// Dion Niazi dn3gy 28 03 2017 inlab8.pdf

Parameter Passing

I created a function by the name of "addSquares()" to examine for this report. This simple function takes in two integer parameters and sums the squares of each parameter and returns that value as in integer. The function's layout is shown below:

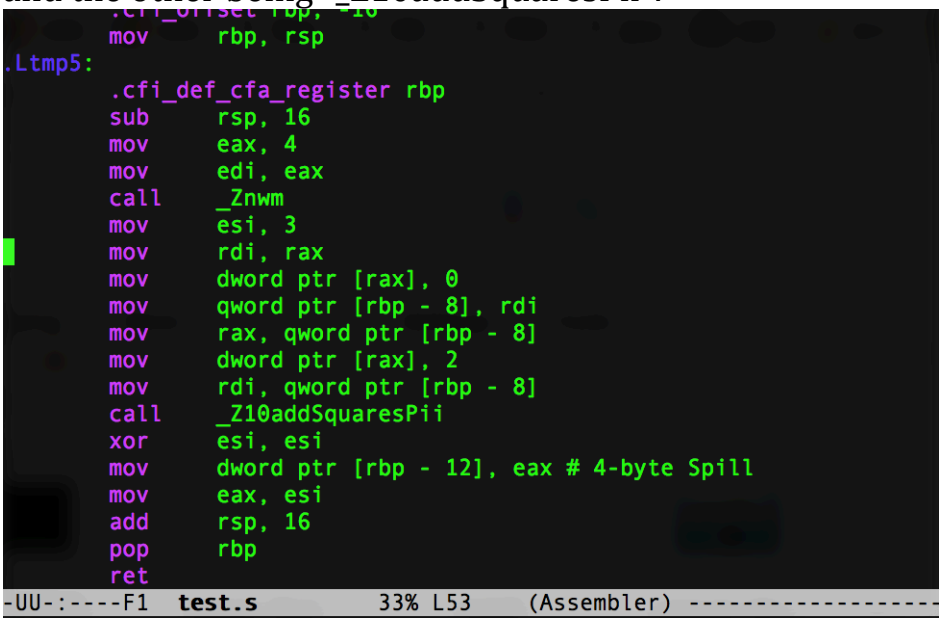
```
int addSquares(int a, int b)
{
  return a*a + b*b;
}
```

Once I placed a function call in the main function and complied the program with the appropriate flags to produce assembly code, I opened the file to see what results were produced. In the assembly file, test.s, the name of the subroutine seems to be really different. The subroutine call name was "_Z10addSquaresii", but the subroutine's name was "addSquares". The first part of the subroutine's name seems to random gibberish or rather maybe some hash value. The middle of the name consists of the original name used earlier. The last part of the name, "ii", represents the parameters of the function and what their types are. The "ii" says that the function has two integer parameters passed in by value. In the main function the first parameter seems to be stored in register edi and the second parameter seems to be stored in register esi. Similar thing happens when the parameters are replaced with different data types, just that the name is slightly modified (the last part) to accommodate the different parameters passed in by value.

When one of the function's parameter was changed to passing by reference (specifically the first parameter), the assembly code had some slight adjustments. The function's name was relatively the same, except for the fact that the function name this time consisted of an "R" in it, making the name become "_Z10addSquaresRii". I believe the "R" in it means passing by reference and one of the main instructions has changed from mov to lea and this would be because of the moving of memory and changing it right then and there, which is what passing by reference does. When the other parameter is passed by reference, the "R" in the name moves to the parameter that passes a value by reference, but just before it. So from the above example when the first parameter is passed in by reference the name is "_Z10addSquaresRii", the "R" placed right before the first parameter type and when the second parameter is passed in by reference the name is "_Z10addSquaresiRi", the "R" placed right before the second parameter type but after the first. When int

was passed by reference, the assembly of the caller was different to that of just passing in int because the memory location is being passed in and assembly needs to accommodate for this change. The callee however remained unchanged because it just needs information that the function passes in.

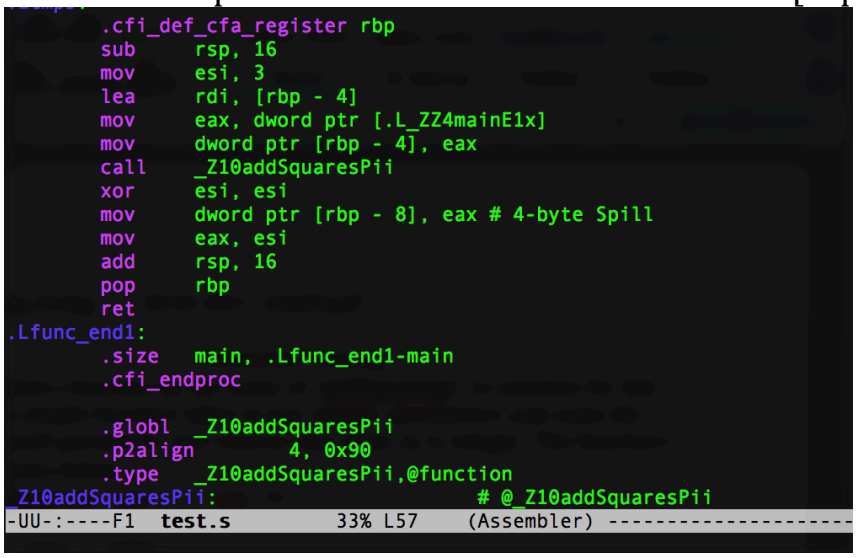
When the first parameter was changed to “int addSquares(int* a, int b)”, changes in assembly had occurred. The caller function (int main()) had two subroutine calls instead of one, one with a really random name being “_Znwm” and the other being “_Z10addSquaresPii”.



The second subroutine call was the original function that contained a “P” in it saying the function has a pointer passed in one of its parameters.

When passing in an int[] in the subroutine, the name of the function remains the same as a pointer being passed in. That is because an array is really a pointer itself. Also the caller function has only to call to just the subroutine and does lea to load up the values from the parameter’s pointer, which is where the base pointer is pointing to in the stack minus the element

to be accessed and for this example it was the first element so it was [rbp – 4], to the rdi register.



Passing in floats is the same process as passing in an int, but has a couple more processes to occur to make it a clean representative to use throughout the function. Floats are complex, so the assembly code has to compensate to make floats simpler. Passing in char is similar to an int, but since memory is smaller its uses one byte to hold memory. Passing in an object is the same as passing an int, except the memory being held depends on what the object contains.