Let's practice our newfound skill and see what we can determine about a running program which we've stopped somewhere in the middle of its execution.

We're told that a computation of fact() is in progress and that the PC of the next instruction to be executed is 0x40.

We're also given the stack dump shown on right.

Since we're in the middle of a fact computation, we know that current stack frame (and possibly others) is an activation record for the fact function.

Using the code on the previous slide we can determine the layout of the stack frame and generate the annotations shown on the right of the stack dump.

With the annotations, it's easy to see that the argument to current active call is the value 3.

Now we want to know the argument to original call to fact.

We'll have to label the other stack frames using the saved BP values.

Looking at the saved LP values for each frame (always found at an offset of -8 from the frame's BP), we see that many of the saved values are 0x40, which must be the return address for the recursive fact calls.

Looking through the stack frames we find the first return address that's *not* 0x40, which must an return address to code that's not part of the fact procedure.

This means we've found the stack frame created by the original call to fact and can see that argument to the original call is 6.

What's the location of the BR that made the original call?

Well, the saved LP in the stack frame of the original call to fact is 0x80.

That's the address of the instruction following the original call, so the BR that made the original call is one instruction earlier, at location 0x7C.

To answer these questions you have to be good at hex arithmetic!

What instruction is about to be executed?

We were told its address is 0x40, which we notice is the saved LP value for all the recursive fact calls.

So 0x40 must be the address of the instruction following the BR(fact,LP) instruction in the fact code.

Looking back a few slides at the fact code, we see that's a DEALLOCATE(1) instruction.

What value is in BP?

Hmm.

We know BP is the address of the stack location containing the saved R1 value in the current stack frame.

So the saved BP value in the current stack frame is the address of the saved R1 value in the *previous* stack frame.

So the saved BP value gives us the address of a particular stack location, from which we can derive the address of all the other locations!

Counting forward, we find that the value in BP must be 0x13C.

What value is in SP?

Since we're about to execute the DEALLOCATE to remove the argument of the nested call from the stack, that argument must still be on the stack right after the saved R1 value.

Since the SP points to first unused stack location, it points to the location after that word, so it has the value 0x144.

Finally, what value is in R0?

Since we've just returned from a call to fact(2) the value in R0 must the result from that recursive call, which is 2.

Wow!

You can learn a lot from the stacked activation records and a little deduction!

Since the state of the computation is represented by the values of the PC, the registers, and main memory, once we're given that information we can tell exactly what the program has been up to.

Pretty neat… Wrapping up, we've been dedicating some registers to help with our various software conventions.

To summarize: R31 is always zero, as defined by the ISA.

We'll also dedicate R30 to a particular function in the ISA when we discuss the implementation of the Beta in the

next lecture.

Meanwhile, don't use R30 in your code!

The remaining dedicated registers are connected with our software conventions: R29 (SP) is used as the stack pointer, R28 (LP) is used as the linkage pointer, and R27 (BP) is used as the base pointer.

As you practice reading and writing code, you'll grow familiar with these dedicated registers.

In thinking about how to implement procedures, we discovered the need for an activation record to store the information needed by any active procedure call.

An activation record is created by the caller and callee at the start of a procedure call.

And the record can be discarded when the procedure is complete.

The activation records hold argument values, saved LP and BP values along with the caller's values in any other of the registers.

Storage for the procedure's local variables is also allocated in the activation record.

We use BP to point to the current activation record, giving easy access the values of the arguments and local variables.

We adopted a "callee saves" convention where the called procedure is obligated to preserve the values in all registers except for R0.

Taken together, these conventions allow us to have procedures with arbitrary numbers of arguments and local variables, with nested and recursive procedure calls.

We're now ready to compile and execute any C program!