Are instruction set and assembly language the same thing? - Stack Overflow

I think everyone is giving you the same answer. Instruction set is is the set (as in math) of all instructions the processor can execute or understand. Assembly language is a programming language.

Let me try some examples based on some of the questions you are asking. And I am going to be jumping around from processor to processor with whatever code I have handy.

Instruction or opcode or binary or machine language, whatever term you want to use for the bits/bytes that are loaded into the processor to be decoded and executed. An example

```
0x5C0B
```

The assembly language, would be

```
add r12,r11
```

For this particular processor. In this case that means r11 = r11 + r12. So I put that text, the add r12,r11 in a text file and use an assembler (a program that compiles/assembles assembly language) to assemble it into some form of binary. Like any programming language sometimes you create object files then link them together, sometimes you can go straight to a binary. And there are many forms of binaries which are in ascii and binary forms and a whole other discussion.

Now what can you do in assembler that is not part of the instruction set? How do they differ? Well for starters you can have macros:

```
.macro add3 arg1, arg2, arg3    add \arg1,\arg3    add \arg2,\arg3.endm.text   add
3 r10,r11,r12
```

Macros are like inline functions, they are not functions that are called but generate code in line. No different than a C macro for example. So you might use them to save some typing or you might use them to abstract something that you want to do over and over again and want the ability to change in one place and not have to touch every instance. The above example essentially generates this:

```
add r10,r12add r11,r12
```

Another difference between the instruction set and assembly langage are pseudo instructions, for this particular instruction set for example there is no pop instruction for popping things off the stack at least not by that name, and I will explain why. But you are allowed to save some typing and use a pop in your code:

```
pop r12
```

The reason why there is no pop is because the addressing modes are flexible enough to have a read from the address in the source register put the value in the destination register and increment the source register by a word. Which in assembler for this instruction set is

```
mov @r1+,r12
```

both the pop and the mov result in the opcode 0x413C.

Another example of differences between the instruction set and assembler, switching instruction sets, is something like this:

```
ldr r0,=bob
```

Which to this assembly language means load the address of bob into register 0, there is no instruction for that, what the assembler does with it is generate something that would look like this if you were to write it in assembler by hand:

```
ldr r0,ZZ123...ZZ123: .word bob
```

Essentially, in a reachable place from that instruction, not in the execution path, a word is created which the linker will fill in with the address for bob. The ldr instruction likewise by the assembler or linker will get encoded with an ldr of a pc relative instruction.

That leads to a whole category of differences between the instruction set and the assembly language

```
call fun
```

Machine code has no way of knowing what fun is or where to find it. For this instruction set with its many addressing modes (note I am specifically and intentionally avoiding naming the instruction sets I am using as that is not relevant to the discussion) the assembler or linker as the case may be (depending on where the fun function ends up being relative to this instruction).

The assembler may choose to encode that instruction as pc relative, if the fun function is 40 bytes ahead of the call instruction it may encode it with the equivalent of call pc+36 (take four off because the pc is one instruction ahead at execution time and this is a 4 byte instruction).

Or the assembler may not know where or what fun is and leave it up to the linker, and in that case the linker may put the absolute address of the function something that would be similar to call #0xD00D.

Same goes for loads and stores, some instruction sets have near and far pc relative, some have absolute address, etc. And you may not care to choose, you may just say

```
mov bob,r1
```

and the assembler or linker or a combination of the two takes care of the rest.

Note that for some instruction sets the assembler and linker may happen at once in one program. These days we are used to the model of compiling to objects and then linking objects, but not all assemblers follow that model.

Some more cases where the assembly language can take some shortcuts:

```
hang: b hang  b .  b 2f1:  b 1b  b 1f1:  b 1b2:
```

The hang: b hang makes sense, branch to the label called hang. Essentially a branch to self. And as the name implies this is an infinite loop. But for this assembly language b . means branch to self, an infinite loop but I didnt have to invent a label, type it and branch to it. Another shortcut is using numbers b 1b means branch to 1 back, the assembler looks for the label number 1 behind or above the instruction. The b 1f, which is not a branch to self, means branch 1 forward, this is perfectly valid code for this assembler. It will look forward or below the line of code for a label number 1: And you can re-use number 1 like crazy in your assembly language program for this assembler, saves on having to invent label names for simple short branches. The second b 1b branches to the second 1. and is a branch to self.

It is important to understand that the company that created the processor defines the instruction set, and the machine code or opcodes or whatever term they or you use for the bits and bytes the processor decodes and executes. Very often that company will produce a document with assembly language for those instructions, a syntax. Often that company will produce an assembler program to compile/assemble that assembly language...using that syntax. But that doesnt mean that any other person on the planet that chooses to write an assembler for that instruction set has to use that syntax. This is very evident with the x86 instruction set. Likewise any psuedo instructions like the pop above or macro syntax or other short cuts like the b 1b have to be honored from one assembler to another. And very often are not, you see this with ARM for example the universal comment symbol of ; does not work with gnu assembler you have to use @ instead. ARMs assembler does use the ; (note I write my arm assembler with ;@ to make it portable). It gets even worse with gnu tools for example you can can put C language things like #define and /* comment */ in your assembler and use the C compiler instead of the assembler and it will work. I prefer to stay as pure as I can for maximum portability, but naturally you may choose to use whatever features the tool offers.