Let's take a closer look at how macros work in UASM.

Here we see the definition of the macro "consec" which has a single parameter "n".

The body of the macro is a sequence of four expressions.

When there's an invocation of the "consec" macro, in this example with the argument 37, the body of the macro is expanded replacing all occurrences of "n" with the argument 37.

The resulting text is then processed as if it had appeared in place of the macro invocation.

In this example, the four expressions are evaluated to give a sequence of four values that will be placed in the next four bytes of the output array.

Macro expansions may contain other macro invocations, which themselves will be expanded, continuing until all that's left are expressions to be evaluated.

Here we see the macro definition for WORD, which assembles its argument into two consecutive bytes.

And for the macro LONG, which assembles its argument into four consecutive bytes, using the WORD macro to process the low 16 bits of the value, then the high 16 bits of the value.

These two UASM statements cause the constant 0xDEADBEEF to converted to 4 bytes, which are then deposited in the output array starting at index 0x100.

Note that the Beta expects the least-significant byte of a multi-byte value to be stored at the lowest byte address.

So the least-significant byte 0xEF is placed at address 0x100 and the most-significant byte 0xDE is placed at address 0x103.

This is the "little-endian" convention for multi-byte values: the least-significant byte comes first.

Intel's x86 architecture is also little-endian.

There is a symmetrical "big-endian" convention where the most-significant byte comes first.

Both conventions are in active use and, in fact, some ISAs can be configured to use either convention!

There is no right answer for which convention to use, but the fact that there two conventions means that we have to be alert for the need to convert the representation of multi-byte values when moving values between one ISA and another, e.g., when we send a data file to another user.

As you can imagine there are strong advocates for both schemes who are happy to defend their point of view at great length.

Given the heat of the discussion, it's appropriate that the names for the conventions were drawn from Jonathan Swift's "Gulliver's Travels" in which a civil war is fought over whether to open a soft-boiled egg at its big end or its little end.

Let's look at the macros used to assemble Beta instructions.

The BETAOP helper macro supports the 3-register instruction format, taking as arguments the values to be placed in the OPCODE, Ra, Rb, and Rc fields.

The ".align 4" directive is a bit of administrative bookkeeping to ensure that instructions will have a byte address that's a multiple of 4, i.e., that they span exactly one 32-bit word in memory.

That's followed by an invocation of the LONG macro to generate the 4 bytes of binary data representing the value of the expression shown here.

The expression is where the actual assembly of the fields takes place.

Each field is limited to requisite number of bits using the modulo operator (%), then shifted left (< to the correct position in the 32-bit word.

And here are the helper macros for the instructions that use a 16-bit constant as the second operand.

Let's follow the assembly of an ADDC instruction to see how this works.

The ADDC macro expands into an invocation of the BETAOPC helper macro, passing along the correct value for the ADDC opcode, along with the three operands.

The BETAOPC macro does the following arithmetic: the OP argument, in this case the value 0x30, is shifted left to occupy the high-order 6 bits of the instruction.

Then the RA argument, in this case 15, is placed in its proper location.

The 16-bit constant -32768 is positioned in the low 16 bits of the instruction.

And, finally, the Rc argument, in this case 0, is positioned in the Rc field of the instruction.

You can see why we call this processing "assembling an instruction".

The binary representation of an instruction is assembled from the binary values for each of the instruction fields.

It's not a complicated process, but it requires a lot of shifting and masking, tasks that we're happy to let a computer handle.

Here's the entire sequence of macro expansions that assemble this ADDC instruction into an appropriate 32-bit binary value in main memory.

You can see that the knowledge of Beta instruction formats and opcode values is built into the bodies of the macro definitions.

The UASM processing is actually quite general.

With a different set of macro definitions it could process assembly language programs for almost any ISA!

All the macro definitions for the Beta ISA are provided in the beta.uasm file, which is included in each of the assembly language lab assignments.

Note that we include some convenience macros to define shorthand representations that provide common default values for certain operands.

For example, except for procedure calls, we don't care about the PC+4 value saved in the destination register by branch instructions, so almost always would specify R31 as the Rc register, effectively discarding the PC+4 value saved by branches.

So we define two-argument branch macros that automatically provide R31 as the destination register.

Saves some typing, and, more importantly, it makes it easier to understand the assembly language program.

Here are a whole set of convenience macros intended to make programs more readable.

For example, unconditional branches can be written using the BR() macro rather than the more cumbersome BEQ(R31,...).

And it's more readable to use branch-false (BF) and branch-true (BT) macros when testing the results of a compare instruction.

And note the PUSH and POP macros at the bottom of page.

These expand into multi-instruction sequences, in this case to add and remove values from a stack data structure

pointed to by the SP register.

We call these macros "pseudo instructions" since they let us provide the programmer with what appears a larger instruction set, although underneath the covers we've just using the same small instruction repertoire developed in Lecture 9.

In this example we've rewritten the original code we had for the factorial computation using pseudo instructions.

For example, CMOVE is a pseudo instruction for moving small constants into a register.

It's easier for us to read and understand the intent of a "constant move" operation than an "add a value to 0" operation provided by the ADDC expansion of CMOVE.

Anything we can do to remove the cognitive clutter will be very beneficial in the long run.