1. The x86 architecture is an instruction set architecture (ISA) series for computer processors. Developed by Intel Corporation, x86 architecture defines how a processor handles and executes different instructions passed from the operating system (OS) and software programs. The “x” in x86 denotes ISA version Key features include:
   * 1. Provides a logical framework for executing instructions through a processor
     2. Allows software programs and instructions to run on any processor in the

Intel 8086 family iii. Provides procedures for utilizing and managing the hardware components of a central processing unit (CPU)

1. The early microprocessors (and microcontrollers) had what would later be called CISC (Complex Instruction Set Computer) architecture. Most would have one or two accumulators. Instructions typically used one of the accumulators as one implied operand, and a memory operand as the second operand. Operations could be any arithmetic or logical operation. The memory operand might have many different types of indexing modes (direct, indirect, indexed, indexed indirect etc.)
2. Calling conventions describe the interface of called code:
   1. The order in which atomic (scalar) parameters, or individual parts of a complex parameter, are allocated
   2. How parameters are passed (pushed on the stack, placed in registers, or a mix of both)
   3. Which registers the called function must preserve for the caller (also known as: callee-saved registers or non-volatile registers)
   4. How the task of preparing the stack for, and restoring after, a function call is divided between the caller and the callee 4.
3. An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term assembly language.
4. A linker is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program. Computer programs are usually made up of multiple modules that span separate object files, each being a compiled computer program.
5. A loader is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution.
6. 4 bytes
7. The %rip register on x86-64 is a special-purpose register that always holds the memoryaddress of the next instruction to execute in the program's code segment. The processor increments %rip automatically after each instruction, and control flow instructions like branches set the value of %rip to change the next instruction. Perhaps surprisingly, %rip also shows up when an assembly program refers to a global variable. See the sidebar under "Addressing modes" below to understand how %rip-relative addressing works.

7.

Address range (hexadecimal)Size Device

|  |  |
| --- | --- |
| 0000–7FFF | 32 KiB RAM |
| 8000–80FF | 256 bytes General-purpose I/O |
| 9000–90FF | 256 bytes Sound controller |
| A000–A7FF | 2 KiB Video controller/text-mapped display RAM |
| C000–FFFF | 16 KiB ROM |

1. Single Instruction, Multiple Data
2. I would prefer AArch32 because AArch32 instructions operate only on registers with a fewinstructions for loading and storing data from/to memory while x86 can use memory or register operands with ALU instructions, sometimes getting the same work done in fewer instructions. Sometimes more because ARM has its own useful tricks like loading a pair of registers in one instruction, or using a shifted register as part of another operation. Up until ARMv8 / AArch64, ARM was a native 32 bit architecture, favoring four byte operations over others. So ARM is a simpler architecture, leading to small silicon area and lots of power save features while x86 becomes a power beast in terms of both power consumption and production.
3. A.

isOdd:

pushq %rbp .seh\_pushreg %rbp movq %rsp, %rbp .seh\_setframe %rbp, 0 .seh\_endprologue movl %ecx, 16(%rbp) movl 16(%rbp), %eax andl $1, %eax testl %eax, %eax jne .L2

movl $0, %eax jmp .L3

B.

someCalcs:

pushq %rbp .seh\_pushreg %rbp movq %rsp, %rbp .seh\_setframe %rbp, 0 .seh\_endprologue movl %ecx, 16(%rbp) movl %edx, 24(%rbp) movl %r8d, 32(%rbp)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| movl 16(%rbp), %edx movl 24(%rbp), %eax addl %eax, %edx movl 32(%rbp), %eax addl %edx, %eax popq %rbp  ret  .seh\_endproc  .globl arrRef1d |  |  |  |  |
| .def arrRef1d; .scl  .seh\_proc arrRef1d C.  arrRef1d:  pushq %rbp .seh\_pushreg %rbp movq %rsp, %rbp .seh\_setframe %rbp, 0 .seh\_endprologue movq %rcx, 16(%rbp) movl %edx, 24(%rbp) movl 24(%rbp), %eax  cltq  leaq 0(,%rax,4), %rdx movq 16(%rbp), %rax addq %rdx, %rax movl (%rax), %eax popq %rbp  ret  .seh\_endproc | 2; | .type | 32; | .endef |
| .def \_\_main; .scl  .section .rdata,"dr" | 2; | .type | 32; | .endef |