What is the difference between an accumulator instruction set architecture and a general-purpose register instruction set architecture?

These represent two points in the design-space for instruction sets. Consider what a CPU instruction needs to do. For now, just consider *compute* instructions. At a minimum, it needs to provide the following information:

What computation to perform (add, subtract, shift, compare)

What values to perform the computation on (the inputs to the computation)

Where to put the result (the output of the computation)

For example, if I wrote the high-level program statement X = Y + Z, you need to tell the processor to fetch the values held in Y and Z, add them, and write the result to X.

In an accumulator architecture, most compute instructions operate on a special register called the accumulator. Most operations, therefore, have the accumulator as an *implicit* argument to the instruction. The accumulator either provides an input to the instruction, receives the output from the instruction, or both. To perform X = Y + Z on an accumulator-based machine, the instruction sequence would look roughly like this:

Load Y into the accumulator

Add Z to the accumulator

Store accumulator to X

If I had a more complex expression, such as "I = J + K + L + M + N + O", the sequence might look like this:

Load J into the accumulator

Add K to the accumulator

Add L to the accumulator

Add M to the accumulator

Add N to the accumulator

Add O to the accumulator

Store accumulator to I

In a **general-purpose register** architecture, compute instructions take multiple arguments to specify which registers to read values from. This makes them more flexible. But, the flexibility comes at a cost: You need more opcode bits to specify which registers to operate on, and you need to provide paths for all of those registers to the arithmetic unit.

For the example expressions above, the code ends up not looking much different:

Load Y into register R0

Load Z into register R1

Add R0 to R1, putting the result in R2

Store R2 to X

General purpose register machines start showing an advantage when you can keep values in registers across many operations. For example, suppose I wanted to run this slightly more complicated program:

X = A + B

Y = A - B

In an accumulator machine, I would have to reload A and B for both operations. In a general purpose machine, I would only load A and B once.