Codesign Challenge

The Challenge

Given is a reference implementation of a correlation engine which sits on an ARM processor. The task is to improve the performance of the reference implementation by optimizing the C code and/or by using a hardware co-processor.

Platform

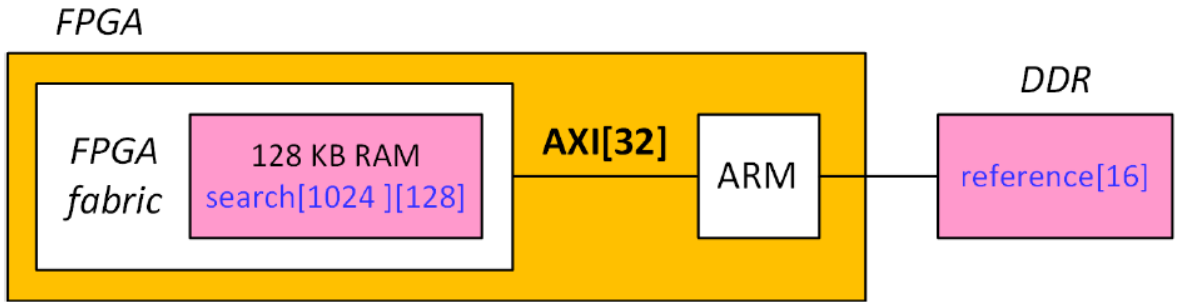
Altera Cyclone V

The Program

To perform correlation of 1024 search waverforms with a reference waveform. The search waveform has 128 samples of 8 bit. The reference waveform has 16 samples of 8-bit. The reference C design that runs on a RISC 32 bit processor completes the task in 460,000,000 cycles.

Given Architecture

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.



Here is the code snippet of the reference implementation:

```
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


 }


}
```

Bottlenecks

The reference implementation completes the task in 460,000,000 cycles, which means the code mentioned above utilizes 460,000,000 cycles. By pulling this code apart, it is seen there are three loops which are run.

Outer loop runs 1024 times, the middle runs 128 times and the innermost runs 16 times. Which makes any line of code found in the innermost loop to be executed 1024 * 128 * 16 times which makes it run for 2,097,152 times. If somehow this is optimized that would result in a big speedup.

The inner most loop is where the magic of correlation engine lies.

```
acc += sig[i * samples + (j + k) % samples] * template[k];
```

Here the samples of the search wave form are multiplied with the samples of the reference waveform and the sum is being accumulated. This is basically a multiplication and add statement which can be done easily in a hardware coprocessor in a parallel fashion.

Approach #1

At first, I designed a simple multiplier in the coprocessor which enabled me to do multiplication of a wave sample with reference sample in a single step in the coprocessor and saving the extra multiplication cycles wasted when it is done on ARM.

#Drawbacks

The fact I was transferring both the reference as well as the search wave sample one by one required extensive bus operations. Especially fetching the search wave samples from the on-chip memory of the fpga to the ARM and then back to the fpga. This added extra cycles compared to the reference implementation.

Approach #2

To tackle the transfer latencies of moving data from on-chip memory to ARM and then back to the fpga, I utilized DMA(Direct Memory Access) controller to move data from on-chip memory to the fpga. This considerably reduced the overhead of bus transactions. For proper utilization of the 32-bit bus width, I moved 4 reference wave samples of 8 bit on a single bus transaction outside all loops to avoid repeated tranfer. In order to exploit the independent nature of the task, I wished to do all 128 * 16 multiplication in parallel.

#Drawbacks

This however is not possible as a maximum of only 87 multipliers that can be fit into the fpga fabric.

Approach #3

To make all computation take place in the coprocessor, I decided to stick with just one block of 16 multipliers that would run in parallel and keep computing the sum. The same task that was being done by the inner most loop of the reference implementation. To avoid sending the sum at every computation to the ARM from the coprocessor and letting the software store the max sum of 16 multiplications across the whole search waveform of 128 samples, I decided to let max sum and the index at which the max sum is found to be computed and stored by the coprocessor. This helps eliminate the inner loop completely. Here is the snippet of my code:

```
unsigned temp;

 for(i=0; i<16;i=i+4){
            temp = (unsigned char)template[i];
            temp = (temp<<8);
            temp = temp + (unsigned char)template[i+1];
            temp = (temp<<8);
            temp = temp + (unsigned char)template[i+2];
            temp = (temp<<8);
            temp = temp + (unsigned char)template[i+3];
            cor_base[i+1] = temp;

 }
 //Sending signal for correlation
 for (i=0; i < 1024; i++) {

            cor_base[23] = 1; //Clearing maxsum in fpga

            for(j=0;j<144;j++)
            cor_base[0] = sig[i * 128 + j];


   c[i] = cor_base[21];  // max correlation index
 }
```

This achieved the speedup of 4.5x

By adding DMA controller, the speedup went to 38.11x with this code:

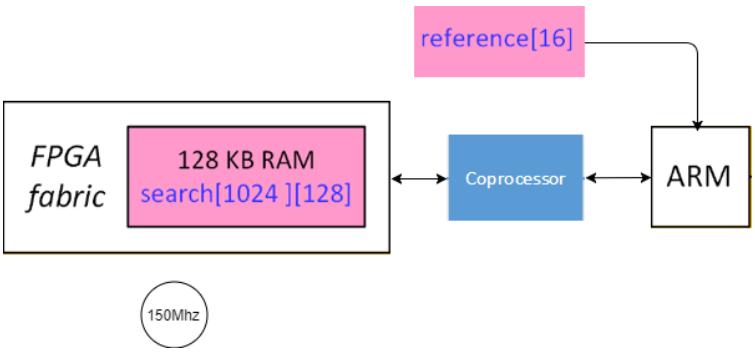
```
for (i=0; i < 1024; i++) {

            cor_base[23] = 1; //Clearing maxsum in fpga

            dmareset(dmabase);
   dmacopyconst(dmabase,
                        0 + (i*128), //Source
                        0x20000,                    //Destination
                        128
                        );
   while (! dmacomplete(dmabase)) ;

   c[i] = cor_base[21];  // max correlation index
 }
```

Architecture



Under the hood of my coprocessor

My coprocessor receives the 16 samples of the reference waveform and stores them in separate registers. The transmission of these reference samples of 8bit happens on a 32bit wide bus. To utilize the bus width, 4 samples are concatenated together and sent. This helps reducing the bus cycles from 16 to 4. But it is important to mention that the shifting operation to concatenate separate samples as a single value to be sent has an added overhead, but the cycle loss due to 12 extra bus cycles is more. As mentioned in class, the key to coprocessor design is analyzing data access.

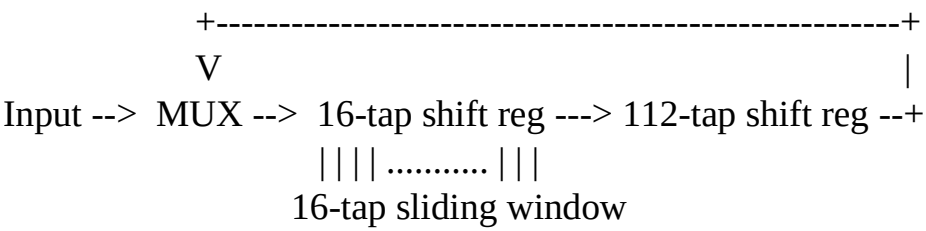
The innermost loop of reference implementation makes a sum of samples coming from memory

search[0] .. search[15] -> first sum
search[1] .. search[16] -> second sum
search[2] .. search[17] -> third sum
...

Given that 16 parallel multiplications take place between the 16 samples of the search waveform and the reference waveform and then the sum is computed, a sliding window mechanism is employed with the help of a 16-byte (sample size) shift register and a 112-byte shift register, designed in Verilog. The software driver sequentially sends the 128 samples of the search wave form. The samples are fed into the 16-byte shift register as they arrive. As soon as the 16-byte shift register is full, a signal is raised to start parallel multiplications of all the reference sample received earlier with the samples in the shift register. Whenever the coprocessor receives a new value, it pushes it in the 16-byte shift register and hence results in the window like structure described above. While a new value is added to the shift register, the first value to enter is pushed onto another shift register of 112-byte. This is done to enable the wrap around mechanism, as shown below.

search[112] .. search[127] -> 113th sum
search[113] .. search[0] -> 114th sum
search[114] .. search[1] -> 115th sum
search[115] .. search[2] -> 116th sum
..
search[127] .. search[14] -> 128th sum

The output of that 112-byte shift register is fed back to the first shift register:



The multiplication operation on the sliding window is valid from cycle 16 to cycle 144. We have to skip the initial 1 to 15 cycles because the shift register is filling up. The resultant value of all the multiplication between the 16 reference samples and the 16 samples present in the window are added and saved in a register. This register value is then compared with the sum that is generated because of the same operation which takes place for the new values that enter the window. The sum value which comes out to be the largest is saved in a separate register called maxsum. This mechanism helps in calculating the largest sum for a search waveform having 128 samples.

In order to calculate the index of the sample which gives out the max sum, I utilized a counter which helps in managing the 16 and 112 byte shift registers. The counter counts from 0 to 143 which make up the total 144 cycles required to go through a search wave. The multiplication and summation begins when counter value is greater than 15 as it marks the point where the window is filled up. Hence at any moment when the counter value is greater than 15, the value in counter minus 16 would give the index of the current sample being correlated. Hence this value is put in the register maxindex as and when maxsum is computed. The only value sent to the ARM from the co-processor is the maxindex per search waveform.

Extra Tweaks

By adding phase locked loop via Qsys I was able to raise the clock frequency of the FPGA to 150Mhz from 50Mhz which resulted in total speedup of 87.87x.

Further Acceleration

Currently only 16 multipliers are being utilized by my coprocessor out of 87 that can be utilized. By adding more parallel multiplication my design can be further accelerated to cross 100x of speed up.

Output

```
REFERENCE
Noise 0.125 error    0
Noise 0.250 error    0
Noise 0.375 error    0
Noise 0.500 error    7
Noise 0.625 error   15
Noise 0.750 error   20
Noise 0.875 error   35
Noise 1.000 error   87
ACCELERATED
Noise 0.125 error    0 delta    0
Noise 0.250 error    0 delta    0
Noise 0.375 error    0 delta    0
Noise 0.500 error    7 delta    0
Noise 0.625 error   15 delta    0
Noise 0.750 error   20 delta    0
Noise 0.875 error   35 delta    0
Noise 1.000 error   87 delta    0
Total Error: 0
Testbench passes!
Reference Execution time 194103481
Accelerated Execution time 5234790
```

Speedup = 460,000,000 / 5234790 = 87.8736