Post-Lab 8

Parameter Passing Part 1

1. Int – By value: I just created a simple c++ function that set variables x and y and called a function that then returned them (y=5, z=6). Each is offset by four and uses DWORD, since we are using an int. I observed the following from this, the caller sets up variables (loads into the registers),:

```
int test(type x, type y){
  return x;
  return y;
}

int main(){
  type y=;
  type z=;
  test(y,z);
}
```

   - Mov DWORD PTR [EBP -4], 0
   - Mov DWORD PTR [EBP -8], 5
   - Mov DWORD PTR [EBP-12], 6
     i. It then moves them into the correct registers to call the callee…

   The Callee performs functions using these set variables (int test):
   - Mov EAX, DWORD PTR [ESP +16]
   - Mov ECX, DWORD PTR [ESP +12]
   - Mov DWORD PTR [ESP+4], ECX
   - Mov EAX, DWORD PTR [ESP +4]
     i. It then continues moving things into eax and returns the result
2. Int – Pass by reference: Caller is different since you are now passing the memory location, not the actual value; callee stays the same because the caller will still pass the correct information to the callee. New Caller:
   - Lea EAX, DWORD PTR [EBP -8]
   - Lea ECX, DWORD PTR [EBP -12]
     i. These lines load the effective addresses.
     ii. The rest is the same
3. Char- By value: I used the same format as above but replaced int with chars (y='y', z='z') caller:
   - Mov DWORD PTR [EBP -4], 0
   - Mov BYTE PTR [EBP -5], 121
   - Mov BYTE PTR [EBP-6], 122
     i. This makes sense, since an ascii char takes up a single byte, so it would use BYTE instead of the DWORD, also it only needs one byte between the local variables. I am assuming the 121 and 122 are the ascii values.

   Callee:
   - Mov AL, BYTE PTR [ESP +10] //only uses part of the register
   - Mov CL, BYTE PTR [ESP +6]
   - Then moves cl into esp +1 and al into esp, and moves the correct information into eax and then returns the values.
4. Char- By reference:

1: http://en.wikipedia.org/wiki/X86_instruction_listings
2: http://x86.renejeschke.de/html/file_module_x86_id_59.html

- Changing char to reference causes the same changes that occurred when using an int, the addition of lea to make sense of the addresses that are being passed.

5. Pointer by value: I changed my code to pass pointers into the test function. The results in the caller were:
   - lea EAX, DWORD PTR [EBP -12]
   - lea ECX, DWORD PTR [EBP -8]
     i. These were the only two additional lines that differed from the caller when using ints, this is expected since it just needs to handle the memory addresses that the pointer passes.

```
int test(int* x, int* y){
  return* x;
  return* y;
}

int main(){
  int x=2;
  int y=3;
  int* xptr=&x;
  int* yptr=&y;
  test(xptr,yptr);
}
```

   Callee:
   - Nothing changed here
6. Floats by value: Changing the variables to floats (x=10.5 and y=11.7) really started to alter the look of the assembly. Notes on both caller and callee:
   - It now used the *movss* command as opposed to the mov command used for all of the others, which I discovered is a special move command for floating point numbers.[1] It still allocated 4 bytes for each number, and beyond the change in the mov command, it was fairly similar.
7. Floats by reference: Passing a float by reference seems to be fairly similar to an int by reference. The major difference is the addition of the lea in the caller and the additional move commands that are required to move the values into the correct locations for the callee.
8. Object: When passing an object, the implementation of the object alters the resulting structure of the assembly code. For example, it will perform similarly if you pass an object that holds two ints.

Part 2

For the array portion, I decided to create an array that would store three ints and add all of the elements together using a loop (diagram 1). The organization of the assembly code seemed a little strange, but made sense eventually. The caller sets up values to be used by the callee in a slightly different way. It appears to have a special section offset for holding the values in the array (diagram 2). The caller then populates these values into eax temporarily and then the correct offset of ebp. For example:

D1

```
int test(int arr[]){
  int x=0;
  for (int i=0;i<3;i++){
    x+= arr[i];
  }
  return x;
}

int main(){
  int arr1[3]={1, 2, 3};
  int ret=test(arr1);
}
```

D2

```
.L_ZZ4mainE4arr1:
        .long   1                       # 0x1
        .long   2                       # 0x2
        .long   3                       # 0x3
        .size   .L_ZZ4mainE4arr1, 12
```

- Mov EAX, .L_ZZ4mainE4arr1
- Mov DWORD PTR [EBP-16], EAX

1: http://en.wikipedia.org/wiki/X86_instruction_listings
2: http://x86.renejeschke.de/html/file_module_x86_id_59.html

- Mov EAX, .L_ZZ4mainE4arr1+4
- Mov DWORD PTR [EBP-12], EAX

The caller then moves the first address of the array [EBP-16] into EAX and then moves EAX into the address of ESP to prepare to call test function.

The Callee first moves the first element of the array [ESP+16] into EAX, which appears to hold the sum and sets the counter to 0 [ESP]. It then compares the counter to 3 (the loop size) and jumps to the exit using the jge command or continues to the loop. It then gets the value of the first element by adding 8 to ESP and moves that into EXC and then gets the value of the next element by multiplying the counter, which has been moved into EAX by 4 and adding it to ECX. It then puts the added value into ECX. It then increments and continues until the end.

```
_Z4testPi:                              # @_
# BB#0:
        sub     ESP, 12
        mov     EAX, DWORD PTR [ESP + 16]
        mov     DWORD PTR [ESP + 8], EAX
        mov     DWORD PTR [ESP + 4], 0
        mov     DWORD PTR [ESP], 0
.LBB0_1:                                # =>
        cmp     DWORD PTR [ESP], 3
        jge     .LBB0_4
# BB#2:                                 #
        mov     EAX, DWORD PTR [ESP]
        mov     ECX, DWORD PTR [ESP + 8]
        mov     EAX, DWORD PTR [ECX + 4*EAX]
        mov     ECX, DWORD PTR [ESP + 4]
        add     ECX, EAX
        mov     DWORD PTR [ESP + 4], ECX
# BB#3:                                 #
        mov     EAX, DWORD PTR [ESP]
        add     EAX, 1
        mov     DWORD PTR [ESP], EAX
        jmp     .LBB0_1
.LBB0_4:
        mov     EAX, DWORD PTR [ESP + 4]
        add     ESP, 12
        ret
```

Part 3

The assembly code for passing by reference and passing a pointer is exactly the same. I would expect this given the similar characteristics that pointers and references share. Below I have included the code for the pointer on the right and reference on the left. As you can see they are the same:

```
_Z4testRiS_:                            # @
# BB#0:
        sub     ESP, 8
        mov     EAX, DWORD PTR [ESP + 16]
        mov     ECX, DWORD PTR [ESP + 12]
        mov     DWORD PTR [ESP + 4], ECX
        mov     DWORD PTR [ESP], EAX
        mov     EAX, DWORD PTR [ESP + 4]
        mov     EAX, DWORD PTR [EAX]
        add     ESP, 8
        ret
```

```
_Z4testPiS_:                            # @_
# BB#0:
        sub     ESP, 8
        mov     EAX, DWORD PTR [ESP + 16]
        mov     ECX, DWORD PTR [ESP + 12]
        mov     DWORD PTR [ESP + 4], ECX
        mov     DWORD PTR [ESP], EAX
        mov     EAX, DWORD PTR [ESP + 4]
        mov     EAX, DWORD PTR [EAX]
        add     ESP, 8
        ret
```

Objects Part 1

In order to test how objects function in assembly, I designed a small class that included one method and an object that held an int pointer, float, int, and char. The first thing it does is decrement the ESP by 40 to make room for the variables.

1: http://en.wikipedia.org/wiki/X86_instruction_listings
2: http://x86.renejeschke.de/html/file_module_x86_id_59.html

Then it places the address of [EBP-24] into the EAX register for future use (it appears that everything above 24 is used for object storage and everything below is used for temporary parameters). It then sets up the parameters with move commands of -4, -28, and -32. It uses the command *cvtsi2ss*, which converts that to 32 bit memory location and places it in an XMM register.[2] It then sets the values of int i and float f before calling the geti function. Of note is the fact that the float is located at EBP-12 and the int is located at EBP-24. It then moves the value of EAX, which is the memory address before the object storage starts [EBP-24] into the memory value of ESP. This appears to remove the unused space (for the variables that I didn't use in the main() call) and cap the memory of ESP at 24. The geti call is fairly straightforward and changes nothing that was described before. One thing I did notice is that the program seems to use many more steps than it needs to use to accomplish its task. Moving so many variables into temporary registers only to be moved again in the next line seems to be very redundant, but I assume that optimization would solve this problem. After further review it does appear as though the lea EAX, DWORD PTR [EBP-24] clarifies where the data members are stored, so they can be easily accessed later in the program.

```
class obj{
public:
  int i;
  char c;
  int* i_ptr;
  float f;
  int geti();
};

int obj::geti(){
  return i;
}

int main(){
  obj o1;
  int tempi=5;
  float tempf=23;
  o1.i=tempi;
  o1.f=tempf;
  o1.geti();
}
```

```
main:                                   # @main
# BB#0:
        push    EBP
        mov     EBP, ESP
        sub     ESP, 40
        lea     EAX, DWORD PTR [EBP - 24]
        mov     ECX, 23
        cvtsi2ss        XMM0, ECX
        mov     DWORD PTR [EBP - 4], 0
        mov     DWORD PTR [EBP - 28], 5
        movss   DWORD PTR [EBP - 32], XMM0
        mov     ECX, DWORD PTR [EBP - 28]
        mov     DWORD PTR [EBP - 24], ECX
        movss   XMM0, DWORD PTR [EBP - 32]
        movss   DWORD PTR [EBP - 12], XMM0
        mov     DWORD PTR [ESP], EAX
        call    _ZN3obj4getiEv
        mov     ECX, DWORD PTR [EBP - 4]
        mov     DWORD PTR [EBP - 36], EAX # 4-byte Spill
        mov     EAX, ECX
        add     ESP, 40
        pop     EBP
        ret
.Ltmp1:
        .size   main, .Ltmp1-main
```

When I compared a public and a private field, I noticed that not much changed. However, in order to access the private field, the assembler had to use the *get* function, which required two additional calls, as opposed to the public field. This is most likely due to the nature of a private field, since you have to use getters to access them.

1: http://en.wikipedia.org/wiki/X86_instruction_listings
2: http://x86.renejeschke.de/html/file_module_x86_id_59.html

Data members can only be accessed from inside this class or a friend class. When accessing from inside, the *this->* pointer command is required. This translates to a call. For example in my code, *o1.i* leads to a call to objc1, which is copied below. As described earlier, the *this* command is not placed directly on the stack, but is instead copied as a memory address, which can have values offset from it. For example, o1 would have its own memory address and would have values offset from it as its components.

```
class obj{
public:
  int i=5;
  int getd();

private:
  int d=10;
};

int obj::getd(){
  return this->d;
}


int main(){
  obj o1;
  int x=o1.i;
  int y=o1.getd();
}
```

```
_ZN3objC1Ev:                                    #
# BB#0:
        push    EBP
        mov     EBP, ESP
        sub     ESP, 8
        mov     EAX, DWORD PTR [EBP + 8]
        mov     DWORD PTR [EBP - 4], EAX
        mov     EAX, DWORD PTR [EBP - 4]
        mov     DWORD PTR [ESP], EAX
        call    _ZN3objC2Ev
        add     ESP, 8
        pop     EBP
        ret
```

1: http://en.wikipedia.org/wiki/X86_instruction_listings
2: http://x86.renejeschke.de/html/file_module_x86_id_59.html