ECE4530 Fall 2013: The Codesign Challenge Hash Collision Search

Assignment posted on Thursday 14 November Noon
Solutions due on Friday 6 December Noon

The Codesign Challenge is the final assignment in ECE 4530. This project is an exercise in performance optimization: you will start from a given reference application on a Nios-II processor. You have to improve the performance of the reference application as much as possible, using the hardware/software codesign techniques covered in this course. Typically, you would design a hardware coprocessor. In addition, you would also optimize the driver software, and/or modify the system architecture. There is an important constraint: the final result must execute on the DE2-115 board under a predefined performance-evaluation testbench. The testbench is written in C, so you have to keep a Nios-II processor present in the final implementation.

Application: Collision Search on SHA-1 Hash

A hash function is a one-way function which converts an arbitrary bitstring into a fixed-length *digest*. In this challenge, we will be using SHA-1, one of the best known and (still) most widely used hash functions. Refer to `http://link.springer.com/chapter/10.1007/978-3-642-04101-3_11` for a good introduction to hash functions and SHA1 in particular.

The one-wayness of SHA-1 is reflected in several properties.

- *Preimage resistance*: If you know the output or *digest* of a hash function, $D = H(x)$, it is very hard to find an x that would result in the same D.
- *Collision resistance*: For a given hash function, it is very hard to find *any* two inputs that result in the same digest D, ie. it is very hard to find an x and a y such that $H(x) = H(y)$.

Hash functions are an important component in cryptographic protocols. They are used during the computation of electronic signatures, in authentication protocols, in file integrity applications, and more. Try the following. Open your Ubuntu VM and install the openSSL tools.

```
sudo apt-get install openssl
```

Next, compute the SHA-1 digest of the word HOKIES as follows:

```
openssl dgst -sha1<press return>
HOKIES<press Ctrl-D>
```

You will see the output digest as follows:

```
openssl dgst -sha1<press return>
HOKIES(stdin)= e017d72b9ceae0544b42c2a9300c0708d015021b
```

The 160-bit word `e017...` is the digest for the string HOKIES. Now let's change one letter of the word. For example, compute the SHA-1 digest of the word GOKIES:

```
openssl dgst -sha1<press return>
GOKIES<press Ctrl-D>
```

In this case the digest is

```
openssl dgst -sha1<press return>
GOKIES(stdin)= 878e4c065d05e5a1ca87fdef81607ab097d83a65
```

As you can see, the resulting digest is completely different. This is common for hash functions: a very small change in the input will generate a large change in the digest. Furthermore, based on the input, it is practically impossible to predict the digest.

Why collisions are hard to find

A hash function maps the set of *all* (ie. an infinite number of) possible input strings to a *finite* set of 2^{160} digests. So, there are an infinite amount of collisions. Why is it so hard to find a collision? Because 2^{160} is an astronomical number! The chance of hitting any particular *chosen* digest with an arbitrary input message is very small: about 2^{-160}.

In a collision search problem, we try to do exactly that: the digest output of SHA-1 is chosen, and next we try to find an input string that will map into the selected digest. Of course, an exact SHA-1 collision is extremely hard and improbable: the probability of finding one is 2^{-160}!. So instead, we define an easier problem as follows:

Find an input string that yields a digest in which the leading n bits are zero, and the remaining (160-n) bits are don't cares.

If n is small, that is relatively easy. Indeed, suppose $n = 1$. Then, for any arbitrary input, the chance of having the first digest bit zero is 1/2. If $n = 3$, then the chance would become 1/8, since we require three leading bits to be zero (fixed). In general, the chance of creating a digest with n initial zeroes from an arbitrary input is 2^{-n}. The call this n the *target*. It is the parameter that defines the difficulty of the collision search problem. For small n, collisions are easy to find, but for large n, collisions are exponentially more complex to find.

This leads to the following collision search problem, used in the codesign challenge.

1. Pick a reference string, for example:
   `XXXX Keep your head cool and your FPGA spinning!`

2. Pick a target, for example 14 bits.
3. Replace the `XXXX` at the front of the string with a 32-bit counter value, starting from 0.
4. Compute the digest of the resulting string and count the number of leading zeroes in the digest. If the number of leading zeroes is equal to, or exceeds, 14 bits, you have found a valid collision for the target. Report the counter value used in this collision and exit.
5. Otherwise, if the number of leading zeroes does not meet the target, then increase the counter and repeat from step 3.

The previous algorithm is bound in speed by the speed at which you can compute a SHA-1 digest. Hence, this problem is very well suited for the codesign challenge: who can build the fastest SHA-1 collision search engine? You may also note it's an *embarrassingly* parallel problem: if you have multiple SHA-1 implementations, you can compute different digests at the same time, for different counter values. In the following, we first define the architecture of the reference testbench implementation, and next the reference testbench software.

Reference Architecture

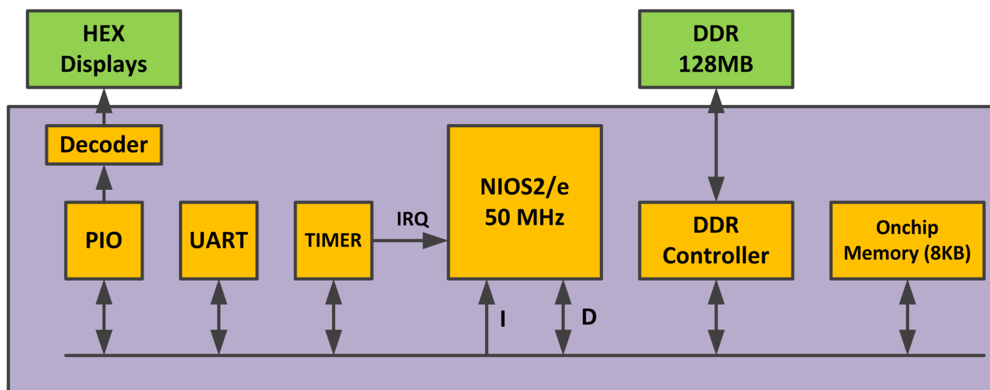

Figure 1: Reference Architecture

The reference implementation uses about 4% of the resources of the FPGA. It includes the following components.

- A NiosII/e microprocessor to execute the reference implementation in software
- A timer to measure the performance of your design. The timer interrupt is connected to the NiosII/e. The timer is used to determine the number of hashes computed per second; this metric is used to determine the ranking (see further under 'Reference Testbench'.

- A UART to print terminal output on your laptop
- A PIO to drive the 7-segment LED displays
- An 8KB on-chip memory, mapped to the instruction- and data-space of the NiosII.
- A memory controlled for an off-chip 128 MByte DDR memory

Your accelerated design may make arbitrary modifications to the design. However, the top-level testbench, explained in the next section, must remain unchanged.

Reference Testbench

Please consult the reference implementation on scholar, `challenge.qar`. Expand the archive in a directory, and consult the files in the software subdirectory. You will find three files.

- `sha1.c` and `sha1.h`: reference implementation of the SHA-1 algorithm. This function is extracted from a library called PolarSSL. This is the SHA-1 implementation used for the reference design, and the implementation used to verify collisions identified by your accelerated design.
- `cinterface.c` and `cinterface.h`: This file contains four functions that make up the interface between your collision searcher and the top-level testbench in the main function. Your implementation must support each of the four functions, which are defined below.
- `collisions.c`: the main testbench for the collision search. This file must remain unchanged in your final design.

At the highest level, your SHA-1 collision search engine must provide an implementation for the following four functions, defined in `cinterface.h` and `cinterface.c`.

- `void setsearchstring(char *s)`: This function is called by the top-level testbench to define the search string. The search string will be shorter than 48 bytes, and will always start with four 'X' characters. The collisions have to be computed over a fixed-length 48-byte search string, padded with 0x0 bytes if the argument to `setsearchstring` is a strong of less than 48 bytes.
- `void settarget(int t)`: This function is called by the top-level testbench to set the target number of bits required for a collision. For example, if the target would be 12, then you would need to find a search string that creates a digest that starts with 12 zeroes.

- `int searchcollision()`: This function performs a collision search, by repeatedly replacing the four leading 'X' characters in the search string with a counter value, and each time computing the SHA-1 digest. A collision is declared when a digest is generated with the required number of zero-bits at the start.
- `int shacomputed()`: This function returns the number of SHA-1 computed since the last call to searchcollision().

Let's take a closer look at `searchcollision()` to see exactly what happens.

```
int searchcollision() {
   currentcount     = 0;
   hascollision     = 0;
   unsigned char digest[20];
   sha1_context ctx;
   while (currentcount < ((unsigned)-1)) {
    currentsearchstring[0] = (char) (currentcount >> 24);
    currentsearchstring[1] = (char) (currentcount >> 16);
    currentsearchstring[2] = (char) (currentcount >>  8);
    currentsearchstring[3] = (char) (currentcount      );
    sha1_starts( &ctx );
    sha1_update( &ctx, currentsearchstring, 48 );
    sha1_finish( &ctx, digest );
    if (testdigest(digest))
      return currentcount;
    currentcount++;
  }
  return 0;
}
```

- A while loop repeatedly computes SHA-1 digests, until a collision is found. Note that computing a SHA-1 digest actually involves three function calls, `sha1_starts()`, `sha1_update()` and `sha1_finish()`. This is a common software interface for SHA-1 and other functions. The input to the SHA-1 function is `currentsearchstring`, and the output is `digest`. The `ctx` variable is a state variable.
- For each iteration of the while loop, a counter value `currentcount` is pasted in the first four bytes of `currentsearchstring`. This will overwrite the four 'X' characters at the start of the search string with counter bytes.
- A SHA-1 digest is 160 bits, and it is stored in a 20-byte `digest` array. The function `testdigest()` evaluates if the computed digest meets the search target. The

testdigest() function is defined in cinterface.c as well, just above searchcollision(). It performs some bit fiddling to ensure that at least n leading bits of the digest are zero, where n is defined by the main testbench through a call to settarget().

The three remaining functions, setsearchstring(), settarget(),and shacomputed(), are easy and compact. Note how setsearchstring clears a block of 48 bytes in memory, and next pastes the 48 first characters of the argument in that region of memory.

```
void setsearchstring(char *v) {
   memset(currentsearchstring, 0, 48);
   strncpy(currentsearchstring, v, 48);
}

void settarget(int n) {
   currenttarget = (n > 0) ? n : 1;
}

int shacomputed() {
   return currentcount;
}
```

Next, open collisions.c and study how the main testbench interacts with the four functions that define the colliision search.

```
int main() {
    alt_alarm alarm;
    char *secretkernel = "XXXX Keep your FPGA spinning!";

    printf("Collision string:                 %s\n",  secretkernel);
    printf("Display update interval (seconds): %4d\n", updateeach);
    printf("Sysclock ticks per second:         %4d\n", (int) alt_ticks_per_second());

    unsigned iteration = 0;
    cbcontext cb;
    do {
        iteration++;
        printf("--------- Iteration %d\n", iteration);
        printf("Target collision (bits):           %4d\n", iteration);
        settarget(iteration);
        setsearchstring(secretkernel);
```

```
        cb.prevcount = 0;
        cb.callbackcount = 0;
        alt_alarm_start(&alarm,
                        updateeach * alt_ticks_per_second(),
                        updatecallback, &cb);

        unsigned cnt;
        cnt = searchcollision(secretkernel);
        alt_alarm_stop(&alarm);
        reportcollision(secretkernel, cnt);
    } while ((cb.callbackcount < 10) && (iteration < 32));

    printf("Terminating Search\n");

    return 0;
}
```

The high-level picture of this testbench is as follows. It will repeatedly test your collision searcher at increasing difficulty, where difficulty is defined by the target. The target starts easy, at 1 (ie. a collision with a single leading zero-bit), and it increases for every iteration. The testbench will stop your collision searcher at the first target that takes more than 100 seconds to compute, *or* is your collision search can work up to a 31 bit target[1]. Depending on the quality of your design, the target may be higher (better). Eventually, what counts is the number of SHA-1 computations per second delivered by your collision searcher.

When `searchcollision()` returns, the resulting collision is printed on the terminal through `reportcollision()`. Next, a test is made how long this collision took to compute (test on `cb.callbackcount`), and, when it took less than 100 seconds, the loop is restarted. At the next loop iteration, the target is increased to the next difficulty level, and the collision search is repeated.

To report feedback during the collision search, the testbench uses an `alarm`, which is a timer interrupt from the system timer. Just before the `searchcollision()` function is called, the testbench initiates the alarm with `alt_alarm_start()`. Exactly 10 seconds later (ie., `updateeach * alt_ticks_per_second()`, where `updateeach` has the value 10), the function `updatecallback()` is called. That function simply reports the current performance of your collision searcher:

```
alt_u32 updatecallback(void *context) {
  ((cbcontextptr) context)->callbackcount++;
  printf("Count %d, SHA1 per sec %d\n",
 shacomputed(),
```

[1]This is not unlikely, given that a 31 bit counter at 50MHz runs out in 43 seconds

```
(shacomputed() - ((cbcontextptr) context)->prevcount) / updateeach);
  ((cbcontextptr) context)->prevcount = shacomputed();

  return updateeach * alt_ticks_per_second();
}
```

The callback function makes use of a structure called a *callback context*, an area of memory that is passed on between different callbacks and the main function. In our implementation, the `context` includes two elements: the number of times the callback function has been called so far (`callbackcount`) and the number of SHA-1 computed at the previous invocation of `updatecallback`. The first field is used as a time-out. It will allow the main testbench to detect when it is starting to take too long for your collision searcher to find new targets. The second field is used to estimate the number of SHA-1 computations per second. This last metric, SHA-1 computations per second, is the key performance metric of your implementation.

The reference testbench output looks as follows. Refer to the quickstart folder in the distributed code: it contains a bitstream `challenge.sof`, and an ELF binary `collisions.elf`. Open two NiosII command shell windows. In the first one, run a `nios2-terminal`. In the second one, download the bitstream and run the ELF binary

```
nios2-configure-sof challenge.sof
nios2-download collisions.elf --go
```

The terminal window will show the following output.

```
Collision string:                  XXXX Keep your FPGA spinning!
Display update interval (seconds):   10
Sysclock ticks per second:         1000
--------- Iteration 1
Target collision (bits):              1
Collision found at Counter Value 0!
Digest: 34379479 17be5513 a84f67df 4446fad0 86e314a4
--------- Iteration 2
Target collision (bits):              2
Collision found at Counter Value 0!
Digest: 34379479 17be5513 a84f67df 4446fad0 86e314a4
--------- Iteration 3
Target collision (bits):              3
Collision found at Counter Value 3!
Digest: 18bd42fe 4d6c58b8 42fb8669 86580eae 1f700131
--------- Iteration 4
Target collision (bits):              4
Collision found at Counter Value d!
```

```
Digest: 0880bab3 4390f0d0 f7c18da1 889a4f1c 54d6d45f
--------- Iteration 5
Target collision (bits):             5
Collision found at Counter Value 2a!
Digest: 023f44fc d322b26b be5ced83 c1523aa2 5101f17a
--------- Iteration 6
Target collision (bits):             6
Collision found at Counter Value 2a!
Digest: 023f44fc d322b26b be5ced83 c1523aa2 5101f17a
--------- Iteration 7
Target collision (bits):             7
Collision found at Counter Value 12b!
Digest: 01308f0d 890c0844 136bc748 610d460f 8f201067
--------- Iteration 8
Target collision (bits):             8
Collision found at Counter Value 14a!
Digest: 007d7bac 61baee27 c6b350d2 23484a20 79e0eb20
--------- Iteration 9
Target collision (bits):             9
Collision found at Counter Value 14a!
Digest: 007d7bac 61baee27 c6b350d2 23484a20 79e0eb20
--------- Iteration 10
Target collision (bits):            10
Collision found at Counter Value 877!
Digest: 0004d5cc 4317283e e07f9eb3 f3a72e17 848d23fb
--------- Iteration 11
Target collision (bits):            11
Collision found at Counter Value 877!
Digest: 0004d5cc 4317283e e07f9eb3 f3a72e17 848d23fb
--------- Iteration 12
Target collision (bits):            12
Collision found at Counter Value 877!
Digest: 0004d5cc 4317283e e07f9eb3 f3a72e17 848d23fb
--------- Iteration 13
Target collision (bits):            13
Collision found at Counter Value 877!
Digest: 0004d5cc 4317283e e07f9eb3 f3a72e17 848d23fb
--------- Iteration 14
Target collision (bits):            14
Count 5490, SHA1 per sec 549
Collision found at Counter Value 26d2!
Digest: 00038f8b 9b792b89 284dbfb9 4356ef70 0e39720d
```

```
--------- Iteration 15
Target collision (bits):            15
Count 5490, SHA1 per sec 549
Count 10980, SHA1 per sec 549
Count 16470, SHA1 per sec 549
Count 21960, SHA1 per sec 549
Count 27450, SHA1 per sec 549
Count 32940, SHA1 per sec 549
Count 38430, SHA1 per sec 549
Count 43920, SHA1 per sec 549
Count 49410, SHA1 per sec 549
Count 54900, SHA1 per sec 549
Count 60390, SHA1 per sec 549
Count 65880, SHA1 per sec 549
Count 71370, SHA1 per sec 549
Count 76860, SHA1 per sec 549
Count 82350, SHA1 per sec 549
Count 87840, SHA1 per sec 549
Count 93330, SHA1 per sec 549
Count 98820, SHA1 per sec 549
Count 104310, SHA1 per sec 549
Count 109800, SHA1 per sec 549
Count 115290, SHA1 per sec 549
Collision found at Counter Value 1cf4b!
Digest: 0000d783 e8ad6e98 76b7a24f 733c2afe 291f6be5
Terminating Search
```

The collision searcher can find the first few targets very quickly. For example, for target 12, the collision searcher returns the counter value 877 (hex), corresponding to the digest 0004dc... (hex). At target 15, the collision searcher computes for more than 100 seconds. Each 10 seconds, a line is printed with the current counter value, and the performance of the collision searcher. A collision is eventually found for counter value 1cf4b (118,603). So, you have to compute over 100K SHA-1 functions to find a 15-bit collision. This highlights a particular feature of collision search: it's a probabilistic process. On the average, we would expect that a 15-bit collision can be found in approximately 2^{15} iterations (32K), but it is very well possible that it takes longer than that.

For this reference testbench, the reported performance is 549 SHA-1 computations per second. This value would be the reported performance for your implementation. *The absolute runtime of your collision search does not matter. The target will be increased until you get a target that requires a runtime of over 100 seconds on your implementation.*

Ranking Criteria

All designs will be strictly ranked from best to worst. This section defines what 'best' means. We will use the following metrics to evaluate the rank of your design.

- Functional Correctness: This requirement is mandatory for all designs. Your design has to work, in order to be considered for ranking. If it does not work, you will be automatically ranked last. Working means: your design identifies collisions.
- Metric 1: The number of SHA-1 computations obtained per second. Higher is better.
- Metric 2: The area efficiency of the resulting design, expressed in terms of 'SHA-1 computations per second per LE'. An LE is a Logic Element, a unit of hardware in a Cyclone IV FPGA. A lower LE cell count corresponds to a smaller design.
- Metric 3: The turn-in time of your design and report, as measured by the turn-in time on Scholar. Turning in the solution earlier is better. Note that, if you turn in the design multiple times, only the latest turn-in time will be used.

Two designs will be compared as follows, to determine their ranking order. If a design is functionally not correct (i.e. fails the testbench), it will automatically moved to the last rank. If multiple designs are not operational, they will all share the same lowest rank. All functionally correct designs will get a better and unique rank.

- First, the SHA-1 computation rate (Metric 1) will be compared. If there is a difference of more than 5% between them (with the fastest design considered 100%), the fastest design will get a better rank. If, one the other hand, the difference is smaller than 5%, Metric 2 will be used as tie-breaker.
- Metric 2 will be used in a similar way to compare two designs. If the difference in area efficiency between two designs is larger than 5%, then the smallest design gets the better rank. Otherwise, if they are separated less than 5%, Metric 3 will be used as tie-breaker.
- Metric 3 will be used as a final metric in case the ranking decision cannot be made using Metric 1 and Metric 2 alone. In this case, the design turned-in earlier wins.

The grade for your project is determined as follows.

- 70 points of the grade are determined by the ranking of your project as described above. The best design gets 70 (out of 70) points, the worst design gets 30 (out of 70) points, and all other designs are linearly distributed between 30 and 70 points.
- 30 points of the grade are determined by the quality of the written documentation you provide with the solution. The requirements are listed below.

Accelerated Testbench

On Scholar, you will find the reference implementation of this design, `challenge.qar`. You need to expand the project, compile the bitstream, and prepare a board support package as with the previous homework.

You are allowed to make arbitrary changes to the design, as long as the top-level testbench remains unchanged. Your accelerated implementation, regardless of a hardware or a software target, needs to provide an implementation for the following four functions:

- `void setsearchstring(char *);`
- `void settarget(int);`
- `int searchcollision();`
- `int shacomputed();`

For example, let's say that you would implement a collision searcher as a memory-mapped hardware module. In that case, the calls to these functions would be interacting with memory-mapped registers.

Here are some tips for accelerated designs.

1. Proceed cautiously. The time you have to optimize the performance of your design is limited. If you are unsure about how to proceed, work in small steps, always making sure that you keep a functional design.

2. Keep it simple. If you cannot clearly explain your optimization idea to yourself, it's probably not a good idea. Don't start off blindly trying things, you will almost certainly fail. Starts by carefully thinking about the problem. Learn how SHA-1 works. Evaluate options to parallelize the implementation. *Think first, implement later.*

3. While being cautious, also think about the big picture. SHA-1 collision search is an embarassingly parallel program. If your implementation supports it, you could compute multiple SHA-1 at the same time, each for different counter values. For example, let's say you implement four hardware coprocessors that each compute a SHA-1. Assign these coprocessors the ID value 0, 1, 2, and 3. Then, for a counter value x, compute the digest for x+0, x+1, x+2 and x+3. In such a parallel implementation, the `searchcollision()` driver can increment x in steps of 4.

4. Don't spend too much time developing your own custom-made implementation of SHA-1. It's easy to find a hardware implementation online, or to find sample Verilog code for it. It is OK to use such code for the codesign challenge. The time you save with (not) coding SHA-1 can be used to work on the integration. Some pointers:

   - `http://rijndael.ece.vt.edu/schaum/slides/ddii/lecture11.pdf`

  - http://opencores.org/project,sha_core and http://opencores.org/project,sha1

5. A well-optimized software implementation may be as good as a lousy hardware implementation. Don't underestimate the performance of software (and multi-core)!

What to turn in

You have to deliver the following items for the project result:

- A qar file of your final design. It must contain a Nios processor that is capable to execute the evaluation testbench. Make sure it is a complete system, and, pay particular attention to the following items.
  - Include the driver software for the resulting system as well. By default, the software is not included as part of a qar file - you have to include it by hand.
  - Include the bitstream (sof file) for your design. This can be selected as an option during qar file generation.
- Any GEZEL code that you've developed. It is not mandatory to use GEZEL coding, and you may directly code your design in VHDL or Verilog. However, if you have not done HDL coding before, it's strongly recommended to develop your coprocessor in GEZEL, and follow one of the methods explained in the previous Homework.
- A PDF document that describes your resulting design. Please explain your design strategy, the architecture of your hardware/software solution, and overall observations on the design.

Good Luck!!