Tour 1)

- Abstractions
- Computer Systems: machine that automatically manipulates and transforms data
- Analogue/Digital Computer
  - Analogue: maps input to output by means of a model
  - Digital: represent data and transformation steps(instructions) in an encoded form
- Von Neumann Architecture
  - Central Processing Unit (Processor)
    - Control Unit
    - Arithmetic/Logic Unit
  - Memory Unit (Memory or Data storage)
- Bit: binary digit
- Byte: 8 bits
- Logic circuits (transistors)
- Electromechanical Relay
  - AND: serial
  - OR: parallel
  - XOR: open contact of one relay is connected to the closed contact of the other

Tour 2)

- Translation & Machine Instructions
  - Compiler: program that translates the whole target program in advance
  - Interpreter: program that translates piecemeal, while the target program is running
- Process of going from the source code to computation (program termination)
- Processor
- Hardware organization
  - CPU, Register, ALU, System bus, etc.
- Launching & Runing the "Hello" Program
  - "The processor is always busy"
- Memory Hierarchy
  - Fast access but High cost vs. Slow access but Low cost
- Caching
  - Data Locality
- Operating System
  - System calls
  - Hardware and Software
  - Abstractions
    - Process
    - Thread
    - Virtual Memory

        - Files
    - Networks
    - SUMMARY

C Basics)

- Int getchar()
- Int putchar(char input)
- EOF: end of file
- Arrays
- Sizeof()
- Strlen
- Dereference (→)

```
root@ubuntu:~# gcc s.c
root@ubuntu:~# ./a.out
Size of Character : 1
Size of short Integer : 2
Size of Integer : 4
Size of Long Integer : 8
Size of Float : 4
Size of Double : 8
Size of Long Double : 16
Size of Integer Pointer : 8
Size of Character Pointer : 8
Size of Float Pointer : 8
root@ubuntu:~#
```

C String: all strings end in 0, or a 'null' (char type value 0, ASCII value 0)
How to write "q" in printf statement? /"q/"? something like that

The Stack)

- Process of executing a function starts with the call instruction
    - Call f1 (f1 is a label. The linker will replace it with an address expression)
- CALL instruction
    - 1. Pushes the return address onto the stack
        - Push: Decrement SP by the machine word size(word size is 64 for 64-bit machine)
        - Copies the contents of the operand register onto memory at the address contained in SP
    - 2. Copies the operand value into IP
        - This causes the CPU to execute the first instruction of the callee function as its next instruction
    - Gcc -S?
    - Objdump -d (stdout)
- Questions on the suffixes!!!

```
Int f1(int x) {
    Return x * 3;
}

Main () {

}
```

```
_f1:
pushq %rbp
movq %rsp, %rbp ('move' means copy; this is done in order to create new stack frame)
movl %esi, -4(%rbp) (rbp refers to the stack? Also study 'q' and 'l' and other suffixes that come after mov)
movl -4(%rbp), %eax
imull $3, %eax (dollar sign for constant)
pop %rbp (copy address of rsp to rbp and increment rsp by 8 bits)
retq
```

```
(old BP)
(main)
(return address)
(old BP)
```

```
Int f2 (int x) {
    Int nums[4];
    Nums[0] = x; (-16(%rbp))
    Nums[1] = x + 1; (-12(%rbp))
    Nums[2] = x + 2; (-8(%rbp))
}
```

SP will decrement by 16bits, in a int nums[4] kind of array. Size of array * size of data type

Practice reconstructing float point notations into decimals!!
Ex) 1 10000011 1110000...

Conversion between binary, hexadecimal, etc.

Assembly code (The last human-readable state of a program)

- Compilation: translates the code text into "Assembly code" (sequence of instructions)
- Assembly Instruction Format = "Label: Mnemonic arg1, arg2, arg3"

- Only the mnemonic is required, so instructions may have between 0 and 3 arguments
- Mnemonic is the human-memorable symbol for a numeric operation code or opcode
- Arguments are typically operands for the operation named by the mnemonic
- Label is a human-readable symbol representing the address in memory of the instruction
  - The compiler uses it for jump instructions elsewhere, to jump to this instruction
  - The linker replaces the symbol with a numeric address as the argument to the jump instruction

- Assembly code example: addl -4(%rbp) , %eax
  - No label → Mnemonic(addl) → arg1(-4(%rbp)) → arg2 (%eax)
  - Meaning: Add the 32-bit integer value stored on the stack at 4 bytes below the address stored in rbp (the base pointer register) to the value stored in register eax (i.e., the 32 lower-order bits of rax), and store the result in eax
    - addl = add + l = integer addition + word size suffix(long – 32bits)
    - -4(%rbp), %eax = source & destination operands
      - CPU makes the ALU add the value stored in the source operand to the value stored in the destination operand and stores the result(sum) in the destination operand
  - %rbp = 64-bit (r prefix) base pointer register
    - (%rbp) = the value stored at the address stored in %rbp. (the parenthesis means dereferencing like *)
    - -4(%rbp) = the value stored at the address that is 4 bytes less than the address stored in %rbp
  - %eax (% is the indication of a register in GCC Assembly, one of the two major dialects of Assembly code for Intel machines)
    - 'e' is the prefix meaning the lower-order 32 bits of the register
    - "ax" is the name of the register since the 8086

Registers)

- Names: AX, BX, CX, DX, SI, DI, BP, SP
  - X = H or L (high/low-order bytes of 16-bit memory addresses)
  - A = accumulator; holds the return value of a function
  - B = "base" register in memory accesses needing a base and an offset
  - C = "counter register," for shift operations and loops
  - D = "data" register for arithmetic and I/O operations
  - SI/DI (source index/destination index) = specialized for stream operations such as copying a string to a buffer or a file to another file, or operating on the elements of an array and storing the result in another array
  - BP/SP = used for stack management, organizing programs into functions using stack frames

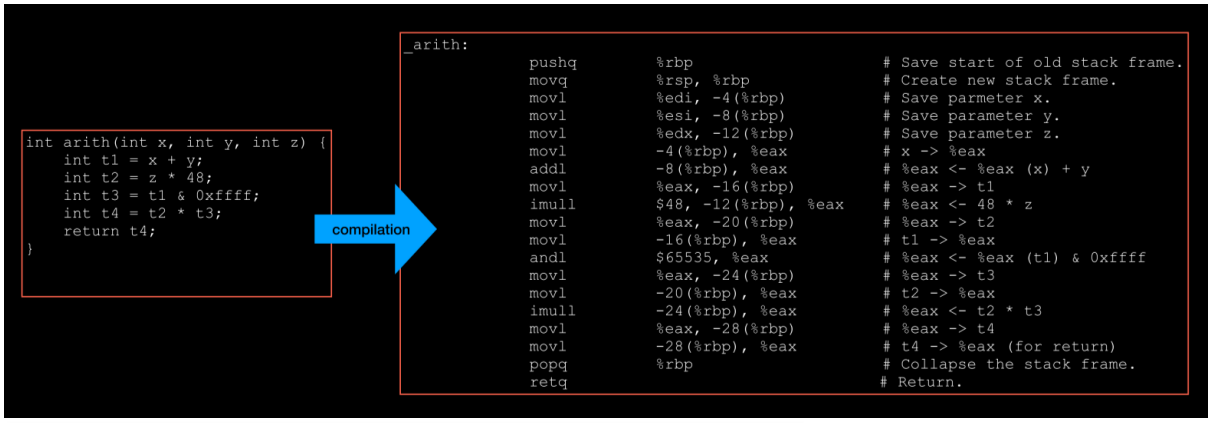
- IP = Instruction pointer

Operand Syntax)

- An operand may be:
  - Constant (immediate) = prefixed by $
    - addl $3, %eax = add a 32-bit 3 to the value in eax and store it in eax
  - Register = prefixed by %
  - Memory location
    - Absolute number (hexadecimal)
    - Address stored in a register or relative to it
    - Syntax = disp(base, index, scale)
- Source operand may be:
  - Immediate
  - Register
  - Memory
- Destination operand may be:
  - Register
  - Memory
- Binary operations take 2 operands: a source and a destination
  - Add, sub, mul (unsinged multiplication), imul (signed multiplication)
- Unary operations take a single operand
  - Inc (increment), dec (decrement), neg (negation), not (complement)
- Shift operations take 2 operands, but the "shift amount" is special
  - It must be an immediate (a constant) or a one-byte value stored in the %cl register

Load Effective Address (lea))

- The lea instruction class computes the address of a memory source operand and stores the address in the destination operand
- The most general form of a memory reference is disp (base, index, scale)
  - disp (the displacement or offset) is an immediate (without the $)
  - base and index are registers
  - scale is an integer either 1, 2, 4, or 8
  - Address expressed by this form is computed as (disp + (base + (index * scale)))
- Lea is used for simple arithmetic, such as addition or multiplication
  - leal %eax, (%edx, %eax) = add the value in %edx to the value in %eax and return it (which is stored in %eax)
  - leal %eax, (%eax, %eax) OR leal &eax, (, %eax, 2) = double the value in %eax



Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Figure 3.7 **Integer arithmetic operations.** The load effective address (`leal`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Mov instruction class – source/destination operands)

- Add = integer addition
  - addb = add two 8-bit values (bytes)
  - addw = add two 16-bit values (words)
  - addl = add two 32-bit values (long words/double words)
  - addq = add two 64-bit values (quadwords)
- mov = copies data from source to destination (most commonly used instruction)

- Push = combines two operations
  - Decrements the value in sp by the system's word size
  - Copies the contents of the operand(register) to memory at the address currently held by sp
- Pop = reverse of Push
  - Copies the value at the address currently held in sp to the operand register (value = size of system's word size)
  - Increments the value in sp by the system's word size
- Call = calls a function
  - Copies the value currently held in the IP register to the location whose address is currently in sp
  - Value held in IP is always the address of the next instruction to be executed
  - Inserts into IP the address given in the operand, which is generally the start address of the first instruction of a function
  - Address copied onto the stack in the first step is called the "return address"
- Ret = returns from a function
  - Copies the value (the return address) at the address currently held in sp into IP
    - This causes the next instruction executed to be the next instruction after call in the caller function

Instruction	Effect
movb	Copy a byte.
movw	Copy a word (2 bytes).
movl	Copy a double word (4 bytes).
movq	Copy a quad word (8 bytes).

Instruction	Effect
movsbw	Copy byte to word, extend sign.
movsbl	Copy byte to double word, extend sign.
movsbq	Copy byte to quad word, extend sign.
movswl	Copy word to double word, extend sign.
movswq	Copy word to quad word, extend sign.
movslq	Copy word to quad word, extend sign.

Instruction	Effect
movzbw	Copy byte to word, pad with 0.
movzbl	Copy byte to double word, pad with 0.
movzbq	Copy byte to quad word, pad with 0.
movzwl	Copy word to double word, pad with 0.
movzwq	Copy word to quad word, pad with 0.
movzlq	Copy word to quad word, pad with 0.

C Declaration	Intel Data Type	Assembly Code Suffix	size (Bytes)
char	byte	b	1
short	word	w	2
int	double word	l	4
long (32-bit systems)	double word	l	4
long long (32-bit systems)	quad word	q	8
long (64-bit systems)	quad word	q	8
long long (64-bit systems)	quad word	q	8
float	single precision	s	4
double	double precision	l	8
long double	extended precision	t	10/12

- movb %ah, %dh —
  Copies the byte without touching any other data in **%rdx**.
  The copied byte is in the same position in **%rdx** (the second-lowest-order byte) as it had been in **%rax**.
- movsbq %ah, %rdx —
  Completely replaces the contents of **%rdx** — the lowest-order byte is now the same as the second-lowest-order byte of **%rax**, and the rest of the bits are either all 0 or all 1, depending on the highest-order bit of **%ah**.
- movzbq %ah, %rdx —
  The same as with **movzbq**, except that all bits to the left of the copied byte have value 0.

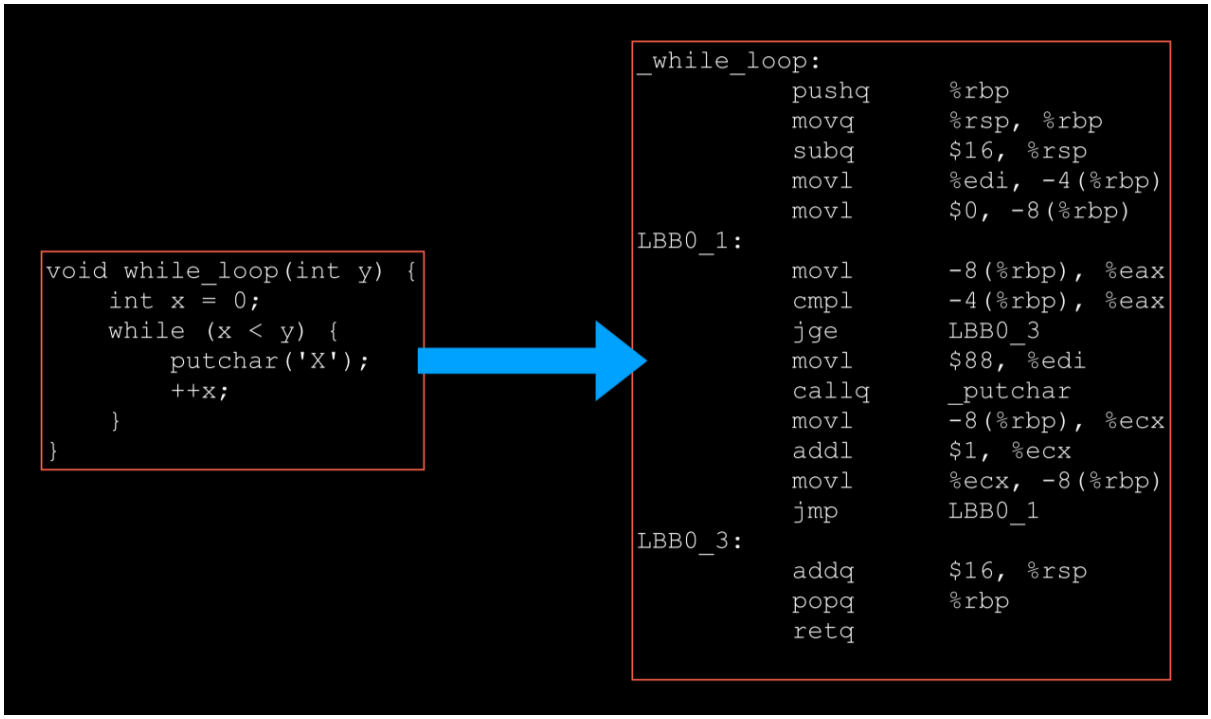
Instruction	Effect
pushq	Decrement %rsp value by system word size. Copy value of operand register to address in %rsp.
popq	Copy value at address in %rsp to operand. Increment %rsp value by system word size.

Instruction	Effect
callq	PUSH — Decrement %rsp by 8 and copy the value in %rip to the memory location whose address is the new value in %rsp. (This creates the *return address*.) Copy the operand value to %rip. (This causes control to jump to the function.)
retq	POP — Copy the value at the address stored in %rsp to %rip and increment %rsp by 8. (This copies the *return address* to the instruction pointer, and causes control to jump back to the instruction in the caller right after the call instruction.)

Control Flow)

- CPU executes instructions in sequence
- The IP always holds the address of the next instruction
- Basic idea is to jump from one place in the program to another under a specified condition
- Conditional jump
  - If, while, for loops
  - Conditional jump instructions checks flags and inserts the new address into IP only if the flags are set as required
    - "Jump on equal" checks the ZF since x = y ⇔ x − y = 0
    - "Jump on less" checks SF and OF, which must be unequal
    - "Jump on greater or equal" checks SF and OF, which must be equal

1111 1110 1110 1101 1111 1010 1100 1110 1101 1110 1010 110 1 0001 001 0 0011 0100



- Condition code
  - When CPU performs an arithmetic or logical operation other the add/mul, it sets bits called condition codes in a register called EFLAGS/RFLAGS
    - CF (carry flag) = set to 1 if the most significant bit (MSB) in the result had a carry out (detects overflow in unsigned arithmetic)
    - ZF (zero flag) = set if the result is 0
    - OF (overflow flag) = set if overflow occurred in signed arithmetic
    - SF (sign flag) = equal to the highest-order bit of the result
- Comparisons
  - "cmp" instruction family performs a subtraction
    - cmp %rdx, %rax = %rdx - %rax
      - Comparing computes x - y = x + (-y)
    - The result is thrown away
    - The purpose is to set the flags

Stack)

- In a 64-bit machine, each register holds one 64-bit datum (x bit machine's register each holds x-bit datum
- Registers are used to hold data during operations
- The stack is a portion of memory used to store values during the execution of functions in the program
- When a program launches, the operating system sets aside a region of memory—a set of contiguous empty memory locations—for the program's use
- The program uses this memory as temporary scratch space for local variables, including

parameters

- Each function gets its own stack space, called a stack frame or activation record
  - This isolation keeps its local variables invisible to the rest of the program
  - When a function returns, its stack frame gets recycled for the next function
- Each new stack frame comes below the previous one (at a lower range of addresses)
  - The stack grows downward
  - When a function returns, the stack shrinks upward (the end of the stack goes to a higher memory address)
- The program uses 2 special registers to set up the stack frame:
  - %bp (base pointer) register = holds the start address (base address) of the stack frame
    - % is the convention used by the GCC assembler to designate a register
  - %sp (stack pointer) register = holds the end address of the stack frame
- Setting up a Stack Frame)
  - 1. Push the current %bp value onto the stack (push %bp)
    - Decrement the value in %sp, then copy the value of %bp to the address stored in %sp
  - 2. Copy %sp to %bp to form the new start address (mov %sp, %bp)
  - 3. Decrease the value of %sp (subtract) to make room for all needed variables
    - ex) sub $16, %sp
    - This moves the end of the stack downward, and all local variables now have addresses relative to %bp (such as %bp – 4)
- Collapsing a Stack Frame)
  - 1. Add back the value earlier subtracted from %sp to create the stack frame
    - ex) add $16, %sp
    - Now the address in %sp is where the value of %bp was last pushed
  - 2. Pop into %bp (pop %bp)
  - 3. Copy the value at the address stored in %sp to %bp, then increment the value in %sp
    - %sp now points to just above where %bp had been pushed, and thus to the end of the previous stack frame

C declaration	32-bit	64-bit
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char *`	4	8
`float`	4	4
`double`	8	8

Figure 2.3 **Sizes (in bytes) of C numeric data types.** The number of bytes allocated varies with machine and compiler. This chart shows the values typical of 32-bit and 64-bit machines.

C data type	Minimum	Maximum
`char`	−127	127
`unsigned char`	0	255
`short [int]`	−32,767	32,767
`unsigned short [int]`	0	65,535
`int`	−32,767	32,767
`unsigned [int]`	0	65,535
`long [int]`	−2,147,483,647	2,147,483,647
`unsigned long [int]`	0	4,294,967,295
`long long [int]`	−9,223,372,036,854,775,807	9,223,372,036,854,775,807
`unsigned long long [int]`	0	18,446,744,073,709,551,615

- An attempt to store a value greater than what the type can accommodate is called *overflow*.
- Overflow behavior differs between signed and unsigned integral types.
- Unsigned: value cycles back through 0.
```
unsigned char u1, u2;
u2 = u1 = 0xff; // 11111111 == 255 — the maximum possible value
++u1;  // u1 == 0x0 == 0 == 00000000
u2 += 2; // u2 == 0x1 == 1 == 00000001
```
- Signed: value “flips” to maximum-magnitude negative.
```
char s1, s2;
s2 = s1 = 0x7f;  // 01111111 == 127 — the maximum for this type
++s1;  // s1 == 0x10000000 == -128
s2 += 2;  // s2 == 0x10000001 == -127
```

C Type	Size	Sign	Exponent	Bias	Mantissa
Float	32	1	8	127	23
Double	64	1	11	1023	52

- The stack
  - For the temporary storage of local variables belonging to a currently-running function.
  - Automatically allocated and de-allocated by the compiler.
  - Relatively small (typically limiited to about 8 MB or less).
- The heap
  - Persistent between function calls.
  - Limited in size only by total available physical memory.
  - Requires the programmer's explicit allocation and deallocation.
- Static data
  - Persistent throughout program execution.
  - Allocated by the compiler according to declarations and never de-allocated.
  - Limited in size only by total available physical memory.
- String literals
  - Persistent throughout program execution, but *read-only*.
  - Otherwise handled like static data

- **void * malloc(size_t size)**
  - Allocates **size** bytes and returns their base address.
  - **size_t** is an alias for **unsigned int** or **unsigned long** and is the type returned by **sizeof**. (On modern 64-bit machines, it is an **unsigned long**.)
  - Cast the **void *** to the appropriate type to ensure compiler type checking.
  - Example: allocating an array of 100 32-bit integers:
    `int * nums = (int *)malloc(100 * sizeof(int));`
  - On success, returns the start address of the allocated chunk; on failure, returns NULL.
- **void * calloc(size_t num, size_t size)**
  - Allocates an *array* of **num** elements, each of size **size**.
  - Example (as above):
    `int * nums = (int *)calloc(100, sizeof(int));`
- **void * realloc(void * ptr, size_t size)**
  - Increases or decreases the size of the memory buffer allocated as **ptr** (the variable).
  - If it is called to enlarge the buffer, if it can, it simply allocates more memory at the end of the buffer without other changes.
  - If contiguous expansion memory is not available, it finds a new chunk and copies the data.
  - Either way, the contents beyond the old data must be assumed to be junk until those locations are initialized.
  - If it shrinks the buffer, the data are truncated.

Storage Class	Memory Segment
Automatic	Stack
Dynamic	Heap
Static	Data, BSS

Integral Types

Signed

Sizes of representation are s

- **short** — typically 16 bits (2 bytes) — $[-2^{15}, 2^{15} - 1]$
  -32 thousand to 32 thousand - 1.
- **int** — typically 32 bits (4 bytes) — $[-2^{31}, 2^{31} - 1]$
  -2 billion to 2 billion - 1.
- **long** — typically 64 bits (8 bytes) — $[-2^{63}, 2^{63} - 1]$
  -8 quintillion to 8 quintillion - 1.

Integral Types

Unsigned

- **unsigned short** — typically 16 bits (2 bytes) — $[0, 2^{16} - 1]$
  0 to 64 thousand - 1.
- **unsigned int** — typically 32 bits (4 bytes) — $[0, 2^{32} - 1]$
  0 to 4 billion - 1.
- **unsigned long** — typically 64 bits (8 bytes) — $[0, 2^{64} - 1]$
  0 to 16 quintillion - 1.

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Aggregate Types

- Enumeration types
  - User-defined types for creating a custom encoding or set of values as a mapping from symbols to integers.
- Arrays
  - Collections of objects of the same type.
- Structures
  - Collections of objects of various types.
- Unions
  - Collections of various type interpretations of the same underlying value in bits.
- **%rax**, **%rcx**, and **%rdx** are *caller-save registers*. The caller must copy those registers to its stack frame before the call and then restore the values to the registers — if it needs those data.
- **%rbx**, **%rsi**, and **%rdi** are *callee-save registers*. The callee must copy their contents to its stack before using the registers, and restore the original contents before returning.

* C data types

    - Integer sizes
    - Signed versus unsigned
    - Floating-point sizes
    - Pointer types
    - Relation between arrays and pointers
    - Complex types
    - C strings—their internal representation and structure
- The stack
    - Creation of the stack frame
    - Collapse of the stack frame
    - Role of SP and BP
    - Precise meanings of the **push**, **pop**, **call**, and **ret** instruction classes
    - Allocation of parameters and variables
    - Allocation of complex objects—structures, unions, arrays
    - Memory addressing modes for stack objects
- Other data storage
    - Read-only text—code and string literals
    - The heap
- Data representation
    - Binary and hexadecimal equivalents to decimal integers
    - Two's complement representations of positive and negative integers
    - Integer overflow for signed (two's complement) and unsigned types
    - **Floating-point representations**
        - **Be able to convert a floating-point representation back into a decimal number.**
    - Array and structure representations
    - Unions—what they are and how to use them
    - How to use **a mask to** reveal the bits of an object—
        - By shifting the mask repeatedly while keeping the input in place
        - By keeping the mask in place while shifting the input
        - With various sizes and configurations of mask—one-bit or spanning a range of bits
- Machine-level representation of programs
    - Operand sizes and their corresponding suffixes on opcode mnemonics
    - Register sizes and corresponding names
    - The **mov** instruction class—source and destination operands
    - Arithmetic and logical instructions
    - Shift operation—left and right, logical and arithmetic
    - The **load effective address** (lea) instruction

- Control instructions
  - Condition codes
  - Comparison instructions
  - Conditional and unconditional jumps
  - How if-constructs and loops are rendered in Assembly code