b                pop   %rbx
7      d:      c3                retq

```

- each of these groups is a single instruction

- Several features of machine code
  - x86-64 can range from 1 to 15 bytes
    - commonly used instructions and instructions with fewer operands require a smaller number of bytes
  - from a given starting position, there is a **unique decoding** of the bytes into machine instructions
  - disassembler determines the assembly code based purely on the byte sequences in the machine-code file
    - **Does not** require access to the source or assembly-code versions of the program
  - uses a slightly different naming convention for the instructions than does the assembly code generated by GCC
    - omitted 'q's' -- size designators (can be omitted)

Suppose we have *main.c*

```

1  #include <stdio.h>
2
3  void multstore(long, long, long *);
4
5  int main() {
6      long d;
7      multstore(2, 3, &d);
8      printf("2 * 3 --> %ld\n", d);
9      return 0;
10 }
```

- To generate the actual **executable code**, needs to **run a linker** on the set of **object-code files**
  - **MUST** contain a function **main**
- Executable files generally have **bigger size** than object files because they contain code used to...
  - **start** and **terminate** the program
  - **interact with the operating system**

```

1  linux> gcc -Og -o prog main.c mstore.c
2  linux> objdump -d prog

```

- **Disassembler** will then **extract code sequences** such as...

```

1  0000000000400540 <multstore>:
2      400540: 53                push  %rbx
3      400541: 48 89 d3          mov   %rdx, %rbx
4      400544: e8 42 00 00 00    callq 40058b <mult2>
5      400549: 48 89 03          mov   %rax, (%rbx)
6      40054c: 5b                pop   %rbx
7      40054d: c3                retq
8      40054e: 90                nop
9      40054f: 90                nop

```

- differences between the one that generated by the disassembly of `mstore.c`
  - **addresses** listed are **different**
    - linker has shifted the location
  - linker has filled in the **address that the `callq` instruction** should use in calling the function `mult2` (Line 4)
    - linker matches function calls with the locations of the executable code for those functions
  - **two additional lines** of code at the end
    - no effect on the program
    - grow the code for the function to 16 bytes for **better placement** of the next block

## H5 ¶3.2.3 Notes on Formatting

- In the full content of the file (assembly code), it contains lines like

```

1  .file    "010-mstore.c"
2  .text
3  .globl   multstore
4  .type    multstore, @function

```



- these are **directives** to guide the assembler and linker
  - but we'll ignore these

```

1 # void multstore(long x, long y, long *dest)
2 # x in %rdi, y in %rsi, dest in %rdx
3 multstore:
4     pushq    %rbx           # Save %rbx
5     movq     %rdx, %rbx     # Copy dest to %rbx
6     call     mult2          # Call mult2(x, y)
7     movq     %rax, (%rbx)    # Store result at *dest
8     popq     %rbx           # Restore %rbx
9     ret                    # Return

```

### H3 §3.3 Data Formats

- Due to its origins as a 16-bit architecture,
  - " **word** " - 16-bit data type
  - " **double word** " - 32-bit data type
  - " **quad word** " - 64-bit data type

C declaration	Intel data type	Assembly-code suffix	Size(bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

- note that the suffix 'l' denotes a 4-byte integer and an 8-byte double-precision floating-point number
  - causes 0 ambiguity

- ∴ Floating-point code uses entirely different set of instructions and registers

### H3 §3.4 Accessing Information

- x86-64 CPU contains **16 general purpose registers** storing 64-bit values
  - store integers & pointers
  - names all begin with **%r**
- **Historical evolution of the instruction set**
  - 8086 - 8x 16-bit registers (%ax ~ %bp)
    - each had a specific purpose (names reflected how they were to be used)
  - IA32 - 8x 32-bit registers (%eax ~ %ebp)
  - **x86-64** - 16x 64-bit registers (8 original + 8 new registers)
    - (%rax ~ %rbp, %r8 ~ %r15)
- Instructions **can operate on data of different sizes** stored in the lower-order bytes of the 16 registers
  - Copying & generating 1-, 2-, 4- and 8-byte values
    - Remaining bytes in the register for instructions that generate < 8 bytes?
      1. 1-, 2- byte: leave the remaining bytes unchanged
      2. 4- bytes: set the upper 4 bytes to 0
- **%rsp**: stack pointer - indicate the end position in the run-time stack

63	31	15	7 0	Usage
%rax	%eax	%ax	%al	<b>Return value</b>
%rbx	%ebx	%bx	%bl	Callee saved
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	Callee saved
%rsp	%esp	%sp	%spl	<b>Stack Pointer</b>
%r8	%r8d	%r8w	%r8b	5th argument
%r9	%r9d	%r9w	%r9b	6th argument
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r13b	Callee saved
%r13	%r13d	%r13w	%r13b	Callee saved
%r14	%r14d	%r14w	%r14b	Callee saved
%r15	%r15d	%r15w	%r15b	Callee saved

## H5 ¶3.4.1 Operand Specifiers

- Most instructions have one or more *operands* - **source** values to use in performing an operation and the **destination** location
- **Source values** - constants or read from registers or memory
- **Results** - can be stored in either registers or memory
- **Three** types of **operands**
  1. **Immediate** - constant values

- '\$' followed by an integer

- \$-577, \$ 0x1F

## 2. **Register** - contents of a register

- one of the 16x 8-, 4-, 2-, 1-byte lower-order portions of the registers for operands having 64, 32, 16, 8 bits
- $r_a$  : arbitrary register     $R[r_a]$  : value with the reference

## 3. **Memory** - some meomory location according to a computed address (effective address)

- $M_b[Addr]$  denotes a reference to the  $b$  -byte value stored in memory starting at address  $Addr$

## • **Operand forms**

Type	Form	Operand value	Name
Immediate	$\$Imm$	$Imm$	Immediate
Register	$r_a$	$R[r_a]$	Register
Memory	$Imm$	$M[Imm]$	Absolute
Memory	$(r_a)$	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base+displacement
Memory	$(r_b, r_i)$	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	$(r_b, r_i, s)$	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

## • **Practice problem**

1	Address	Value	Register	Value
2	-----			
3	0x100	0xFF	%rax	0x100
4	0x104	0xAB	%rcx	0x1
5	0x108	0x13	%rdx	0x3
6	0x10C	0x11		

1	Operand	Value	Comment
2	-----		
3	%rax	0x100	Register
4	0x104	0xAB	Absolute address
5	\$0x108	0x108	Immediate
6	(%rax)	0xFF	Address 0x100
7	4(%rax)	0xAB	Address 0x104
8	9(%rax,%rdx)	0x11	Address 0x10C
9	260(%rcx,%rdx)	0x13	Address 0x108
10	0xFC(,%rcx,4)	0xFF	Address 0x100
11	(%rax,%rdx,4)	0x11	Address 0x10C

## H5 ¶3.4.2 Data Movement Instructions

- copying data from one location to another
- **MOV class**
  - movb, movw, movl, movq -->. 1, 2, 4, 8 bytes
  - copy data from a source location to a destination location without any transformation

1	Instruction	Effect	Description
2	-----		
3	MOV S, D	D ← S	Move
4	movb		Move byte
5	movw		Move word
6	movl		Move double word
7	movq		Move quad word
8	movabsq I, R	R ← I	Move absolute quad word

- **Source operand**
  - immediates, values stored in register / memory

- **Destination**
  - registers or memory addresses
- In **x86-64**, **move** instruction **cannot have both** operands refer to **memory** locations
  1. load the source value into a register
  2. write this register value to the destination
- **Register operands** can be any of the 16 registers, but **must match** the size designated by the last character of the instruction (**b, w, l, q**)
- For most cases, **MOV** instructions only update the specific register bytes / memory locations indicated by the destination operand
- **Exception!** - when **movl** has a **register** as the **destination**, it will also set the **high-order 4 bytes of the register to 0**
- **Examples**

```

1  movl $0x4050, %eax      #Immediate--Register, 4B
2  movw %bp, %sp           #Register--Register, 2B
3  movb (%rdi,%rcx),%al    #Memory-Register, 1B
4  movb $-17, (%esp)       #Immediate--Memory, 1B
5  movq %rax, -12(%rbp)    #Register--Memory, 8B

```

- Copying a smaller source value to a larger destination
  - **MOVZ** class
    - **Zero-extending** data movement instructions
    - fill out the remaining bytes of the destination with **0's**

1	Instruction	Effect	Description
2	MOVZ S,R	R ← ZeroExtend(S)	Move with zero extension
3	movzbw to word		Move zero-extended byte
4	movzbl to double word		Move zero-extended byte
5	movzwl to double word		Move zero-extended word
6	movzbq to quad word		Move zero-extended byte
7	movzwq to quad word		Move zero-extended word

- Note that there is **no** such instruction like **movzlw** which moves zero-extended double word to quad word
  - **movl** does the same thing when it has **register** as its destination
- **MOVS** class
  - **Sign-extending** data movement instructions
  - fill out the remaining bytes by **sign extension** -- replicating copies of the **most significant bit** of the source operand

1	Instruction	Effect	Description
2	MOVS S, R	R ← SignExtend(S)	Move with sign extension
3	movsbw		Move sign-extended byte to word
4	movsbl		Move sign-extended byte to double word
5	movswl		Move sign-extended word to double word
6	movsbq		Move sign-extended byte to quad word
7	movswq		Move sign-extended word to quad word
8	movslq		Move sign-extended double word to quad word
9	cltq	%rax ← SignExtend(%eax)	Sign-extend %eax to %rax

- **cltq** instruction
  - no operands
  - always uses **%eax** as its source and **%rax** as its destination for the **sign-extended result**
- each instruction name has **size designators** as its **final two characters**
  - 1st - source size, 2nd - destination size

## H5 ¶3.4.3 Data Movement Example

- C code

```

1  long exchange(long *xp, long y) {
2      long x = *xp;
3      *xp = y;
4      return x;
5  }

```

- Assembly code

```

1  # long exchange(long *xp, long y)
2  # xp in %rdi, y in %rsi
3  exchange:
4      movq    (%rdi), %rax    # Get x at xp, Set as return value
5      movq    %rsi, (%rdi)    # Store y at xp,
6      ret                                # Return.

```

- register **%rax** will be used **to return a value** from the function
- Two features to note
  1. **calling** "pointers" with **simple addresses** / **dereferencing** pointers involve **copying the pointer into a register**, and then using this **register in a memory reference**
  2. local variables ( *x* and *y* ) are often kept in **registers** than in memory locations ( ∴ register access is much faster than memory access)

## H5 ¶3.4.4 Pushing and Popping Stack Data

- **Stack** takes a vital role in the **handling of procedure calls**
  - values can be added or deleted according to a **"a last-in, first-out" (LIFO)** discipline
  - **add** data with **push** operation / **remove** data with **pop** operation
    - value popped will always be the value that was most recently pushed
  - can be implemented as an **array**
    - end of the array is called the **top** of the stack
- With x86-64, the program stack is stored in some region of memory
- **Stack grows downward**
  - **top** element of the stack has the **lowest address** of all stack elements
  - by convention, it is norm to draw stacks upside down (stack "top" shown at the bottom)



1	Instruction	Effect	Description
2	-----		
3	<code>pushq S</code>	<code>R[%rsp] &lt;- R[%rsp]-8;</code>	Push quad word
4		<code>M[R[%rsp]] &lt;- S</code>	
5			
6	<code>popq D</code>	<code>D &lt;- M[R[%rsp]];</code>	Pop quad word
7		<code>R[%rsp] &lt;- R[%rsp]+8</code>	

- **pushq** and **popq** take a single operand
  - **pushq** - data source for pushing
    - first decrement the stack pointer by 8
    - write the value at the new top-of-stack address
    - **pushq %rbp**

```

1  subq $8,%rsp      # Decrement stack pointer
2  movq %rbp, (%rsp) # Store %rbp on stack

```

- **popq** - data destination for popping

```

1  movq (%rsp),%rax  # Read %rax from stack
2  addq $8,%rsp      # Increment stack pointer

```

- even after 'pop' instruction, the value remains at memory location until it is overwritten
  - While **stack top** is always considered to be the address indicated by **%rsp**
- programs can access **arbitrary positions** within the stack

### H3 §3.5 Arithmetic and Logical Operations

- Most of the operations are given as instruction classes
  - ∃ different variants with different operand sizes (except **leaq** )

#### H5 ¶3.5.1 Load Effective Address

1	Instruction	Effect	Description
2	-----		
3	leaq     S,D	D <- &S	Load effective address

- variant of the **movq** instruction
  - reads from memory to a register **BUT**, it does **not reference memory** at all
  - **copies** the **effective address** to the **destination**
  - used to compactly describe common arithmetic operations
    - if %rdx contains value x,
    - then `leaq 7(%rdx, %rdx, 4), %rax --> %rax = 5x + 7`
  - **destination** operand **must** be a **register**

## H5 ¶3.5.2 Unary and Binary Operations

1	Instruction	Effect	Description
2	-----		
3	INC     D	D <- D+1	Increment
4	DEC     D	D <- D-1	Decrement
5	NEG     D	D <- -D	Negate
6	NOT     D	D <- ~D	Complement

- Unary operations takes **single operand** serving as both **source** and **destination**
  - can be either a **register** or a **memory location**
    - e.g. `incq(%rsp)` --> increments the 8-byte element on the top of the stack
- reminiscent of the C increment(++ ) and decrement(-- ) ops

1	Instruction	Effect	Description
2	-----		
3	ADD     S,D	D <- D + S	Add
4	SUB     S,D	D <- D - S	Subtract
5	IMUL    S,D	D <- D * S	Multiply
6	XOR     S,D	D <- D ^ S	Exclusive-or
7	OR      S,D	D <- D   S	Or
8	AND     S,D	D <- D & S	And

- **Binary operations** takes the **second operand** which is used as both a **source** and a **destination**
- reminiscent of the C assignment operators (e.g `x -= y;`)
- **First operand** can be
  - Immediate value, register, memory location
- **Second operand** can be
  - register, memory location
- **Two operands cannot both be memory locations**
  1. processor must read the value from memory
  2. perform the operation
  3. write the result back to memory

## H5 ¶3.5.3 Shift Operations

1	Instruction	Effect	Description
2	-----		
3	SAL <i>k,D</i>	$D \leftarrow D \ll k$	Left shift
4	SHL <i>k,D</i>	$D \leftarrow D \ll k$	Left shift (same as SAL)
5	SAR <i>k,D</i>	$D \leftarrow D \gg Ak$	Arithmetic right shift
6	SHR <i>k,D</i>	$D \leftarrow D \gg LK$	Logical right shift

- **First operand = shift amount**
  - specified as an **immediate value** or with the **single-byte** register **%cl**
    - possible to encode shift amounts up to  $2^8 - 1 = 255$
  - With x86-64, shift instruction operating on data values that are  $w$  bits long determines the shift amount from the lower-order  $m$ -bits of register **%cl**, where  $2^m = w$ 
    - **higher-order bits are ignored**

e.g. If **%cl** = 0xFF, `salb` would shift by 7, `salw` would shift by 15, `salq` would shift by 31
- **Second operand = value to shift**
  - can be **register** or **memory location**
- SAL & SHL have the **same effect** (filling from the right with **0's**)
- SAR - arithmetic shift (fill with copies of the sign bit (**MSB**))
- SHR - logical shift (fill with **0's**)

## H6 ¶3.5.4 Discussion

- most of the instructions can be used for either **unsigned** or **two's-complement arithmetic**
- Only Right Shifting needs to **differentiate between signed vs. unsigned data**
- Example [C code - Assembly code] for Arithmetic functions

```
1 long arith(long x, long y, long z) {
2     long t1 = x ^ y;
3     long t2 = x * 48;
4     long t3 = t1 & 0xF0F0F0F;
5     long t4 = t2 - t3;
6     return t4;
7 }
```

```
1 # long arith(long x, long y, long z)
2 # x in %rdi, y in %rsi, z in %rdx
3 arith:
4     xorq    %rsi, %rdi          # t1 = x ^ y
5     leaq    (%rdx,%rdx,2), %rax  # 3*z
6     salq    $4, %rax            # t2 = 16 * (3*z)
7     andl    $252645135, %edi     # t3 = t1 & 0xF0F0F0F
8     subq    %rdi, %rax          # return t2 - t3
9     ret
```

## H5 ¶3.5.5 Special Arithmetic Operations

1	Instruction	Effect	Description
2	-----	-----	-----
3	<code>imulq S</code>	<code>R[%rdx]:R[%rax]&lt;-SxR[%rax]</code>	Signed full multiply
4	<code>mulq S</code>	<code>R[%rdx]:R[%rax]&lt;-SxR[%rax]</code>	Unsigned full multiply
5	<code>cqto</code>	<code>R[%rdx]:R[%rax]&lt;-SignExtend(R[%rax])</code>	Convert to oct word
6	<code>idivq S</code>	<code>R[%rdx]&lt;-R[%rdx]:R[%rax]mod S;</code>	Signed divide
7		<code>R[%rax]&lt;-R[%rdx]:R[%rax]/S</code>	
8	<code>divq S</code>	<code>R[%rdx]&lt;-R[%rdx]:R[%rax]mod S;</code>	Unsigned divide
9		<code>R[%rax]&lt;-R[%rdx]:R[%rax]/S</code>	

- X86-64 instruction set provides *limited* support for operations involving **128-bit (16-byte)** numbers (oct word)
- **`imulq`** instruction has **two different forms**
  1. as a member of the **IMUL** instruction class (as mentioned before)
    - takes two operands and generates a 64-bit product
    - $*_{64}^u$  and  $*_{64}^t$  -- same bit-level behavior
      - truncates the product to 64 bits
  2. two different "**One-operand**" multiply instructions to compute the **full 128-bit** product of **64-bit values**
    1. **`mulq`** - used for **unsigned** 128-bit products
    2. **`imulq`** - used for **two's-complement** 128-bit products
      - One argument must be in register **`%rax`** , and the other given as the **instruction source operand**
      - **Product stored** in **`%rdx`** ( **high** -order 64 bits) and **`%rax`** ( **low** -order 64 bits)

[Example C code]

```

1  #include <inttype.h>
2
3  typedef unsigned __int128 uint128_t;
4
5  void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {
6      *dest = x * (uint128_t) y;
7  }

```

[Corresponding Assembly code]

```

1  # void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
2  # dest in %rdi, x in %rsi, y in %rdx
3  store_uprod:
4      movq    %rsi, %rax    # Copy x to multiplicand
5      mulq    %rdx          # Multiply by y
6      movq    %rax, (%rdi)  # Store lower 8 bytes at dest
7      movq    %rdx, 8(%rdi) # Store upper 8 bytes at dest+8
8      ret

```

- storing the product requires **two movq** instructions
  1. for the low-order 8 bytes (L6)
  2. for the high-order 8 bytes (L7)
- Similarly, for divisions with 128-bit quantity
  - **idivl**
    - takes as its **dividend** the 128-bit quantity in registers **%rdx (high-order 64 bits)** and **%rax (low-order 64 bits)**
    - **divisor** is given as the **instruction operand**
    - **stores quotient** in **%rax** and **remainder** in **%rdx**
  - **cqto**
    - used for sign extension
    - reads the **sign bit** from **%rax** and **copies** it across all of **%rdx**

[Example C code]

```

1 void remdiv(long x, long y, long *qp, long *rp) {
2     long q = x/y;
3     long r = x%y;
4     *qp = q;
5     *rp = r;
6 }

```

[Corresponding Assembly code]

```

1 # void remdiv(long x, long y, long *qp, long *rp)
2 # x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
3 remdiv:
4     movq    %rdx, %r8    # Copy qp
5     movq    %rdi, %rax    # Move x to lower 8 bytes of dividend
6     cqto    # Sign-extend to upper 8 bytes of dividend
7     idivq   %rsi          # Divide by y
8     movq    %rax, (%r8)   # Store quotient at qp
9     movq    %rdx, (%rcx)  # Store remainder at rp
10    ret

```

- **qp must** first be saved in a **different register**
  - (∵ **%rdx** is required for the division operation)
- After the division, **quotient** in **%rax** gets stored at **qp** (L8), and the **remainder** in **%rdx** gets stored at **rp** (L9)
- Similarly, **unsigned division** uses **divq** instruction with **%rdx** filled with **0** beforehand.

### H3 §3.6 Control

- **Conditionals, loops, switches** -- require conditional execution
  - operations depend on the outcomes of tests
- Two basic low-level mechanisms
  1. altering control flow
  2. altering data flow

based on the results of tests
- Normally, statements and instructions are executed **sequentially**
- **jump** instructions are used to indicate that control should pass to some other part of the program

## H5 ¶3.6.1 Condition Codes

- CPU maintains a set of **single-bit condition code** registers
  - describes the **most recent arithmetic or logical** operation
  - can be tested to perform **conditional branches**
- Commonly used condition codes
  - **CF** : carry flag -- unsigned overflow
  - **ZF** : zero flag -- zero
  - **SF** : sign flag -- negative value
  - **OF** : overflow flag -- two's-complement overflow

Suppose we performed the C assignment  $t = a + b$ , condition codes would be set when

CF  $(\text{unsigned})\ t < (\text{unsigned})\ a$  Unsigned overflow

ZF  $(t == 0)$  Zero

SF  $(t < 0)$  Negative

OF  $(a < 0 == b < 0) \&\&(t < 0 != a < 0)$  Signed overflow

- **leaq** instruction does not alter any condition codes
  - it is intended to be used in **address computation**
- Otherwise, all of the aforementioned instructions cause the condition codes to be set

1	Instruction	Based on	Description
2	-----		
3	CMP S1,S2	S2 - S1	Compare
4	cmpb		Compare byte
5	cmpw		Compare word
6	cmpl		Compare double word
7	cmpq		Compare quad word
8			
9	TEST S1,S2	S1 & S2	Test
10	testb		Test byte
11	testw		Test word
12	testl		Test double word
13	testq		Test quad word

- **CMP** instructions and **TEST** instructions set condition codes without altering any other registers



- **CMP** - sets condition codes according to the **differences** of its two operands
  - $\approx$  SUB instruction but **DOES NOT** update its destination
- **TEST** - sets condition codes according to the result of **AND** operation applied to its two operands
  - $\approx$  AND instruction but **DOES NOT** update its destination
  - typically, the same operand is repeated (testq %rax,%rax)
    - to see whether %rax is negative, zero, or positive
  - or one of the operands is a mask indicating which bits should be tested

## H5 ¶3.6.2 Accessing the Condition Codes

- $\exists$  3 common ways of using the condition codes (rather than reading the condition codes directly)
  1. set a **single byte** to **0** or **1** depending on some combination of the condition codes
  2. conditionally jump to some other part of the program
  3. conditionally transfer data
- **SET** instructions

1	Instruction	Synonym	Effect	Set condition
2	-----			
3	sete	D setz	D $\leftarrow$ ZF	Equal/zero
4	setne	D setnz	D $\leftarrow$ $\sim$ ZF	Not equal/not zero
5	sets	D	D $\leftarrow$ SF	Negative
6	setns	D	D $\leftarrow$ $\sim$ SF	Nonnegative
7	setg	D setnle	D $\leftarrow$ $\sim$ (SF $\wedge$ OF) $\wedge$ $\sim$ ZF	Greater (signed >)
8	setge	D setnl	D $\leftarrow$ $\sim$ (SF $\wedge$ OF)	Greater or equal (signed >=)
9	setl	D setnge	D $\leftarrow$ SF $\wedge$ OF	Less (signed <)
10	setle	D setng	D $\leftarrow$ (SF $\wedge$ OF)   ZF	Less or equal (signed <=)
11	seta	D setnbe	D $\leftarrow$ $\sim$ CF $\wedge$ $\sim$ ZF	Above (unsigned >)
12	setae	D setnb	D $\leftarrow$ $\sim$ CF	Above or equal (unsigned >=)
13	setb	D setnae	D $\leftarrow$ CF	Below (unsigned <)
14	setbe	D setna	D $\leftarrow$ CF   ZF	Below or equal (unsigned <=)

- Assembly code that computes  $a < b$

```

1 # int comp(data_t a, data_t b)
2 # a in %rdi, b in %rsi
3 comp:
4     cmpq    %rsi, %rdi    # Compare a:b
5     setl    %al           # Set low-order byte of %eax to 0 or 1
6     movzbl  %al, %eax     # Clear rest of %eax (and rest of %rax)
7     ret

```

- Note that...
  - **signed comparison tests** are based on combinations of **SF ^ OF** and **ZF**
  - **unsigned comparison tests** are based on combinations of **CF** and **ZF**

## H5 ¶3.6.3 Jump Instructions

### 2. conditionally jump to some other part of the program

- causes the execution to switch to a completely new position in the program
- destinations are indicated in **label**

```

1     movq    $0, %rax      # Set %rax to 0
2     jmp     .L1           # Goto .L1
3     movq    (%rax), %rdx   # Null pointer dereference (skipped)
4 .L1:
5     popq    %rdx          # Jump target

```

Instruction	Synonym	Jump condition	Description
-----			
jmp Label		1	Direct jump
jmp *Operand		1	Indirect jump
je Label	jz	ZF	Equal/zero
jne Label	jnz	~ZF	Not equal/not zero
js Label		SF	Negative
jns Label		~SF	Nonnegative
jg Label	jnle	~(SF^OF)&~ZF	Greater (signed >)
		~(SF^OF)	Greater or equal (signed >=)
jle Label	jnge	SF^OF	Less (signed <)
jle Label	jng	(SF^OF) ZF	Less or equal (signed <=)
ja Label	jnbe	~CF&~ZF	Above (unsigned >)

13	<code>jae</code>	<code>Label</code>	<code>jnb</code>	<code>~CF</code>	Above or equal (unsigned >=)
14	<code>jb</code>	<code>Label</code>	<code>jnae</code>	<code>CF</code>	Below (unsigned <)
15	<code>jbe</code>	<code>Label</code>	<code>jna</code>	<code>CF ZF</code>	Below or equal (unsigned <=)

- while generating the object-code file, the assembler determines the address of all labeled instructions and encodes the **jump targets** (addresses of the destination instructions)
- **jmp** instruction **jumps unconditionally**
  - **direct jump**: jump target encoded as part of the instruction  
e.g. `jmp .L1`
  - **indirect jump**: jump target is read from a register or a memory location  
e.g. `jmp %rax` --> uses the value in register `%rax`  
`jmp *(%rax)` --> reads the jump target from memory, using `%rax` as the read address
- Remaining jump instructions are **conditional**
  - either jump or continue executing at the next instruction in the code sequence, **depending on** some combination of the **condition codes**
  - Can **only** be **direct**

## H5 ¶3.6.4 Jump Instruction Encodings

- important to understand how the targets of jump instructions are encoded --> **CH7. Linking**
- In assembly code, jump targets are written using **symbolic labels**
  - **Assembler** and **linker** generate the proper encodings of the jump target
- **Two** commonly used **encodings** for jump targets
  - **PC relative**
    - **difference** between the address of the **target instruction** and the address of the **instruction immediately following the jump**
    - offsets encoded using **1, 2, or 4 bytes**

Example Assembly Code

```

1    movq    %rdi, %rax
2    jmp     .L2
3  .L3:
4    sarq    %rax
5  .L2:
6    testq   %rax, %rax
7    jg      .L3
8    rep; ret

```

Corresponding **.o** generated by the **assembler**

```

1  0:  48 89 f8          mov    %rdi,%rax
2  3:  eb 03              jmp     8 <loop+0x8>
3  5:  48 d1 f8          sar    %rax
4  8:  48 85 c0          test   %rax,%rax
5  b:  7f f8            jg      5 <loop+0x5>
6  d:  f3 c3          repz retq

```

- e.g. 03 in L2 + 05 in L3 = 0x08 (jump target address)
- value of the **PC** = instruction **following the jump**

Disassembled version **after linking**

```

1  4004d0: 48 89 f8          mov    %rdi,%rax
2  4004d3: eb 03              jmp     8 <loop+0x8>
3  4004d5: 48 d1 f8          sar    %rax
4  4004d8: 48 85 c0          test   %rax,%rax
5  4004db: 7f f8            jg      5 <loop+0x5>
6  4004dd: f3 c3          repz retq

```

- instructions have been **relocated** to different addresses
  - encodings of the jump targets **remain unchanged**
- With **PC-relative** encoding, instructions can be **compactly encoded**, and object code can be shifted to different positions in memory **without alteration**
- **Absolute**
  - **4 bytes** to directly specify the target address

## H5 ¶3.6.5 Implementing Conditional Branches with Conditional Control

- most general way to translate conditional expressions and statements from C into machine code = **conditional** + **unconditional** jumps

[Example C code]

```
1  long lt_cnt = 0;
2  long ge_cnt = 0;
3
4  long absdiff_se(long x, long y) {
5      long result;
6      if (x < y) {
7          lt_cnt++;
8          result = y - x;
9      }
10     else {
11         ge_cnt++;
12         result = x - y;
13     }
14     return result;
15 }
```

[Equivalent goto version - C code]

```
1  long gotodiff_se(long x, long y) {
2      long result;
3      if(x >= y)
4          goto x_ge_y;
5      lt_cnt++;
6      result = y - x;
7      return result;
8  x_ge_y:
9      ge_cnt++;
10     result = x - y;
11     return result;
12 }
```

[Generated assembly code]

```
1  # long absdiff_se(long x, long y)
```

```

2  # x in %rdi, y in %rsi
3  absdiff_se:
4      cmpq    %rsi, %rdi          # Compare x:y
5      jge     .L2                 # If >= goto x_ge_y
6      addq    $1, lt_cnt(%rip)    # lt_cnt++
7      movq    %rsi, %rax
8      subq    %rdi, %rax          # result = y - x
9      ret                                # Return
10  .L2:                             # x_ge_y:
11      addq    $1, ge_cnt(%rip)    # ge_cnt++
12      movq    %rdi, %rax
13      subq    %rsi, %rax          # result = x - y
14      ret                                # return

```

As shown above, the assembly implementation typically adheres to the following form

```

1      t = test-expr;
2      if(!t)
3          goto false;
4      then-statement
5      goto done;
6  false:
7      else-statement
8  done:

```

- compiler generates separate blocks of code for `then-statement` and `else-statement`

## H5 ¶3.6.6 Implementing Conditional Branches with Conditional Moves

- Conventional way to implement conditional operations is through a **conditional transfer of control**
  - program follows one execution path when a condition holds and another when it does not
  - can be very **inefficient** on modern processors
- Alternate strategy: --> through **conditional transfer of data**
  - computes **both outcomes of a conditional operation** and then selects one based on whether or not the condition holds

- can be implemented by a simple conditional move

[Example C code]

```
1 long absdiff(long x, long y) {
2     long result;
3     if (x < y)
4         result = y - x;
5     else
6         result = x - y;
7     return result;
8 }
```

--> similar to the one used in **3.5.5**, however this does not have *lt\_cnt* or *ge\_cnt* (side effects).

[Implementation using **conditional assignment**]

```
1 long cmovdiff(long x, long y) {
2     long rval = y - x;
3     long eval = x - y;
4     long ntest = x >= y;
5     /* Line below requires single instruction */
6     if (ntest) rval = eval;
7     return rval;
8 }
```

--> computes both  $y-x$  and  $x-y$

[Generated **Assembly code**]

```

1  # long absdiff(long x, long y)
2  # x in %rdi, y in %rsi
3  absdiff:
4      movq    %rsi, %rax
5      subq    %rdi, %rax    # rval = y-x
6      movq    %rdi, %rdx
7      subq    %rsi, %rdx    # eval = x-y
8      cmpq    %rsi, %rdi    # Compare x:y
9      cmovge  %rdx, %rax    # If >=, rval = eval
10     ret      # Return rval

```

--> cmovge instruction implements the conditional assignment (L6) of cmovdiff

- Reason why **conditional data transfers** can **outperform conditional control transfers**
  - processors use **pipelining** to achieve high performance
    - instruction is processed via a **sequence of stages**, each performing **one small portion** of the required operations
    - high performance** by **overlapping** the steps of the successive instructions
  - When the machine **encounters a conditional jump** ("branch"), **cannot determine** which way the branch will go **until it has evaluated the branch condition**
    - ∃ **branch prediction logic** to guess
      - misprediction** can incur a **serious penalty** (≈15-30 clock cycles) --> **degradation of program performance**
        - dominates** the performance of the function
- conditional move instructions**

1	Instruction	Synonym	Move condition	Description
2	-----			
3	cmovz S,R	cmovz	ZF	Equal/zero
4	cmovne S,R	cmovnz	~ZF	Not equal/not zero
5	cmovs S,R		SF	Negative
6	cmovns S,R		~SF	Nonnegative
7	cmovg S,R	cmovnle	~(SF^OF)&~ZF	Greater (signed >)
8	cmovge S,R	cmovnl	~(SF^OF)	Greater or equal (signed >=)
9	cmovl S,R	cmovnge	SF^OF	Less (signed <)
10	cmovle S,R	cmovng	(SF^OF) ZF	Less or equal (signed <=)



11	<code>cmova</code>	<code>S,R</code>	<code>cmovnbe</code>	<code>~CF&amp;~ZF</code>	Above (unsigned >)
12	<code>cmovae</code>	<code>S,R</code>	<code>cmovnb</code>	<code>~CF</code>	Above or equal (unsigned >=)
13	<code>cmovb</code>	<code>S,R</code>	<code>cmovnae</code>	<code>CF</code>	Below (unsigned <)
14	<code>cmovbe</code>	<code>S,R</code>	<code>cmovna</code>	<code>CF ZF</code>	Below or equal (unsigned <=)

- Not all conditional expressions can be compiled using conditional moves
  - if one of the two expressions could **possibly generate an error condition** or a **side effect** --> **invalid behavior**
  - put **side effects** to **force** GCC to implement the function using **conditional transfers**

[Example C code]

```
1 long cread(long *xp) {
2     return (xp ? *xp : 0);
3 }
```

[Corresponding assembly code with **Conditional move**]

```
1 # long cread(long *xp)
2 # Invalid implementation of function cread
3 # xp in register %rdi
4 cread:
5     movq    (%rdi), %rax    # v = *xp
6     testq   %rdi, %rdi     # Test x
7     movl    $0, %edx       # Set ve = 0
8     cmovbe  %rdx, %rax     # If x==0, v = ve
9     ret                               # Return v
```

- **Invalid** implementation --> dereferencing of xp by the `movq` instruction (L5) occurs even when the test fails --> causing a null pointer dereferencing error
- Conditional moves **does not always** improve code efficiency
  - when either evaluation requires a **significant computation**
    - effort wasted when the corresponding condition does not hold

- GCC only uses **conditional moves** only when the two expressions **can be computed very easily**
  - e.g. single add instruction
  - uses **conditional control transfers** even where the cost of branch misprediction would **exceed** even more complex computations

## H5 ¶3.6.7 Loops

- C provides several looping constructs
  - **do-while, while, for**
- no corresponding instructions exist in machine code
  - > implemented with combinations of **conditional tests** and **jumps**

## H6 Do-While Loops

```

1  do
2      body-statement
3      while (test-expr);

```

- repeatedly execute **body-statement**, evaluate **test-expr** and continue the loop if the **evaluation result is nonzero (true)**
  - body-statement is executed at least once

[Corresponding **goto** statements]

```

1  loop:
2      body-statement
3      t = test-expr;
4      if(t)
5          goto loop;

```

- program evaluates the body statement, then the test expression
  - if test succeeds, the program goes back for **another iteration**

[Example C code]

```

1  long fact_do(long n) {
2      long result = 1;
3      do {
4          result *= n;
5          n = n-1;
6      } while (n > 1);
7      return result;
8  }

```

- computes the **factorial** of its argument ( $n!$ )

[Equivalent **goto** version]

```

1  long fact_do_goto(long n) {
2      long result = 1;
3      loop:
4          result *= n;
5          n = n-1;
6          if (n > 1)
7              goto loop;
8      return result;
9  }

```

[Corresponding **assembly-language** code]

```

1  # long fact_do(long n)
2  # n in %rdi
3  fact_do:
4      movl    $1, %eax          # Set result = 1
5  .L2:                                # loop:
6      imulq   %rdi, %rax        # Compute result *= n
7      subq    $1, %rdi          # Decrement n
8      cmpq    $1, %rdi          # Compare n:1
9      jg      .L2               # If >, goto loop
10     rep; ret                  # Return

```

- jump instruction **jg** (L9) is the key instruction in implementing a loop
  - **determines** whether to **continue iterating** or to **exit**

## H6 While Loops

- loop is potentially terminated before the first execution of **body-statement**

### 1. *jump to middle method*

- generated by **GCC** when optimization is specified as -Og.

```
1     goto test;
2 loop:
3     body-statement
4 test:
5     t = test-expr;
6     if(t)
7         goto loop;
```

[Example C code]

```
1 long fact_while(long n) {
2     long result = 1;
3     while (n > 1) {
4         result *= n;
5         n = n-1;
6     }
7     return result;
8 }
```

[Equivalent **goto** version]

```
1 long fact_while_jm_goto(long n) {
2     long result = 1;
3     goto test;
4 loop:
5     result *= n;
6     n = n-1;
7 test:
8     if (n > 1)
9         goto loop;
10    return result;
11 }
```

[Corresponding **Assembly-language** code]

```

1  # long fact_while(long n)
2  # n in %rdi
3  fact_while:
4      movl    $1, %eax          # Set result = 1
5      jmp     .L5               # Goto test
6  .L6:                          # loop:
7      imulq   %rdi, %rax        # Compute result *= n
8      subq    $1, %rdi          # Decrement n
9  .L5:                          # test:
10     cmpq    $1, %rdi          # Compare n:1
11     jg      .L6               # If >, goto loop
12     rep; ret                  # Return

```

## 2. guarded do method

```

1  t = test-expr;
2  if(!t)
3      goto done;
4  loop:
5      body-statement
6      t = test-expr;
7      if (t)
8          goto loop;

```

- transforms the code into a **do-while** loop by using a conditional branch to skip over the loop if the initial test fails
- **GCC** uses this when compiling with higher levels of optimization (-O1)
- compiler can optimize the initial test to **always hold**

[C code]

```

1  long fact_while (long n) {
2      long result = 1;
3      while (n > 1) {
4          result *= n;
5          n = n-1;
6      }
7      return result;
8  }

```

[Equivalent goto version]

```

1  long fact_while_gd_goto(long n) {
2      long result = 1;
3      if (n <= 1)
4          goto done;
5      loop:
6          result *= n;
7          n = n-1;
8          if (n != 1)
9              goto loop;
10     done:
11         return result;
12 }

```

[Corresponding **assembly**-language code (**-O1**)]

```

1  # long fact_while(long n)
2  # n in %rdi
3  fact_while:
4      cmpq    $1, %rdi      # Compare n:1
5      jle     .L7           # If <=, goto done
6      movl    $1, %eax      # Set result = 1
7  .L6:                # loop:
8      imulq   %rdi, %rax    # Compute result *= n
9      subq    $1, %rdi      # Decrement n
10     cmpq    $1, %rdi      # Compare n:1
11     jne     .L6           # If !=, goto loop
12     rep; ret              # Return
13  .L7:                # Done:
14     movl    $1, %eax      # Compute result = 1
15     ret                # Return

```

- **jle** instruction on L5 skips over the loop code when the **initial test fails**

## H6 For Loops

```

1  for (init-expr; test-expr; update-expr)
2      body-statement

```

- code generated by **GCC** for a **for loop** then follows one of the two translation strategies
  1. **jump-to-middle**

```

1    init-expr;
2    goto test;
3  loop:
4    body-statement
5    update-expr;
6  test:
7    t = test-expr;
8    if (t)
9      goto loop;

```

## 2. guarded-do

```

1    init-expr;
2    t = test-expr;
3    if(!t)
4      goto done;
5  loop:
6    body-statement
7    update-expr;
8    t = test-expr;
9    if(t)
10     goto loop;
11  done:

```

[Example C code - **for loop**]

```

1  long fact_for(long n) {
2    long i;
3    long result = 1;
4    for (i = 2; i <= n; i++)
5      result *= i;
6    return result;
7  }

```

[Corresponding **jump-to-middle** transformation]

```

1  long_fact_for_jm_goto(long n) {
2      long i = 2;
3      long result = 1;
4      goto test;
5  loop:
6      result *= i;
7      i++;
8  test:
9      if (1 <= n)
10         goto loop;
11     return result;
12 }

```

[Assembly code with -Og optimization level]

```

1  # long fact_for(long n)
2  # n in %rdi
3  fact_for:
4      movl    $1, %eax        # Set result = 1
5      movl    $2, %edx        # Set i = 2
6      jmp     .L8             # Goto test
7  .L9:                        # loop:
8      imulq   %rdx, %rax      # Compute result *= i
9      addq    $1, %rdx        # Increment i
10 .L8:                        # test:
11     cmpq    %rdi, %rdx      # Compare i:n
12     jle     .L9             # If <=, goto loop
13     rep; ret                # Return

```

## H5 ¶3.6.8 Switch Statements

- **switch** statement -> provides a **multiway branching capability** based on the value of an integer index
- **useful** when dealing with tests where there can be a **large number of possible outcomes**
- allow an **efficient implementation** using a data structure called a **jump table**
  - array where entry **i** is the **address** of a code segment implementing the action the program should take when the **switch index equals i**
  - code performs an **array reference** into the jump table using the **switch index** to determine the **target for a jump instruction**



- **ADVANTAGE**

- **time** taken to perform the **switch** is **independent** of the **number of switch cases**
- **GCC** selects the method of translating **switch statement** based on
  1. **number of cases (4 or more)**
  2. **span a small range of values**

[Example C code]

```
1 void switch_eg (long x, long n, long *dest) {
2     long val = x;
3
4     switch (n) {
5         case 100:
6             val *= 13;
7             break;
8
9         case 102:
10            val == 10;
11            // Falls through
12
13        case 103:
14            val += 11;
15            break;
16
17        case 104:
18        case 105:
19            val *= val;
20            break;
21
22        default:
23            val = 0;
24    }
25    *dest = val;
26 }
```

- Features to note
  1. range **NOT contiguous**
  2.  $\exists$  cases that **fall through** to other cases  
( $\therefore$   $\nexists$  break statement)

[Translation into **extended C**]

```
1 void switch_eg_impl (long x, long n, long *dest) {
2     // Table of code pointers
3     static void *jt[7] = {
4         &&loc_A, &&loc_def, &&loc_B,
5         &&loc_C, &&loc_D, &&loc_def, &&loc_D
6     };
7     unsigned long index = n - 100;
8     long val;
9
10    if (index > 6)
11        goto loc_def;
12    // Multiway branch
13    goto *jt[index];
14
15    loc_A:    // Case 100
16        val = x * 13;
17        goto done;
18    loc_B:    // Case 102
19        x = x + 10;
20        // Falls through
21    loc_C:    // Case 103
22        val = x + 11;
23        goto done;
24    loc_D:    // Case 104, 106
25        val = x * x;
26        goto done;
27    loc_def:  // Default case
28        val = 0;
29    done:
30        *dest = val;
31 }
```

- makes use of support provided by **GCC** for **jump table**
- array **jt** contains 7 entries, each of which is the **address of a block of code**
  - **locations** are defined by labels in the code & indicated in the entries in **jt** by code pointers (&&: pointer for a code location)
- **computed goto** is supported by **GCC** as an extension to the C language

[Assembly code]

```

1  # void switch_eg (long x, long n, long *dest)
2  # x in %rdi, n in %rsi, dest in %rdx
3  switch_eg:
4      subq    $100, %rsi          # Compute index = n-100
5      cmpq    $6, %rsi           # Compare index:6
6      ja      .L8                 # If >, goto loc_def
7      jmp     *.L4(,%rsi,8)       # Goto *jg[index]
8  .L3:                          # loc_A:
9      leaq    (%rdi,%rdi,2), %rax # 3*x
10     leaq    (%rdi,%rax,4), %rdi  # val = 13*x
11     jmp     .L2                 # Goto done
12  .L5:                          # loc_B:
13     addq    $10, %rdi           # x = x + 10
14  .L6:                          # loc_C:
15     addq    $11, %rdi           # val = x + 11
16     jmp     .L2                 # Goto done
17  .L7:                          # loc_D:
18     imulq   %rdi, %rdi          # val = x * x
19     jmp     .L2                 # Goto done
20  .L8:                          # loc_def:
21     movl     $0, %edi            # val = 0
22  .L2:                          # done:
23     movq     %rdi, (%rdx)        # *dest = val
24     ret                          # Return

```

- **jmp** instruction has an operand prefixed with \*
  - ≈ **computed goto** in extended C

#### [Jump table]

```

1      .section      .rodata
2      .align 8      # Align address to multiple of 8
3  .L4:
4      .quad .L3      # Case 100: loc_A
5      .quad .L8      # Case 101: loc_def
6      .quad .L5      # Case 102: loc_B
7      .quad .L6      # Case 103: loc_C
8      .quad .L7      # Case 104: loc_D
9      .quad .L8      # Case 105: loc_def
10     .quad .L7      # Case 106: loc_D

```

- **.rodata**: "read-only data"
- sequence of seven **"quad"** (8-byte) words

- value of each word is given by the instruction address associated with the indicated assembly-code labels
- **.L4** marks the start of the allocation
  - associated address serves as the **base for the indirect jump**
- Use of a **jump table** allows a very **efficient way** to implement a **multiway branch**
  - even with 100+ cases, can be handled by a single jump table access

### H3 §3.7 Procedures

- **Procedures:** key abstraction in software
  - package code that implements some functionality with a designed set of arguments and an optional return value
  - **function** can be **invoked** from **different points** in a program
  - **hides** detailed implementation + **provide** a **clear & concise** interface definition

Suppose procedure **P** calls procedure **Q**, and **Q** returns back to **P** after execution.

One or more of the following mechanisms need to be involved

#### 1. **Passing control**

- **PC** must be set to the **starting address of the code for Q** upon entry --> set to the **instruction in P following the call to Q** upon return

#### 2. **Passing data**

- **P** must be able to provide one or more **parameters** to **Q**, and **Q** must be able to **return** a value back to **P**

#### 3. **Allocating and deallocating memory**

- **Q** may need to **allocate space for local variables** when it begins, and then **free that storage before it returns**

- **x86-64** implementation has tried to **minimize the overhead** involved in invoking a procedure
  - implements **only as much** of the above set of mechanisms as is required for each particular procedure

## H5 ¶3.7.1. The Run-Time Stack

- key feature of the **procedure-calling** mechanism is **LIFO** (last-in, first-out) memory management discipline provided by a **stack** data structure

Suppose P is calling Q

- while Q is executing, P and any of the procedures in the chain of calls up to P is **temporarily suspended**
  - only Q will need the ability to **allocate new storage for its local variables** or to **set up a call to another procedure**
    - when **Q returns** , **local storage can be freed**
- Hence, program can manage the **storage required by its procedures** using a **Stack**
  - As P calls Q, **control & data information** are added to the **end of the stack**
    - gets **deallocated** when **Q returns**
- **Stack grows toward lower addresses**, and the **stack pointer %rsp** points to the **top element of the stack (lowest address)**
- **pushq**: Store data on stack
- **popq**: retrieve data from stack
- can **allocate space** for data with no specific initial value by **decrementing the stack pointer**
  - **deallocate** by **incrementing the stack pointer**
- When procedure requires storage beyond what it can hold in registers, it allocates **space on the stack** (**Stack frame**)
  - **frame** for the **currently executing procedure** is always at the **top of the stack**
    - When P calls Q, it will **push** the **return address onto the stack** (where within P the program should **resume execution** once Q returns)
- Stack frames for most procedures are of **fixed size** , allocated at the **beginning of the procedure**
- Some require **variable-size frames** (¶3.10.5)
- **P** can pass up to **six integral values** on the **registers** , but if **Q requires more arguments** , these values can be **stored by P** within its stack frame before the call

- Due to **space & time efficiency**, x86-64 procedures allocate **only the portions of the stack frames they require**
  - many procedures have six or fewer arguments (**can be passed in registers**)
  - **Majority** of functions **do not require** a stack frame
    - local variables can be held in registers
    - function does not call any other functions ( **leaf procedure** )

## H5 ¶3.7.2 Control Transfer

1	Instruction	Description
2	-----	
3	call     Label	Procedure call (direct)
4	call *Operand	Procedure call (indirect)
5	ret	Return from call

- **passing control** from P to Q by setting **PC (%rip)** to the **starting address of Q**
- when Q returns, the processor **must have** some record of the **code location** where it should **resume the execution of P**
  - recorded in x86-64 by **invoking procedure Q** with the instruction **call Q**
    - pushes an address **A (return address)** onto the stack and sets **PC** to **the beginning of Q**
      - **return address** is computed as the address of the instruction **immediately following the call instruction**
  - **ret** instruction **pops** an address **A** off the stack, and sets **PC to A**
- **call** instruction has a **target** indicating the **address of the instruction** where the **called procedure starts**
  - can be **direct** or **indirect**
    - **direct** - target given as a **label**  
e.g. callq 400540
    - **indirect** - target given by '\*' followed by an **operand specifier**

## H5 ¶3.7.3 Data Transfer

- procedure **calls** may involve **passing data** as arguments + **returning** from a procedure may also involve **returning a value**
  - with x86-64, most of data passing take place via **registers**
    - arguments passed in registers %rdi, %rsi, and others
    - values returned in register **%rax**
- when P calls Q, **code for P must first copy the arguments into the proper registers**
- when Q returns back to P, **code for P can access the returned value in %rax**
- with x86-64, **six integral arguments** (integer, pointer) can be passed **via registers**
  - used in a **specified order**

1	Operand	Argument number					
2	size(bits)	1	2	3	4	5	6
3	-----						
4	64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
5	32	%edi	%esi	%edx	%ecx	%r8d	%r9d
6	16	%di	%si	%dx	%cx	%r8w	%r9w
7	8	%dil	%sil	%dl	%cl	%r8b	%r9b

- arguments are allocated to these registers according to their **ordering in the argument list**
- arguments smaller than 64 bits can be accessed using the **appropriate subsection** of the 64-bit register
- if a function has **more than six integral arguments**, the other ones are **passed on the stack**
  - suppose P calls Q with  $n$  integrals, s.t.  $n > 6$ , then P **must allocate a stack frame with enough storage for arguments 7 ~  $n$** 
    - argument **7 at the top of the stack**
- when passing parameters on the stack, **all data sizes are rounded up to be multiples of eight**
- program executes a **call** instruction with the **arguments in place**
  - Q can access its arguments via **registers & stack**

- If, Q also calls another function with 6+ arguments, **Q allocates space within its stack frame**

[Example C code]

```

1 void proc (long a1, long *a1p,
2             int a2, int *a2p,
3             short a3, short *a3p,
4             char a4, short *a4p) {
5     *a1p += a1;
6     *a2p += a2;
7     *a3p += a3;
8     *a4p += a4;
9 }

```

```

1 - Function with 8 arguments with different sizes & types

```

[Generated **Assembly code**]

```

1 # void proc(a1, alp, a2, a2p, a3, a3p, a4, a4p)
2 # Arguments passed as follows:
3 # a1 in %rdi (64b)
4 # alp in %rsi (64b)
5 # a2 in %edx (32b)
6 # a2p in %rcx (64b)
7 # a3 in %r8w (16b)
8 # a3p in %r9 (64b)
9 # a4 at %rsp+8 (8b)
10 # a4p at %rsp+16 (64b)
11 proc:
12 movq 16(%rsp), %rax # Fetch a4p (64b)
13 addq %rdi, (%rsi) # *alp += a1 (64b)
14 addl %edx, (%rcx) # *a2p += a2 (32b)
15 addw %r8w, (%r9) # *a3p += a3 (16b)
16 movl 8(%rsp), %edx # Fetch a4 (8b)
17 addb %dl, (%rax) # *a4p += a4 (8b)
18 ret # Return

```

- the first **six arguments** are passed in **registers**
- **the last two** are passed on the **stack**



[Generated **Stack frame**]

```
1  -----
2  |          a4p          | 16
3  -----
4  |          | a4 |      8
5  -----
6  | Return address | 0  <-- Stack pointer (%rsp)
7  -----
```

- 1 - Return address is pushed onto the stack as part of the **\*\*procedure call\*\***
- 2 - the two arguments, a4 and a4p are at positions 8 and 16 relative to the **\*\*stack pointer (%rsp)\*\***

## H5 ¶3.7.4 Local Storage on the Stack

- At times, **local data** must be stored in memory
  1. **not enough registers** to hold all of the local variables
  2. **address operator (&)** applied to a **local variable**, and we must be able to generate an address for it
  3. some local variables are **arrays** or **structures** and hence **must be accessed by array or structure references**
- **procedure** allocates space on the **stack frame** by decrementing the stack pointer
  - stack frame labeled "local variables"

[C code for swap\_add and calling function]

```
1  long swap_add(long *xp, long *yp) {
2      long x = *xp;
3      long y = *yp;
4      *xp = y;
5      *yp = x;
6      return x + y;
7  }
8
9  long caller() {
10     long arg1 = 534;
```

```

11     long arg2 = 1057;
12     long sum = swap_add(&arg1, &arg2);
13     long diff = arg1 - arg2;
14     return sum * diff;
15 }

```

- function **caller** creates **pointers to local variables** arg1 and arg2, then passes these to **swap\_add**

[Generated **Assembly code** for calling function]

```

1  # long caller()
2  caller:
3      subq    $16, %rsp           # Allocate 16 bytes for stack frame
4      movq    $534, (%rsp)       # Store 534 in arg1
5      movq    $1057, 8(%rsp)     # Store 1057 in arg2
6      leaq    8(%rsp), %rsi      # Compute &arg2 as second argument
7      movq    %rsp, %rdi         # Compute &arg1 as first argument
8      call    swap_add           # Call swap_add(&arg1, &arg2)
9      movq    (%rsp), %rdx       # Get arg1
10     subq    8(%rsp), %rdx       # Compute diff = arg1 - arg2
11     imulq   %rdx, %rax          # Compute sum * diff
12     addq    $16, %rsp          # Deallocate stack frame
13     ret                          # Return

```

- decrementing the stack pointer by 16 (L3) --> **allocates 16 bytes on the stack**
  - Let **S** be the value of the stack pointer
    - **&arg2 = S+8, &arg1 = S**
      - local variables arg1 and arg2 are **stored within the stack frame** at offsets 0 and 8 relative to the stack pointer ( **%rsp** )
  - **caller** retrieves the two values **from the stack** (L9-10) by **(%rsp), 8(%rsp)**
  - then the function **deallocates** its **stack frame** by **incrementing the stack pointer** by 16

[C code for calling function]

```

1  long call_proc() {
2      long  x1 = 1; int    x2 = 2;
3      short x3 = 3; char  x4 = 4;
4      proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
5      return (x1+x2) * (x3-x4);
6  }

```

[Generated **Assembly code**]

```

1  # long call_proc()
2  call proc:
3      # Set up arguments to proc
4      subq    $32, %rsp          # Allocacte 32-byte stack frame
5      movq    $1, 24(%rsp)       # Store 1 in &x1
6      movl    $2, 20(%rsp)       # Store 2 in &x2
7      movw    $3, 18(%rsp)       # Store 3 in &x3
8      movb    $5, 17(%rsp)       # Store 4 in &x4
9      leaq    17(%rsp), %rax      # Create &x4
10     movq    %rax, 8(%rsp)       # Store &x4 as arg8
11     movl    $4, (%rsp)          # Store 4 as arg7
12     leaq    18(%rsp), %r9       # Pass &x3  as arg6
13     movl    $3, %r8d            # Pass 3 as arg5
14     leaq    20(%rsp), %rcx      # Pass &x2 as arg4
15     movl    $2, %edx            # Pass 2 as arg3
16     leaq    24(%rsp), %rsi      # Pass &x1 as arg2
17     movl    $1, %edi            # Pass 1 as arg1
18     # Call proc
19     call    proc
20     # Retrieve changes to memory
21     movslq   20(%rsp), %rdx      # Get x2 and convert to long
22     addq     24(%rsp), %rdx      # Compute x1+x2
23     movswl   18(%rsp), %eax      # Get x3 and convert to int
24     movsbl   17(%rsp), %ecx      # Get x4 and convert to int
25     subl     %ecx, %eax          # Compute x3-x4
26     cltq
27     imulq    %rdx, %rax          # Compute (x1+x2) * (x3-x4)
28     addq     $32, %rsp          # Deallocate stack frame
29     ret

```

[Stack frame for function]

```

1  -----
2  |      Return address      | 32
3  -----
4  |           x1             | 24
5  -----
6  |      x2      |   x3   | x4 | 16
7  -----
8  |      Argument 8 = &x4      | 8
9  -----
10 |                      arg7->|4| 0  <-- %rsp
11 -----

```

## H5 ¶3.7.5 Local Storage in Registers

- set of program registers acts as a **single resource shared by all of the procedures**
- Thus, we must make sure that when one procedure (**caller, P**) calls another (**callee, Q**), **the callee does not overwrite some register value that the caller planned to use later**
- **uniform set of conventions** for register usage that must be respected by all procedures
  - **%rbx, %rbp, %r12-%r15** -- **callee-saved** registers
    - Q must **preserve the values of these registers**
      - ensure that they have the **same** values when Q returns to P as they did when Q was called
        1. by **not changing values at all**
        2. by **pushing the original value** on the **stack**, altering it, and then popping the **old value from the stack before returning**
          - creates the portion labeled **"Saved registers"**
      - With this convention, P can **safely store a value in a callee-saved register**
  - all other registers (except for %rsp) are **caller-saved** registers
    - **can be modified by any function**
    - **incumbent upon P** to save the data

[Example calling function]

```

1  long P(long x, long y) {
2      long u = Q(y);
3      long v = Q(x);
4      return u + v;
5  }

```

- P calls Q **twice**
  - during the first call, **must retain x** to use later
  - during the second call, **must retain Q(y)**

[Generated **Assembly code**]

```

1  # long P (long x, long y)
2  # x in %rdi, y in %rsi
3  P:
4      pushq    %rbp          # Save %rbp
5      pushq    %rbx          # Save %rbx
6      subq     $8, %rsp      # Align stack frame
7      movq     %rdi, %rbp    # Save x
8      movq     %rsi, %rdi    # Move y to first argument
9      call     Q              # Call Q(y)
10     movq     %rax, %rbx     # Save result
11     movq     %rbp, %rdi    # Move x to first argument
12     call     Q              # Call Q(x)
13     addq     %rbx, %rax     # Add saved Q(y) to Q(x)
14     addq     $8, %rsp      # Deallocate last part of stack
15     popq     %rbx          # Restore %rbx
16     popq     %rbp          # Restore %rbp
17     ret

```

- uses **two callee-saved registers**
  - **%rbp** to hold x, **%rbx** to hold Q(y)
  - **popped in the reverse order from how they were pushed** at the very end of the function

## H5 ¶3.7.6 Recursive Procedures

- the conventions we have covered allow x86-64 procedures to **call them recursively**
- **each procedure call** has its **own private space on the stack**

- **local variables** of the multiple outstanding calls **do not interfere with one another**

[Example C code]

```
1  long rfact(long n) {
2      long result;
3      if (n <= 1)
4          result = 1;
5      else
6          result = n * rfact(n-1);
7      return result;
8  }
```

[Generated **Assembly code**]

```
1  # long rfact(long n)
2  # n in %rdi
3  rfact:
4      pushq    %rbx                # Save %rbx
5      movq     %rdi, %rbx          # Store n in callee-saved register
6      movl     $1, %eax             # Set return value = 1
7      cmpq     $1, %rdi             # Compare n:1
8      jle      .L35                 # if <=, goto done
9      leaq     -1(%rdi), %rdi       # Compute n-1
10     call     rfact                # Call rfact(n-1)
11     imulq    %rbx, %rax            # Multiply result by n
12 .L35:                                # done:
13     popq     %rbx                # Restore %rbx
14     ret                                # Return
```

- assembly code uses **%rbx** to hold **n**
  - saves the existing value on the stack (L4), (**pushq %rbx**)
  - restores the value before returning (L13), (**popq %rbx**)
- Due to the stack discipline & register-saving conventions, we can be assured when the recursive call to **rfact(n-1)** returns that...
  1. **result** of the call will be held in register **%rax**
  2. value of argument **n** will be held in register **%rbx**
  - Hence, multiplying 1. and 2. computes the desired result

- **Stack discipline** provides a mechanism where **each invocation of a function** has its **own private storage for state information**
  - saved values of the return location & callee-saved registers
- also provides **storage for local variables**

### H3 §3.8 Array Allocation and Access

- with C, we can generate **pointers to elements within arrays** and **perform arithmetic** with pointers
  - address computations in machine code
- Optimizing compilers simplify the address computations used by array indexing

#### H5 ¶3.8.1 Basic Principles

- For data type **T** and integer constant **N**, consider a declaration **T A[N];**
  - Suppose it has a starting location of  $x_A$ . It has two effects
    1. allocates a contiguous region of  $L \cdot N$  bytes in memory, where  $L$  is the size (in bytes) of data type T
    2. introduces an identifier **A** that can be used as a **pointer to the beginning of the array**
      - value of this pointer will be  $x_A$
      - array elements can be accessed using an integer index ranging 0 to  $N - 1$
      - array element  $i$  will be stored at address  $x_A + L \cdot i$

- For example, consider the following declarations

char     A[12];

char     \*B[8];

int       C[6];

double   \*D[5];

these declarations will generate arrays with the following parameters

1	Array	Elem.size	Total.size	Start.address	Elem.i
2	-----				
3	A	1	12	x_A	x_A+i
4	B	8	64	x_B	x_B+8i
5	C	4	24	x_C	x_C+4i
6	D	8	40	x_D	x_D+8i

- note that B and D are arrays of pointers => array elements are 8 bytes each

- **memory referencing** example:

Suppose **E** is an array of values of type **int** and we wish to evaluate **E[i]**, where the **address of E** is stored in register **%rdx** and **i** is stored in register **%rcx**

```
1  movl  (%rdx,%rcx,4),%eax
```

which will perform  $x_E + 4i$

- allowed scaling factors of **1, 2, 4, 8** cover the sizes of the common primitive data types

## H5 ¶3.8.2 Pointer Arithmetic

- C allows **arithmetic on pointers** -- computed value is scaled **according to the size of the data type** referenced by the pointer

if  $p$  is a pointer to data of type  $T$ , and the value of  $p$  is  $x_p$ ,

then the expression  $p + i$  has value  $x_p + L \cdot i$ , where  $L$  is the size of data type  $T$

- **'&'** -- **generates** pointers, **'\*'** -- **dereferences** pointers

Suppose  $Expr$  denotes some object.

=>  $\&Expr$  - pointer giving the **address of the object**

$AExpr$  denotes an address,

=>  $*AExpr$  - **value at the address**

Hence,

$$Expr = *\&Expr$$



- **array subscripting operation** can be applied to **both arrays and pointers**

$$A[i] = *(A + i)$$

Suppose

**%rdx** -- **starting address** of integer array E

**%rcx** -- **integer index i**

1	Exp	Type	Value	Assembly code
2	-----			
3	E	int*	x_E	movl %rdx,%rax
4	E[0]	int	M[x_E]	movl (%rdx),%eax
5	E[i]	int	M[x_E+4i]	movl (%rdx,%rcx,4),%eax
6	&E[2]	int*	x_E+8	leaq 8(%rdx),%eax
7	E+i-1	int*	x_E+4i-4	leaq -4(%rdx,%rcx,4),%rax
8	*(E+i-3)	int	M[x_E+4i-12]	movl -12(%rdx,%rcx,4),%eax
9	&E[i]-E	long	i	movq %rcx,%rax

- Note that **data** are stored in **%eax** and **pointers** are stored in **%rax**
- Return data types, operations, and registers **match**
  - pointers - leaq, movq - %rax
  - data - movl - %eax

## H5 ¶3.8.3 Nested Arrays

- general principles of array allocation and referencing hold

For instance, the declaration

```
1 int A[5][3]
```

is equivalent to

```
1 typedef int row3_t[3];
2 row3_t A[5];
```

```

1 - row3_t : array of three integers
2 - A: five such elements --> each requiring **12 bytes** to store the
   three integers

```

- Hence, A has a size of  $4 \cdot 5 \cdot 3 = 60$  bytes (array of 15 integers)
- array A can be viewed as a **two-dimensional array** with **5 rows & 3 cols**
- array elements are ordered in memory in **row-major order**
  - all elements of row 0 are followed by all elements of row 1 ...
  - **Consequence of nested declaration**
- To **access** elements of **multidimensional arrays**, the compiler generates code to compute the offset of the desired element and then uses one of the **MOV** instructions with the **start of the array** as the base address and the **offset** (scaled) as an index
  - For an array  $T D[R][C]$ ;
 

array element  $D[i][j]$  is at memory address  $\&D[i][j] = x_D + L(C \cdot i + j)$ , where  $L$  is the size of data type  $T$  in bytes
- Accessing  $A[i][j]$  in assembly code

```

1 # A in %rdi, i in %rsi, and j in %rdx
2 leaq    (%rsi,%rsi,2), %rax    # Compute 3i
3 leaq    (%rdi,%rax,4), %rax    # Compute x_A+12i
4 movl    (%rax,%rdx,4), %eax    # Read from M[x_A+12i+4j]

```

## H5 ¶3.8.4 Fixed-Size Arrays

- C compiler can make many **optimizations** for code operating on **multi-dimensional arrays of fixed size**

[Example C code]

```

1  #define N 16
2  typedef int fix_matrix[N][N];
3
4  /* Compute i,k of fixed matrix product */
5  int fix_prod_ele (fix_matrix A, fix_matrix B, long i, long k) {
6      long j;
7      int result = 0;
8
9      for(j = 0; j < N; j++)
10         result += A[i][j] * B[j][k];
11
12     return result;
13 }

```

Whenever a program uses **some constant** as an **array dimension** or **buffer size**, set name with **#define** declaration

--> easy to change the value by modifying the #define declaration

[**Optimized** C code (with -O1 flag)]

```

1  /* Compute i,k of fixed matrix product */
2  int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k) {
3      int *Aptr = &A[i][0];    // points to elements in row i and A
4      int *Bptr = &B[0][k];    // points to elements in column k of B
5      int *Bend = &B[N][k];    // marks stopping point for Bptr
6      int result = 0;
7      do {
8          result += *Aptr * *Bptr; // add next product to sum
9          Aptr++;                // move Aptr to next col
10         Bptr += N;              // move Bptr to next row
11     } while(Bptr != Bend);      // test for stopping pt
12     return result;
13 }

```

- optimization accomplished by...
  1. removing **integer index j**
  2. converting **array references** -> **pointer dereferences**
    1. **Aptr** --> successive elements in row **i** of **A**
      - init value = `&A[i][0]`

2. **Bptr** --> successive elements in col **k** of **B**
  - init value =  $\&B[0][k]$
3. **Bend** --> value of **Bptr** when terminating the loop
  - $\&B[N][k]$  ( $\because$  terminates at  $(n + 1)$  st elem of col  $j$  of  $B$ )

[Generated **Assembly code**]

```

1  # int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k)
2  # A in %rdi, B in %rsi, i in %rdx, k in %rcx
3  fix_prod_ele:
4      salq    $6, %rdx                # compute 64*i
5      addq    %rdx, %rdi              # compute Aptr = x_A+64i = &A[i][0]
6      leaq    (%rsi,%rcx,4), %rcx     # compute Bptr = x_B+4k = &B[0][k]
7      leaq    1024(%rcx), %rsi        # compute Bend = x_B+4k+1024 = &B[N]
      [k]
8      movl    $0, %eax                # set result = 0
9      .L7:                             # loop:
10     movl    (%rdi), %edx             # read *Aptr
11     imull    (%rcx), %edx            # multiply by *Bptr
12     addl    %edx, %eax               # add to result
13     addq    $4, %rdi                 # increment Aptr++
14     addq    $64, %rcx                # increment Bptr += N
15     cmpq    %rsi, %rcx              # Compare Bptr:Bend
16     jne     .L7                     # If !=, goto loop
17     rep; ret                         # Return

```

## H5 ¶3.8.5 Variable-Size Arrays

- Historically, **C** only supported **multidimensional arrays** where the sizes could be determined at compile time
  - had to allocate storage with **malloc** or **calloc**
  - needed to explicitly encode the mapping of multidimensional arrays into single-dimension via **row-major indexing**
- **ISO C99** introduced the capability of having **array dimension expressions** that are computed as the array is being allocated
- can declare an array
 

$int A[expr1][expr2]$ 
  - either as a **local variable** or as an **argument to a function**

- dimensions are determined by **evaluating the expressions** at the time the declaration is encountered

[Example C code]

```
1 int var_ele(long n, int A[n][n], long i, long j) {
2     return A[i][j];
3 }
```

[Generated Assembly code]

```
1 # int var_ele(long n, int A[n][n], long i, long j)
2 # n in %rdi, A in %rsi, i in %rdx, j in %rcx
3 var_ele:
4     imulq    %rdx, %rdi          # compute n x i
5     leaq     (%rsi,%rdi,4), %rax  # compute x_A+4(nxi)
6     movl     (%rax,%rcx,4), %eax  # read from M[x_A+4(nxi)+4j]
7     ret
```

- **GCC** computes the address of element  $i, j$  as  $x_A + 4(n \cdot i) + 4j = x_A + 4(n \cdot i + j)$
- Address computation is **similar to that of the fixed-size array** **except**
  1. register usage changes (': parameter  $n$  added)
  2. multiply instruction is used (L4) to compute  $n \cdot i$  rather than an *leaq* instruction
    - Can incur a **significant performance penalty** but it is **unavoidable** in **variable-sized array**
- when variable-sized arrays are referenced **within a loop**, compiler often **optimizes** the **index computations** by exploiting the **regularity of the access patterns**

[Example C code]

```

1 // Compute i, k of variable matrix product
2 int var_prod_ele(long n, int A[n][n], int B[n][n], long i, long k) {
3     long j;
4     int result = 0;
5
6     for(j = 0; j < n; j++)
7         result += A[i][j] * B[j][k];
8
9     return result;
10 }

```

[Optimized C code]

```

1 // Compute i, k of variable matrix product
2 int var_prod_ele_opt(long n, int A[n][n], int B[n][n], long i, long
3 k) {
4     int *Arow = A[i];
5     int *Bptr = &B[0][k];
6     int result = 0;
7     long j;
8     for(j = 0; j < n; j++) {
9         result += Arow[j] * *Bptr;
10        Bptr += n;
11    }
12    return result;
13 }

```

- uses (**array-based code**) a different style from the optimized code for the **fixed-size array**
- **retains** loop variable **j** to
  1. detect when the loop has terminated
  2. index into an array consisting of the elements of row **i** of **A**

[Generated **Assembly code -O1 option**]

```

1  # Registers:  n in %rdi, Arow in %rsi, Bptr in %rcx
2  #              4n in %r9, result in %eax, j in %edx
3  .L24:                                # loop:
4      movl  (%rsi,%rdx,4), %r8d         # Read Arow[j]
5      imull (%rcx), %r8d               # Multiply by *Bptr
6      addl  %r8d, %eax                 # Add to result
7      addq  $1, %rdx                   # j++
8      addq  %r9, %rcx                  # Bptr += 4n
9      cmpq  %rdi, %rdx                 # Compare j:n
10     jne   .L24                       # If !=, goto loop

```

- program uses both a scaled value **4n (%r9)** for **incrementing Bptr (L8)** and **n** to **check the loop bounds**
  - with **Optimizations enabled**, **GCC** recognizes **patterns** that arise when a program steps through **multidimensional arrays**
    - then generates either the **pointer-based** or the **array-based** code that **significantly improve** program performance

### H3 §3.9 Heterogeneous Data Structures

- C provides **two mechanisms** for creating data types by **combining objects of different types**
  1. **structures**
    - declared using **struct**
    - aggregate **multiple objects** into a **single unit**
  2. **unions**
    - declared using **union**
    - allow an object to be **referenced using several different types**

#### H5 ¶3.9.1 Structures

- C **struct** creates a data type that **groups objects of possibly different types into a single object**
- All of the components of a structure are stored in a **contiguous region of memory**, and a **pointer to a structure** is the **address of its first byte**
- compiler maintains **byte offset** of each field, and generates **references to structure elements** using these **offset** as **displacements in memory referencing instructions**

For instance, consider

```

1 struct rec {
2     int i;
3     int j;
4     int a[2];
5     int *p;
6 };

```

will have a total of 24 bytes of data

1	offset	0	4	8		16	24
2	-----						
3	Contents	i	j	a[0]	a[1]		p
4							

Suppose  $r$  of type `struct rec *` is in `%rdi`

- to access the fields of a structure, the compiler generates code that **adds the appropriate offset** to the **address of the structure**

```

1 # Registers: r in %rdi
2 movl    (%rdi), %eax      # Get r->i
3 movl    $eax, 4(%rdi)     # Store in r->j

```

--> copies element  $r \rightarrow i$  to element  $r \rightarrow j$

(See how the code **adds offset 4** to the **address of x** (L3))

- to generate a **pointer** to an object within a structure, can simply **add** the **field's offset** to the **structure address**

```

1 # Registers: r in %rdi, i in %rsi
2 leaq    8(%rdi,%rsi,4), %rax # Set %rax to &r->a[i]

```

(generates the pointer value  $\&(r \rightarrow a[1])$  by adding offset  $8 + 4 \cdot 1$ )

- for complex statement such as  $r \rightarrow p = \&r \rightarrow a[r \rightarrow i + r \rightarrow j];$  ,



```

1  # Registers: r in %rdi
2  movl    4(%rdi), %eax          # Get r->j
3  addl    (%rdi), %eax          # Add r->i
4  cltq                               # Extend to 8 bytes
5  leaq    8(%rdi,%rax,4), %rax   # Compute &r->a[r->i + r->j]
6  movq    %rax, 16(%rdi)        # Store in r->p

```

## H5 ¶3.9.2 Unions

- **Unions** allows a **single object** to be **referenced according to multiple types**
- **identical syntax** (to that for structures), **different semantics**
  - **all reference the same block**

Consider  $\exists S3, U3$ , s.t.

```

1  struct S3 {
2      char c;
3      int i[2];
4      double v;
5  };
6
7  union U3 {
8      char c;
9      int i[2];
10     double v;
11 };

```

when compiled on an x86-64 Linux machine, the **offsets** & **total size** are...

	Type	c	i	v	Size
1					
2					
3	S3	0	4	16	24
4	U3	0	0	0	8

- **pointer p** of type union U3 \* (p->c, p->i[0], p->i[1], p->v) would **all reference the**

## beginning of the data structure

- **overall size of a union** = **maximum** size of any of its fields

- Unions can be useful in multiple contexts

(can also lead to bugs (∴ bypass the safety))

1. know in advance that the **use of two different fields** in a data structure will be **mutually exclusive**

- declaring two fields as part of a union will **reduce the total space allocated**

Suppose we are implementing a **binary tree data structure**

```
1  struct node_s {
2      struct node_s *left;
3      struct node_s *right;
4      double data[2];
5  };
```

--> with struct, every node requires **32 bytes**, with **16 bytes wasted** for each type of node

```
1  union node_u {
2      struct {
3          union node_u *left;
4          union node_u *right;
5      } internal;
6      double data[2];
7  };
```

--> with union, every node requires **16 bytes**

However, with this encoding, cannot determine whether a given node is a leaf or an internal node

--> use **enumerated type**, defining the different possible choices for the union

```

1  typedef enum { N_LEAF, N_INTERNAL } nodetype_t;
2
3  struct node_t {
4      nodetype_t type;
5      union {
6          struct {
7              struct node_t *left;
8              struct node_t *right;
9          } internal;
10         double data[2];
11     } info;
12 };

```

--> requires **24 bytes** (4 for type + 8 each for left, right or 16 for info.data + 4 bytes of padding between type and elements)

## 2. to access the **bit patterns** of **different data types**

```

1  unsigned long double2bits(double d) {
2      union {
3          double d;
4          unsigned long u;
5      } temp;
6      temp.d = d;
7      return temp.u;
8  };

```

- **stored** the argument in the union **using one data type** and **access it using another**
  - **u** will have the **same bit representation** as **d**
- when using **unions** to **combine data types of different sizes**, byte-ordering issues can become important

```

1  double uu2double(unsigned word0, unsigned word1) {
2      union {
3          double d;
4          unsigned u[2];
5      } temp;
6
7      temp.u[0] = word0;
8      temp.u[1] = word1;
9      return temp.d;
10 }

```

- on **Little-endian machines**, **word0** will become the **low**-order 4 bytes of **d**, while **word1** will become the **high**-order 4 bytes of **d**
- on **Big-endian machines**, vice versa

## H5 ¶3.9.3 Data Alignment

- many computer systems place **restrictions** on the **allowable addresses** for the primitive data types
  - **addresses** for some objects **must be** a **multiple of K** (2, 4, 8)
- **alignment restrictions** - **simplify the design** of the hardware forming the interface between the processor and the memory system
  - For instance, if a processor always fetches 8 bytes from memory with an address that must be a multiple of 8
    - > **guarantees** any **double** will be aligned to have its address be **multiple of 8**
    - > value **can be read / written** with a **single memory operation**
- **Intel** recommends that **data be aligned to improve memory system performance**
  - any primitive object of **K bytes** must have an **address** that is a **multiple of K**

1	K	Types
2	-----	
3	1	char
4	2	short
5	4	int, float
6	8	long, double, char *

- Compiler places **directives** in the **assembly code** indicating the **desired alignment for global data**
  - e.g. `.align 8` from jump table  
--> ensures that the data following will start with an address that is a **multiple of 8**
- Compiler may need to **insert gaps** in the **field allocation** to **ensure** that each structure element **satisfies its alignment requirement**
  - For example,

```

1 struct S1 {
2     int i;
3     char c;
4     int j;
5 };

```

will be aligned as...

```

1 offset | 0      | 4  | 5      | 8      | 12
2 Contents -----
3       |  i    | c  | xxx   |  j    |
4       -----

```

--> compiler **inserts a 3-byte gap** to satisfy **4-byte alignment requirements**

- Compiler may need to **add padding to the end of the structure** so that each element in an **array of structures** will **satisfy its alignment requirement**
  - For example,

```

1  struct S2 {
2      int i;
3      int j;
4      char c;
5  };
6
7  struct S2 d[4];

```

--> rather than packing each structure into **9 bytes** (∴ cannot satisfy the alignment requirement for each element of d),

compiler allocates **12 bytes** for each structure with the **final 3 bytes being wasted space**

1	offset	0	4	8	9	12
2	Contents	-----				
3		i	j	c	xxx	
4		-----				

--> then **elements of d** will have addresses  $x_d$ ,  $x_d + 12$ ,  $x_d + 24$ ,  $x_d + 36$ . --> as long as  $x_d$  is a multiple of 4, all of the **alignment restrictions will be satisfied**

### H3 §3.10 Combining Control and Data in Machine-Level Programs

- how data and control interact with each other
  1. **Pointers**
  2. symbolic debugger **GDB**
  3. **buffer overflow** --> causing security vulnerability
  4. **varying stack storage** required by a function one execution to another

#### H5 ¶3.10.1 Understanding Pointers

- **Pointers** - uniform way to **generate references to elements** within different data structures
- some **key principles** of pointers
  1. Every pointer has an **associated type**
    - indicates what kind of object the pointer points to

```

1  int *ip;           // pointer to int
2  char **cpp;        // pointer to a pointer to char

```

- in general, if the object has type  $T$ , then the pointer has type  $*T$
  - **special case:**  $void *$  --> **generic pointer**
    - e.g.) *malloc* function returns a generic pointer
    - can be converted to a typed pointer by
      1. explicit cast
      2. implicit casting of the assignment operation
  - pointer types are **not part of machine code**
    - abstraction provided by C to avoid addressing errors
2. Every pointer **has a value**
- **address** of some object of the designated type
  - **special case** : **NULL (0)** indicates that the pointer **does not point anywhere**
3. Pointers are **created** with the **'&'** operator
- can be applied to any C expression that is categorized as an **lvalue**
    - expression that can appear on the left side of an assignment
    - variables, elems of structures, unions, arrays
  - in machine code, **leaq** instruction is used
    - to compute the **address of a memory reference**
4. Pointers are **dereferenced** with the **\*** operator
- result: value having the type associated with the pointer
  - dereferencing is implemented by a memory reference
    - storing to or retrieving from the **specified address**
5. Arrays and pointers are closely related
- name of an array can be referenced (but not updated) as if it were a pointer variable
  - array referencing has the **exact same** effect as a **pointer arithmetic** and **dereferencing**

$$a[3] \text{ is same as } *(a+3)$$
  - both array referencing and pointer arithmetic require scaling the offsets by the object size
 
$$p+i \text{ for pointer } p \text{ with value } p \text{ will have the address of } p + L \cdot i, \text{ where } L \text{ is the size of the data type associated with } p$$

## 6. Casting from one type of pointer to another **changes its type** but **not its value**

- **changes any scaling of pointer arithmetic**

If  $p$  is a pointer of type `char *` having value  $p$ ,

`(int *) p+7`  $\Rightarrow p + 28$

`(int *) (p+7)`  $\Rightarrow p + 7$

--> casting has higher precedence than addition

## 7. Pointers can also **point to functions**

- provides a powerful capability for **storing and passing references** to code
  - can be invoked in some other part of the program

```
1  int fun(int x, int *p);
2
3  int (*fp)(int, int *);
4  fp = fun;
5
6  int y = 1;
7  int result = fp(3, &y);
```

- value of a function pointer = **address of the first instruction**

## H5 ¶3.10.2 Life in the Real World: Using the GDB Debugger

- supports the **run-time evaluation & analysis** of **machine-level programs**
- possible to study to behavior by watching the program in action while having considerable control over its execution
- start GDB with

```
1  linux> gdb prog
```

- set breakpoints near points of interest
  - just after the entry of a function
  - at a program address
- when one of the breakpoints is hit during execution, the program will halt and return control to the user
  - examine different registers & memory locations in various formats
  - can single-step the program / few instructions at a time / proceed to the next breakpoint



## H5 ¶3.10.3 Out-of-Bounds Memory References and Buffer Overflow

- C **does not** perform **bounds checking for array references**, and + local variables on the stack + state information
  - > can cause serious program errors
- state stored on the stack gets **corrupted** by a **write** to an **out-of-bounds array element**
- particularly common source of state corruption is known as **Buffer overflow**
  - e.g. some character array is allocated on the stack to hold a string, but the **size of the string exceeds the space allocated for the array**

```
1 // Implementations of library function gets()
2 char *gets(char *s) {
3     int c;
4     char *dest = s;
5
6     while ((c = getchar()) != '\n' && c != EOF)
7         *dest++ = c;
8     if (c == EOF & dest == s)
9         // No characters read
10        return NULL;
11    *dest++ = '\0'; // Terminate string
12    return s;
13 }
14
15 // Read input line and write it back
16 void echo() {
17     char buf[8]; // way too small!
18     gets(buf);
19     puts(buf);
20 }
```

- preceding code: implementation of the library function **gets**
  - **reads** a line from the **standard input**, **stopping** when either a **terminating newline character** or some **error condition** is encountered
  - **copies** the string **to the location designated** by argument **s** and **terminates** the string with a **null** character
  - **No way to determine whether sufficient space has been allocated to hold the entire string**

- with code above, any string longer than seven characters will cause an **out-of-bound** write

[Generated **Assembly code**]

```

1  # void echo()
2  echo:
3      subq    $24, %rsp    # Allocate 24 bytes on stack
4      movq    %rsp, %rdi   # Compute buf as %rsp
5      call    gets        # Call gets
6      movq    %rsp, %rdi   # Compute buf as %rsp
7      call    puts        # Call puts
8      addq    $24, %rsp    # Deallocate stack space
9      ret

```

- the program allocates **24 bytes** on the stack
- character **buf** is positioned at the **top of the stack**
  - %rsp copied to %rdi to be used as the argument to the calls to both **gets** and **put**
- 16 bytes between **buf** and the stored **return pointer** are not used
  - No serious consequence occurs for strings of up to **23** characters, **but beyond that**,
    - **the value of the return pointer + possibly additional saved state will be corrupted**
  - > **ret** (L9) will cause the program to jump to a totally unexpected location
- Commonly used library functions (**strcpy**, **strcat**, **sprintf**) can generate a byte sequence without being given any indication of the size of the destination buffer --> **buffer overflow!**
- **Buffer overflow** is used to get a program to perform a function that it would otherwise be unwilling to do
  - **exploit code** : byte encoding of some executable code + **extra bytes to overwrite the return address** (to make **ret** jump to the **exploit code** )

H5 ¶3.10.4 Thwarting Buffer Overflow Attacks

H6 Stack Randomization

- in order to insert exploit code into a system, the attacker needs to **inject exploit code + pointer to this code** as part of the attack string
  - needs to **know the stack address** where the string will be located
- Historically, the stack addresses for a program were highly predictable --> easy for attackers to inject worms
- **Stack randomization** --> makes the position of the stack vary from one run of a program to another
  - even if many machines are running identical code, they would **all be using different stack addresses**
  - implemented by **allocating a random amount of space between 0 and n bytes** on the stack
    - allocation range **n** needs to be **large enough** to get **sufficient variations** in the stack address, yet **small enough** that it **does not waste too much space** in the program
- **Address-space layout randomization (ASLR)**
  - **different parts of the program** (program code, library code, stack, global variables, heap data) are loaded into **different regions** of memory **each time a program is run**
    - program running on one machine will have different address mappings than the same program running on other machines

## H6 Stack Corruption Detection

- to detect when a stack has been corrupted
- **stack protector**
  - stores **canary value (guard value)** between any local buffer and rest of the stack
    - generated **randomly** each time the program runs --> hard to determine what it is
  - before restoring the register state and returning from the function, the program checks **if the canary has been altered**
    - if so, the program **aborts with an error**
- Recent versions of **GCC** inserts **stack protector** automatically
  - to prevent GCC from inserting it, use option `-fno-stack-protector`

[Generated **Assembly code with stack protector enabled**]

```
1  # void echo()
2  echo:
3      subq $24, %rsp      # Allocate 24B on stack
4      movq %fs:40, %rax    # Retrieve canary
5      movq %rax, 8(%rsp)   # Store on stack
6      xorl %eax, %eax      # Zero out register
7      movq %rsp, %rdi      # Compute buf as %rsp
8      call gets            # Call gets
9      movq %rsp, %rdi      # Compute buf as %rsp
10     call puts            # Call puts
11     movq 8(%rsp), %rax    # Retrieve canary
12     xorq %fs:40, %rax    # Compare to stored val
13     je    .L9            # If =, goto ok
14     call  __stack_chk_fail # Stack corrupted!!!!
15 .L9                    # ok:
16     addq $24, %rsp        # Deallocate stack space
17     ret
```

- retrieves a value from memory (L4) and **stores it on the stack at offset 8 from %rsp** (just beyond the region allocated for buf (L5))
  - **%fs:40** - canary value is read from memory using **segmented addressing**
    - marked as "read only"
- before returning, the function **compares the value stored at the stack location with the canary value** (xorq instruction on L12)
  - If the two are identical (0) --> function completes normally
  - non-zero value --> calls an error routine

## H6 Limiting Executable Code Regions

- limit which memory regions hold executable code
  - only the portion of memory holding the code generated by the compiler need be executable
  - other portions **restricted** to **read/write only**
- Historically, x86 architecture **merged the read and execute access controls into a single 1-bit flag**
  - any page marked as readable = executable
  - various schemes were implemented to be able to limit some pages to being readable but not executable, but led to **significant inefficiencies**

- More recently, **AMD** introduced an **NX** ("no-execute") bit
  - separating the read and execute access modes
  - stack can be marked as being readable and writable, **but not executable**
  - **performed in hardware, with no penalty in efficiency**

## H5 ¶3.10.5 Supporting Variable-Size Stack Frames

- some functions require a **variable amount of local storage**

[Example C code]

```

1  long vframe(long n, long idx, long *q) {
2      long i;
3      long *p[n];
4      p[0] = &i;
5      for(i = 1; i < n; i++)
6          p[i] = q;
7      return *p[idx];
8  }
```

- declares local array  $p$  of  $n$  pointers --> requires  $8n$  bytes on the stack
  - $n$  may vary from one call of the function to another
    - compiler cannot determine how much space it must allocate for the function's stack frame

[Portions of generated **assembly code**]

```

1  # long vframe(long n, long idx, long *q)
2  # n in %rdi, idx in %rsi, q in %rdx
3  # Only portions of code shown
4  vframe:
5      pushq %rbp                # Save old %rbp
6      movq %rsp, %rbp           # Set frame pointer
7      subq $16, %rsp            # Allocate space for i (%rsp = s_1)
8      leaq 22(,%rdi,8), %rax
9      andq $-16, %rax
10     subq %rax, %rsp            # Allocate space for array p (%rsp =
    s_2)
11     leaq 7(%rsp), %rax
```

```

12     shrq    $3, %rax
13     leaq    0(,%rax,8), %r8        # Set %r8 to &p[0]
14     movq    %r8, %rcx              # Set %rcx to &p[0] (%rcx = p)
15     #...
16     # Code for initialization loop
17     # i in %rax and on stack, n in %rdi, p in %rcx, q in %rdx
18     .L3:                          # loop:
19     movq    %rdx, (%rcx,%rax,8)    # Set p[i] to q
20     addq    $1, %rax               # Increment i
21     movq    %rax, -8(%rbp)         # Store on stack
22     .L2:
23     movq    -8(%rbp), %rax         # Retrieve i from stack
24     cmpq    %rdi, %rax             # Compare i:n
25     jl      .L3                   # If <, goto loop
26     #...
27     # Code for function exit
28     leave   %rsp                  # Restore %rbp and %rsp
29     ret                                # Return

```

- to manage a variable-size stack frame, x86-64 uses register **%rbp** to serve as a **frame pointer** (base pointer)
  - **Must save** the previous version of **%rbp** on the stack (∴ **callee-saved register**)
    - keeps **%rbp** pointing to **fixed-length local variables (i)**, at offsets relative to **%rbp**
  - was used with every function call in earlier versions of x86 code
  - with x86-64, used only where **stack frame** may be of **\*\*variable size**