* Compiled implementations of languages have the advantage of being faster because they translate directly to the native code of the specific machine.
* Interpreted implementations of languages tend to be more portable as well.
* Interpreted implementations of languages are generally easier to create because writing compilers is difficult.
* **Advantages of compiled languages**
* Assembler, COBOL, PL/I, C/C++ are all translated by running the source code through a compiler. This results in very efficient code that can be executed any number of times. The overhead for the translation is incurred just once, when the source is compiled; thereafter, it need only be loaded and executed.
* Interpreted languages, in contrast, must be parsed, interpreted, and executed each time the program is run, thereby greatly adding to the cost of running the program. For this reason, interpreted programs are usually less efficient than compiled programs.
* **Advantages of interpreted languages**
* There are reasons for using languages that are compiled and reasons for using interpreted languages. There is no simple answer as to which language is "better"—it depends on the application. Even within an application we could end up using many different languages. For example, one of the strengths of a language like CLIST is that it is easy to code, test, and change. However, it is not very efficient. The trade-off is machine resources for programmer time.
* Keeping this in mind, we can see that it would make sense to use a compiled language for the intensive parts of an application (heavy resource usage), whereas interfaces (invoking the application) and less-intensive parts could be written in an interpreted language. An interpreted language might also be suited for ad hoc requests or even for prototyping an application.

Assembly Language allows you to carefully control the individual instructions the processor executes. As noted in answers, this fine-grain control can allow the following benefits:

* Use special processor instructions such as SIMD, memory controller manipulation, processor identification
* Access to special processor features such as supervisor instructions and hardware timers
* Use cache instructions, port I/O, memory mapped peripherals, and DMA registers
* Can handle interrupts and low (even zero) memory situations and context switches with high predictability
* Performance - one can directly optimize instructions, memory, and coherency for maximum computational and memory efficiency

Challenges are many too:

* Assembly language is specific to a processor. Even within a family (such as 80x86) there are vast differences between the instructions available for a 8086, 80386, and 80x86 that supports 64-bit instructions and that is before dealing with special instructions noted above such as SIMD. This means your available instructions (and there for the “language” you are programming in) is dependent on the actual processor you are running on!
* Very painstaking — even operations such as multiplying two numbers together requires the overhead of “getting” the number and deciding where to put the result… (which register? on the stack? write back to memory?)
* Interfacing to other languages can be complicated — you have push / pop your own variables and you have to pack the data in the form that language or library expects.
* Few built-ins in the language - a lot of “lifting” is required by the programmer including memory management and memory-object layout

In nutshell Assembly gives you control to do some things that high level languages cannot but due to the level of expressiveness requires many more lines of code. I would say assembly languages are “Computer-time efficient” and high level languages are “Human-time efficient”. Sometimes this is important such as when writing multimedia codecs but other times human time is much more important (bringing up applications quickly!).