ECE4530 Fall 2017
Codesign Challenge - Correlation Engine

Assignment posted on 14 November
Solutions due on 7 December 11:59PM

The Codesign Challenge is the finishing assignment in ECE 4530. This project is an exercise in performance optimization: you will start from a reference application on an ARM 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.

The major constraints in the design are as follows.

- The final result has to run on a DE1-SoC board
- The final result has to be turned in before the submission deadline, 7 December 11:59PM
- The testbench needs to run in C on the ARM core; the assignment defines the required API for the accelerated design.

Application: Correlation Engine

Digital correlation is a signal processing operation used to identify or locate signals. For example, correlation is used in the reception of GPS signals, in radar applications, as well for tracking of objects in real-time video data.

Figure 1 illustrates the computations involved in correlation. A reference waveform is captured as discrete-time data samples, and is the signal we wish to locate in a longer search waveform. The search waveform is a discrete-time sampled-data signal which contains a time-shifted version of the reference waveform. The search waveform may also contain noise, which is not shown in Figure 1. The objective of correlation is to determine the position of the reference waveform in the search waveform. This is done by repeatedly computing the dot product of the reference waveform with the search waveform. If we assume that the reference waveform holds `L` samples, and the search waveform holds `N` samples, then the correlation result would be a waveform `corr` of `N` samples, computed as follows.

```
for (j=0; j<N; j++) {
  for (i=0; i<L; i++) {
    corr[j] = corr[j] + reference[i] * search[(j+i) % N];
  }
}
```

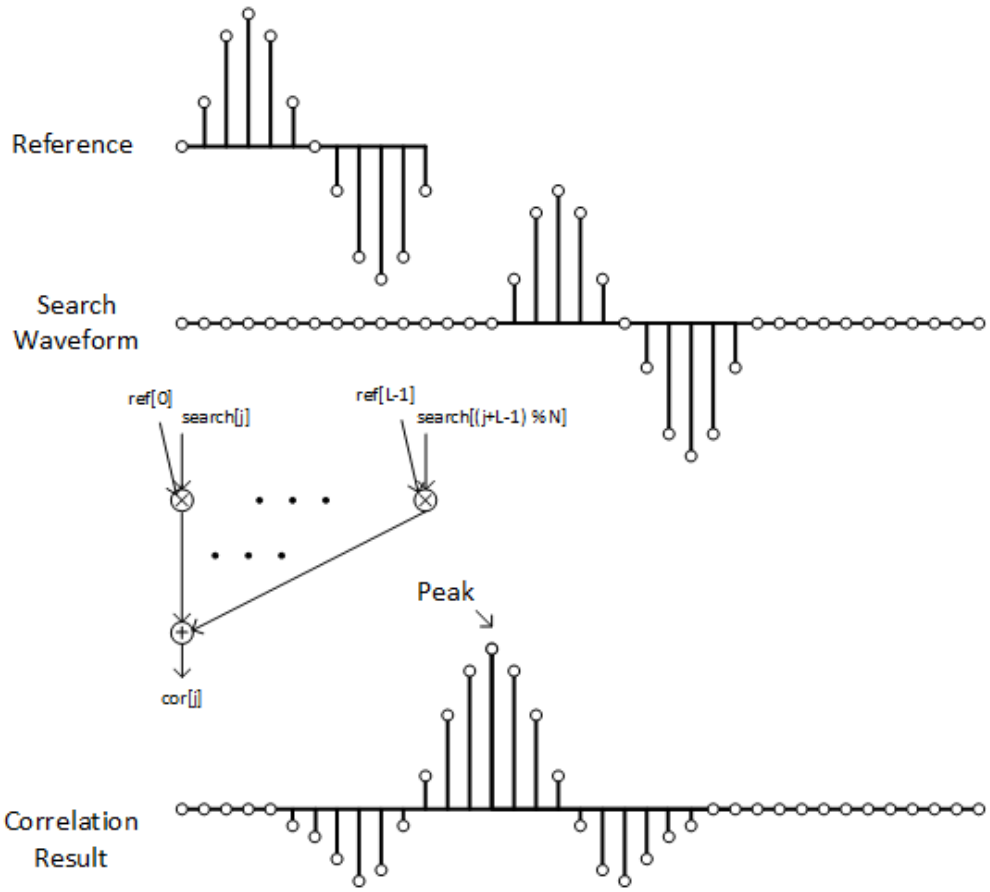

Figure 1: Discrete-time correlation

Each point on the correlation result is the sum of `L` multiplications between samples from the reference and samples from the search waveform.

Note the 'modulo-N' operation in the index computation of the search waveform. It indicates that, as the reference wave shifts beyond the end of the search waveform, the missing samples will be taken from the beginning of the search waveform.

Figure 1 shows the resulting correlation waveform, which has a distinct peak when there is maximum overlap between the reference and the samples of the search waveform. The most probable starting point of the reference within the search waveform is given by the correlation peak.

Computing the correlation is a compute-intensive operation: each point of the correlation result requires `L` multiplications, and determining the correlation peak requires tt L*N multiplications. Hence, the correlation operation is a suitable target for hardware acceleration.

Problem parameters The specific design parameters for the correlation engine are as follows. The reference waveform is a sine wave samples over 16 samples, and the sine wave amplitude is stored with a resolution of 8 bit, using integer values between -128 and 127. The

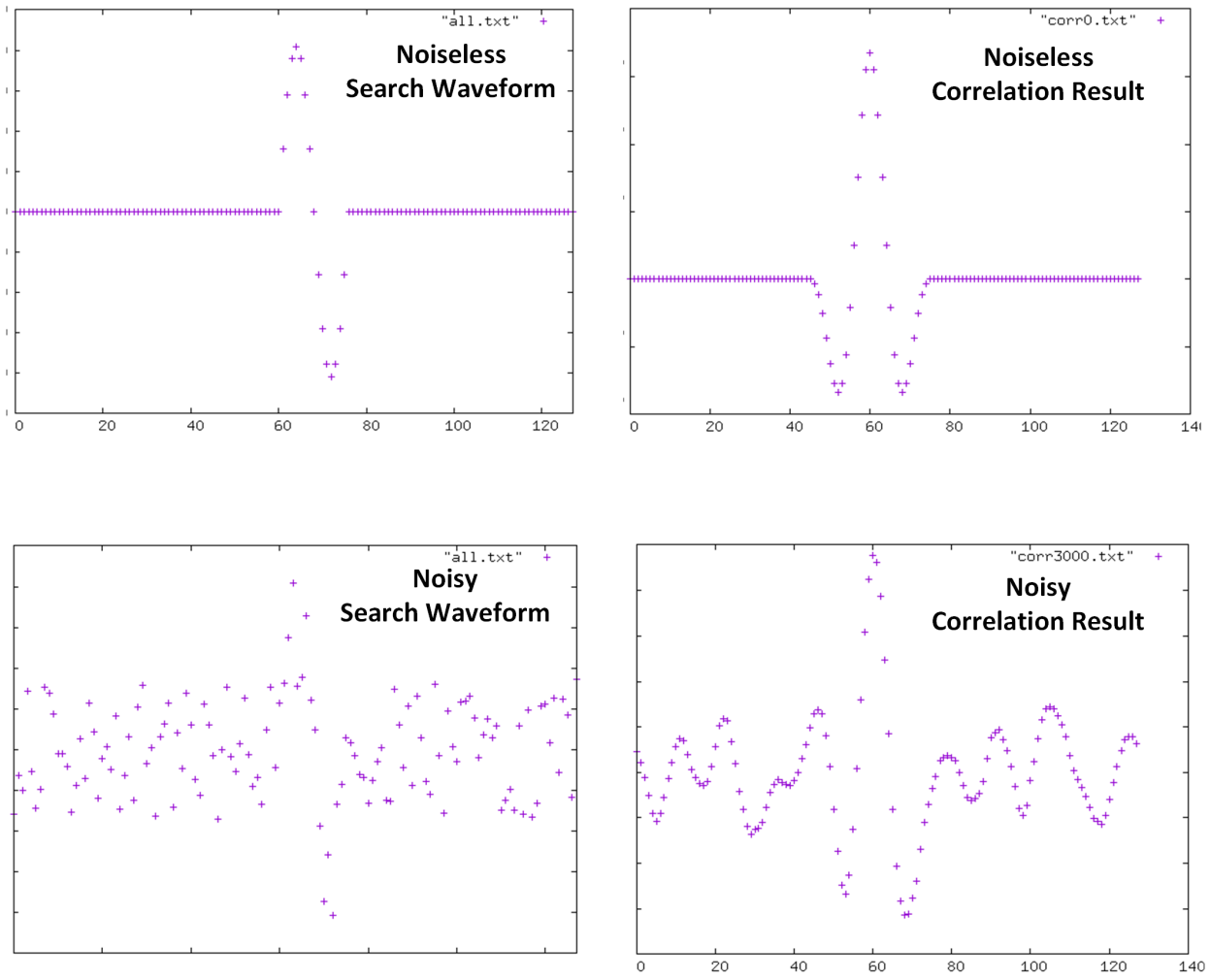

Figure 2: Correlation of a noiseless and a noisy signal

search waveform contains 128 samples, and it is stored with a resolution of 8 bit. The search waveform contains a single copy of the search waveform, starting at sample 60. Figure 3 upper left, and Figure 3 lower left show a noiseless and a noisy version of the search waveform. In each case, the search waveform is scaled such that it uses the full range of 8 bit.

The right side of Figure 3 shows the correlation result of the reference waveform with the search waveform. In the noiseless case, there is a clear peak at position 60, the location where the reference is stored in the search waveform. In the noisy case, there is still a single peak, but there is distortion in the correlation output. It is quite likely that, at high noise levels, the correlation peak will not be exactly at position 60, but may shift backward or forward a few samples. Still, it is obvious that the correlation operation greatly reduces the effect of noise, when we compare the noisy search waveform and the noisy correlation result.

The correlation engine we will build has two inputs: one for a reference waveform (16 samples of 8 bit) and one for a search waveform (128 samples of 8 bit). It has one output:

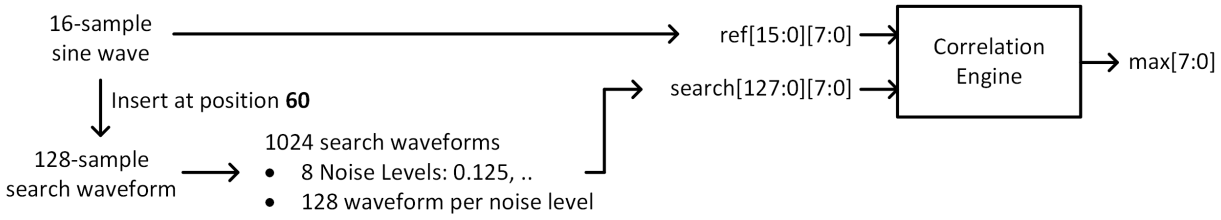

Figure 3: The testbench for the correlation engine tests 1024 waveforms

the index of the correlation maximum in the search waveform (a number from 0 to 127).

Testbench Figure 4 shows the overall testbench of the correlation engine. In total, 1024 different search waveforms are tested. All of them have the 16-sample reference inserted at position 60. There are 8 different noise levels, and there are 128 search waveforms at each noise level. The noise levels start at 0.125 and move up to 1.0. A noise level of 1.0 means that the noise samples being added can have a maximum amplitude as high as the reference sine wave.

The testbench computes an error metric for the outcome as follows. It takes the absolute difference between the correlation maximum found by the correlation engine, and the analytical position used by the testbench (60). In the noiseless case, the difference is always zero. In a noisy waveform, the difference can become bigger than zero. By accumulating the error over multiple search waveforms, we can determine how well the correlator performs. The testbench will use this error metric to compare the functionality of your accelerated design with the reference implementation in software.

Reference Architecture Figure 4 illustrates the reference architecture. The initial reference implementation executes on the ARM, and program is stored in off-chip DDR. The search waveforms are stored in the FPGA, in a 128KB RAM (1024 waveforms of 128 samples of one byte each). The on-chip RAM is configured as an avalon slave, and connects to the AXI bus between the ARM and the FPGA fabric. The on-chip RAM is a single-port RAM with a byte-wide port.

Your accelerated result can modify the memory structure of this design as desired, with the requirement that the testbench code must be able to initialize the memory as part of the testbench initialization. In other words, you cannot hardcode the values of the search and reference waveforms in your design. On the other hand, the resolution of the samples (8-bit), the number of samples per search waveform (128) and the number of samples in the reference waveform (16) are fixed.

The Reference Code The following pages contain a discussion of the reference code, which is released as part of the codesign challenge.

There is only a single file, `correlate.c`. The file can be compiled for your laptop, as well as for the DE1SoC board. There is a makefile that supports compilation.

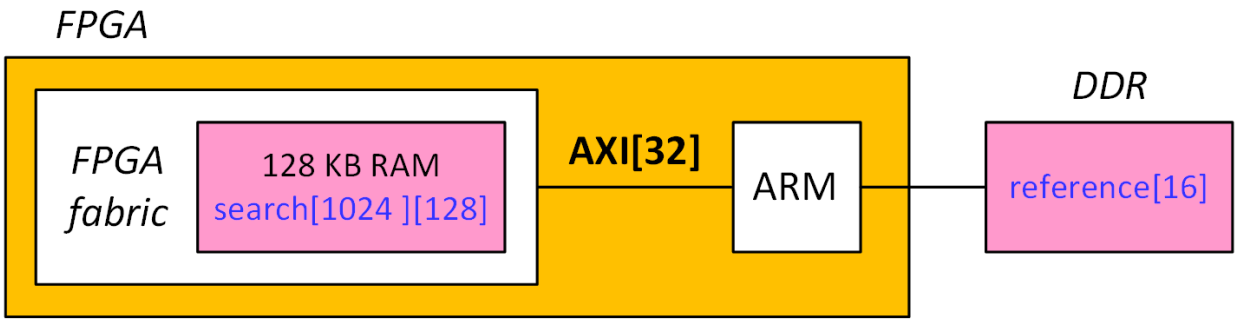

Figure 4: The reference architecture

The initial sections of the code contain necessary include files, as well as code to measure clock cycles on the ARM. There is a macro `NODE1SOC` which is used for code which is not meant for the ARM. The conditional macro `#ifndef NODE1SOC` is used for code that is meant for the ARM.

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <math.h>
#include <assert.h>
#include <string.h>

#ifndef NODE1SOC
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/mman.h>
#include <linux/perf_event.h>
#include "hps_0.h"
#include "socal/hps.h"

static int fddev = -1;
__attribute__((constructor)) static void init(void) {
        static struct perf_event_attr attr;
        attr.type = PERF_TYPE_HARDWARE;
        attr.config = PERF_COUNT_HW_CPU_CYCLES;
        fddev = syscall(__NR_perf_event_open, &attr, 0, -1, -1, 0);
}

__attribute__((destructor)) static void fini(void) {
  close(fddev);
}

static inline long long cpucycles(void) {
```

```
  long long result = 0;
  if (read(fddev, &result, sizeof(result)) < sizeof(result)) return 0;
  return result;
}
#endif
```

Throughput the program, you will find also several special datatypes. `sample_t` is the data type used for samples. `sample_hptr` is the data type used to point to samples stored in a hardware (FPGA) memory. `index_t` is used for the index of samples in an array. These data types are meant to improve the readability of the code. The are defined using `typedef`.

```
typedef signed char           sample_t;
typedef volatile signed char *sample_hptr;
typedef unsigned              index_t;
```

Next, there are several helper functions that support the generation of the reference waveform and the search waveform.

The function `initrng` reseeds the random number generator. The effect is that every execution of the program is unique, as it will use a unique set of noise samples. If you would like to have deterministic execution (which is useful during debugging), then you would comment-out the call to `initrng()` in the main program.

```
// reseeds the PRNG with a true random number
void initrng() {
  unsigned s;
  int f = open("/dev/random", O_RDONLY);
  read(f, &s, sizeof(s));
  srand(s);
  close(f);
}
```

The waveforms are initially constructed as floating-point, and later quantized to integers. Hence, the following functions use floating-point computations. The function `gensin` generates a sine wave. The function `gennoise` generates a noisy signal of a given amplitude. The function `buildsignal` creates a search waveform by inserting a sine wave at a given position, and by adding noise.

```
// generates a sine wave v[] of n samples with double precision
void gensin(double *v, unsigned n) {
  unsigned i;
  assert(n > 0);
  for (i=0; i<n; i++)
    v[i] = sin(2.0 * M_PI * i/n);
}

// generates a noise form v[] of n samples,
// amplitude -level to +level
void gennoise(double *v,
```

```
          unsigned n,
          double level) {
  unsigned i;
  for (i=0; i<n; i++)
    v[i] = level * (2.0 * rand() /  INT32_MAX - 1.0);
}

// creates a search waveform of s samples in sig
//   - noise level level
//   - template inserted at pos
void buildsignal(double *sig, unsigned s,  // output buffer
         double level,                     // noise level
         double *template, unsigned t,     // template to insert
         unsigned pos                      // position of insertion
         ) {
  unsigned i;
  gennoise(sig, s, level);                 // build noisy signal
  for (i=0; i<t; i++)
    sig[(i + pos) % s] += template[i];     // insert template
}
```

Next, there are several utility functions. `scalesig` converts a waveform from floating point to integer by scaling it to a given precision. `findmax` find the highest value in waveform. `samplesave` dumps the contents of a waveform as an ASCII file, which can be used for plotting the waveform.

```
// finds max in sig[] of s samples
index_t findmax(double *sig, index_t s) {
  index_t i, max = 0;
  for (i=0; i<s; i++) {
    if (fabs(sig[i]) > fabs(sig[max]))
      max = i;
  }
  return max;
}

// scales sig[] so that +max or -max of sig equals +(1 << precision)-1
// or -(1 << precision) and stores the result in v[]
void scalesig(double *sig, index_t s,
      unsigned precision,
      sample_hptr v) {

  index_t m = findmax(sig, s);
  unsigned i;
  double scale;

  if (sig[m] > 0)
    scale = ((1 << precision) - 1) * 1.0 / sig[m];
  else
    scale = -(1 << precision) * 1.0 / sig[m];
```

```
  for (i=0; i<s; i++)
    v[i] = floor(sig[i] * scale);
}

// saves an int signal of s samples in file name
void samplesave(sample_hptr sig, index_t s,
char *name) {
  index_t i;
  FILE *f = fopen(name, "w");
  for (i=0; i<s; i++)
    fprintf(f,"%d\n", sig[i]);
  fclose(f);
}
```

The main workhorse of the reference implementation is the function `bulkcorrelate`. This function computes the correlation between the reference waveform, and a set of search waveforms (hence the name bulk-correlate). For each of these correlation operations, it finds the index of the correlation maximum, and stores this in the output array.

The variable `sig` contains the input data. While it is a one-dimensional array, this array really contains `waveforms` separate waveforms of `samples` samples, stored one after the other. Hence, sample `i` of waveform `j` can be accessed as `sig[j * samples + i]`. The correlation loop hence has three levels: the outer counts waveforms, the middle loop counts sample positions within a waveform, and the inner loop counts samples of the template. The correlation values are not stored, since we are only interested in the correlation peak. Therefore, the correlation loop tracks the maximum of the correlation computed over the current waveform, and stores a single maximum per search waveform. Furthermore, only the index of the maximum is kept, and not the maximum value itself.

When `bulkcorrelate` executes on a laptop, it will create a file `wave_corr.txt` which contains the index of the correlation peaks over all search waveforms. Ideally, we would expect all these values to be 60 (the position where the reference is inserted in the search waveform).

```
void  bulkcorrelate(sample_hptr sig,
    index_t     waveforms,        // total number of waveforms
    index_t     samples,          // samples per waveform
    sample_hptr template,         // search template
    index_t     templatesamples,  // samples in search template
    index_t    *c                 // store max index of correlation of each waveform
    ) {
  index_t i, j, k;
  int     acc;
  int     max;
  index_t maxindex;

  for (i=0; i < waveforms; i++) {
    max      = 0;
    maxindex = 0;
    for (j=0; j < samples; j++) {
```

```
      acc = 0;
      for (k=0; k < templatesamples; k++) {
    acc += sig[i * samples + (j + k) % samples] * template[k];
      }
      if (acc > max) {
    max = acc;
    maxindex = j;
      }
    }
    c[i] = maxindex;  // max correlation index
  }

#ifdef NODE1SOC
  {
    FILE *f = fopen("wave_corr.txt", "w");
    for (i=0; i<waveforms; i++)
      fprintf(f, "%d\n", c[i]);
    fclose(f);
  }
#endif

}
```

The objective of the codesign challenge is to accelerate `bulkcorrelate` using hardware/software codesign techniques. You have to capture your accelerated design in `hw_bulkcorrelate`. This function can contain an interface to a hardware coprocessor, or a highly optimized software design. The default implementation of `hw_bulkcorrelate` simply calls `bulkcorrelate`.

Right before the main function, we find a set of functions that compute the median (used for cycle-count evaluation), as well as a set of macros `T1`, `S1`, `Q1` and `N1`. These macros represent the number of samples in the reference waveform, the number of samples in the search waveform, the number of waveforms per noise level, and the number of noise levels, respectively. The last one, in combination with the `NOISELEVEL` macro, determines the noise levels used.

The global variable `bulk` contains all of the generated search waveform samples. There is a considerable amount of those samples: 8 noise levels * 128 waveforms/level * 128 samples/waveform = 128K samples.

```
#ifndef NODE1SOC

#define TIMINGMEASUREMENTS 5

int compare_unsigned(const void *a, const void *b) {
  const unsigned long long *da = (const unsigned long long *) a;
  const unsigned long long *db = (const unsigned long long *) b;
  return (*da > *db) - (*da < *db);
}

unsigned long long median(unsigned long long *thist) {
  qsort(thist, TIMINGMEASUREMENTS, sizeof(unsigned long long), compare_unsigned);
```

```
  return thist[TIMINGMEASUREMENTS >> 1];
}

#endif

//-----------------------------------------------------------------------
// T1 - reference search template is 16 samples
// S1 - each search waveform holds 128 samples
// Q1 - processing batch size is 128 waveforms per noise level
// N1 - number of noise levels to test is 8
#define T1 16
#define S1 128
#define Q1 128
#define N1 8
#define NOISELEVEL(A) (0.125 + 0.125 * A)

#ifdef NODE1SOC
// bulk storage in main memory
sample_t    bulk[N1*Q1*S1];
#else
// bulk storage in FPGA on AXI bus
sample_hptr bulk; // [N1*Q1*S1];
#endif
```

The main function builds the reference waveform and the search waveforms, and quantizes all of them to 8 bit. Next, it runs the bulk correlation for the reference implementation as well as for the accelerated implementation. When the code runs on an ARM, the timing will be evaluated, and the correlation will be computed `TIMINGMEASUREMENTS` number of times.

The final part of the main function shows the error evaluation, which compares the results between the reference version and the accelerated version. While it is possible to compute the result exactly (the reference implementation only includes computations on 8-bit signals), the error evaluation allows for some mismatch. In total, over 1024 waveforms, the total mismatched-index distance has to be smaller than 1024.

```
int main() {
  index_t  i, j;
  double   wave       [T1];
  double   sig        [S1];
  sample_t isig       [S1];

  sample_t intwave    [T1];
  index_t  ref_intmax [N1*Q1];
  index_t  acc_intmax [N1*Q1];

#ifndef NODE1SOC
  int memfd;
  if( ( memfd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
    printf( "ERROR: could not open \"/dev/mem\"...\n" );
    return( 1 );
  }
```

```
  bulk = mmap( NULL,
       ONCHIP_MEMORY2_0_SPAN,
       ( PROT_READ | PROT_WRITE ),
       MAP_SHARED,
       memfd,
       0xC0000000 + ONCHIP_MEMORY2_0_BASE );

  if( bulk == MAP_FAILED ) {
    printf( "ERROR: mmap() failed...\n" );
    close( memfd );
    return(1);
  }
#endif

  unsigned TARGET = 60;

  // Prepare the signal

  // - reseed the random number generator
  initrng();

  // - generate a template waveform
  gensin (wave, T1);

  // - scale it to 8 bit (7 bit 2-c)
  scalesig    (wave, T1, 7, intwave);

#ifdef NODE1SOC
  samplesave(intwave, T1, "wave_ref.txt");
#endif

  // - build signal
  for (i=0; i<N1; i++) {   // for every noise level
    for (j=0; j<Q1; j++) {     // for all waveforms at this noise level

      buildsignal (sig,           // waveform
   S1,            // number of samples in waveform
   NOISELEVEL(i), // noise level
   wave,          // target waveform to hide
   T1,            // number of samples in target waveform
   TARGET         // target insertion index
   );

      scalesig    (sig,
   S1,            // number of samples in waveform
   7,             // 8 bit = 7 bit 2-c
   isig
   );

      memcpy((void *) (bulk + (j + i * Q1) * S1),   // store it in bulk memory
```

```
     isig,
     sizeof(char) * S1);
#ifdef NODE1SOC
  // on x86, save sample waveforms
      {
char buf[64];
if (j == 0) {
  snprintf(buf,64,"wave_level_%d.txt", i);
  samplesave(isig, S1, buf);
}
      }
#endif
    }
  }

  //------------------------------ reference computation

#ifndef NODE1SOC
  unsigned long long ref_cycles[TIMINGMEASUREMENTS];
  unsigned timingloop;

  for (timingloop=0; timingloop<TIMINGMEASUREMENTS; timingloop++) {
    ref_cycles[timingloop] = cpucycles();
#endif

    // reference computation
    bulkcorrelate(bulk,        // waveform[num][samples]
  N1*Q1,       // N1*Q1 waveforms
  S1,          // S1 samples per waveform
  intwave,     // search template
  T1,          // of T1 samples
  ref_intmax   // max index found
  );

#ifndef NODE1SOC
    ref_cycles[timingloop] = cpucycles() - ref_cycles[timingloop];
  }
#endif

  // error analysis
  printf("REFERENCE\n");
  for (i=0; i<N1; i++) {
    unsigned errorlevel = 0;
    for (j=0; j<Q1; j++)
      errorlevel += abs(ref_intmax[i*Q1 + j] - TARGET);
    printf("Noise %4.3f error %4d\n", NOISELEVEL(i), errorlevel);
  }

  //------------------------------ accelerated computation

#ifndef NODE1SOC
```

```
  unsigned long long acc_cycles[TIMINGMEASUREMENTS];

  for (timingloop=0; timingloop<TIMINGMEASUREMENTS; timingloop++) {
    acc_cycles[timingloop] = cpucycles();
#endif

    // accelerated computation
    hw_bulkcorrelate(bulk,        // waveform[num][samples]
     N1*Q1,       // N1*Q1 waveforms
     S1,          // S1 samples per waveform
     intwave,     // search template
     T1,          // of T1 samples
     acc_intmax   // max index found
     );

#ifndef NODE1SOC
    acc_cycles[timingloop] = cpucycles() - acc_cycles[timingloop];
  }
#endif

  unsigned totalerror = 0;

  // error analysis
  printf("ACCELERATED\n");
  for (i=0; i<N1; i++) {
    unsigned errorlevel_ref = 0;
    unsigned errorlevel_acc = 0;
    for (j=0; j<Q1; j++) {
      errorlevel_ref += abs(ref_intmax[i*Q1 + j] - TARGET);
      errorlevel_acc += abs(acc_intmax[i*Q1 + j] - TARGET);
    }
    printf("Noise %4.3f error %4d delta %4d\n",
   NOISELEVEL(i),
   errorlevel_acc,
   abs(errorlevel_ref - errorlevel_acc));
    totalerror +=  abs(errorlevel_ref - errorlevel_acc);
  }

  printf("Total Error: %d\n", totalerror);
  if (totalerror <1024)
    printf("Testbench passes!\n");
  else
    printf("Testbench fails!\n");

#ifndef NODE1SOC
  printf("Reference Execution time %lld\n", median(ref_cycles));
  printf("Accelerated Execution time %lld\n", median(acc_cycles));

  if ( munmap( (void *) bulk, ONCHIP_MEMORY2_0_SPAN ) != 0 ) {
    printf( "ERROR: munmap() failed...\n" );
    close( memfd );
```

```
    return( 1 );
  }
  close( memfd );
#endif

  return 0;
}
```

Compiling and Running the Code The code can be compiled and run on your laptop as well as on the DE1SoC board. After typing 'make' you will see that an executable is being made for each platform.

```
arm-linux-gnueabihf-gcc -g -Wall -Dsoc_cv_av -I../conversion \
   -Ic:/intelFPGA_lite/17.0/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av \
   -Ic:/intelFPGA_lite/17.0/embedded/ip/altera/hps/altera_hps/hwlib/include/ \
   -g -Wall -lm correlate.c -o correlate
arm-linux-gnueabihf-objdump -D correlate > correlate.lst
arm-linux-gnueabihf-size correlate
   text    data     bss     dec     hex filename
   5945     376      84    6405    1905 correlate
gcc -lm -DNODE1SOC correlate.c -o correlate_x86
```

To run on X86, execute `correlate_X86`. You will see the following output. In this case, there is no difference between the reference version and the accelerated version (delta is always zero). Note that the 'error' identified in each case is not a functional error, but a result of correlating a noisy signal: it is possible that the exact position of the reference waveform is not found due to the impact of noise.

```
$ ./correlate_x86.exe
REFERENCE
Noise 0.125 error    0
Noise 0.250 error    0
Noise 0.375 error    1
Noise 0.500 error    6
Noise 0.625 error   21
Noise 0.750 error   32
Noise 0.875 error   35
Noise 1.000 error   46
ACCELERATED
Noise 0.125 error    0 delta    0
Noise 0.250 error    0 delta    0
Noise 0.375 error    1 delta    0
Noise 0.500 error    6 delta    0
Noise 0.625 error   21 delta    0
Noise 0.750 error   32 delta    0
Noise 0.875 error   35 delta    0
Noise 1.000 error   46 delta    0
Total Error: 0
Testbench passes!
```

To execute the testbench on the board, proceed as follows. First, compile the bitstream (in baseline/hardware). Next, convert the bitstream to a compressed bitstream (in baseline/conversion). Finally, compile the software as shown above (in baseline/software). Copy the compressed bitstream and the executable `correlate` to the board, configure the FPGA, and run the executable. You will find that the reference implementation (which does not use compiler optimization) takes around 460 million clock cycles to compute 1024 correlations.

```
root@socfpga:~# ./hps_config_fpga baseline.rbf
INFO: alt_fpga_control_enable().
INFO: alt_fpga_control_enable OK.
alt_fpga_control_enable OK  next config the fpga
INFO: MSEL configured correctly for FPGA image.
baseline.rbf file file open success
INFO: FPGA Image binary at 0x72cad008.
INFO: FPGA Image size is 2395416 bytes.
INFO: alt_fpga_configure() successful on the 1 of 5 retry(s).
INFO: alt_fpga_control_disable().
root@socfpga:~# ./correlate
REFERENCE
Noise 0.125 error    0
Noise 0.250 error    0
Noise 0.375 error    3
Noise 0.500 error    8
Noise 0.625 error   16
Noise 0.750 error   31
Noise 0.875 error   36
Noise 1.000 error   43
ACCELERATED
Noise 0.125 error    0 delta    0
Noise 0.250 error    0 delta    0
Noise 0.375 error    3 delta    0
Noise 0.500 error    8 delta    0
Noise 0.625 error   16 delta    0
Noise 0.750 error   31 delta    0
Noise 0.875 error   36 delta    0
Noise 1.000 error   43 delta    0
Total Error: 0
Testbench passes!
Reference Execution time 458017728
Accelerated Execution time 457982103
```

Objective and rules of implementation

- The accelerated design must have the same functionality as the reference software design - the testbench must announce 'testbench passes!'.
- You are allowed to change anything inside of the function `hw_bulkcorrelate()`. However, you cannot modify the function call itself: the API must run unmodified by the testbench in `main`.

- In particular, it's completely your decision on how you will accelerate `hw_bulkcorrelate`. This can include building hardware versions of the correlation engine, optimizing the software, transforming how data (such as the reference waveform) is stored.
- You are allowed to change tool settings and enable optimizations as you wish. You are allowed to change the operating frequency of the board as you wish.
- When in doubt if a certain change is permissible or not, ask by email. In general, we will allow any optimization that sticks to the spirit of the testbench (ie. computing correlations). Hardcoding the results to 'fake' the solution, or tampering with the cycle counter, is obviously not allowed.
- The speedup of your design is equal to the execution time of the original reference design (460 million cycles) to the accelerated design. Your speedup will be computed as (460,000,000 / accelerated plotting time). The value 460,000,000 can be considered absolute and will not be affected when you use software optimization compiler flags.

Ranking Criteria

All designs will be strictly ranked from best to worst. 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 (the testbench must pass), in order to be considered for ranking. If it does not work, you will be automatically ranked last.
- **Metric 1**: The speedup of your design following the formula given above. Higher is better.
- **Metric 2**: The area efficiency of the resulting design, expressed in ALMs. An ALM is an Adaptive Logic Module, a unit of hardware in a Cyclone V FPGA. A lower ALM cell count corresponds to a smaller design. You can find the ALM count in the 'Flow Summary' of Quartus.
- **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 incorrect (i.e. fails the testbench), it will automatically be moved to the lowest rank. If multiple designs are not operational, they will all share the same lowest rank. Every functionally correct design will get a better and unique rank.

- First, the speedup (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, on 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 turn-in time is the time of your last commit on the git repository.

The grade for your project is determined as follows.

- 60 points of the grade are determined by the ranking of your project as described above. The best design gets 60 (out of 60) points, the worst design gets 30 (out of 60) points, and all other designs are linearly distributed between 30 and 60 points.

- 40 points of the grade are determined by the quality of the written documentation you provide with the solution.

Acceleration Strategies

The main objective of this assignment is to make the testbench run as fast as possible, using hardware software codesign. There are several layers to this problem and it is worthwhile to carefully think them through.

- First, there is the correlation loop (the innermost loop in bulkcorrelate). You can study the code of the reference implementation and observe that this is a simple for loop that sequentially performs multiplications. You can envisage unrolling this loop and, using 16 parallel multipliers, compute a correlation value in parallel.

- Next, there is a sequence of 128 points in the search waveform (the middle loop in bulkcorrelate), and there are 1024 such waveforms (the outer loop in bulkcorrelate). This means that there is a huge amount of data to process, and it is unlikely that it can all be stored in hardware. Instead, you would need to loop for an efficient data transportation mechanism.

- Note also, that each waveform correlation in bulkcorrelate is independent of any other correlation. If you have parallel correlation engines, you can make then work in parallel provided you can access data fast enough from memory.

- The reference testbench takes 460 million clock cycles to correlate 1024 waveforms. Each waveform needs 128*16 = 8192 data multiplications. So we have around 2 million data multiplications, or about 230 cycles per data multiplication. Since a multiplication on the ARM takes just one clock cycle, there are 229 clock cycles used for 'overhead' such as moving data. The reference implementation is very inefficient!

There are several optimization mechanisms to consider. However, it's vital to think and act strategically.

- Avoid using a random lets try this and see strategy. Very likely you will forget something, or it will lead you into a local optimum while loosing sight of the big picture. Always keep an objective in mind (eg. a desired acceleration factor), and measure your progress in reaching your objective. Such a final objective should be derived through a reasonable back-of-the-envelope calculation. Always look for the bottleneck in the overall system.

- Take only a small step at a time, and verify, verify, verify. Make sure that you have always something that works and that you can fall back on.

- Despite the tremendous computational requirements, the algorithm can be extensively parallelized. Hence, your optimizations should initially be guided by 'how can I do more computations in parallel'. A second very, very important design decision is to minimize the data movement required in the system. If you have less data to copy, you get more cycles available for computations.

- The key to succeed in this assignment is NOT in a single monster session of back-to-back all-nighters a few days before the deadline. The key to succeed is to work on this design in small steps over a longer period. Design is a creative process, and that takes time. Ideas take time to develop. Think about your strategy and discuss with your colleagues. This is an open-ended assignment, and you are competitively graded, so its not in your advantage to disclose your ideas. However, a brainstorm session or two with your peers may do wonders to sharpen your insight in this assignment.

What to turn in

You have to deliver the following items for the project result. Everything has to be provided through github.

- A rbf (bitstream) file of your final design, and a compiled executable (elf) of your final design.

- The source code of your optimized testbench driver, as a C file.

- The source code of your Verilog design, in case you added a new coprocessor to the QSYS system.

- 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. Note that the PDF document counts for 40 % of the grade, so it's worth to do this carefully. Work on the PDF before you're running out of time.