# MITOCW | MIT6_004S17_11-02-06-01_300k

In this problem, we will examine how compilers translate high level language descriptions into assembly language. We will be given several code fragments and asked to help the compiler in figuring out the dependencies of the program so that it produces valid code. Let's begin with the code fragment: a = b + 3*c. We can assume that our variables: a, b, and c are stored in memory. We can also assume that registers may be used to store intermediate results. Given the following partially completed assembly code, let's determine the missing values that the compiler would have had to determine.

We begin with XXX which is the first instruction. The first instruction is trying to put the value of c into register R1. Since c comes from memory, that means that instruction XXX must be a LD operation where c is the address of the variable to be loaded.

Note that LD(c, R1) is actually a macro instruction that is equal to LD(R31, c, R1).

The load operation would add the constant c to the value of register R31, which is always 0, thus ending up with the address c as the source address of the load operation.

R1 is a temporary register that will hold the value of variable c.

Next, we need to multiply c by 3. Multiply operations are generally very expensive, so it is the compilers job to figure out that this operation could potentially be achieved by 2 simpler and faster operations. The comment tells us that it first tries to compute 2 * c and store that result into R0. Since R1 = c, and the constant in this operation is a 1, we need to realize that the inexpensive operation that the compiler would use for this is a logical shift to the left by one position.

In binary, this produces the same result as multiplying a number by 2.

So YYY = SHLC. Note that we use the constant version of the SHL operation since the amount to shift is given by a constant in our instruction rather than being read from another register. The next instruction is provided for us and it adds R0 which equals 2*c to R1 which equals c in order to produce 3*c.

This intermediate result is then stored back into R0.

Next we want to once again get the value of a variable from memory.

As we saw before, XXX = LD in order to load the contents of address b into register R1.

We are almost done, we now just need to add R1 = b to R0 = 3*c and then store the result back into memory variable a. Since the store instruction is using R0 as its source, that means that ZZZ must also be R0 so that the correct value ends up in variable a. Next, we will take a look at how a conditional statement would be compiled into assembly language.

The statement says that if a is greater than b then c should be assigned the value 17.

Once again we are given the semi-complete translation of the high level language code into beta assembly. For this example, we first load the values of our variables a and b into temporary registers R0 and R1.

Now we want to check if a is greater than b and if so set c = 17.

We know that XXX must be some kind of beta comparison operation.

However, the beta does not provide a compare greater than operation, so instead we need to make use of the compare less than (CMPLT) or compare less than or equal (CMPLE) operations.

Since we see that the store into label c is skipped when the code branches to label _L2, we want to make sure that the branch is not taken when a is greater than b.

This is equivalent to the branch being taken when a is less than or equal to b.

So if we make XXX = CMPLE of R0, which equals a, and R1, which equals b, then the result stored into R0 will be 1 if a <= b. We then set YYY to R0 to ensure that we take the branch when a b.

Finally, if we set ZZZ = 17, then when the branch is not taken, we will move 17 into R0 and then store that value into the location pointed to by address c.

So the complete translation of this conditional statement to beta assembly is shown here.

For this next code segment, we are going to take a look at how a compiler would convert array accesses into beta code. Once again we are given partially completed assembly code to help us understand how the compiler translates this high level code into beta assembly. We begin with a load of the value stored at location i into register R0. I is the index into our array.

However, since the beta is byte addressed, but it deals with 32 bit values, that means that each array element requires 4 bytes of storage.

So in order to point to the correct location in memory, we need to multiply i by 4.

As we saw earlier, shifting to the left by 1 bit is equivalent to multiplying by 2, so here we shift to the left by 2 bits in order to multiply by 4.

So XXX = 2. Now that R0 = 4 * i, in order to load a[i], we would load the location a + 4*i. In order to load a[i-1], we need to load the location that is 4 bytes before that, so location a + 4*i – 4.

This means that in this load operation which actually wants to load a[i-1], we need to set YYY = a-4. So this load operation places array element a[i-1] into R1. Now we want to store the contents of R1 into array element a[i] which is located at address a + 4*i.

Since R0 already equals 4*i, then adding a to R0 will give us the desired destination address of our store. This means that we just need to set ZZZ to R1 since that is the value that we want to store into a[i].

Let's take a look at one last example. Here we have a variable sum that is initialized to 0, followed by a loop that increments the value of sum by i for every value of i between 0 and 9. Our partial mostly completed compiled code is shown here. The first thing that the compiler does, is it initializes the two variables sum and i to 0.

This is done by storing the value of register R31, which is always 0 in the beta, into the locations pointed to by sum and by i. _L7 is a label that indicates the beginning of our loop. The first thing that needs to happen in the loop is to load the current values of sum and i from memory.

Next, sum should be incremented by i. Since R0 is stored back into sum, we want XXX = R0 to be the destination register of the ADD.

Now the loop index needs to be incremented. Since R1 = i, that means that we want to increment R1 by 1,so YYY = R1. Finally, we need to determine whether the loop needs to be repeated or we are done. This is done by checking whether i is less than 10. The beta provides the CMPLTC operation to do just that. Since R1 holds the latest value of i, comparing R1 to the constant 10 will produce the result we want in R0.

So ZZZ = 10. If the comparison was true, then we need to repeat the loop so we branch back to _L7. If not, we proceed to the instruction after the branch.