Lab: SHA512

File `SHA.cry` and `SHA512.cry` contain Cryptol functions that implement the `SHA512` hash plus create a digest for a given text string. Of specific interest is the function `SHAImp msg` in `SHA.cry` which produces a digest from input text `msg`. The function is defined like this:

```
SHAImp : {n} (fin n) => [n][8] -> [digest_size]
SHAImp msg = SHAFinal (SHAUpdate SHAInit msg)
```

Run Cryptol and load `SHA512.cry` – `SHA.cry` is automatically loaded as well. An example run of this function is as follows:

```
SHA512> SHAImp "Hello"
0x3615f80c9d293ed7402687f94b22d58e529b8cc7916f8fac7fddf7fbd5af4cf777d3d795a7a00a
16bf7e7f3fb9561ee9baae480da9fe7a18769e71886b03f315
```

Notice from the signature of `SHAImp` that the output is `digest_size` bits wide and `digest_size` is defined as `SHAImp` in `SHA512.cry`. File `sha512.c` contains C code for producing a `SHA512` digest. Adding a `main` like this:

```
int main(int args, char **argv) {
   int i=0;
   uint8_t *out = (uint8_t *)malloc(SHA512_DIGEST_LENGTH);
   int len = strlen(argv[1]);
   out = SHA512(argv[1], len, out);
   printf("0x");
   for (i=0 ; i < 64 ; i++)
      if (out[i] < 16) printf("0%1x",out[i]); else printf("%2x",out[i]);
   printf("\n");
}
```

and compiling allows one to display the digest given a text string as input like this:

```
[prompt]$ sha512 "Hello"
0x3615f80c9d293ed7402687f94b22d58e529b8cc7916f8fac7fddf7fbd5af4cf777d3d795a7a00a
16bf7e7f3fb9561ee9baae480da9fe7a18769e71886b03f315
```

Observe that for the same input, the digest is the same for both `SHAImp` and `SHA512`. It is desired to verify formally that the C function for computing the digest is functionally identical to the Cryptol “gold standard”.

Observe again that `SHAImp msg = SHAFinal (SHAUpdate SHAInit msg)`. Observe also that this is similar to the C function `SHA512`:

```
uint8_t *SHA512(const uint8_t *data, size_t len,
                uint8_t out[SHA512_DIGEST_LENGTH]) {
  SHA512_CTX ctx;
  const int ok = SHA512_Init(&ctx) &&
                 SHA512_Update(&ctx, data, len) &&
                 SHA512_Final(out, &ctx);
  memset(&ctx, 0, sizeof(ctx));
  return out; }
```

where `SHA512_ctx` carries the 8 initial hash values for `SHA512`, the message digest length, and parameters `Nl`, `Nh`, and 128 byte message length. Corresponding to this is type `SHAState` in Cryptol. The plan is to show equivalence for the `Init`, `Update`, and `Final` functions then use those to show equivalence for `SHA512` and `SHAImp`. The C functions depend on `sha512_block_data_order` which corresponds to `processBlock_Common` in SHA.cry.

The Cryptol gold standard specification comes directly from the NIST/FIPS 180-4 publication which is presented as the background material for this lab. Several functions are defined in the documentation. These are defined in Cryptol and implemented in C. Start with the following functions on Page 11 of the publication.

$$\textstyle\sum_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \qquad (4.10)$$

$$\textstyle\sum_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \qquad (4.11)$$

$$\sigma_0^{\{512\}}(x) = ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \qquad (4.12)$$

$$\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x) \qquad (4.13)$$

where

$ROTR^n(x)$ is the rotate right (circular right shift) operation, where `x` is a `w`-bit word and `n` is an integer with $0 \le n < w$, and is defined by $ROTR^n(x) = (x >> n) \vee (x << w - n)$ and $SHR^n(x)$ is the right shift operation, where `x` is a `w`-bit word and `n` is an integer with $0 \le n < w$, and is defined by $SHR^n(x) = x >> n$. The Cryptol versions of these are in `SHA512.cry`:

```
SIGMA_0 x = (x >>> 28) ^ (x >>> 34) ^ (x >>> 39)
SIGMA_1 x = (x >>> 14) ^ (x >>> 18) ^ (x >>> 41)
sigma_0 x = (x >>> 1) ^ (x >>> 8) ^ (x >> 7)
sigma_1 x = (x >>> 19) ^ (x >>> 61) ^ (x >> 6)
```

The C implementation of these is the following in sha512.c:

```
static inline uint64_t Sigma0(uint64_t x) {
   return CRYPTO_rotr_u64((x), 28) ^ CRYPTO_rotr_u64((x), 34) ^
      CRYPTO_rotr_u64((x), 39);
}

static inline uint64_t Sigma1(uint64_t x) {
   return CRYPTO_rotr_u64((x), 14) ^ CRYPTO_rotr_u64((x), 18) ^
      CRYPTO_rotr_u64((x), 41);
}

static inline uint64_t sigma0(uint64_t x) {
   return CRYPTO_rotr_u64((x), 1) ^ CRYPTO_rotr_u64((x), 8) ^ ((x) >> 7);
}

static inline uint64_t sigma1(uint64_t x) {
   return CRYPTO_rotr_u64((x), 19) ^ CRYPTO_rotr_u64((x), 61) ^ ((x) >> 6);
}
```

where `CRYPTO_rotr_u64` is defined in `sha512.c`:

```
static inline uint64_t CRYPTO_rotr_u64(uint64_t value, int shift) {
   return (value >> shift) | (value << ((-shift) & 63));
}
```

Verifying equivalence of the above functions will help verify `sha512_block_data_order` against `processBlock_Common` so these are considered first.

Exercise 1:
Develop a SAW file for proving that the `sigma` and `SIGMA` functions above are equivalent to their corresponding Crypto specifications. Run `saw` on the file and report the result. ■

The next target is `sha512_block_data_order` because `SHA512` in `sha512.c`, the end goal in proving correctness for this example, depends on `SHA512_Init`, `SHA512_Update`, and `SHA512_Final`, all of which depend on `sha512_block_data_order` and the proof for `sha512_block_data_order` runs faster with the above functions proved first. The specification given in the publication is on Page 25. The Cryptol specification is `processBlock_Common` in `SHA.cry`. In `sha512.c` `sha512_block_data_order` is used in `SHA512_Init`, `SHA512_Update`, and `SHA512_Final` like this:

```
sha512_block_data_order (c->h, p, 1)
```

where, from `sha512.h`, `c->h` is an array of 8 64 but integers, `p` is an array of 128 8 bit integers and from the declaration of `SHA512_block_data_order`, 1 is of type `size_t` which is a 64 bit integer. Therefore, for the `SHA512_block_data_order` setup in SAW file, say `s2.saw`, the `llvm_execute_func` line looks like this:

```
llvm_execute_func [state_ptr, data_ptr, llvm_term {{ 1:[64]}}];
```

where `state_ptr` is part of a pair `(state,state_ptr)` obtained from

```
(state,state_ptr) <- pointer_to_fresh "state" (llvm_array 8 (llvm_int 64));
```

and `data_ptr` is part of a pair `(data,data_ptr)` obtained from

```
(data,data_ptr) <- pointer_to_fresh "data" (llvm_array 128 (llvm_int 8));
```

The state and data variables are used by the Cryptol specification in

```
llvm_points_to state_ptr
   (llvm_term {{ processBlock_Common state (split (join data)) }});
```

(see `SHA.cry` for why `(split (join date))` is used in the above). `The llvm_verify` line:

```
sha512_bdo_ov <- llvm_verify m "sha512_block_data_order"
                 [Sigma0_ov, Sigma1_ov, sigma0_ov, sigma1_ov] true
                 sha512_block_data_order_setup ...;
```

If `...` is replaced by, say `z3`, the SAW file, with all the sigma functions, declarations of pointer_to_fresh, and the completed setup function for SHA512_block_data_order, the proof will not finish, at least not in a reasonable time. For this particular case the only way to get a reasonable result is to use an uninterpreted function library. An example, which works in this case is to replace `...` with

```
(w4_unint_z3 [“SIGMA_0”,”SIGMA_1”,”sigma_0”,”sigma_1”])
```

The `w4_unint_z3` command is used to specify that some Cryptol functions should be considered uninterpreted. You will have to study `SMT` solvers, such as `Z3`, to understand precisely what that means for a solver and you likely do not want to do this, expecting that `SAW` should take care of this. Perhaps `SAW` will eventually be able to do so. For now, a rough explanation is offered. Suppose some function `f` is given and a proof of `f(x) == f(y)` is needed. Normally, `SAW` will expand `f` and pass the whole thing to the `SMT` solver. However, by calling `(w4_unint_z3 ["f"])`, then `SAW` will leave `f` uninterpreted and the solver will only be able to prove `f(x) == f(y)` if `x == y`. This works because Cryptol functions are pure and so if the arguments are equal then the return values must be equal. So, the downside of declaring functions uninterpreted is the desired proof may be missed but, if the above is enough, a fast proof will be generated. The problem is knowing which functions can be deemed uninterpreted. Here is some advice. When a SAW proof hangs there are a couple things to try first:

1. If the C code has some complexity that you think might be slowing the proof down, prove that bit separately and use it as an override.
2. If the Cryptol specification has some complexity that you think might be slowing the proof down and the same exact Cryptol function appears on both sides of the equality, then leave that function uninterpreted. It's important that the arguments for this function on both sides of the equality are themselves equal for the uninterpreted trick to work. We will call this idea "equals for equals".

For this problem, compare the sigma functions in `sha512.c` and `SHA512.cry`. These are very simple and obey "equals for equals". The next question is why were these picked for being uninterpreted? Sorry, the answer is it was tried and it worked. Other functions could have been deemed uninterpreted but they were unnecessary. By the way `(w4_unint_z3 [])` is the same as `z3`.

Exercise 2:
Verify C function `SHA512_block_data_order` is functionally equivalent to the Cryptol specification `processBlock_Common`. Write the SAW file, `s2.saw`, and run it. ■

Next consider the equivalence of the `SHA512_Update` function in C and the `SHAUpdate` specification in Cryptol. The goal is to add the following `llvm_verify` line to s2.saw which will become s3.saw:

```
update_ov <-llvm_verify m "SHA512_Update"
   [sha512_block_data_order_ov] false SHA512_Update_setup
   (w4_unint_z3 ["processBlock_Common"]);
```

The above makes use of `sha512_block_data_order_ov`, found earlier, and recognizes that `processBlock_Common` should be treated as uninterpreted. These are hints intended to save the reader time figuring these out on their own.

The next step is to complete `SHA512_Update_setup`. Start with

```
llvm_execute_func [sha_ptr, data_ptr, llvm_term {{ `128 : [64] }}];
```

These parameters correspond to the `SHA512_Update` function of `sha512.c`:

```
int SHA512_Update(SHA512_CTX *c, const void *in_data, size_t len)
```

The `sha_ptr` will be the `SHA512_CTX` struct pointer `c`. The `data_ptr` will be the `in_data` pointer. The `llvm_term` that is 128 will be the `len` parameter: block size is 128 bytes. The in_data parameter is an array of 128 bytes, which can be cast in the `SHA512_Update_setup` as

```
(data, data_ptr) <- pointer_to_fresh "data" (llvm_array 128 (llvm_int 8));
```

where `pointer_to_fresh` has been completed earlier. The return value of the `SHA512_Update` function is 1 which, as a SAW directive in the setup function, becomes this:

```
llvm_return (llvm_term {{ 1 : [32] }});
```

All that's left for the setup function is to take care of the pointer to the `SHA512_CTX` struct. This is the hard part. The following is needed for the `sha_ptr` parameter in the setup function:

```
(sha512_ctx, sha_ptr) <- pointer_to_fresh_sha512_state_st 0;
```

where the 0 is a parameter so as to be able to reuse this function for the next exercise. The pointer_to_fresh_sha512_state_st function

Exercise 3:
Create a function digest_in_bytes in `SHA256.cry` with signature

```
digest_in_bytes : {i} (fin i, 64 >= width (8*i)) => [i][8] -> [32][8]
```

That takes a message `msg` as input and outputs 32 bytes that is the digest of `msg`. Run

```
digest_in_bytes “Hello World Folks”
```

and verify the result matches the above except that the output is now a sequence of bytes. ■

Exercise 4:
The digest function of Cryptol does not require the length of the input message as input. The C function should not either. Write a C function named SHA256_Buf_Wrapper that takes, as input, the message, as a `char[]`, and the 32 byte digest array and inserts the digest of the message into the digest array. The prototype for the wrapper function is the following:

```
void SHA256_Buf_Wrapper(char *input, uint8_t digest[32]);
```

Then change `main` to look like this:

```
int main (int argc, char** argv) {
   // usage: sha256 <input>
   int i;
   uint8_t digest[32];
   SHA256_Buf_Wrapper(argv[1],digest);
   for (i=0 ; i < 32 ; i++)
      if (digest[i] < 16) printf("0%x",digest[i]);
         else printf("%x",digest[i]);
         printf("\n");
```

Run sha-256 “Hello World Folks” and verify the output is as above. ■

Exercise 5:
Following the pattern of the previous lab, construct a saw file that verifies that the C function `SHA256_Buf_Wrapper` on input `msg` produces the same output as the Cryptol function `digest_in_bytes` on `msg`. ■