Assignment 5 Solution

Brought to you by Julie Zelenski and Jerry Cain.

Problem 1: Binary numbers and bit operations

- A few examples are below. Checking your results in decimal is the easiest way to verify you've got it correct.

```
 58 = 32 + 16 + 8 + 2          00111010
+47 = 32 + 8 + 4 + 2 + 1     + 00101111
105                           01101001  = 64 + 32 + 8 + 1 = 105

100 = 64 + 32 + 4             01100100
-27 = 16 + 8 + 2 + 1         - 00011011
 73                           01001001  = 64 + 8 + 1 = 73

20 = 16 + 4                   00010100
*3 =  2 + 1                  * 00000011
60                            00010100
                             00010100
                              00111100 = 32 + 16 + 8 + 4 = 60
```

- Adding one to the maximum signed short (32767) gives the most negative short – 32768. Adding one to the maximum unsigned short (65535) returns zero. In both cases, it is doing the usual binary arithmetic and carrying the last bit into the sign bit or off the end, wrapping back into range. No error is reported in either case.

- Assigning from larger to smaller type just truncates to the lower byte(s). For example, assigning the short 1025 to a char produces 1. The other direction has no problem or truncation.

- The remainder of a number when divided by 4 can be found in the bits in the places below 4 (i.e. the bits in the 2 and 1's place). Thus we want to get the value of just those bits from the number. A bitwise AND can pull out a specific subset of bits, by AND'ing with something that has just those bits on. The integer 3 has only the 2 and 1 bits on, so if we take a number and bitwise AND it with 3, we can quickly compute its remainder when divided by 4:

  ```
  printf("Remainder of %d is %d\n", num, num & 3);
  ```

  If the upper byte of a short is all zero bits, then we can assign it to a char without data loss. Using a similar approach as above, we can extract just the high order bits from a number to see whether they are all zeros. The short with all the upper bits on is 32512 (32767-255).

  ```
  printf("High byte of %d clear? %d \n", num, (num & 32512) == 0));
  ```

There are easier and more portable ways to compute the number to AND against (using bitmask or bit shifting), but we won't get into it for now.

- To change a number's sign in two's complement, we need to invert all the bits and add one. Inverting all the bits can be done with a bitwise XOR with that number that has all bits on. This operation will turn all bits previously off on and all bits on off, exactly what we want. The value with all bits on is –1.

```
printf("Start with %d, change sign %dn", num, (num ^ -1) + 1);
```

- Here's a simple encryption/decryption program that reads a file from standard input and writes the result to standard output. It encrypts every character with a constant key of 'Z'. You could adapt it to encrypt by shorts or ints and using different keys if you felt like it. If you run this program twice on the same file, you will get the original contents back.

```
#include <stdio.h>

#define KEY 'Z'

int main(void)
{
   int ch;

   while ((ch = getc(stdin)) != EOF)
      putc(ch ^ KEY, stdout);
   return 0;
}
```

Probelm 2: ASCII and extended ASCII

As expected, the mapping for characters in the bottom half of the range is well-established, but the upper half is full of variation. For example, the ASCII value 153 is mapped to the trademark symbol on the Mac, but to the u with a circumflex over it on UNIX.

Setting up a web page or sending an email with extended ASCII characters is therefore not a reliable activity. Unless the receiver is using a system with the same character mappings, there is no guarantee your intention is properly interpreted.

Problem 3: It's just bits and bytes

This is from an **`epic`**, results can differ due to machine endianness or instruction encoding:

- The string `"hi!"` as an integer: `1751720192`, as an instruction: `call 0xa1a59c00`.
- The integer `3` as float: `4.20389539e-45`, which is definitely not `3.0`.
- ```
  struct binky b;
  printf("Printing 4-byte struct as float %f\n", *(float *)&b);
  ```
- ```
  int i;
  char *s = "Hi there!";
  i = *(int *)s;
  ```
- ```
  short s = 1;          // s will have 1 in LSB, 0 in MSB
  char ch = *(char *)&s;     // read out first byte of short
  printf("Endian is %s.\n". (ch == 0 ? "big" : "little"));
  ```
- ```
  float f = 3.14159;
  ```
- printf("Value %d\n", f); // printf will use unconverted bits

Problem 4: Identical Outputs

As we've stressed in lecture, type information is used at compile-time to make decisions about how many bytes of data to load/store and calculating offsets and the like, but all that is left at runtime is a sequence of instructions and data laid out in memory for it to operate on. Looking at the generated code tells you little about what a given 4 bytes is: an **`int`**? a **`struct fraction *`**? a sequence of **`4`** characters? It is possible for many different C functions to compile to the same output— the exact same sequence of CPU instructions is appropriate if the functions access the same memory locations in the same pattern. For example, look at this set of machine instructions:

```
R2 = Mem[SP]
R3 = R2 + 4
R4 = Mem[SP + 4]
Mem[R4] = R3
```

a)

```
void PtrOnly(void) {
   int **b, *a;
   *b = &a[1];    // or equivalently, *b = a + 1
}
```

b)

```
void IntOnly(void) {
   int arr[2];
   *(int *)(arr[1]) = arr[0] + 4;
}
```

Problem 5: Find the linked list

The basic strategy is to scan the heap from bottom to top, at each location try to interpret the bytes as though they were the head node of such a list. We check what would be the data field and if it matches what we want, we access what would be the next field, verify that it points into a valid area of the heap and then dereference and see if what follows matches the rest of the list.

```
const void *const kHeapEnd =
   ((char *)kHeapStart + kHeapSize - sizeof(list))

bool isListOnHeap(void)
{
   const char *pos;

   for (pos = kHeapStart; pos <= kHeapEnd; pos++) {
      const struct list *current = (struct list *)position;
      if ((current->data == 1) && isInHeap(current->next)) {
         current = current->next;
         if ((current->data == 2) && isInHeap(current->next)) {
            current = current->next;
            if (current->data == 3) && (current->next == NULL))
               return true;
         }
      }
   }

   return false;
}

static bool isInHeap(const void *ptr)              // Helper function
{
   return (ptr >= kHeapStart && ptr <= kHeapEnd);
}
```

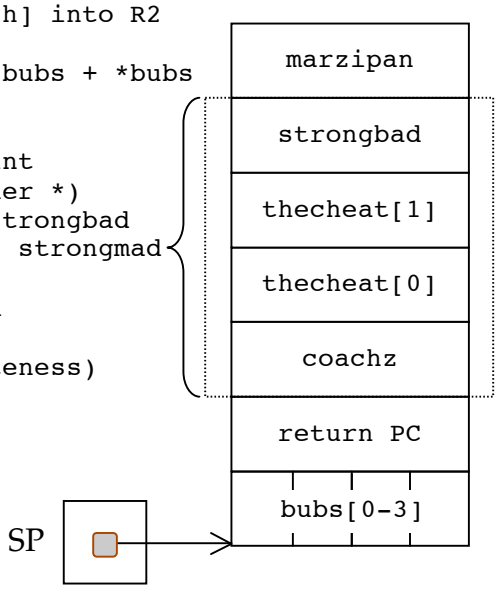
Problem 6: Homestar Runner: The System Is Down

a) Consider the following record definition:

```
typedef struct {
   int coachz;
   short *thecheat[2];
   homestarrunner **strongbad;
} homestarrunner;

void pompom(homestarrunner strongmad, homestarrunner *marzipan)
{
line 1    char bubs[4];
line 2    bubs[*bubs] = *(marzipan->thecheat[strongmad.coachz]);
line 3    ((homestarrunner *)(strongmad.thecheat))->strongbad += *(int *)bubs;
}
```

Generate code for the entire **pompom** function.

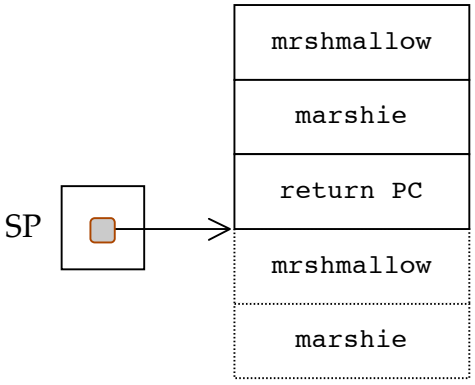
```
// line 1
SP = SP – 4;       // make space for 4 characters on the stack

// line 2
R1 =.1 M[SP];      // load *bubs
R1 = SP + R1;      // prepare bubs + *bubs

R2 = M[SP + 24];   // load marzipan
R2 = R2 + 4;       // advance R2 to address thecheat[0] instead of coachz
R3 = M[SP + 8];    // load strongmad.coachz
R3 = R3 * 4;       // scale offset by sizeof(short *)
R2 = R2 + R3;      // advance R2 to address thecheat[strongmad.coach]
R2 = M[R2];        // pull thecheat[strongmad.coach] into R2
R2 =.2 M[R2];      // pull short
M[R1] =.1 R2;      // assign short to one byte at bubs + *bubs

// line 3
R1 = M[SP];        // load bubs[0-3] as a single int
R1 = R1 * 4;       // scale by sizeof(homestarrunner *)
R2 = SP + 24;      // prepare address of pretend strongbad
R3 = M[R2];        // load old strongbad value
R3 = R3 + R1;      // do pointer arithmetic
M[R2] = R3;        // flush new value to strongbad

// clean up (not require, though here for completeness)
SP = SP + 4;
RETURN;
```



b) Now generate code for the **puppetthing** function. You needn't draw the stack frame out, but it can only help.

```
homestarrunner **puppetthing(homestarrunner *marshie,
                             homestarrunner& mrshmallow)
{
   return (**puppetthing(&mrshmallow, *marshie)).strongbad;
}
```

```
R1 = M[SP + 8];       // &mrshallow is address of referenced objects
R2 = M[SP + 4];       // passing *marshie by ref requires marshie be passed
SP = SP – 8;          // make space for two params of recursive call
M[SP] = R1;           // put &mrshmallow at the bottom
M[SP + 4] = R2;       // place marshie above that
CALL <puppetthing>    // call, expecting RV to be populated with a double ptr
SP = SP + 8;          // clean up params
R1 = M[RV];           // dereference RV to get homestarrunner *
RV = M[R1 + 12];      // replace RV with contents of strongbad field relative
                      // to R1 address
RET;              // answer is in RV… let's get out of here…
```



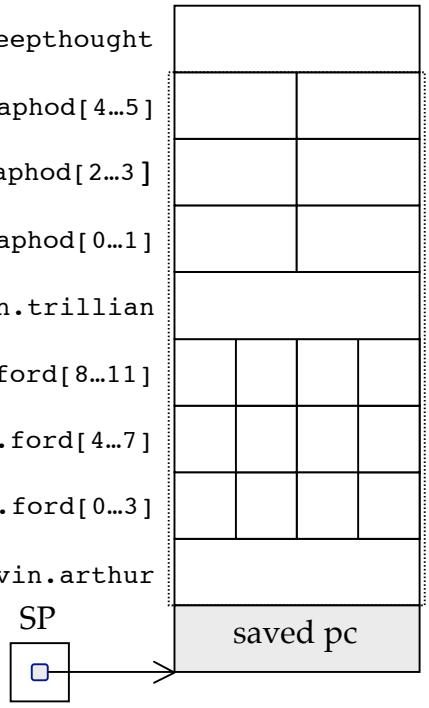
Problem 7: The Hitchhiker's Guide To The Galaxy

Consider the following type and function definition:

```
   typedef struct {
      short **arthur;
      char ford[12];
      int trillian;
      short zaphod[6];
   } galaxy;

   static galaxy *hitchhikersguide(galaxy *mice, short **dolphins);
   static short *thanksforallthefish(galaxy marvin, int *deepthought)
   {
line 1    marvin.zaphod[100] = deepthought[*marvin.ford];
line 2    ((galaxy *)((galaxy *)(marvin.zaphod))->ford)->trillian = **marvin.arthur;
line 3    return hitchhikersguide(&marvin + 1, marvin.arthur)->arthur[10];
   }
```

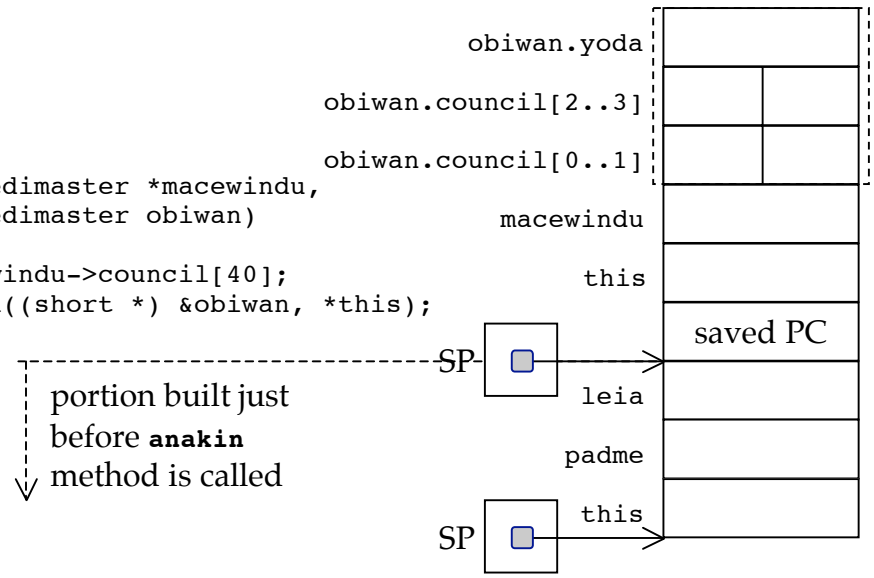
Generate code for the entire **`thanksforallthefish`** function. Be clear about what assembly code corresponds to what line.

```
// line 1
R1 = M[SP + 36];      // load deepthought
R2 =.1 M[SP + 8];     // load *marvin.ford
R3 = R2 * 4;          // scale by sizeof(int)
R4 = R1 + R3;         // compute &deepthought[*marvin.ford]
R5 = M[R4];           // load int of rhs
M[SP + 224] =.2 R5;   // flush to marvin.zaphod[100]

// line 2
R1 = M[SP + 4];       // load marvin.arthur
R2 = M[R1];           // load *marvin.arthur
R3 =.2 M[R2];         // load **marvin.arthur
M[SP + 44] = R3;      // flush to trillian field

// line 3
R1 = SP + 36;         // load &marvin + 1
R2 = M[SP + 4];       // load marvin.arthur
SP = SP – 8;          // make space for params
M[SP] = R1;           // load &marvin + 1
M[SP + 4] = R2;       // load marvin.arthur
CALL <hitchhikersguide>
SP = SP + 8;          // clean up params
R1 = M[RV];           // load arthur field of galaxy addressed by return value
RV = M[R1 + 40];      // load 10th pointer to short in array of short pointers
RET;
```



Problem 8: C++'s Dark Side

```
class jedimaster {

   public:
      int luke(jedimaster *macewindu, jedimaster obiwan);
      int& anakin(short *padme, jedimaster& leia);

   private:
      short council[4];
      short *yoda;
};

int jedimaster::luke(jedimaster *macewindu,
                     jedimaster obiwan)
{
line 1   obiwan.yoda += macewindu->council[40];
line 2   return obiwan.anakin((short *) &obiwan, *this);
}
```



This was straightforward code generation, save for the fact that you needed to understand that k-argument methods operate just like k+1-argument functions, where the address of the relevant object is spliced in as argument 0 and everything else is shifted over to make room.

```
// obiwan.yoda += macewindu->council[40];
R1 = M[SP + 20];        // load obiwan.yoda
R2 = M[SP + 8];         // load macewindu
R3 =.2 M[R2 + 80];      // load macewindu->council[40];
R4 = R3 * 2;            // scale offset by sizeof(short)
R5 = R1 + R4;           // advance old value of obiwan.yoda by scaled
offset
M[SP + 20] = R5;        // flush back new value to obiwan.yoda

// return obiwan.anakin((short *) &obiwan, *this);
R1 = SP + 12;           // load &obiwan (and look! it's a short *!, too)
R2 = M[SP + 4];         // load this
SP = SP – 12;           // make space for three params (including this!)
M[SP] = R1;             // initialize this
M[SP + 4] = R1;         // initialize padme
M[SP + 8] = R2;         // initialize leia
CALL <jedimaster::anakin>
SP = SP + 12;           // clean up parameters
RV = M[RV];             // convert reference into a copy
RET;
```