One of the most useful abstractions provided by high-level languages is the notion of a procedure or subroutine, which is a sequence of instructions that perform a specific task.

A procedure has a single named entry point, which can be used to refer to the procedure in other parts of the program.

In the example here, this code is defining the GCD procedure, which is declared to return an integer value.

Procedures have zero or more formal parameters, which are the names the code inside the procedure will use to refer the values supplied when the procedure is invoked by a "procedure call".

A procedure call is an expression that has the name of the procedure followed by parenthesized list of values called "arguments" that will be matched up with the formal parameters.

For example, the value of the first argument will become the value of the first formal parameter while the procedure is executing.

The body of the procedure may define additional variables, called "local variables", since they can only be accessed by statements in the procedure body.

Conceptually, the storage for local variables only exists while the procedure is executing.

They are allocated when the procedure is invoked and deallocated when the procedure returns.

The procedure may return a value that's the result of the procedure's computation.

It's legal to have procedures that do not return a value, in which case the procedures would only be executed for their "side effects", e.g., changes they make to shared data.

Here we see another procedure, COPRIMES, that invokes the GCD procedure to compute the greatest common divisor of two numbers.

To use GCD, the programmer of COPRIMES only needed to know the input/output behavior of GCD, i.e., the number and types of the arguments and what type of value is returned as a result.

The procedural abstraction has hidden the implementation of GCD, while still making its functionality available as a "black box".

This is a very powerful idea: encapsulating a complex computation so that it can be used by others.

Every high-level language comes with a collection of pre-built procedures, called "libraries", which can be used to

perform arithmetic functions (e.g., square root or cosine), manipulate collections of data (e.g., lists or dictionaries), read data from files, and so on - the list is nearly endless!

Much of the expressive power and ease-of-use provided by high-level languages comes from their libraries of "black boxes".

The procedural abstraction is at the heart of object-oriented languages, which encapsulate data and procedures as black boxes called objects that support specific operations on their internal data.

For example, a LIST object has procedures (called "methods" in this context) for indexing into the list to read or change a value, adding new elements to the list, inquiring about the length of the list, and so on.

The internal representation of the data and the algorithms used to implement the methods are hidden by the object abstraction.

Indeed, there may be several different LIST implementations to choose from depending on which operations you need to be particularly efficient.

Okay, enough about the virtues of the procedural abstraction!

Let's turn our attention to how to implement procedures using the Beta ISA.

A possible implementation is to "inline" the procedure, where we replace the procedure call with a copy of the statements in the procedure's body, substituting argument values for references to the formal parameters.

In this approach we're treating procedures very much like UASM macros, i.e., a simple notational shorthand for making a copy of the procedure's body.

Are there any problems with this approach?

One obvious issue is the potential increase in the code size.

For example, if we had a lengthy procedure that was called many times, the final expanded code would be huge!

Enough so that inlining isn't a practical solution except in the case of short procedures where optimizing compilers do sometimes decide to inline the code.

A bigger difficulty is apparent when we consider a recursive procedure where there's a nested call to the procedure itself.

During execution the recursion will terminate for some values of the arguments and the recursive procedure will eventually return answer.

But at compile time, the inlining process would not terminate and so the inlining scheme fails if the language allows recursion.

The second option is to "link" to the procedure.

In this approach there is a single copy of the procedure code which we arrange to be run for each procedure call - all the procedure calls are said to link to the procedure code.

Here the body of the procedure is translated once into Beta instructions and the first instruction is identified as the procedure's entry point.

The procedure call is compiled into a set of instructions that evaluate the argument expressions and save the values in an agreed-upon location.

Then we'll use a BR instruction to transfer control to the entry point of the procedure.

Recall that the BR instruction not only changes the PC but saves the address of the instruction following the branch in a specified register.

This saved address is the "return address" where we want execution to resume when procedure execution is complete.

After branching to the entry point, the procedure code runs, stores the result in an agreed-upon location and then resumes execution of the calling program by jumping to the supplied return address.

To complete this implementation plan we need a "calling convention" that specifies where to store the argument values during procedure calls and where the procedure should store the return value.

It's tempting to simply allocate specific memory locations for the job.

How about using registers?

We could pass the argument value in registers starting, say, with R1.

The return address could be stored in another register, say R28.

As we can see, with this convention the BR and JMP instructions are just what we need to implement procedure call and return.

It's usual to call the register holding the return address the "linkage pointer".

And finally the procedure can use, say, R0 to hold the return value.

Let's see how this would work when executing the procedure call fact(3).

As shown on the right, fact(3) requires a recursive call to compute fact(2), and so on.

Our goal is to have a uniform calling convention where all procedure calls and procedure bodies use the same convention for storing arguments, return addresses and return values.

In particular, we'll use the same convention when compiling the recursive call fact(n-1) as we did for the initial call to fact(3).

Okay.

In the code shown on the right we've used our proposed convention when compiling the Beta code for fact().

Let's take a quick tour.

To compile the initial call fact(3) the compiler generated a CMOVE instruction to put the argument value in R1 and then a BR instruction to transfer control to fact's entry point while remembering the return address in R28.

The first statement in the body of fact tests the value of the argument using CMPLEC and BT instructions.

When n is greater than 0, the code performs a recursive call to fact, saving the value of the recursive argument n-1 in R1 as our convention requires.

Note that we had to first save the value of the original argument n because we'll need it for the multiplication after the recursive call returns its value in R0.

If n is not greater than 0, the value 1 is placed in R0.

Then the two possible execution paths merge, each having generated the appropriate return value in R0, and finally there's a JMP to return control to the caller.

The JMP instruction knows to find the return address in R28, just where the BR put it as part of the original procedure call.

Some of you may have noticed that there are some difficulties with this particular implementation.

The code is correct in the sense that it faithfully implements procedure call and return using our proposed convention.

The problem is that during recursive calls we'll be overwriting register values we need later.

For example, note that following our calling convention, the recursive call also uses R28 to store the return address.

When executed, the code for the original call stored the address of the HALT instruction in R28.

Inside the procedure, the recursive call will store the address of the MUL instruction in R28.

Unfortunately that overwrites the original return address.

Even the attempt to save the value of the argument N in R2 is doomed to fail since during the execution of the recursive call R2 will be overwritten.

The crux of the problem is that each recursive call needs to remember the value of its argument and return address, i.e., we need two storage locations for each active call to fact().

And while executing fact(3), when we finally get to calling fact(0) there are four nested active calls, so we'll need 4*2 = 8 storage locations.

In fact, the amount of storage needed varies with the depth of the recursion.

Obviously we can't use just two registers (R2 and R28) to hold all the values we need to save.

One fix is to disallow recursion!

And, in fact, some of the early programming languages such as FORTRAN did just that.

But let's see if we can solve the problem another way.