Compress

The function that we want to sketch is defined as follows. Given an input bit-vector x and a mask m, pack all the bits in x corresponding to 1's in the mask to the beginning of the word. The specification for this problem is shown below.

```
int W = 8;

bit[W] compress(bit[W] x, bit[W] m){
  int i=0;
  bit[W] out = 0;
  for(int j=0; j<W; ++j){
      if(m[j]){
        out[i] = x[j];
        i = i+1;
      }
  }
  return out;
}
```

For example, suppose you have an 8-bit vector (for clarity, we name each bit with a letter):

```
x = a b t n y p e k
```

And a mask

```
m = 1 0 1 0 0 1 1 0
```

Then, the result will be the following bit-vector:

```
out = 0 0 0 0 a t p e
```

In order to get an efficient implementation, though, we want to take advantage of the fact that we can shift all the bits in a word with a single instruction.

In order to do this, we can make use of the following function:

```
bit[W] maskedShift(bit[W] in, bit[W] mask, int s){
        bit[W] t = in & mask;
        return (in ^ t) | t >> s;
}
```

The function above shifts all the bits selected by the mask by an amount s, and leaves the remaining bits in their place. Therefore, in order to implement compress efficiently, we want to us this function to move the selected bits from their initial positions to their final position in as few iterations as possible.

The idea for the algorithm we want to implement is as follows. First, each of the selected bits has a distance that it has to travel in order to get to its final position. For example, for the bit-vector shown above

```
bit a is shifted by 4
bit t is shifted by 3
bit p is shifted by 1
bit e is shifted by 1
```

Now, if we expand each of the shift amounts into powers of two, we find that

```
bit a is shifted by 4
bit t is shifted by 2 + 1
bit p is shifted by 1
bit e is shifted by 1
```

Thus, we can call maskedShift with s = 1 to shift bits t, p and e. Then, we can call it with s=2 to shift bit t again, and then we can call it with s=4 to shift bit a, and that will get all the bits to their final position.

In fact, it doesn't matter what bits we want to select, we can shift them all to their final positions by shifting by powers of two each time.

```
x = maskedShift( x, ??, 1);
x = maskedShift( x, ??, 2);
x = maskedShift( x, ??, 4);
```

The trick, however, is knowing which bits to shift at each step.

Exercise 1.

If you know what the mask to compress is going to be, it's easy to write a sketch that selects the correct mask for each step. Try creating a sketch to implement the following function.

```
int W = 8;

bit[W] fixedMask(bit[W] x){
   bit[W] mask = {0, 1, 0, 0, 1, 0, 0, 1};
   return  compress(x, mask);
}
```

Run a few experiments with different fixed masks and different sizes of W. Make use of the repeat(??) construct to get a sketch that works for different sizes of W.

A note on array initializers:

For the initialization of the bit mask, `{0, 1, 0, 0, 1, 0, 0, 1}` is an array initializer. The initializer initializes the array so that `mask[0] = 0`, `mask[1] = 1`, `mask[2] = 0`, etc; where `mask[0]` is the least significant bit, so the least significant bit is the leftmost bit in the initializer. For some, this may be a little counterintuitive, given that the right shift `>> n` actually moves bit `i` to bit `i-n` (just like it would in C).

Now, we want to exploit the trick above for the case where we don't know the mask at compile time, so we need an algorithm to allow us to compute the masks for each of the maskedShifts from the original mask.

To understand how this can be done, consider the mask for the first maskedShift. In the first maskedShift, we want to shift by one all the bits for which the total travel distance will be odd. Now, note that the travel distance for each bit is equal to the number of zeros to the right of it. Thus, the first mask should select all the bits that have an odd number of zeros to the right of them.

To do this, the following function will come in very handy.

```
bit[W] xor_reduce(bit[W] in){
  bit[W] out = 0;
  out[0] = in[0];
  for(int i=1; i<W; ++i){
      out[i] = in[i] ^ out[i-1];
  }
  return out;
}
```

The function has the property that bit out[i] in the output will be one if the number of ones between 0 and i is odd. This is very close to what we need, except, we need to consider the number of zeros, rather than ones, and we need to count them from 0 to i-1, not from 0 to i. However, this is not hard to fix. We can get the mask we want with the following statements.

```
        mk = (!m) << 1;
        mp = xor_reduce(mk);
        mv = mp & m;
       //mv is the mask for the first shift
```

So now we know how to compute the first mask, but there is a slight problem, the implementation of xor_reduce shown above is very inefficient.

Exercise 2.

Sketch an implementation of xor_reduce that uses only 3 shifts and 3 bit-vector xors for an 8 bit word.

The masks for subsequent maskedShifts can be computed using similar ideas. For example, for the next iteration, you want to shift all the bits whose remaining travel distance is a multiple of two but not of 4; in other words, odd multiples of 2. It's easy to see that you can get such a mask by running xor_reduce on (mk & !mp). Think about it. If you run xor_reduce(m), you get all the bits that have an odd number of ones from them to the end of the bit-vector. If you run xor_reduce(m & !xor_reduce(m)), you get all the bits where the number of ones from them to the end of the bit-vector is an odd multiple of two.

Example:

m = 1 0 1 0 1 0 1 0 1
v = xor_reduce(m) = 1 0 0 1 1 0 0 1 1 ;
 // v has ones for bits with an odd number of ones at or ahead of them.
 // masking with !v zeros all of those bits, leaving only those which are

```
        // multiples of 2.
       // And then xor_reducing leaves only those which are odd multiples of 2.
  xor_reduce( m & !v) = 0 0 0 0 0 0 1 0 0
```

Exercise 3. Putting it all together.

Sketch an efficient implementation of compress that uses all the tricks that we have described so far.