So far we've talked about assembling instructions.

What about data?

How do we allocate and initialize data storage and how do we get those values into registers so that they can be used as operands?

Here we see a program that allocates and initializes two memory locations using the LONG macro.

We've used labels to remember the addresses of these locations for later reference.

When the program is assembled the values of the label N and factN are 0 and 4 respectively, the addresses of the memory locations holding the two data values.

To access the first data value, the program uses a LD instruction, in this case one of convenience macros that supplies R31 as the default value of the Ra field.

The assembler replaces the reference to the label N with its value 0 from the symbol table.

When the LD is executed, it computes the memory address by adding the constant (0) to the value of the Ra register (which is R31 and hence the value is 0) to get the address (0) of the memory location from which to fetch the value to be placed in R1.

The constants needed as values for data words and instruction fields can be written as expressions.

These expressions are evaluated by the assembler as it assembles the program and the resulting value is used as needed.

Note that the expressions are evaluated at the time the assembler runs.

By the time the program runs on the Beta, the resulting value is used.

The assembler does NOT generate ADD and MUL instructions to compute the value during program execution.

If a value is needed for an instruction field or initial data value, the assembler has to be able to perform the arithmetic itself.

If you need the program to compute a value during execution, you have to write the necessary instructions as part of your program.

One last UASM feature: there's a special symbol ".", called "dot", whose value is the address of the next main memory location to be filled by the assembler when it generates binary data.

Initially "." is 0 and it's incremented each time a new byte value is generated.

We can set the value of "." to tell the assembler where in memory we wish to place a value.

In this example, the constant 0xDEADBEEF is placed into location 0x100 of main memory.

And we can use "." in expressions to compute the values for other symbols, as shown here when defining the value for the symbol "k".

In fact, the label definition "k:" is exactly equivalent to the UASM statement "k = ." We can even increment the value of "." to skip over locations, e.g., if we wanted to leave space for an un initialized array.

And that's assembly language!

We use assembly language as a convenient notation for generating the binary encoding for instructions and data.

We let the assembler build the bit-level representations we need and to keep track of the addresses where these values are stored in main memory.

UASM itself provides support for values, symbols, labels and macros.

Values can be written as constants or expressions involving constants.

We use symbols to give meaningful names to values so that our programs will be more readable and more easily modified.

Similarly, we use labels to give meaningful names to addresses in main memory and then use the labels in referring to data locations in LD or ST instructions, or to instruction locations in branch instructions.

Macros hide the details of how instructions are assembled from their component fields.

And we can use "." to control where the assembler places values in main memory.

The assembler is itself a program that runs on our computer.

That raises an interesting "chicken and egg problem": how did the first assembler program get assembled into binary so it could run on a computer?

Well, it was hand-assembled into binary.

I suspect it processed a very simple language indeed, with the bells and whistles of symbols, labels, macros, expression evaluation, etc. added only after basic instructions could be assembled by the program.

And I'm sure they were very careful not loose the binary so they wouldn't have to do the hand-assembly a second time!