

We'll use the stack to hold a procedure's activation record.

That includes the values of the arguments to the procedure call.

We'll allocate words on the stack to hold the values of the procedure's local variables, assuming we don't keep them in registers.

And we'll use the stack to save the return address (passed in LP) so the procedure can make nested procedure calls without overwriting its return address.

The responsibility for allocating and deallocating the activation record will be shared between the calling procedure (the "caller") and the called procedure (the "callee").

The caller is responsible for evaluating the argument expressions and saving their values in the activation record being built on the stack.

We'll adopt the convention that the arguments are pushed in reverse order, i.e., the first argument will be the last to be pushed on the stack.

We'll explain why we made this choice in a couple of slides... The code compiled for a procedure involves a sequence of expression evaluations, each followed by a PUSH to save the calculated value on the stack.

So when the callee starts execution, the top of the stack contains the value of the first argument, the next word down the value of the second argument, and so on.

After the argument values, if any, have been pushed on the stack, there's a BR to transfer control to the procedure's entry point, saving the address of the instruction following the BR in the linkage pointer, R28, a register that we'll dedicate to that purpose.

When the callee returns and execution resumes in the caller, a DEALLOCATE is used to remove all the argument values from the stack, preserving stack discipline.

So that's the code the compiler generates for the procedure.

The rest of the work happens in the called procedure.

The code at the start of the called procedure completes the allocation of the activation record.

Since when we're done the activation record will occupy a bunch of consecutive words on the stack, we'll sometimes refer to the activation record as a "stack frame" to remind us of where it lives.

The first action is to save the return address found in the LP register.

This frees up LP to be used by any nested procedure calls in the body of the callee.

In order to make it easy to access values stored in the activation record, we'll dedicate another register called the "base pointer" (BP = R27) which will point to the stack frame we're building.

So as we enter the procedure, the code saves the pointer to the caller's stack frame, and then uses the current value of the stack pointer to make BP point to the current stack frame.

We'll see how we use BP in just a moment.

Now the code will allocate words in the stack frame to hold the values for the callee's local variables, if any.

Finally, the callee needs to save the values of any registers it will use when executing the rest of its code.

These saved values can be used to restore the register values just before returning to the caller.

This is called the "callee saves" convention where the callee guarantees that all register values will be preserved across the procedure call.

With this convention, the code in the caller can assume any values it placed in registers before a nested procedure call will still be there when the nested call returns.

Note that dedicating a register as the base pointer isn't strictly necessary.

All accesses to the values on the stack can be made relative to the stack pointer, but the offsets from SP will change as values are PUSHed and POPed from the stack, e.g., during procedure calls.

It will be easier to understand the generated code if we use BP for all stack frame references.

Let's return to the question about the order of argument values in the stack frame.

We adopted the convention of PUSHing the values in reverse order, i.e., where the value of the first argument is the last one to be PUSHED.

So, why PUSH argument values in reverse order?

With the arguments PUSHed in reverse order, the first argument (labeled "arg 0") will be at a fixed offset from the base pointer regardless of the number of argument values pushed on the stack.

The compiler can use a simple formula to determine the correct BP offset value for any particular argument.

So the first argument is at offset -12, the second at -16, and so on.

Why is this important?

Some languages, such as C, support procedure calls with a variable number of arguments.

Usually the procedure can determine from, say, the first argument, how many additional arguments to expect.

The canonical example is the C printf function where the first argument is a format string that specifies how a sequence of values should be printed.

So a call to printf includes the format string argument plus a varying number of additional arguments.

With our calling convention the format string will always be in the same location relative to BP, so the printf code can find it without knowing the number of additional arguments in the current call.

The local variables are also at fixed offsets from BP.

The first local variable is at offset 0, the second at offset 4, and so on.

So, we see that having a base pointer makes it easy to access the values of the arguments and local variables using fixed offsets that can be determined at compile time.

The stack above the local variables is available for other uses, e.g., building the activation record for a nested procedure call!