MITOCW | MIT6_004S17_12-02-06-01_300k

In order to understand how procedures are implemented on the beta, we will take a look at a mystery function and its translation into beta assembly code.

The mystery function is shown here: The function f takes an argument x as an input.

It then performs a logical AND operation on the input x and the constant 5 to produce the variable a. After that, it checks if the input x is equal to 0, and if so returns the value 0, otherwise it returns an unknown value which we need to determine. We are provided with the translation of this C code into beta assembly as shown here. We take a closer look at the various parts of this code to understand how this function as well as procedures in general work on the beta. The code that calls the procedure is responsible for pushing any arguments onto the stack. This is shown in pink in the code and on the stack. If there are multiple arguments then they are pushed in reverse order so that the first argument is always in the same location relative to the BP, or base pointer, register which we will see in a moment.

The BR instruction branches to label f after storing the return address, which is b, into the LP, or linkage pointer, register. In yellow, we see the entry sequence for the procedure. The structure of this entry sequence is identical for all procedures. The first thing it does is PUSH(LP) which pushes the LP register onto the stack immediately after the arguments that were pushed onto the stack by the caller. Next it pushes the BP onto the stack in order to save the most recent value of the BP register before updating it.

Now, a MOVE(SP, BP) is performed. SP is the stack pointer which always points to the next empty location on the stack. At the time that this MOVE operation is executed, the SP, points to the location immediately following the saved BP.

This move instruction makes the BP point to the same location that the SP is currently pointing to, which is the location that is immediately following the saved BP.

Note, that once the BP register is set up, one can always find the first argument at location BP – 12 (or in other words, 3 words before the current BP register).

If there was a second argument, it could be found in location BP – 16, and so on.

Next, we allocate space on the stack for any local variables.

This procedure allocates space for one local variable.

Finally, we push all registers that are going to be modified by our procedure onto the stack.

Doing this makes it possible to recover the registers' original values once the procedure completes execution. In this example, register R1 is saved onto the stack. Once the entry sequence is complete, the BP register still points to the location immediately following the saved BP.

The SP, however, now points to the location immediately following the saved R1 register.

So for this procedure, after executing the entry sequence, the stack has been modified as shown here. The procedure return, or exit, sequence for all beta procedures follows the same structure. It is assumed that the return value for the procedure has already been placed into register R0.

Next, all registers that were used in the procedure body, are restored to their original values. This is followed by deallocating all of the local variables from the stack. We then restore the BP, followed by the LP register. Finally, we jump to LP which contains the return address of our procedure. In this case, LP contains the address b which is the address of the next instruction that should be executed following the execution of the f procedure. Taking a closer look at the details for our example, we see that we begin our exit sequence with POP(R1) in order to restore the original value of register R1. Note that this also frees up the location on the stack that was used to store the value of R1.

Next, we get rid of the local variables we stored on the stack.

This is achieved using the MOVE(BP, SP) instruction which makes the SP point to the same location as the BP thus specifying that all the locations following the updated SP are now considered unused. Next, we restore the BP register.

Restoring the BP register is particularly important for nested procedure calls.

If we did not restore the BP register, then upon return to the calling procedure, the calling procedure would no longer have a correct BP, so it would not be able to rely on the fact that it's first argument is located at location BP-12, for example.

Finally, we restore the LP register and JMP to the location of the restored LP register.

This is the return address, so by jumping to LP, we return from our procedure call and are now ready to execute the next instruction at label b.

Now let's get back to our original procedure and its translation to beta assembly.

We will now try to understand what this mystery function is actually doing by examining the remaining sections of our assembly code highlighted here.

Let's zoom into the highlighted code. The LD instruction loads the first argument into register R0. Recall that the first argument can always be found at location BP – 12, or in other words, 3 words before the current BP register.

This means that the value x is loaded into R0.

Next we perform a binary AND operation between R0 and the constant 5, and store the result of that operation into register R1. Note that its okay to overwrite R1 because the entry sequence already saved a copy of the original R1 onto the stack.

Also, note that overwriting R0 is considered fine because we ultimately expect the result to be returned in R0, so there is no expectation of maintaining the original value of R0.

Looking back at the c code of our function, we see that the bitwise AND of x and 5 is stored into a local variable named a. In our entry sequence, we allocated 1 word on the stack for our local variables. That is where we want to store this intermediate result. The address of this location is equal to the contents of the BP register. Since the destination of a store operation is determined by adding the contents of the last register in the instruction to the constant, the destination of this store operation is the value of BP + 0.

So as expected, variable a is stored at the location pointed to by the BP register.

Now we check if x equals 0 and if so we want to return the value 0.

This is achieved in beta assembly by checking if R0 is equal to 0 since R0 was loaded with the value of x by the LD operation. The BEQ operation checks whether or not this condition holds and if so, it branches to label bye which is our exit sequence.

In that situation, we just saw that R0 = 0, so R0 already contains the correct return value and we are ready to execute our return sequence.

If x is not equal to 0, then we perform the instructions after label xx.

By figuring out what these instructions do, we can identify the value of our mystery function labeled ?????. We begin by decrementing R0 by 1.

This means that R0 will be updated to hold x-1.

We then push this value onto the stack and make a recursive call to procedure f.

In other words, we call f again with a new argument which is equal to x-1.

So far we know that our mystery function will contain the term f(x-1).

We also see that LP gets updated with the new return address which is yy + 4.

So just before our recursive call to f with the new argument x-1, our stack looks like this. After the procedure entry sequence is executed in the first recursive call, our stack looks like this.

Note that this time the saved LP is yy + 4 because that is our return address for the recursive procedure call. The previous BP points to where the BP was pointing to in the original call to f. Another term for this group of stack elements is the activation record. In this example, each activation record consists of 5 elements. These are the argument to f, the saved LP, the saved BP, the local variable, and the saved R1.

Each time that f is called recursively another activation record will be added to the stack.

When we finally return from all of these recursive calls, we are back to a stack that looks like this with a single activation record left on the stack plus the first argument with which the recursive call was made. The DEALLOCATE(1) instruction then removes this argument from the stack. So the SP is now pointing to the location where we previously pushed the argument x-1. R0 holds the return value from the recursive call to f which is the value of f(x-1). Now, we execute a LD into register R1 of the address that is the contents of register BP + 0.

This value is a. We then ADD R1 to R0 to produce our final result in R0. R0 is now equal to a + f(x-1), so we have discovered that our mystery function is a + f(x-1).

Before we continue with analyzing a stack trace from this problem, let's answer a few simpler questions. The first question is whether or not variable a, from the statement a = x & 5, is stored on the stack and if so where is it stored relative to the BP register. Earlier we saw that our assembly program allocates space for one local variable on the stack. It then stores R1, which holds the result of performing a binary ANDC between x and the constant 5, into the location pointed to by the BP register, as shown here. Next, we want to translate the instruction at label yy into its binary representation. The instruction at label yy is BR(f, LP).

This instruction is actually a macro that translates to a BEQ(R31, f, LP).

Note that because R31 always equals 0, this branch is always taken.

The format of the binary representation for this instruction is a 6-bit opcode, followed by a 5 bit Rc identifier, followed by another 5 bits which specify Ra, and then followed by a 16 bit literal. The opcode for BEQ is 011100.

Rc = LP which is register R28. The 5-bit encoding of 28 is 11100.

Ra is R31 whose encoding is 11111. Now, we need to determine the value of the literal in this instruction. The literal in a branch instruction stores the offset measured in words from the instruction immediately following the branch, to the destination address. Looking at our assembly code for this function, we see that we want to count the number of words from the DEALLOCATE(1) instruction back to label f. Recall that the PUSH and POP macros are actually each made of up two instructions so each of those counts as 2 words.

Counting back, and accounting for the two instructions per push and pop, we see that we have to go back 16 instructions, so our literal is -16 expressed as a 16 bit binary number. Positive 16 is 0000 0000 0001 0000, so -16 is 1111 1111 1111 0000. Now, suppose that the function f is called from an external main program, and the machine is halted when a recursive call to f is about to execute the BEQ instruction tagged xx. The BP register of the halted machine contains 0x174, and the hex contents of a region of memory are shown here.

The values on the left of the stack are the addresses of each location on the stack.

We first want to determine the current value of the SP, or stack pointer, register.

We were told that the machine is halted when a recursive call to f is about to execute the BEQ instruction tagged xx. And that the BP register was 0x174 at that point. We see that after the BP was updated to be equal to the SP in the MOVE operation, two additional entries were made on the stack.

The first was an ALLOCATE instruction which allocated space for one local variable, thus making the SP point to location 0x178, and then PUSH(R1), which saves a copy of R1 on the stack, thus moving the SP register one further location down to 0x17C.

We now want to answer some questions about the stack trace itself to help us better understand its structure. We first want to determine the value of local variable a in the current stack frame. We know that a is stored at location BP + 0. So a is the variable stored at address 0x174, and that value is 5. From here, we can label all the entries in the stack trace as follows. We saw earlier, that each activation record consists of 5 words, the argument x followed by the saved LP, the saved BP, the local variable, and the saved register R1. We can apply this structure to label our stack trace. Now that our stack trace is fully labeled we can take a closer look at the details of implementing procedures on the beta.

We begin by looking at the multiple LP values that were stored on the stack.

Note that the first one at address 0x144 has a value of 0x5C whereas the following two have a value of 0xA4. This occurs because the LP value stored at address 0x144 is the return address from the main procedure that originally called procedure f, whereas the following two LP values are the return addresses from recursive calls to f made from within the f function itself. Using this information you can now answer the question: What is the address of the BR instruction that made the original call to f from the external main program? Recall that the value stored in the LP register is actually the return address which is the address of the instruction immediately following the branch instruction. So if the original LP value was 0x5C, that means that the address of the branch instruction was 0x58.

We can also answer the question, what is the value of the PC when the program is halted.

We know that the program was halted just before executing the instruction at label xx.

We also know, that the instruction at label yy makes a recursive call to f.

We know that the LP value from the recursive calls is 0xA4.

This means that the address of the DEALLOCATE(1) instruction is 0xA4.

Counting backwards by 4 bytes, and accounting for the fact that a PUSH operation consists of two instructions, we see that label xx = 0x90 and that is the value of the PC when the program is halted. As our last question, we want to consider the following: Suppose that you are told that you could delete 4 instructions from your program without affecting the behavior of the program.

The 4 instructions to be removed are a LD, a ST, an ALLOCATE, and a MOVE instruction.

Removing these instructions would make our program shorter and faster.

So, our goal is to determine whether or not this is possible without affecting the behavior of the program. Let's first consider removing the ALLOCATE instruction. If this instruction is removed, that means that we will not be saving space on the stack for local variable a.

However, if we take a closer look at the 3 lines of code highlighted in yellow, we see that the actual value of a is first computed in R1 before storing it in local variable a. Since R1 is going to be saved on the stack during each recursive call, we could get away without saving a on the stack because we can find its value in the stored R1 of the next activation record as shown in the highlighted pairs of a and R1 on the stack. This means that we could safely remove the ALLOCATE instruction. As a result, this also means that we don't need the ST operation that stores a on the stack or the LD operation that reloads a into register R1. Finally, because there are no longer any local variables stored on the stack, then the instruction MOVE(BP,SP) which is normally used to deallocate all local variables, can be skipped because after popping R1, the BP and SP registers will already point to the same location. So, in conclusion the 4 operations can be removed from the program without changing the behavior of the code.