The problem we need to solve is where to store the values needed by procedure: its arguments, its return address, its return value.

The procedure may also need storage for its local variables and space to save the values of the caller's registers before they get overwritten by the procedure.

We'd like to avoid any limitations on the number of arguments, the number of local variables, etc.

So we'll need a block of storage for each active procedure call, what we'll call the "activation record".

As we saw in the factorial example, we can't statically allocate a single block of storage for a particular procedure since recursive calls mean we'll have many active calls to that procedure at points during the execution.

What we need is a way to dynamically allocate storage for an activation record when the procedure is called, which can then be reclaimed when the procedure returns.

Let's see how activation records come and go as execution proceeds.

The first activation record is for the call fact(3).

It's created at the beginning of the procedure and holds, among other things, the value of the argument n and the return address where execution should resume after the fact(3) computation is complete.

During the execution of fact(3), we need to make a recursive call to compute fact(2).

So that procedure call also gets an activation record with the appropriate values for the argument and return address.

Note that the original activation record is kept since it contains information needed to complete the computation of fact(3) after the call to fact(2) returns.

So now we have two active procedure calls and hence two activation records.

fact(2) requires computing fact(1), which, in turn, requires computing fact(0).

At this point there are four active procedure calls and hence four activation records.

The recursion terminates with fact(0), which returns the value 1 to its caller.

At this point we've finished execution of fact(0) and so its activation record is no longer needed and can be discarded.

fact(1) now finishes its computation returning 1 to its caller.

We no longer need its activation record.

Then fact(2) completes, returning 2 to its caller and its activation record can be discarded.

And so on… Note that the activation record of a nested procedure call is always discarded before the activation record of the caller.

That makes sense: the execution of the caller can't complete until the nested procedure call returns.

What we need is a storage scheme that efficiently supports the allocation and deallocation of activation records as shown here.

Early compiler writers recognized that activation records are allocated and deallocated in last-in first-out (LIFO) order.

So they invented the "stack", a data structure that implements a PUSH operation to add a record to the top of the stack and a POP operation to remove the top element.

New activation records are PUSHed onto the stack during procedure calls and the POPed from the stack when the procedure call returns.

Note that stack operations affect the top (i.e., most recent) record on the stack.

C procedures only need to access the top activation record on the stack.

Other programming languages, e.g. Java, support accesses to other active activation records.

The stack supports both modes of operation.

One final technical note: some programming languages support closures (e.g., Javascript) or continuations (e.g., Python's yield statement), where the activation records need to be preserved even after the procedure returns.

In these cases, the simple LIFO behavior of the stack is no longer sufficient and we'll need another scheme for allocating and deallocating activation records.

But that's a topic for another course!

Here's how we'll implement the stack on the Beta: We'll dedicate one of the Beta registers, R29, to be the "stack pointer" that will be used to manage stack operations.

When we PUSH a word onto the stack, we'll increment the stack pointer.

So the stack grows to successively higher addresses as words are PUSHed onto the stack.

We'll adopt the convention that SP points to (i.e., its value is the address of) the first unused stack location, the location that will be filled by next PUSH.

So locations with addresses lower than the value in SP correspond to words that have been previously allocated.

Words can be PUSHed to or POPed from the stack at any point in execution, but we'll impose the rule that code sequences that PUSH words onto the stack must POP those words at the end of execution.

So when a code sequence finishes execution, SP will have the same value as it had before the sequence started.

This is called the "stack discipline" and ensures that intervening uses of the stack don't affect later stack references.

We'll allocate a large region of memory to hold the stack located so that the stack can grow without overwriting other program storage.

Most systems require that you specify a maximum stack size when running a program and will signal an execution error if the program attempts to PUSH too many items onto the stack.

For our Beta stack implementation, we'll use existing instructions to implement stack operations, so for us the stack is strictly a set of software conventions.

Other ISAs provide instructions specifically for stack operations.

There are many other sensible stack conventions, so you'll need to read up on the conventions adopted by the particular ISA or programming language you'll be using.

We've added some convenience macros to UASM to support stacks.

The PUSH macro expands into two instructions.

The ADDC increments the stack pointer, allocating a new word at the top of stack, and then initializes the new top-of-stack from a specified register value with a ST instruction.

The POP macro LDs the value at the top of the stack into the specified register, then uses a SUBC instruction to decrement the stack pointer, deallocating that word from the stack.

Note that the order of the instructions in the PUSH and POP macro is very important.

As we'll see in the next lecture, interrupts can cause the Beta hardware to stop executing the current program between any two instructions, so we have to be careful about the order of operations.

So for PUSH, we first allocate the word on the stack, then initialize it.

If we did it the other way around and execution was interrupted between the initialization and allocation, code run during the interrupt which uses the stack might unintentionally overwrite the initialized value.

But, assuming all code follows stack discipline, allocation followed by initialization is always safe.

The same reasoning applies to the order of the POP instructions.

We first access the top-of-stack one last time to retrieve its value, then we deallocate that location.

We can use the ALLOCATE macro to reserve a number of stack locations for later use.

Sort of like PUSH but without the initialization.

DEALLOCATE performs the opposite operation, removing N words from the stack.

In general, if we see a PUSH or ALLOCATE in an assembly language program, we should be able to find the corresponding POP or DEALLOCATE, which would indicate that stack discipline is maintained.

We'll use stacks to save values we'll need later.

For example, if we need to use some registers for a computation but don't know if the register's current values are needed later in the program, we can PUSH their current values onto the stack and then are free to use the registers in our code.

After we're done, we can use POP to restore the saved values.

Note that we POP data off the stack in the opposite order that the data was PUSHed, i.e., we need to follow the last-in first-out discipline imposed by the stack operations.

Now that we have the stack data structure, we'll use it to solve our problems with allocating and deallocating activation records during procedure calls.