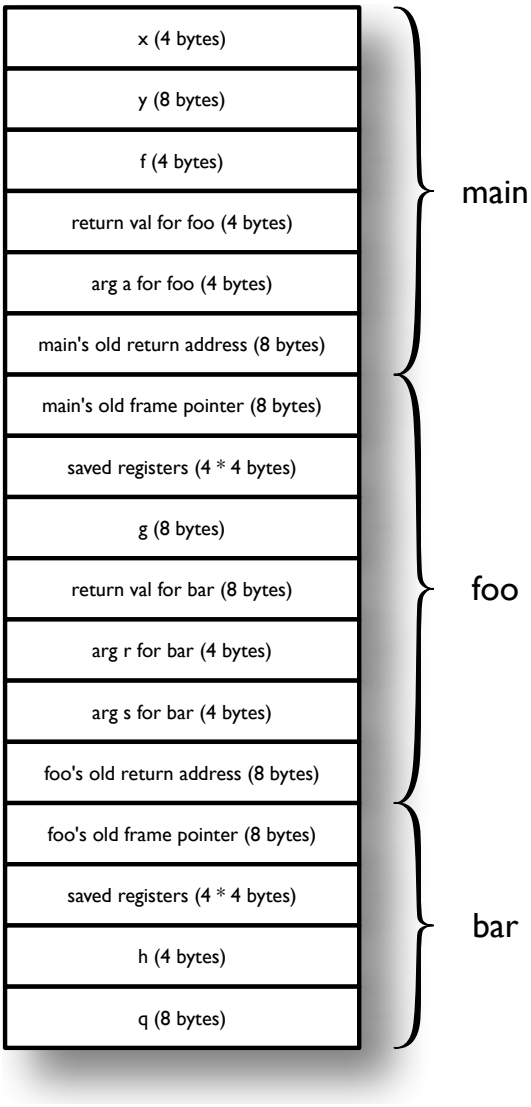
ECE 468

Problem Set 4: Function calls and Common Subexpression Elimination.

For the following problems, consider the following program:

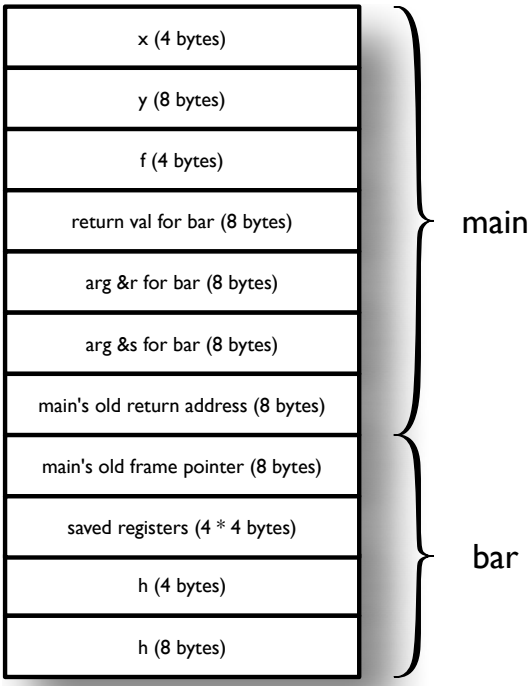
```
void main() {
   int x;
   double y;
   float f;
   ... //some computations
   f = foo(x + y);
   f += bar(x, y);
   ... //some computations
}

float foo(int a) {
   double g;
   g = bar(a, a);
   return g;
}

double bar(int r, int s) {
   float h;
   h = 1.0 * (r + s);
   if (r == s) {
     double q = 2.0;
     h = h * q;
   }
   return h;
}
```

1. Assume your program is running on a machine with 4 registers, using callee saves. Assume addresses (i.e., pointers) are 8 bytes, floats are 4 bytes, ints are 4 bytes, and doubles are 8 bytes. Draw the *complete stack* (i.e., the stack including all active activation records) for the program *right after foo has called bar, and before bar returns.* For each slot in the stack, indicate what is stored there, and how much space that slot takes up.



Answer:

2. Now assume that the call signature for bar is `double bar(int &r, int &s)` (in other words, bar is now a pass-by-reference call). Draw the call stack *right after main calls bar, and before bar returns.*



Answer:

For the following problems, consider the following piece of three-address code:

```
1.  READ(A)
2.  READ(B)
3.  C = A + B
4.  A = A + B
5.  B = C * D
6.  T1 = C * D
7.  T2 = T1 + C
8.  F = A + B
9.  C = F + B
10. G = A + B
11. T3 = F + B
12. WRITE(T3)
```

1. Show the result of performing Common Subexpression Elimination (CSE) on the above code. Rather than generating assembly, just give new 3AC. If you're reusing the results of an expression, just generate code that looks like X = Y, where Y is the variable/temporary that held the result of the original calculation.

Answer: In each row, we will show in parentheses what expressions are available *before the expression is evaluated*. We will not perform CSE yet.

```
1.  READ(A)
2.  READ(B)
3.  C = A + B
4.  A = A + B     (A + B)
5.  B = C * D     (nothing -- writing to A kills A + B)
6.  T1 = C * D    (C * D)
7.  T2 = T1 + C   (C * D)
8.  F = A + B     (T1 + C, C * D)
9.  C = F + B     (A + B, T1 + C, C * D)
10. G = A + B     (A + B, F + B)
11. T3 = F + B    (A + B, F + B)
12. WRITE(T3)     (A + B, F + B)
```

Using this information, we can perform CSE by replacing expressions with already-computed expressions that are available:

```
1.  READ(A)
2.  READ(B)
3.  C = A + B
4.  A = C         (A + B)
5.  B = C * D     (nothing -- writing to A kills A + B)
6.  T1 = B        (C * D)
7.  T2 = T1 + C   (C * D)
8.  F = A + B     (T1 + C, C * D)
9.  C = F + B     (A + B, T1 + C, C * D)
10. G = F         (A + B, F + B)
11. T3 = C        (A + B, F + B)
12. WRITE(T3)     (A + B, F + B)
```

2. Suppose F and A were aliased. How would that change the results of CSE?

   Answer: If we know that F and A are aliased, this has two effects: writing to F will kill any expression that uses A (and vice versa), and also, computing F + B is the same as computing A + B:

```
1.  READ(A)
2.  READ(B)
```

```
3.  C = A + B
4.  A = A + B     (A + B, F + B)
5.  B = C * D     (nothing -- writing to A kills A + B and F + B)
6.  T1 = C * D    (C * D)
7.  T2 = T1 + C   (C * D)
8.  F = A + B     (T1 + C, C * D)
9.  C = F + B     (T1 + C, C * D -- writing to F kills A + B)
10. G = A + B     (A + B, F + B)
11. T3 = F + B    (A + B, F + B)
12. WRITE(T3)     (A + B, F + B)
```

This results in the following CSE:

```
1.  READ(A)
2.  READ(B)
3.  C = A + B
4.  A = C         (A + B, F + B)
5.  B = C * D     (nothing -- writing to A kills A + B and F + B)
6.  T1 = B        (C * D)
7.  T2 = T1 + C   (C * D)
8.  F = A + B     (T1 + C, C * D)
9.  C = F + B     (T1 + C, C * D -- writing to F kills A + B)
10. G = C         (A + B, F + B -- note that we're using C, not F)
11. T3 = C        (A + B, F + B)
12. WRITE(T3)     (A + B, F + B)
```