For this lab, you will implement serial-port communication on an FPGA to load an image from the computer, compute a 2D FFT (a discrete-time version of a Fourier transform), and transfer the results back to the computer. You will need to use the Xilinx FFT cores, on-chip memory, and serial UARTs. You will need to design a state-machine to control these components. This time, you will need to synthesize the design and have it working in hardware. In this lab, the size of the image to be transformed is variable and must be sent as the first byte from the computer. The image sizes are restricted to $N \times N$ where $N$ is a power of two and the size is sent as $\log_2 N$.

Figure 1: Hardware. Image data should be sent to the FPGA though the serial port. The FPGA should compute the 2D FFT result and send the result (real and imaginary components) back to the computer through the serial port. For the simulation part of the project, you can implement a just an 8x8 size computation to speed simulation. In this lab, the size of the image to be transformed is sent as the first byte from the computer.

1 Description

I provided some in-class explanation of this project and the 2-D FFT computation and will not repeat everything here.

A 2-D FFT is computed by performing an FFT transformation independently on each of the the rows, storing the intermediate results, and then performing an FFT transformation on each of the columns of the intermediate image results (image is also generic term for two-dimensional data). The data can be streamed in one row at a time: left to right scanning of each row with top row going first. Memory is required to store the full results of the row-transformed data so that the data can be fed into the next FFT operation in a different order, one column at a time: top to bottom with left-most column first. The results after the column transforms are final and may be sent back to computer immediately. You should alternate the real an imaginary data words when it is sent back to the computer: 16-bits real, 16 bits imaginary.
Figure 2: 2D FFT computation. The image is scanned in one row at a time. THE FIRST FFT OPERATION MUST BE PERFORMED ON EACH ROW INDEPENDENTLY. Intermediate results are stored until all row calculations are completed. The intermediate results are then fed column-wise into the second. EACH COLUMN FFT MUST BE COMPUTED INDEPENDENTLY. Each FFT requires real and imaginary data in and outputs real and imaginary data. Since the input is real data, zero may be sent into the imaginary data input for the first FFT.

Of key importance is the storage and retrieval of the intermediate results. The 2-D image data will be stored in an ordered memory that for practical purposes is a 1-D array. It would be natural to store the first row followed by the second row and so on until the last row is stored. Here is piece that may be confusing: for the second FFT, the intermediate 2D image must be accessed one logical column at a time. This means the first 1024-point set of data comes from the first points of each of the 1024 stored rows. The second set of data comes from all the second points. The exact indexing depends on how you stored your data.

Here is another way to describe the memory access. Consider all of the \( N^2 \) points (each point is represented by a real and an imaginary result) written to memory to be assigned an index according to the order they were written. That means the writes to memory correspond to indices \( 0,1,2,\ldots,(N-1) \) for the first vector, then \( N+1, N+2, N+3,\ldots,N+(N-1) \) for the second vector, then \( 2N+1, 2N+2, 2N+3,\ldots,2N+(N-1) \) for the third vector, and so on. The order they are read is \( 0, N, 2N, \ldots,(N-1)N \) for the first vector, then \( 1, N+1, 2N+1,\ldots,1+(N-1)N \) for the second vector, then \( 2, 2+N, 2+2N,\ldots,2+(N-1)N \) for the third vector, and so on until the read for the last vector, \( (N-1), (N-1)+N, (N-1)+2N,\ldots,(N-1)+(N-1)N \). So, the writes always increase by 1 within a vector and the start of each vector is \( 0, N, 2N,\ldots,(N-1)N \); whereas the read indices skip by \( N \) within each vector and the start of each read vector is \( 0, 1, 2, 3,\ldots,N-1 \).

Figure 3: Register Data Flow. Each row should be sent through the FFT and the results should be stored in memory. There are \( N \) rows each with \( N \) points for a total of \( N^2 \) points. Then the results must be fed back through the FFT module before being sent back to the computer. The trick is to read the data out of memory in the right order. Note that I added a multiply by a constant \( S \) to compensate for the fact that the “scaled” FFT option we use reduces the output by a factor \( \frac{1}{N} \). A value of 128 should work well (too large and we overflow 16-bit representation).
2 Design Approach and Requirement Details

You will need to use the Xilinx IP Core Generator to generate 1-D FFT module and an external memory interface. Download a serial, unbuffered UART from the internet (cite source). Write a case-statement-based state machine to control the modules and data flow.

- **Interface Summary**
  - We will only work with square images, but you need to support a variable size $N \times N$ where $N$ is a power of two and no greater than 1024.
  - Computer starts by sending the size of the image as a byte $\log_2 N$. Computer then sends the 2D data, one byte per pixel, scanning the rows from left to right starting with the top row.
  - Computer waits to receive results, which should be $N \times N \times 2$ 16-bit words ($N \times N \times 2 \times 2$ bytes). Each pixel should be sent as a contiguous 4 bytes where the bytes are [High-Byte Real, Low-Byte Real, High-Byte Imaginary, Low-Byte Imaginary]. The pixel data is expected to be sent corresponding to scanning the image columnwise (leftmost column first starting from top, then second column, etc...)

- **FFT**
  - Use Xilinx core generator “FFT” module
  - Generate a 1024-point capable FFT, but check the run-time size configuration option so that you can change the size to implement a smaller transform for shorter simulation.
  - Check the clock enable option so that you can control the rate of data flow
  - Use the “Natural” ordering option to make the coefficients come out in a more manageable order (0,1,2,3,...)
  - You’ll only need to instantiate of them.

- **PLL** (nevermind, PLL is not available on the FPGA we are using)
  - Use the PLL to generate as many clocks as you need.

- **Memory**
  - Use the external RAM. Use the posted document and your TA for instructions on how to generate the memory interface block and simulate it in your design. Note: A model of the external RAM itself is generated with the memory interface for the purpose of simulation.

- **UART** (universal asynchronous receiver-transmitter)
  - Download a Verilog implementation of an unbuffered UART from the internet (cite source in report)
    - specs: 8-bit data, no parity bit, no hardware or software flow control
  - Do not need hardware-handshaking signals, only RX and TX will be used

- **Control and misc.**
  - The center of control should be a case-statement-based state machine
  - Extra components or registers may be generated separately if needed/desired
  - For simulation, use an image of at least size 8x8.
  - The input data word size is 8 bits.
  - The output should be sent back by alternating real and imaginary words, 16-bit words sent big-endian.

- **Testbenches**
  - Generate a testbench to test an image size of at least 8x8
  - Image data should be read from disk and results should be written to disk
  - I can and will provide you with test data and a way to verify results in Matlab or other environments as requested.
3 What to turn in

- Design files including a “top” module implemented in Verilog NOT SCHEMATIC.
- Top module should only have pins rx, tx, clk_50mhz, and reset (I’ll keep you posted if this changes)
  - Adherence to this is required for the TA to test your design and give you points for correct implementation.
- Testbench and input files you used to test your design
- Report including design overview, design choices, and verification
- Bit file so that the TA can test your design in hardware