The Codesign Challenge

Objectives

In the codesign challenge, your task is to accelerate a given software reference implementation as much as possible. You can use any of the previously discussed techniques to accelerate the implementation: optimize the software, build a coprocessor, optimize the hardware/software communication. You can use library modules you find on the Internet.

The constraints of the solution are:

1. that it must be completed by 11/29/2010 at 5:00PM;
2. that it must run correctly on the Spartan 3E starter kit or the Altera DE2 kit (depending on the type of kit you have used during Homework assignments);
3. and that it follows the given testing procedure to demonstrate the performance of your implementation.

The solutions will be ranked in two categories: one for Altera kits, and a second one for Xilinx kits. Within each category, all solutions will be strictly ranked and graded accordingly. Thus, there will be a single 'best' Xilinx solution, a single 'best' Altera solution, and so on. There is no perfect solution; the only 'perfect' solution is the one that ranks higher than all the others within its category.

The quality of your solution will be evaluated using the following criteria:

1. The resulting clock cycle count of the design. The clock cycle count is defined in the system test-bench, to be specified further in this homework.
2. The product of the logic utilization and the cycle count. Logic utilization is defined as:
    a. for Xilinx devices: the sum of LUTs and registers
    b. for Altera devices: the sum of Logic Elements and registers
3. The time when the solution was turned in (before the deadline, but earlier is better).

The clock cycle count is most important, the area-time product is second-most important, and the design time is third-most important. Faster (but correct) designs will always win. For solutions which have a cycle count within 5% of each other, the area-time product (item 2) will be used as a tie-breaker. In case two solutions are within 5% of each with respect to item 1 as well as item 2, the time of posting the solution will be used to resolve the ranking of the two designs.

Hence, it is in your interest to find the highest possible performance that can still be accommodated on the Xilinx/Altera board, and to find that solution as quickly as possible.

The Fast Fourier Transform

Your task is to accelerate a Discrete Fourier Transform as much as possible. The Discrete Fourier Transform (DFT) is an algorithm to evaluate the spectrum of a sampled-data signal. The DFT is extensively used in Digital Signal Processing applications that need spectral analysis.

The particular variant of the DFT you need to accelerate is a Fast Fourier Transform. You will receive a reference implementation of the FFT in C. The FFT operates on complex data, which means that each data sample has a real part and an imaginary part.

As the FFT is an extensively studied algorithm, there is plenty of documentation that can be found online. In this assignment, we only include a summary discussion, emphasizing the flow of data in the algorithm, and the operations it performs.

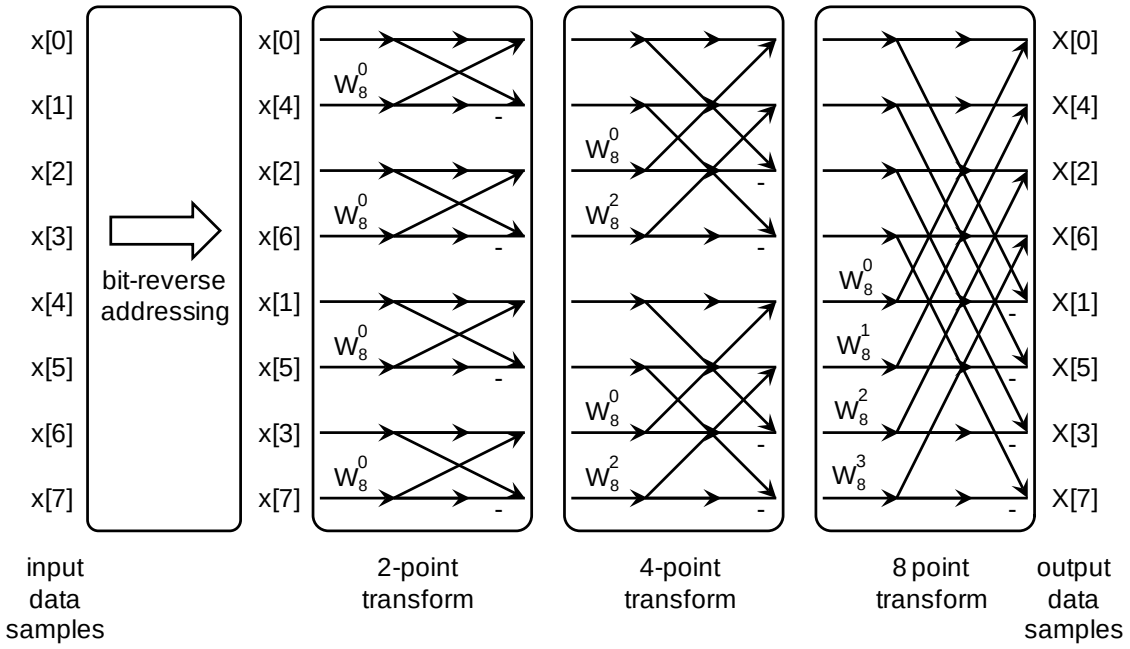

Figure 1: Data-flow in an 8-point decimation-in-time FFT

Figure 1 illustrates the data operations involved in an 8-point FFT. This algorithm transforms 8 samples on the left of the figure, x[0] through x[7], into 8 frequency samples on the right of the figure, X[0] through X[7]. The FFT goes through the following steps:

- Initial bit-reversal, which reshuffles each element to its bits-reversed index. For example, element x[3] has index 3 = (011)b. The bit-reversed index is (110)b = 6. Hence, the fourth position of the bit-reversed array is occupied by x[6].
- Four 2-point transforms
- Two 4-point transforms
- One 8-point transform

The notation for the transforms is further clarified as follows. Each cross in a transform is called a *butterfly*; it's an operation on two complex data samples, resulting in two transformed data samples.

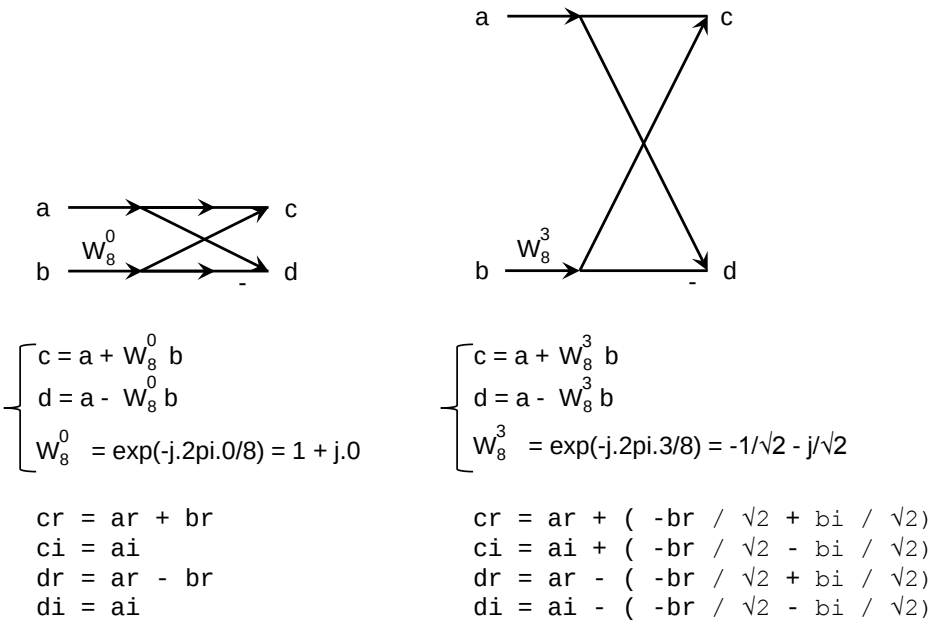

Figure 2: FFT Butterfly Operations

Figure 2 describes two sample butterflies present in Figure 1. The left butterfly can be found in the 2-point transform; the right butterfly can be found in the 8-point transform. A butterfly consists of a complex addition and a complex subtraction. The b-operand of the butterfly is multiplied with a complex weight factor W before the addition/subtraction. These W can be expressed as exp(-j.a) = cos(a) - j.sin(a). For example,

$W_8^0 = \exp(-j.2\pi.0/8) = \cos(0) - j.\sin(0) = 1 + j.0$.

Similarly,

$W_8^3 = \exp(-j.2\pi.3/8) = \cos(2\pi.3/8) - j.\sin(2\pi.3/8) = -1/\sqrt{2} - j/\sqrt{2}$.

When observing Figure 1 closely, one can see that the FFT has a high degree of symmetry. In fact, a 16-point FFT looks very similar to an 8-point FFT. It consists of an initial bit-reversal and four passes of transforms:

- Initial bit-reversal
- Eight 2-point transforms
- Four 4-point transforms
- Two 8-point transforms
- One 16-point transform

In the Challenge, you will implement a 1024-point FFT, consisting of an initial bit-reversal followed by 10 passes of transforms.

Fast Fourier Transform on Fixed-point Data

The reference implementation you receive works on 32-bit data in <32,16> fixed-point representation. This data format has 32 bit of resolution and 16 bit of fractional precision. The following examples illustrate operations on <32,16> fixed-point data.

```
int a;     // a is an integer
double d;  // d is a floating point data type
int afix;  // afix is  a <32,16> data type
int bfix;  // bfix is  a <32,16> data type
int cfix;  // cfix is  a <32,16> data type

// to convert integer to <32,16>, shift up by 16
// this may cause overflow of afix
afix = a << 16;

// to convert <32, 16> to integer, shift down by 16
// fractional bits in afix will be lost
a = afix >> 16;

// to convert double to <32,16>, multiple by (1 << 16)
// this may cause overflow of afix
afix = (int) (d * (1 << 16));

// to convert <32, 16> to double, divide by (1 << 16)
d = afix * 1.0 / (1 << 16);

// addition of <32,16> is easy
afix = bfix + cfix;

// multiplication of <32,16> requires scaling
afix = (bfix * cfix) >> 16;
```

The reference architecture

Figure 3 illustrates the reference architecture (in Xilinx or Altera). Obviously, you can modify this as part of solving the codesign challenge.

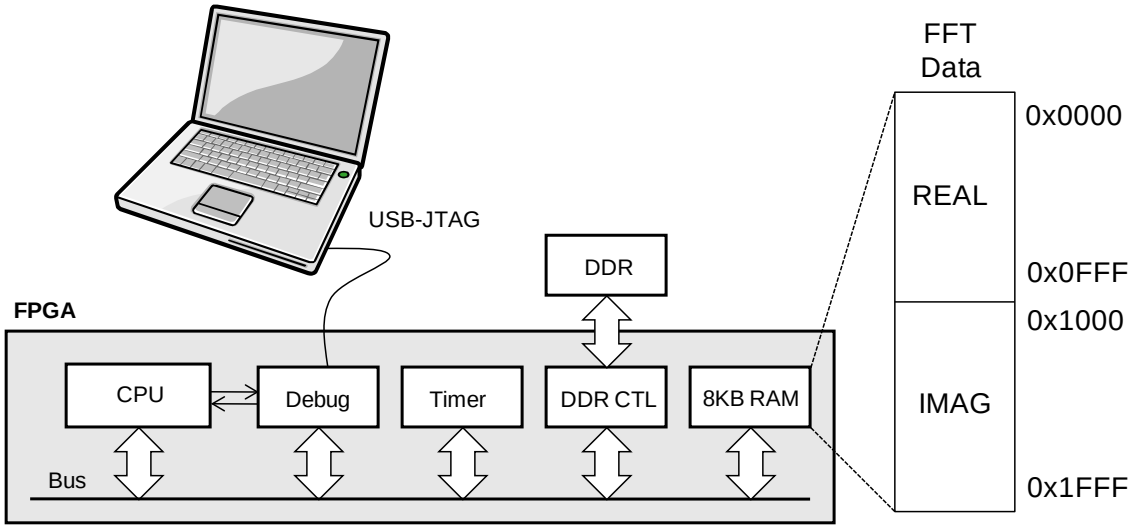

Figure 3: Architecture for the reference implementation

The configuration includes a soft-core CPU, a timer, an off-chip memory controller, and 8 Kilobyte of on-chip RAM. The data that needs to be transformed by the FFT is stored in the 8KB RAM as a 1K-element array of real data followed by a 1K-element array of imaginary data.

You may make arbitrary modifications to the architecture but you must stick to the following restrictions:

- The clock cycles are counted against a 50MHz system clock
- The data transformed by the FFT must be fetched from the 8KB RAM module.
- The 8KB RAM must remain attached to a PLB bus (Xilinx) or an Avalon bus (Altera)

Example 'modifications' of the architecture could include the following:

- Adding CPUs, coprocessors, custom instructions, or memory-mapped coprocessors.
- Segmenting the bus in smaller parts and promoting parallel access.
- Adding storage elements to repartition program/temporary data in the system,

The testbench

On scholar, you will find the following files:

- Xilinx project with reference implementation: xilinxchallenge.tgz
- Altera project with reference implementation: alterachallenge.qar and alterasw.tgz
- Reference C code: fftref.c. The reference C code is for illustration purposes; the Xilinx and Altera projects are self-contained.

The reference implementation of the FFT in C is found in fftref.c:

```
void reference_fftfix(unsigned m, int *x, int *y)
```

Your implementation needs to implement a function

```
void my_fftfix(unsigned m, int *x, int *y)
```

In these functions, `m` is the size of the FFT (given as log2(number_of_points)), `x` is a pointer to the real part of the FFT data, and `y` is a pointer to the imaginary part of the FFT data. By default, `my_fftfix` calls `reference_fftfix`. Of course, when you implement a coprocessor, you will include coprocessor calls inside of `my_fftfix`.

The testbench (on Xilinx as well as on Altera), in fftverify.c, performs the following actions:

1. Verify the correctness of your implementation in `my_fftfix` using a reference pulse of 8 samples wide and unit-height.
2. Evaluate the timing of `reference_fftfix` for a dataset of 1024, 512, 256, and 128 points
3. Evaluate the timing of `my_fftfix` for a dataset of 1024, 512, 256, and 128 points
4. Evaluate the speedup ratio achieved by your optimized `my_fftfix`

The 'resulting clock cycle count' for the design (see criteria on page one) is the cycle count timing required to execute `my_fftfix` for a dataset of 1024 points. The the timing result on 512, 256, and 128 points will not be included. However, your implementation must support these data set sizes as well in order to be counted as correct.

The comparison test between `my_fftfix` and `reference_fftfix` allows for a limited amount of error (see function `comparefft()`). This margin allows for hardware implementations that are not 100% bit-accurate to the reference implementation in C. However, keep in mind that the margin is limited. The testbench will tell you if the accumulated error is within acceptable range.

Testbench Output on Xilinx/Altera

If you implement the reference design and run the testbench, you will observe the following output on Xilinx:

```
XMD% run

RUNNING> XMD% Functional Correctness Test
... Reference FFT
... Optimized FFT
... Error accumulation
... Accumulated error: 0
... Error is within acceptable range.
Performance Test
... Reference FFT size  128   1785412  1785465  1785553  1785675
                              1785673  1785640  1785574  1785675
... Reference FFT size  256   3708992  3709033  3708953  3708963
                              3709069  3708963  3709069  3708963
... Reference FFT size  512   7652425  7652665  7652665  7652665
                              7652665  7652665  7652665  7652665
... Reference FFT size 1024   15759643 15759584 15759549 15759584
                              15759549 15759584 15759549 15759584
... Optimized FFT size  128   1785902  1785719  1785743  1786108
                              1785813  1785813  1786108  1785813
... Optimized FFT size  256   3709372  3709208  3709390  3709243
                              3709355  3709243  3709355  3709243
... Optimized FFT size  512   7653925  7653623  7653960  7653588
                              7653862  7653623  7653960  7653588
... Optimized FFT size 1024   15761271 15761102 15761154 15761154
                              15761154 15761154 15761154 15761154
     N        REF        OPT SPEEDUPx100
   128    1785583    1785877          99
   256    3709000    3709301          99
   512    7652635    7653766          99
  1024   15759578   15761162          99
```

The performance test measures the reference FFT and the optimized FFT for four different sizes (128, 256, 512, 1024 sampels). Each FFT is measured 8 times, and the resulting cycle count is the average of these measrements. The speedup is measured as the ratio of 100 times the reference cycle count divided by the optimized cycle count.

On Altera, you will see the following output:

```
Functional Correctness Test
... Reference FFT
... Optimized FFT
... Error accumulation
... Accumulated error: 0
... Error is within acceptable range.
Performance Test
... Reference FFT size  128   3285454  3159913  3159371  3180393
                              3159561  3159699  3159513  3159513
... Reference FFT size  256   6774687  6734040  6745421  6733883
```

```
                                6733569  6734213  6733597  6734109
... Reference FFT size  512     14278582 14246954 14246997 14246709
                                14246997 14246734 14247041 14246903
... Reference FFT size 1024     30040310 30008968 30008654 30008654
                                30008654 30008594 30008899 30008428
... Optimized FFT size  128     3160064  3160078  3209700  3160425
                                3171855  3160155  3159915  3179741
... Optimized FFT size  256     6734270  6775085  6734639  6745821
                                6734512  6746151  6734304  6745881
... Optimized FFT size  512     14279344 14247429 14247322 14247346
                                14247542 14247620 14247346 14247542
... Optimized FFT size 1024     30040408 30009162 30008987 30009598
                                30009686 30009342 30009342 30009342
     N        REF        OPT SPEEDUPx100
   128    3177927    3170241         100
   256    6740439    6743832          99
   512   14250864   14251436          99
  1024   30012645   30013233          99
```

How to start

On Scholar, download the baseline reference implementation, for Xilinx or Altera. Use your experience of previous Homework to implement the reference design. Execute the testbench and verify that you can observe a similar output as shown above. Then, start optimizing your design!

Hints and Tips

- If you build a coprocessor, it is highly recommended to **construct a cosimulation model** of your design using GEZEL. While you can develop coprocessor hardware directly in VHDL, it will require you to take care of many details at once. Going through cosimulation first enables you to test your idea before taking it to the board.
- Also, when developing hardware, **first test your ideas on 'small' designs.** When the low level components work fine, next verify how well it scales up.
- Also, carefully **consider tradeoffs**. There are obviously many more implementation alternatives than you have time to try out. Thus, you will have to think before you implement, and to experiment to find the largest acceleration as quickly as possible.
- Always **focus on the bottleneck** in the overall system. Remember the earlier examples we discussed. Hardware parallelism is useless if you cannot provide data sufficiently fast.
- Also, **make use of your homework assignments/solutions** to see examples on how a memory-mapped interface or an FSL interface can be created.

What to turn in

By the deadline, post the following information on Scholar.

- A report, ***username_report.pdf***, that summarizes the main characteristics of your design. Mention the cycle count of the resulting implementation, and the area (LUT/LE + registers).
    - In the report, provide a block-diagram of the optimized system.
    - In the report, include a screenshot of the optimized design as it runs through the testbench (similar to the output shown in the section "Testbench Output on Xilinx/Altera").
- If you developed a cosimulation model in GEZEL, provide the cosimulation model (C driver and FDL file). Call the model ***username_cosimulation.fdl***
- The optimized implementation in XPS or Quartus. Zip the project directory as ***username_project.zip***. Make sure you clean your project before turning it in (remove binaries, intermediate compilation files, etc).

Grading

The ranked performance of your design will count for 50% of the grade. The other 50% of the grade will be determined by the quality of the report and the quality of the optimized design (eg coprocessor design, use of cosimulation, etc).

This is your chance to explore new ideas and to try out what you have learned in this class! We will discuss the design in detail in the class of 1 December.

Good luck!