Video explanation of solution is provided below the problem.

For all Beta related questions, you should make use of the Beta documentation, the Beta Instruction Summary, and the Beta Diagram.

Compilers

3/12 points (ungraded)

Below you are given the partial results of hand-compiling the following C fragments into Beta assembly language. Please fill in the correct values for XXX, YYY and ZZZ for each of the code fragments. Make sure to only use valid beta instructions or macros. Also, do not include any spaces in your responses.

You can assume that the necessary storage allocation for each variable or array has been done, and that a label has been defined that indicates the first storage location for that variable or array. All of the variables are stored in main memory (in the first 32k bytes of main memory so that they can be addressed by a 16-bit literal). You can also assume that all varaibles and arrays are C integers, i.e., 32-bit values.

```
a = b + 3*c;
       XXX(c,R1)
                           // Make R0 = 2*c
       YYY(R1,1,R0)
       ADD(R0,R1,R0)
       XXX(b,R1)
       ADD(R1,R0,ZZZ)
       ST(R0,a)
XXX =
                     X Answer: LD
YYY =
                     X Answer: SHLC
ZZZ =
```

X Answer: R0

Explanation

In order to perform the C statement [a = b + 3*c], you begin by loading the constant c into a register, say R1, so XXX = LD. Next you need to produce 3*c. In order to do that without a multiply operation, you can shift left (YYY = SHLC) by one position the contents of R1 in order to produce 2*c. That intermediate result is stored into another register, R0 in this case. Next you need to add 2*c + c in order to arrive at the value of 3*c. This can be done by adding the contents of registers R0 and R1 together. That result can then be stored back into R0 since the previous value of R0 is no longer needed. Next, we LD b into R1 and ADD the latest contents of R1 and R0 together. Since R1 = b and R0 = 3*c, adding the two together will produce the desired result of a. Since the final instruction performs a store (ST) of register R0 into address a, that means that ZZZ = R0 in order to place the result in the register that is used for the store operation.

The resulting compiled code is shown below:

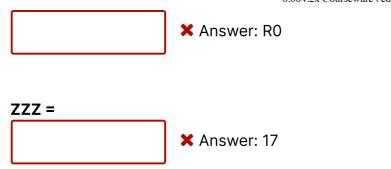
```
LD(c,R1)
SHLC(R1,1,R0)
ADD(R0,R1,R0)
LD(b,R1)
ADD(R1,R0,R0)
ST(R0,a)
```

```
if (a > b) c = 17;
```

```
LD(a,R0)
     LD(b,R1)
     XXX(R0,R1,R0)
     BT(YYY,_L2)
     CMOVE(ZZZ,R0)
     ST(R0,c)
_L2:
```

```
XXX =
                   X Answer: CMPLE
```

YYY =



Explanation

In order to perform the C statement [if (a > b) c = 17], you begin by loading the constant a into R0 and b into R1. You then want to check if a is greater than b, however, the beta does not provide a compare greater than operation, so instead we need to make use of the compare less than (CMPLTC) or compare less than or equal (CMPLE) operations. Since we see that the store into label c operation is skipped when the code branches to label _L2, we want to make sure that occurs when a is less than or equal to b, so the missing XXX operation is CMPLEC. This instruction sets R0 to 1 if the comparison returns true and 0 otherwise. The BT operation, then branches to L2 if the register YYY is 1. This means that we want YYY = R0, since that is the register that the comparison stored its result into. Finally, when the assignment statement is executed, we want to store the value 17 into c, so ZZZ = 17.

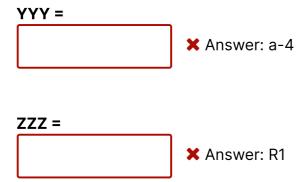
The resulting compiled code is shown below:

```
LD(a,R0)
     LD(b,R1)
     CMPLE(R0,R1,R0)
     BT(R0, L2)
     CMOVE(17,R0)
     ST(R0,c)
_L2:
```

```
a[i] = a[i-1];
```

```
LD(i,R0)
SHLC(R0,XXX,R0)
LD(R0, YYY, R1)
ST(ZZZ,a,R0)
```

```
XXX =
                     X Answer: 2
```



Explanation

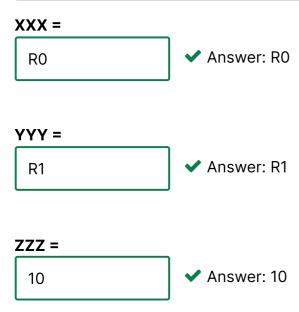
In order to perform the C statement a[i] = a[i-1]; , you begin by loading the constant i into RO. You then want to generate the offset of element a[i] from the label a. Since each data element is 32 bits wide, it uses up 4 bytes, so consecutive array locations are 4 bytes apart in memory. In order to multiply the index i by 4, we can shift it to the left by 2 locations so XXX = 2. So R0 now holds the value 4*i. In order to load the value of a[i-1], we want to load the value located at a + 4*i - 4. Since R0 currently contains 4*i, that means that YYY = a -4 in order to produce the address of element a[i-1]. So after performing the LD, R1 = a[i-1]. We now want to store the contents of R1 into memory location a + 4*i. Since R0 still holds 4*i, adding a to R0 will produce the address of element a[i]. When then want to store the contents of R1 into a[i], so ZZZ = R1.

The resulting compiled code is shown below:

```
LD(i,R0)
SHLC(R0,2,R0)
LD(R0,a-4,R1)
ST(R1,a,R0)
```

```
4.
           sum = 0;
           for(i=0;i<10;i=i+1) sum += i;
```

```
ST(R31, sum)
     ST(R31,i)
_L7:
     LD(sum, R0)
     LD(i,R1)
     ADD(R0,R1,XXX)
     ST(R0, sum)
     ADDC(YYY,1,R1)
     ST(R1,i)
     CMPLTC(R1,ZZZ,R0)
     BT(R0, L7)
```



Explanation

In order to perform the C statements | sum = 0;

for (i=0; i<10; i=i+1) sum += i;, you begin by initializing sum and i to 0. This is done by storing the contents of R31, which is always 0, into the locations pointed to by sum and i. Then we enter the loop. In the loop, the current value of sum is first loaded into R0, and the current value of i is loaded into R1. Next we add R0 and R1 in order to produce the result of sum + i. This result then needs to be stored back into sum. Since R0 is the register that holds the latest value of sum, XXX = R0. Next, we want to increment i by 1. Since R1 is the register that holds i, YYY = R1. The updated value of i is stored into location i. Then we check whether or not the loop should be repeated. This is done by comparing R1 = i to ZZZ, so ZZZ = 10. If R1 is less than 10, then R0 is set to 1, and the branch on true, BT, instruction branches back to the beginning of the loop. If R1 is not less than 10, then we are done.

The resulting compiled code is shown below:

```
ST(R31, sum)
     ST(R31,i)
_L7:
     LD(sum, R0)
     LD(i,R1)
     ADD(R0,R1,R0)
     ST(R0, sum)
     ADDC(R1,1,R1)
     ST(R1,i)
     CMPLTC(R1, 10, R0)
     BT(R0, L7)
```

Submit

1 Answers are displayed within the problem

Compilers

Start of transcript. Skip to the end.



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.



Video

▲ Download video file

Transcripts

- ▲ Download Text (.txt) file

Discussion

Hide Discussion

Topic: 11. Compilers / WE11.1

Add a Post

