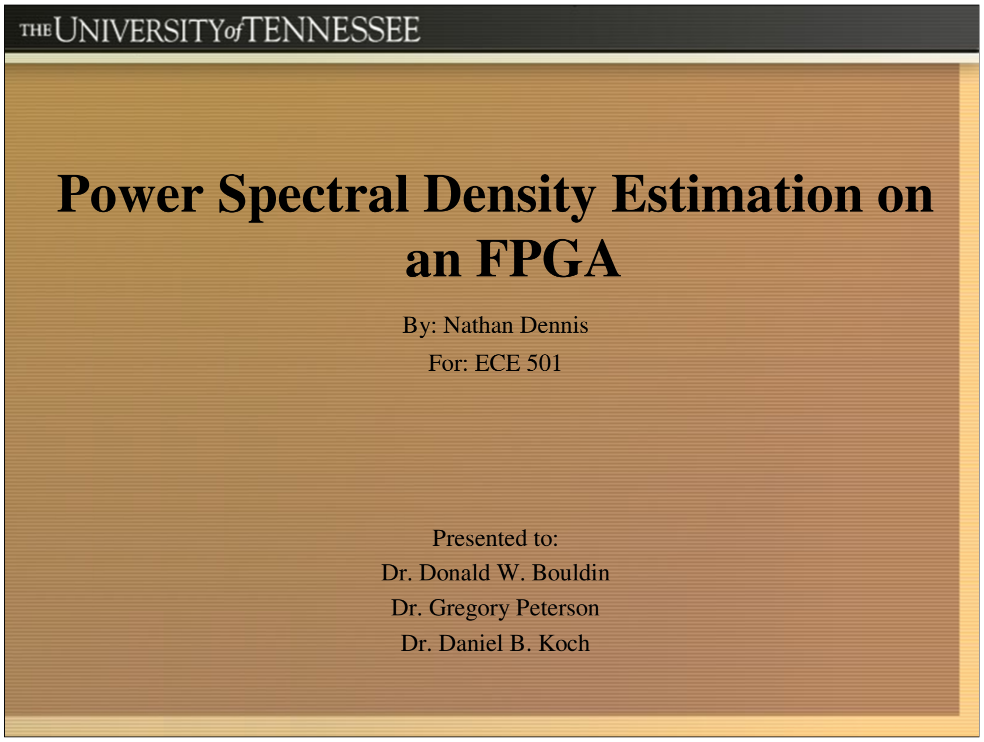


THE UNIVERSITY of TENNESSEE

Power Spectral Density Estimation on an FPGA

By: Nathan Dennis

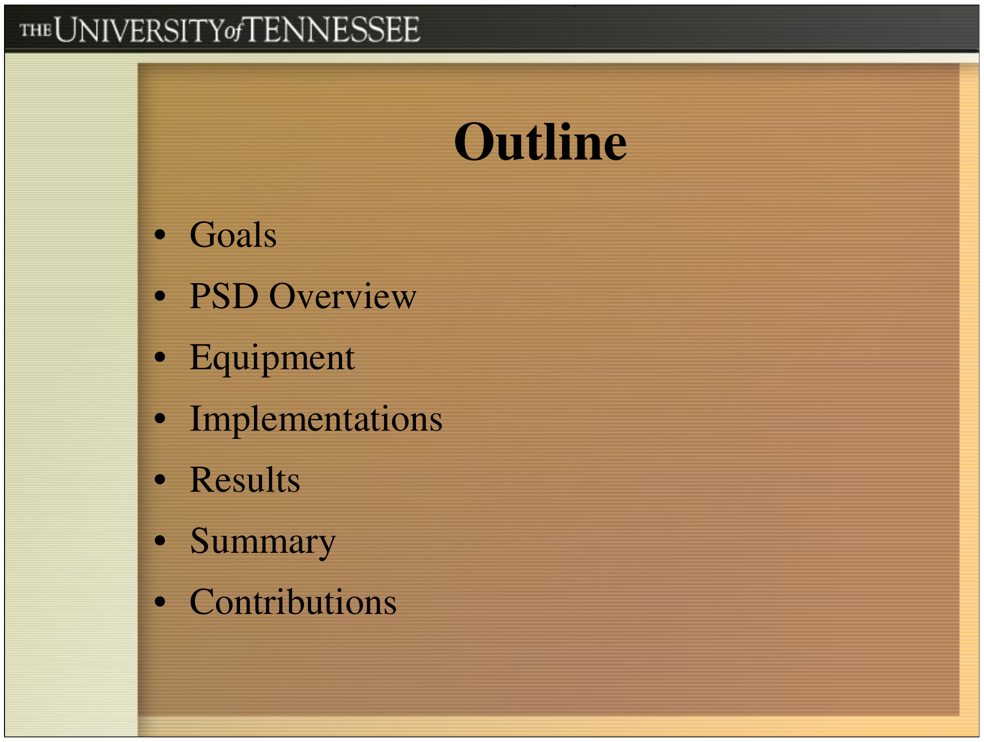
For: ECE 501

Presented to:

Dr. Donald W. Bouldin

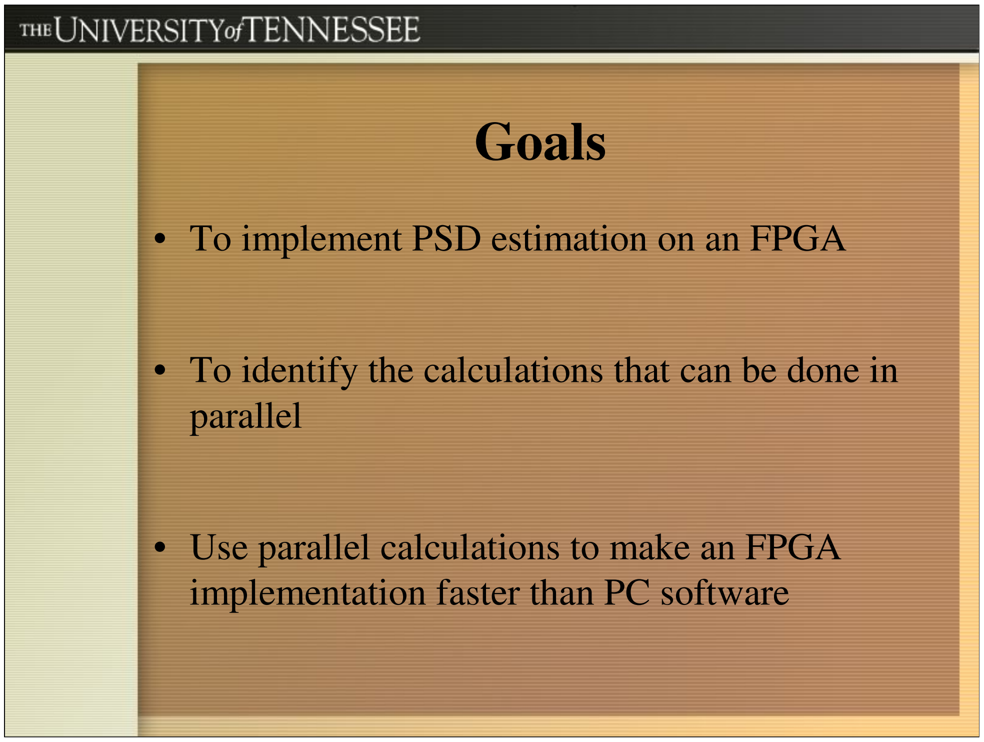
Dr. Gregory Peterson

Dr. Daniel B. Koch



Outline

- Goals
- PSD Overview
- Equipment
- Implementations
- Results
- Summary
- Contributions



THE UNIVERSITY of TENNESSEE

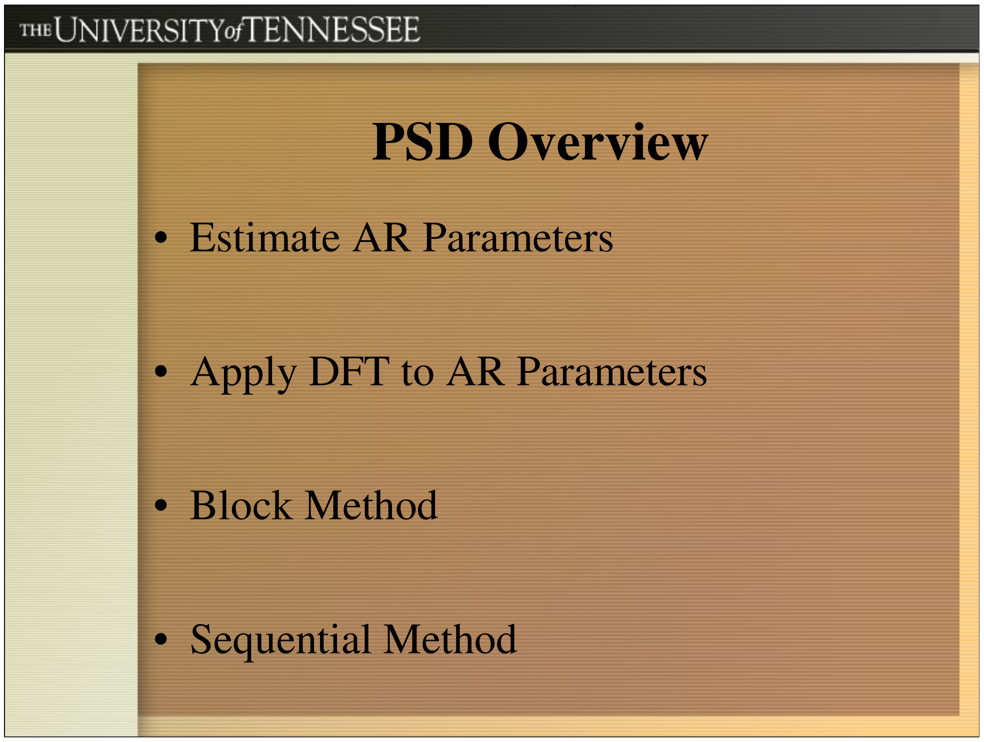
Goals

- To implement PSD estimation on an FPGA
- To identify the calculations that can be done in parallel
- Use parallel calculations to make an FPGA implementation faster than PC software

I have studied two major areas of computer engineering in graduate school. One was digital signal processing and the other was digital system design. I thought that it would make an interesting project for ECE 501 to somehow combine the two.

In my digital system design classes I learned how an FPGA can be used to do numerous calculations in parallel which gives it a performance advantage over ordinary CPUs found in today's PCs. In my digital signal processing classes I noticed that the FFT, in particular the radix-2 algorithm, provided a way to calculate a DFT using many calculations that are independent of each other, and so could be done in parallel. I thought that this would be a great candidate for implementation on an FPGA.

I chose the implement PSD estimation instead since I assumed that there are already many FPGA implementations of the FFT alone. PSD estimation requires calculation of a set of coefficients that characterize a random signal. Then the FFT is applied to those coefficients to produce a PSD.



THE UNIVERSITY of TENNESSEE

PSD Overview

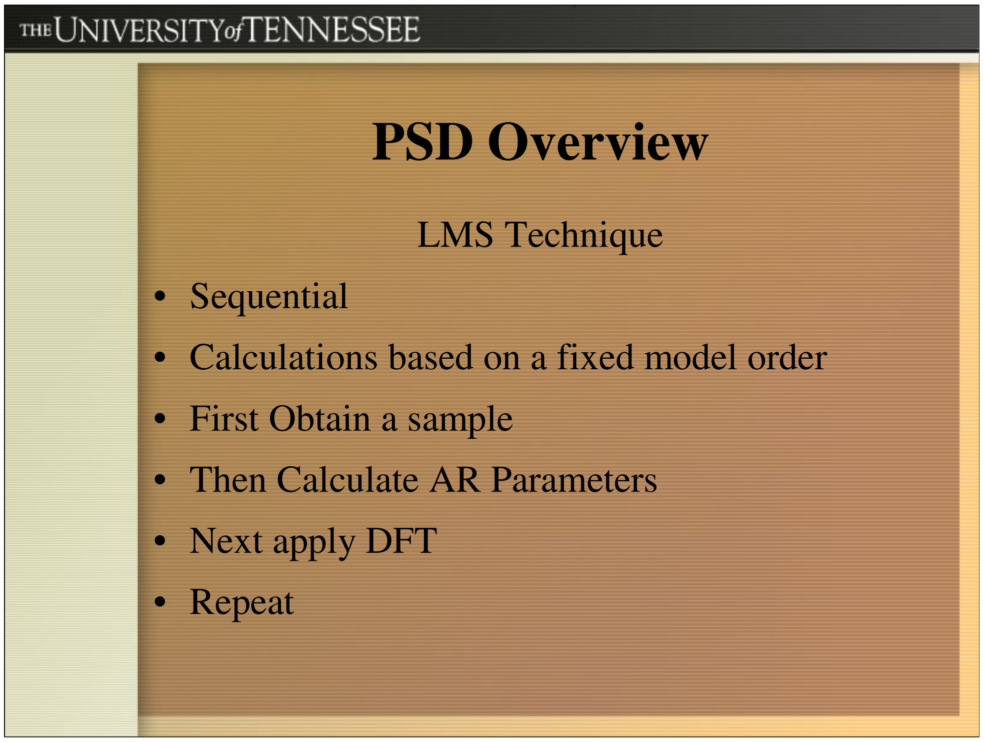
- Estimate AR Parameters
- Apply DFT to AR Parameters
- Block Method
- Sequential Method

One way to obtain the PSD for a random signal is to estimate its AR or autoregressive parameters. These parameter estimates can be used as coefficients in a digital filter. If the estimates were good then if white noise is the input to this filter, a random signal with similar statistical properties as the original random signal would be the output. Of course we do not intend to construct this filter, the purpose of estimating these coefficients is to obtain a model for the random signal.

After these coefficients are obtained then the DFT is applied to them. This produces the PSD.

Some PSD algorithms require the entire data set to be available. The AR parameters are estimated once, and the DFT is applied once. These are known as block methods.

The algorithm I chose is known as a sequential method. Here a new set of AR parameters is estimated each time new data is made available.


THE UNIVERSITY of TENNESSEE

PSD Overview

LMS Technique

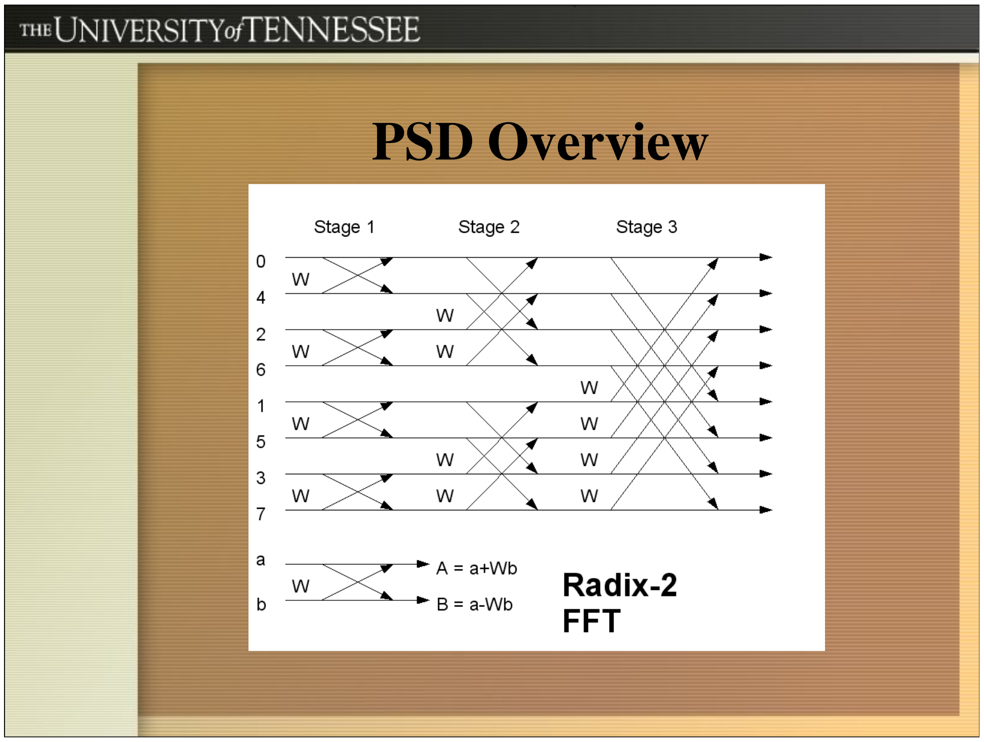
- Sequential
- Calculations based on a fixed model order
- First Obtain a sample
- Then Calculate AR Parameters
- Next apply DFT
- Repeat

The LMS or least mean square technique for AR parameter estimation is a sequential method which works by minimizing the mean square error. Where the error is the difference between the desired signal and the actual signal. Initially an AR parameter estimation is made based on the first sample. Then when the next sample is received it is compared with the value that the previous AR parameter estimation produced. This is the error, and a new AR parameter estimation is made that reduces the mean square error. This process is repeated with each new sample. The estimate becomes more accurate with each new sample.

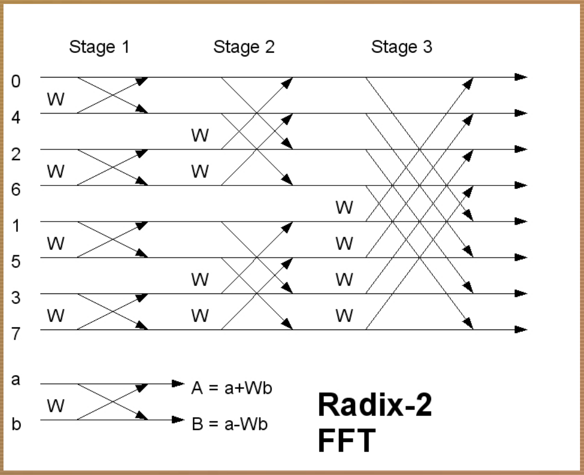
The speed at which this estimation converges depends on the step size. This number must be chosen and placed in the algorithm prior to evaluation.

Another requirement for this algorithm is knowledge of the filter length, needed to model the random signal. This is the number of AR parameters per estimation. Different methods exist for estimating the filter length. I used Matlab and adjusted the filter length until a PSD was produced that looked acceptable.

Each time a new sample is made available the LMS algorithm is applied to obtain a new estimate of the AR parameters. The DFT is also applied to each new set of AR parameters. So thirty samples requires thirty LMS calculations and thirty DFTs.

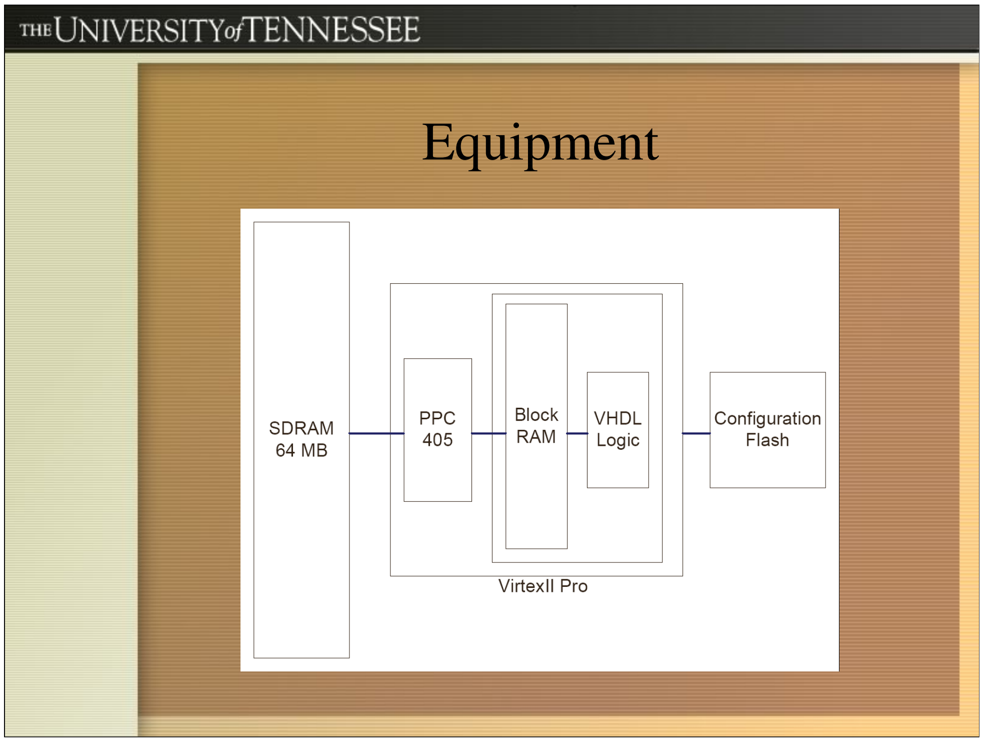

THE UNIVERSITY of TENNESSEE

PSD Overview

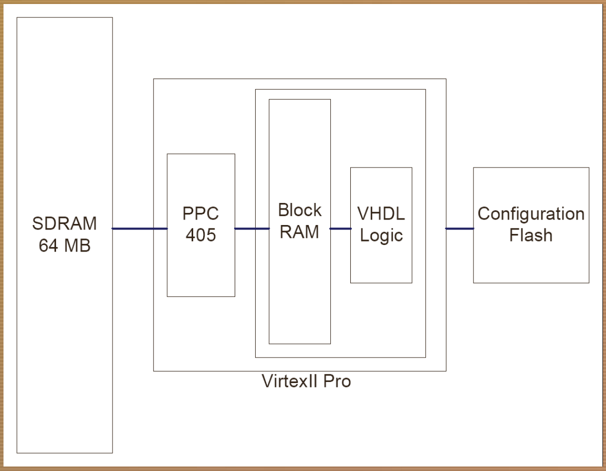


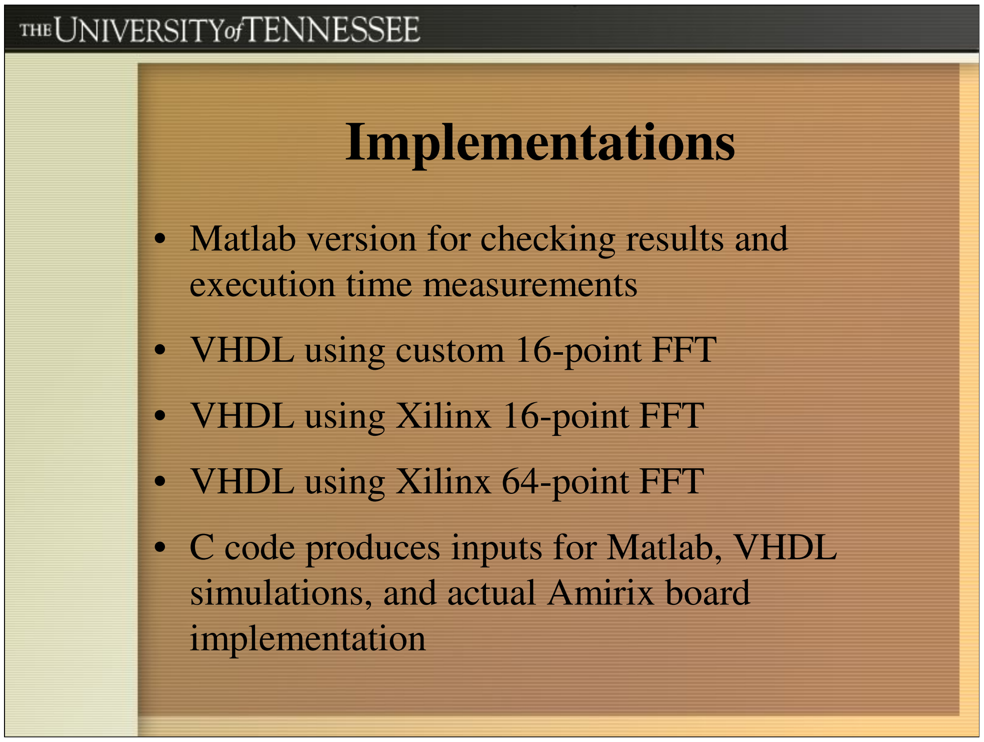
This is a diagram of the Radix-2 FFT algorithm. It calculates an 8-point FFT. The numbers on the left of the diagram show the indexes of the data that go in. Notice that they are not in order. They are in bit reversed order. This order is obtained by taking each index as if they were in order, converting the indices to binary, reversing the binary numbers, and converting back to decimal. For instance, assuming indices 0-7 we would want the second line to have input index 1. In binary this is 001. Reverse the binary and we get 100. This is 4 in decimal. So the second line requires input 4. The order is different for varying FFT sizes.

Also notice that this FFT has 3 stages. A 16-point FFT will have four stages. If you multiply the FFT size by two you add one more stage to the computation. Each stage is made up of what are called butterfly operations. Each butterfly operation has its own twiddle factor denoted by the variable W. A basic butterfly operation is shown at the bottom of the diagram. Notice that since all of the butterfly operations contained in one stage are independent of each other they could be calculated simultaneously. This will be attempted in VHDL.


THE UNIVERSITY of TENNESSEE

Equipment




THE UNIVERSITY of TENNESSEE

Implementations

- Matlab version for checking results and execution time measurements
- VHDL using custom 16-point FFT
- VHDL using Xilinx 16-point FFT
- VHDL using Xilinx 64-point FFT
- C code produces inputs for Matlab, VHDL simulations, and actual Amirix board implementation

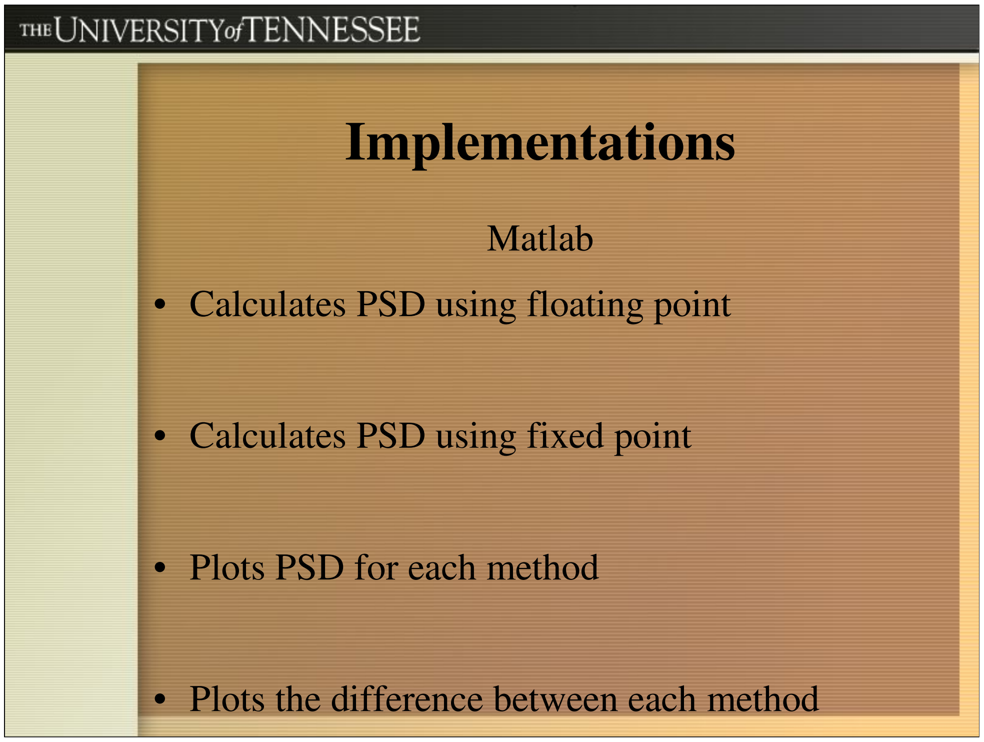
I decided to use Matlab for the software implementation of my project. This implementation would be used to make time measurements for software execution of the algorithm as well as for verifying that the VHDL results were correct. All calculations in Matlab are done in double precision floating point which make the results it produces ideal for checking VHDL accuracy.

I produced three different VHDL implementations. One uses custom 16-point FFT code that I wrote myself. This FFT is computed using the Radix-2 algorithm. All complex multiplications required for each individual stage are done simultaneously. All inputs to the FFT are applied simultaneously.

I created another implementation that uses a Xilinx 16-point FFT block for the purpose of comparing its performance with my FFT code. I also wanted to see how much space the Xilinx block would occupy on the chip, to see if higher order FFTs could be used.

After seeing that the Xilinx 16-point FFT used very little space, I then created a version using a Xilinx 64-point FFT block. This design fits on the chip and offers better results.

Matlab, VHDL simulation, and the actual Amirix board require the inputs do be in different forms. I wrote C code, that mixes a 100 Hz, 200 Hz, and 300 Hz, sine wave together. It uses a sample frequency of 1000. Each sine wave has a different amplitude. The 100 Hz signal was assigned .1. The 200 Hz signal was assigned .3, and the 300 Hz signal was assigned .5. The C program also takes other parameters, such as step size for the LMS algorithm, fixed point multiplier, filter length, and number of samples. All of this information is used to produce inputs for the floating point Matlab code, and the fixed point Matlab code. Fixed point data is also produced for the VHDL simulator and the Amirix board. New data sets for all can be made instantly by changing the parameters in the C code and executing again.


THE UNIVERSITY of TENNESSEE

Implementations

Matlab

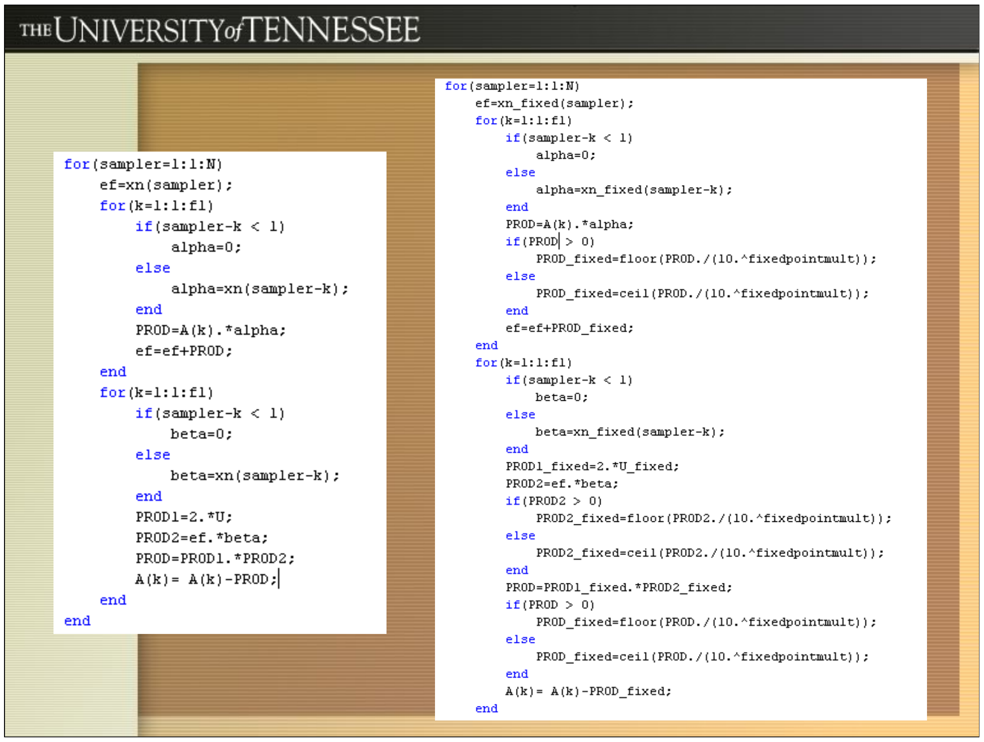
- Calculates PSD using floating point
- Calculates PSD using fixed point
- Plots PSD for each method
- Plots the difference between each method

I knew that my VHDL implementation would need to use fixed point arithmetic to do its calculations. So I wanted to first create a Matlab program that calculates the PSD using floating point arithmetic and then a fixed point version. This would allow me to measure the error introduced from using fixed point calculations. It would also allow me to choose the best fixed point multiplier.

I had previously made a Matlab program for the LMS technique in another class. I made some small changes to my original code to make it easier to port to VHDL. This included splitting all arithmetic operations so that one output is produced from two inputs. I also altered the loop indexing to make the algorithm easier to understand. After the changes I checked to make sure that the new LMS procedure gave the same outputs as the original. I used Matlab's built-in FFT function for the floating point version. I then added code to plot the result. This was my floating point implementation.

Next I developed a fixed point version by first looking at the LMS portion. The input is assumed to already be in fixed point form. The C program produces the fixed point inputs which are the floating point inputs multiplied by 10,000 and are all whole numbers. When there is a multiplication in the algorithm I must divide the product by 10,000. When complete this algorithm produces AR parameters that are the floating point AR parameters multiplied by 10,000 and any remaining decimal truncated.

I wrote custom FFT code in Matlab for the fixed point portion. I wrote it so that it would compute in a similar way that the VHDL would compute it, except nothing would be done in parallel. The built in Matlab FFT function takes a data set of any length and computes the FFT for that number of points. It does this by

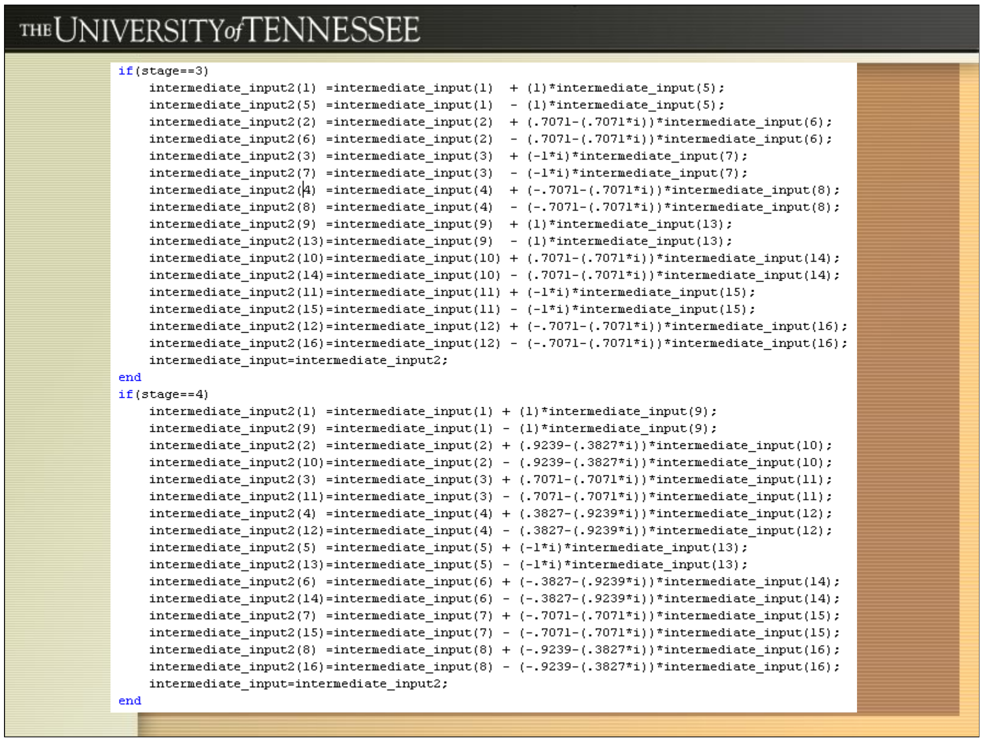


```
for(sampler=1:1:N)
    ef=xn(sampler);
    for(k=1:1:fl)
        if(sampler-k < 1)
            alpha=0;
        else
            alpha=xn(sampler-k);
        end
        PROD=A(k).*alpha;
        ef=ef+PROD;
    end
    for(k=1:1:fl)
        if(sampler-k < 1)
            beta=0;
        else
            beta=xn(sampler-k);
        end
        PROD1=2.*U;
        PROD2=ef.*beta;
        PROD=PROD1.*PROD2;
        A(k)= A(k)-PROD;
    end
end
```

```
for(sampler=1:1:N)
    ef=xn_fixed(sampler);
    for(k=1:1:fl)
        if(sampler-k < 1)
            alpha=0;
        else
            alpha=xn_fixed(sampler-k);
        end
        PROD=A(k).*alpha;
        if(PROD > 0)
            PROD_fixed=floor(PROD./(10.^fixedpointmult));
        else
            PROD_fixed=ceil(PROD./(10.^fixedpointmult));
        end
        ef=ef+PROD_fixed;
    end
    for(k=1:1:fl)
        if(sampler-k < 1)
            beta=0;
        else
            beta=xn_fixed(sampler-k);
        end
        PROD1_fixed=2.*U_fixed;
        PROD2=ef.*beta;
        if(PROD2 > 0)
            PROD2_fixed=floor(PROD2./(10.^fixedpointmult));
        else
            PROD2_fixed=ceil(PROD2./(10.^fixedpointmult));
        end
        PROD=PROD1_fixed.*PROD2_fixed;
        if(PROD > 0)
            PROD_fixed=floor(PROD./(10.^fixedpointmult));
        else
            PROD_fixed=ceil(PROD./(10.^fixedpointmult));
        end
        A(k)= A(k)-PROD_fixed;
    end
```

These are my floating and fixed point algorithms for the LMS technique. The outer loop iterates N times where N is the number of samples I intend to operate on. Throughout this project N=30. The variable fl stands for filter length. I chose this to be 16 for my particular data set. This filter order is higher than it should be for the data set I produced. I used this number because the maximum filter length possible in my VHDL implementation is 16, and I wanted to base my time measurements on using the highest possible filter order.

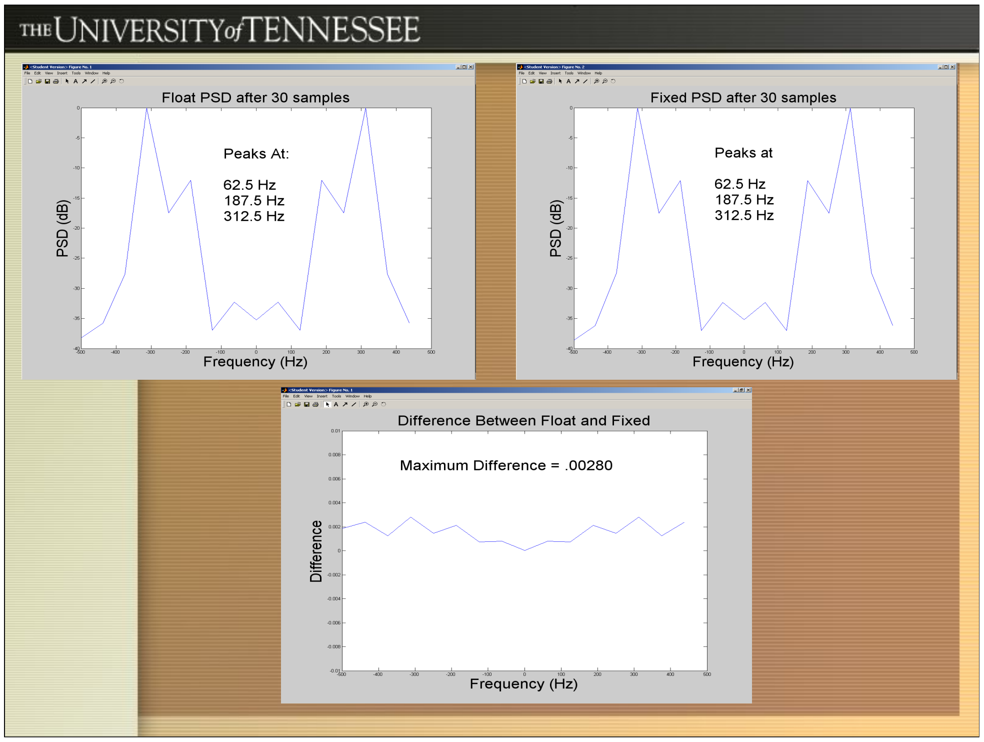
The step size for my algorithm is represented by the variable U, seen in the second inner loop. The arithmetical operations have been split up with regard to VHDL because most arithmetic done in VHDL requires that no more than two inputs be used for one operation.



THE UNIVERSITY of TENNESSEE

```
if(stage==3)
    intermediate_input2(1) =intermediate_input(1)  + (1)*intermediate_input(5);
    intermediate_input2(5) =intermediate_input(1)  - (1)*intermediate_input(5);
    intermediate_input2(2) =intermediate_input(2)  + (.7071-(.7071*i))*intermediate_input(6);
    intermediate_input2(6) =intermediate_input(2)  - (.7071-(.7071*i))*intermediate_input(6);
    intermediate_input2(3) =intermediate_input(3)  + (-1*i)*intermediate_input(7);
    intermediate_input2(7) =intermediate_input(3)  - (-1*i)*intermediate_input(7);
    intermediate_input2(4) =intermediate_input(4)  + (-.7071-(.7071*i))*intermediate_input(8);
    intermediate_input2(8) =intermediate_input(4)  - (-.7071-(.7071*i))*intermediate_input(8);
    intermediate_input2(9) =intermediate_input(9)  + (1)*intermediate_input(13);
    intermediate_input2(13)=intermediate_input(9)  - (1)*intermediate_input(13);
    intermediate_input2(10)=intermediate_input(10) + (.7071-(.7071*i))*intermediate_input(14);
    intermediate_input2(14)=intermediate_input(10) - (.7071-(.7071*i))*intermediate_input(14);
    intermediate_input2(11)=intermediate_input(11) + (-1*i)*intermediate_input(15);
    intermediate_input2(15)=intermediate_input(11) - (-1*i)*intermediate_input(15);
    intermediate_input2(12)=intermediate_input(12) + (-.7071-(.7071*i))*intermediate_input(16);
    intermediate_input2(16)=intermediate_input(12) - (-.7071-(.7071*i))*intermediate_input(16);
    intermediate_input=intermediate_input2;
end
if(stage==4)
    intermediate_input2(1) =intermediate_input(1) + (1)*intermediate_input(9);
    intermediate_input2(9) =intermediate_input(1) - (1)*intermediate_input(9);
    intermediate_input2(2) =intermediate_input(2) + (.9239-(.3827*i))*intermediate_input(10);
    intermediate_input2(10)=intermediate_input(2) - (.9239-(.3827*i))*intermediate_input(10);
    intermediate_input2(3) =intermediate_input(3) + (.7071-(.7071*i))*intermediate_input(11);
    intermediate_input2(11)=intermediate_input(3) - (.7071-(.7071*i))*intermediate_input(11);
    intermediate_input2(4) =intermediate_input(4) + (.3827-(.9239*i))*intermediate_input(12);
    intermediate_input2(12)=intermediate_input(4) - (.3827-(.9239*i))*intermediate_input(12);
    intermediate_input2(5) =intermediate_input(5) + (-1*i)*intermediate_input(13);
    intermediate_input2(13)=intermediate_input(5) - (-1*i)*intermediate_input(13);
    intermediate_input2(6) =intermediate_input(6) + (-.3827-(.9239*i))*intermediate_input(14);
    intermediate_input2(14)=intermediate_input(6) - (-.3827-(.9239*i))*intermediate_input(14);
    intermediate_input2(7) =intermediate_input(7) + (-.7071-(.7071*i))*intermediate_input(15);
    intermediate_input2(15)=intermediate_input(7) - (-.7071-(.7071*i))*intermediate_input(15);
    intermediate_input2(8) =intermediate_input(8) + (-.9239-(.3827*i))*intermediate_input(16);
    intermediate_input2(16)=intermediate_input(8) - (-.9239-(.3827*i))*intermediate_input(16);
    intermediate_input=intermediate_input2;
end
```

This is a portion of my custom fixed point FFT code. Nothing in this portion indicates that it is fixed point because it would be redundant in Matlab. Notice each line is the sum of a number and a product. Some of these products contain decimals. For these calculations to work in fixed point the decimal twiddle factors would need to be multiplied by 10,000. Then multiplied by its corresponding intermediate input. Then divided by 10,000. So rather than including a multiply and divide by 10,000, I left it out of the Matlab code, and only put it in the VHDL. The VHDL and Matlab results will still be the same.

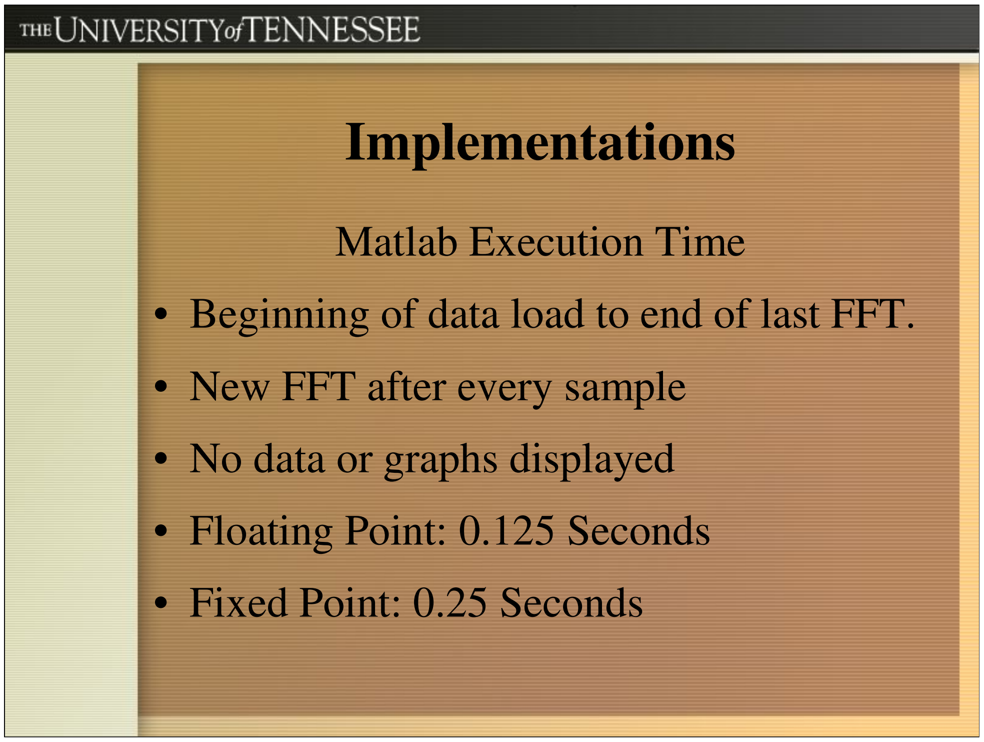


These three figures show results from Matlab code.

The upper left figure shows the floating point PSD. Its peaks are located at 62.5 Hz, 187.5 Hz, and 312.5 Hz. These do not match 100 Hz, 200, Hz, and 300 Hz exactly. That is because this is only an estimation. The peak at 62.5 Hz is a particularly bad estimation for the 100 Hz signal, but the 100 Hz signal had a very low amplitude compared to the others.

The fixed point version yielded peaks at the same frequency locations shown in upper right figure. The only differences are in height or y-values.

Next I wanted to better visualize the error introduced by the fixed point arithmetic. I restructured my code to save the PSD from the last sample in floating point and in fixed point. Then I subtracted these from each other. A plot of the result is show at the bottom. The maximum difference is 0.0028.


THE UNIVERSITY of TENNESSEE

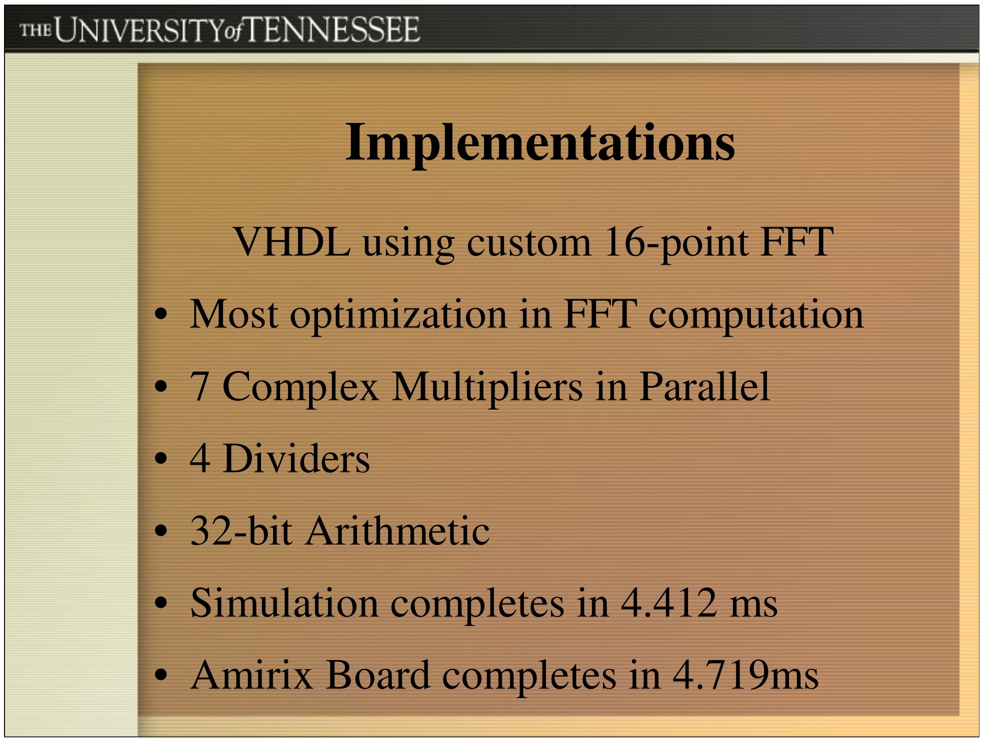
Implementations

Matlab Execution Time

- Beginning of data load to end of last FFT.
- New FFT after every sample
- No data or graphs displayed
- Floating Point: 0.125 Seconds
- Fixed Point: 0.25 Seconds

I wanted to measure the execution time of this algorithm in software. I removed the code that actually plots the data, and I disabled all output to the command line. This makes the software run faster. Then I placed Matlab commands in the code to measure elapsed time. I ran the code so that it would perform the LMS routine and 16-point FFT for each of 30 samples. I began the timer at the beginning of data loading, until the last FFT. The Floating point version takes 0.125 Seconds. The fixed point version takes 0.25 seconds. The fixed point version takes longer because all calculations in Matlab are done in double precision floating point and extra calculations must be done to convert the numbers to simulate fixed point.

This software was ran on my PC. It has a 2GHz AMD processor and 1GB of RAM.


THE UNIVERSITY of TENNESSEE

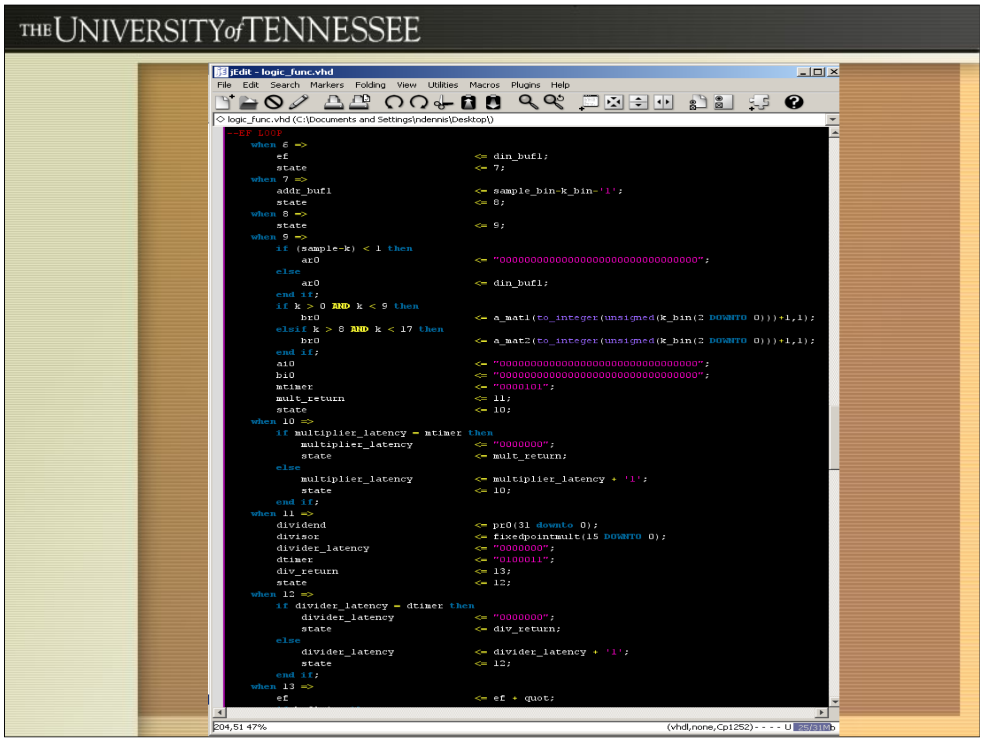
Implementations

VHDL using custom 16-point FFT

- Most optimization in FFT computation
- 7 Complex Multipliers in Parallel
- 4 Dividers
- 32-bit Arithmetic
- Simulation completes in 4.412 ms
- Amirix Board completes in 4.719ms

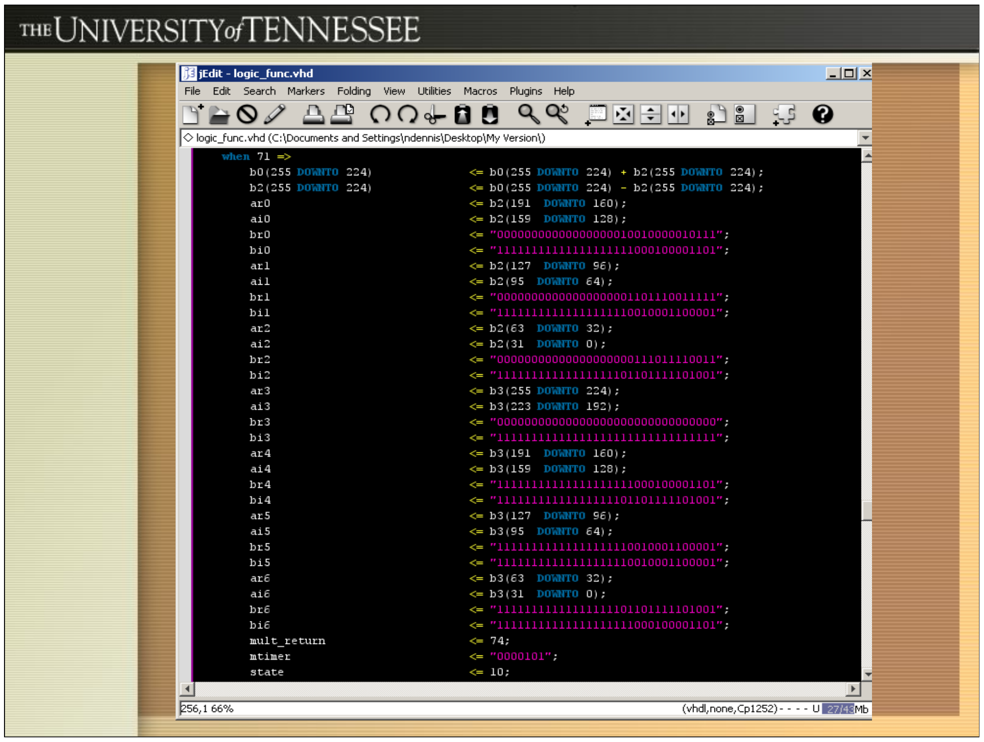
For this implementation I chose to use Xilinx divider and complex multiplier blocks. I began by writing custom code for a 64-point FFT. It was similar to the Matlab algorithm. I hard coded all twiddle factors and butterfly operations. There were enough complex multiplier blocks to do all butterfly operations in each stage simultaneously. This design simulated successfully, but the design was too big to fit on the actual FPGA. I tried many different thing to try to fit this design on the FPGA. I reduced the number of complex multiplier blocks, reduced the number of dividers, and reduced the maximum filter length size. None of these worked, so I reduced it to a 32-point FFT. It simulated successfully, but again it would not fit. I went through the same routine to make it fit, I even tried breaking the VHDL into smaller pieces. The design would pass the mapper, and placement, but there were too many signals for it to route.

I finally decided to go with a 16-point FFT. This design fit and I was able to use as many complex multipliers as I needed. I also included dividers for meeting the fixed point requirement. I multiplied all twiddle factors by 10,000 and then hard coded these values in the VHDL. The dividers are used after the twiddle factors are multiplied by their corresponding intermediate inputs. The dividers occupy a lot of space.


THE UNIVERSITY of TENNESSEE
jEdit - logic_func.vhd
File Edit Search Markers Folding View Utilities Macros Plugins Help
logic_func.vhd (C:\Documents and Settings\ndennis\Desktop\)

```
--EF LOOP
    when 6 =>
        ef                              <= din_bufl;
        state                           <= 7;
    when 7 =>
        addr_bufl                       <= sample_bin-k_bin-'1';
        state                           <= 8;
    when 8 =>
        state                           <= 9;
    when 9 =>
        if (sample-k) < 1 then
            ar0                         <= "00000000000000000000000000000000";
        else
            ar0                         <= din_bufl;
        end if;
        if k > 0 AND k < 9 then
            br0                         <= a_mat1(to_integer(unsigned(k_bin(2 DOWNTO 0)))+1,1);
        elsif k > 8 AND k < 17 then
            br0                         <= a_mat2(to_integer(unsigned(k_bin(2 DOWNTO 0)))+1,1);
        end if;
        ai0                             <= "00000000000000000000000000000000";
        bi0                             <= "00000000000000000000000000000000";
        mtimer                          <= "0000101";
        mult_return                     <= 11;
        state                           <= 10;
    when 10 =>
        if multiplier_latency = mtimer then
            multiplier_latency          <= "0000000";
            state                       <= mult_return;
        else
            multiplier_latency          <= multiplier_latency + '1';
            state                       <= 10;
        end if;
    when 11 =>
        dividend                        <= pr0(31 downto 0);
        divisor                         <= fixedpointmult(15 DOWNTO 0);
        divider_latency                 <= "0000000";
        dtimer                          <= "0100011";
        div_return                      <= 13;
        state                           <= 12;
    when 12 =>
        if divider_latency = dtimer then
            divider_latency             <= "0000000";
            state                       <= div_return;
        else
            divider_latency             <= divider_latency + '1';
            state                       <= 12;
        end if;
    when 13 =>
        ef                              <= ef + quot;
```

Here is a sample from my LMS code. I would like to note a few things. In state nine the two arrays that hold the AR parameters can be seen. They are named a_mat1 and a_mat2. Each can hold eight 32-bit numbers. That is why the maximum filter length my design can hold is 16. I had to define this array earlier in the code. I made the array small so I could easily add and remove arrays to change the maximum filter length. Arrays had to be removed in order to fit the design on the chip.


THE UNIVERSITY of TENNESSEE

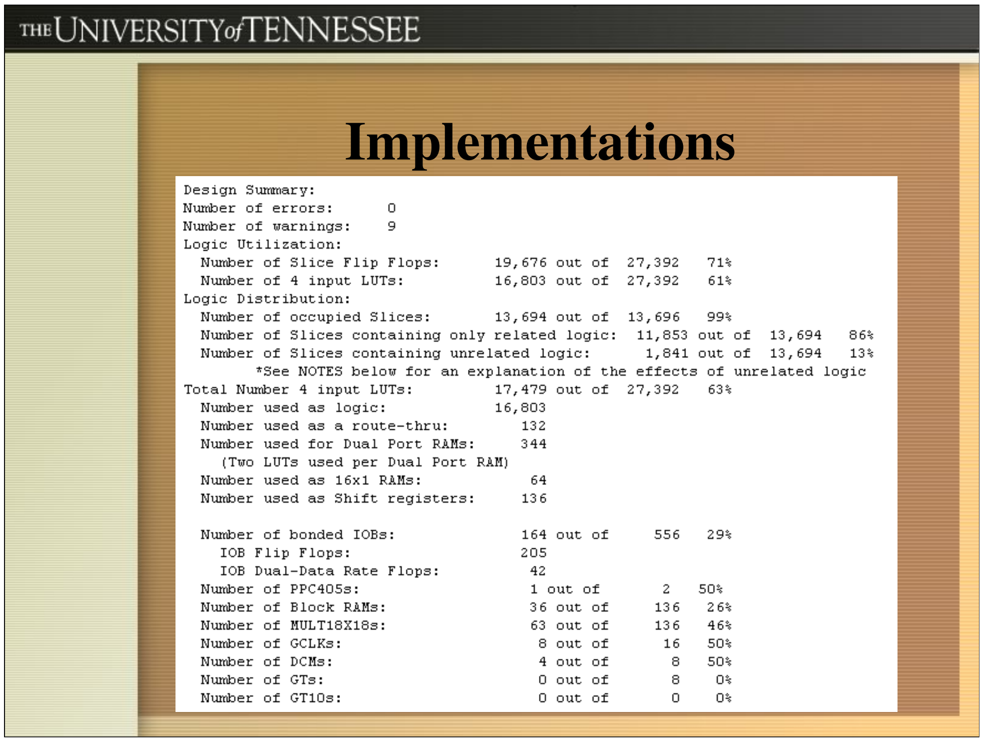
jEdit - logic_func.vhd
File Edit Search Markers Folding View Utilities Macros Plugins Help
◇ logic_func.vhd (C:\Documents and Settings\ndennis\Desktop\My Version\)

```
    when 71 =>
        b0(255 DOWNTO 224)              <= b0(255 DOWNTO 224) + b2(255 DOWNTO 224);
        b2(255 DOWNTO 224)              <= b0(255 DOWNTO 224) - b2(255 DOWNTO 224);
        ar0                             <= b2(191  DOWNTO 160);
        ai0                             <= b2(159  DOWNTO 128);
        br0                             <= "00000000000000000010010000010111";
        bi0                             <= "11111111111111111111000100001101";
        ar1                             <= b2(127  DOWNTO 96);
        ai1                             <= b2(95  DOWNTO 64);
        br1                             <= "00000000000000000001101110011111";
        bi1                             <= "11111111111111111110010001100001";
        ar2                             <= b2(63  DOWNTO 32);
        ai2                             <= b2(31  DOWNTO 0);
        br2                             <= "00000000000000000000111011110011";
        bi2                             <= "11111111111111111101101111101001";
        ar3                             <= b3(255 DOWNTO 224);
        ai3                             <= b3(223 DOWNTO 192);
        br3                             <= "00000000000000000000000000000000";
        bi3                             <= "11111111111111111111111111111111";
        ar4                             <= b3(191  DOWNTO 160);
        ai4                             <= b3(159  DOWNTO 128);
        br4                             <= "11111111111111111111000100001101";
        bi4                             <= "11111111111111111101101111101001";
        ar5                             <= b3(127  DOWNTO 96);
        ai5                             <= b3(95  DOWNTO 64);
        br5                             <= "11111111111111111110010001100001";
        bi5                             <= "11111111111111111110010001100001";
        ar6                             <= b3(63  DOWNTO 32);
        ai6                             <= b3(31  DOWNTO 0);
        br6                             <= "11111111111111111101101111101001";
        bi6                             <= "11111111111111111111000100001101";
        mult_return                     <= 74;
        mtimer                          <= "0000101";
        state                           <= 10;
```

I wanted to maintain 32-bit arithmetic capability throughout my design so I defined my complex numbers as a 32-bit real part plus a 32-bit imaginary part. So each complex number is defined with 64-bits. In my original 64-point FFT I tried designating a 64-bit register for each of the FFT inputs. The resulting design could not be the routed. I then tried using Xilinx single port block RAMs. These use fewer logic slices. I used the maximum width, 256, which can store 4 complex numbers each. As I reduced the FFT size to 16-points I continued using the single port block RAMs. I believed though, at this size, I could switch back to using registers, and save some clock cycles. This idea worked and the design routed successfully. The easiest way for me to switch from the block RAMs to the registers was to maintain the 256-bit width. This slide shows a portion of my FFT code. You can see how b0 through b3, the registers used for storing intermediate values, are indexed from 255 to 0. Using four 256-bit registers is sufficient for keeping track of 16 complex numbers, which are 64-bits each.

This portion of code also shows how all seven complex multipliers are being used. The inputs to these multipliers are ar's, ai's, br's, and bi's numbered 0 through 6. The "r's" mean real part and the "i's" mean imaginary part. You can also see all of the twiddle factors, which have been multiplied by 10,000 prior to code entry. All of these factors have been assigned to the b inputs of the complex multipliers.

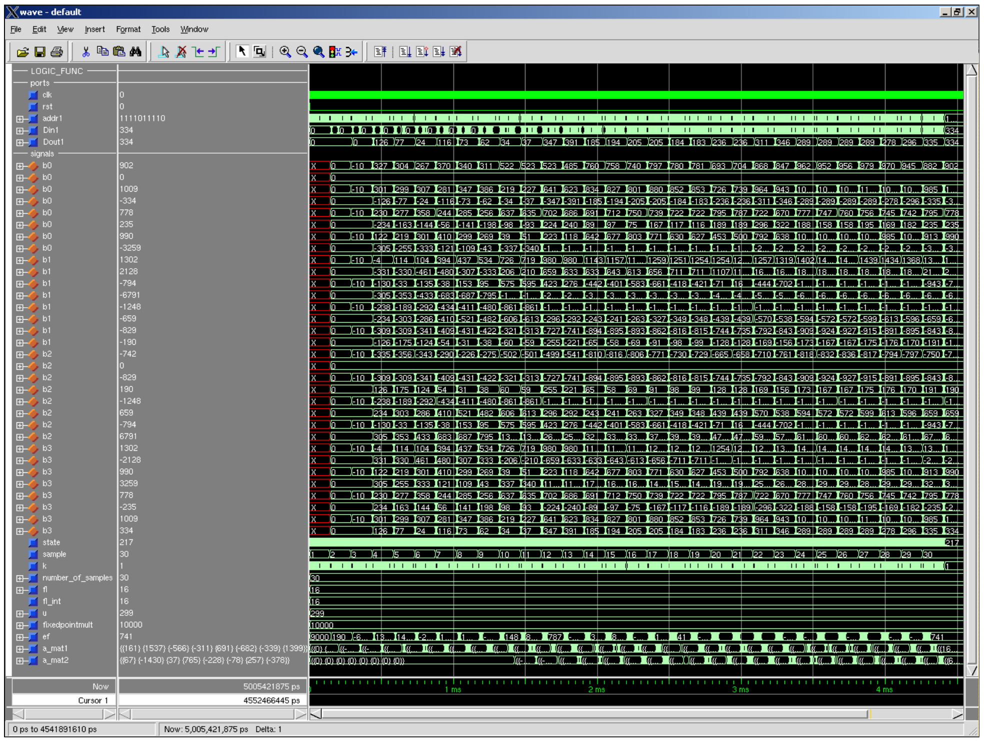
The variable at the bottom dubbed "mtimer" designates how many clock cycles must elapse before the multiplier output is ready. I adjusted this by running the design through the simulator. Similar adjustments had to be made in adjusting for the latency of the dividers.


THE UNIVERSITY of TENNESSEE

Implementations

```
Design Summary:
Number of errors:      0
Number of warnings:    9
Logic Utilization:
  Number of Slice Flip Flops:      19,676 out of  27,392   71%
  Number of 4 input LUTs:          16,803 out of  27,392   61%
Logic Distribution:
  Number of occupied Slices:       13,694 out of  13,696   99%
  Number of Slices containing only related logic:  11,853 out of  13,694   86%
  Number of Slices containing unrelated logic:      1,841 out of  13,694   13%
        *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:         17,479 out of  27,392   63%
  Number used as logic:            16,803
  Number used as a route-thru:        132
  Number used for Dual Port RAMs:     344
    (Two LUTs used per Dual Port RAM)
  Number used as 16x1 RAMs:            64
  Number used as Shift registers:     136

  Number of bonded IOBs:              164 out of     556   29%
    IOB Flip Flops:                   205
    IOB Dual-Data Rate Flops:          42
  Number of PPC405s:                   1 out of       2   50%
  Number of Block RAMs:               36 out of     136   26%
  Number of MULT18X18s:               63 out of     136   46%
  Number of GCLKs:                     8 out of      16   50%
  Number of DCMs:                      4 out of       8   50%
  Number of GTs:                       0 out of       8    0%
  Number of GT10s:                     0 out of       0    0%
```

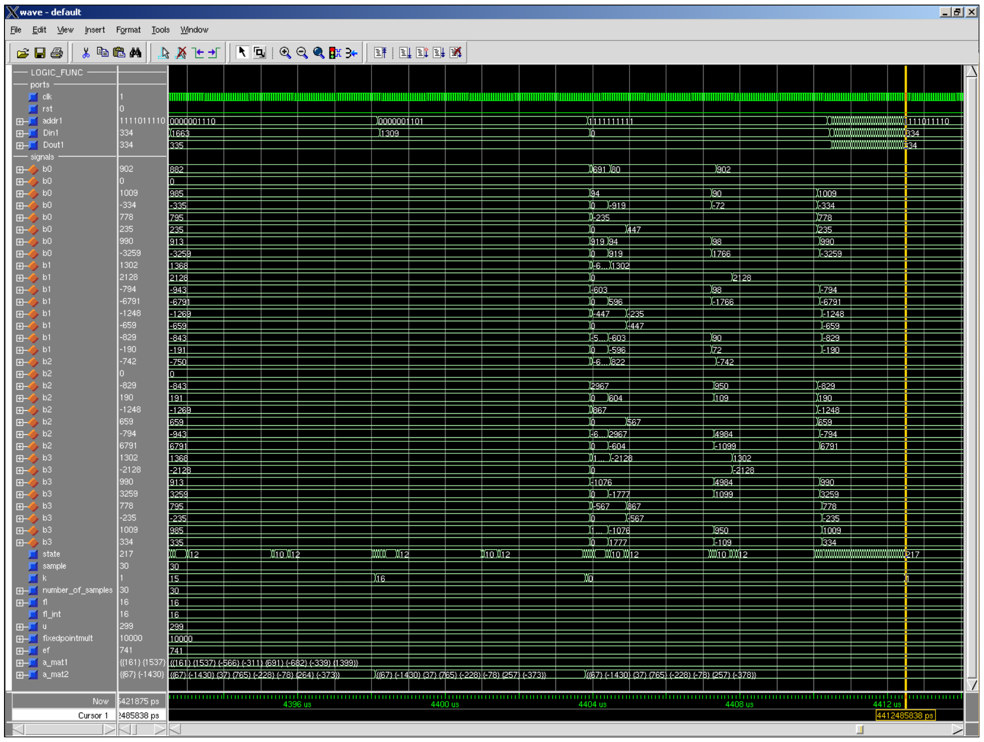
This is an excerpt from the log file produced by the Xilinx mapper. It shows how much of the FPGA’s resources my design uses. Throughout the development process, the number that was most important was the number of occupied slices. I encountered many overmapping errors when I was attempting higher order FFTs. When the design would overmap the number of occupied slices was usually the overmapped resource. I desired to add another divider to my current design which would save about 3 clock cycles per FFT computation, but this causes overmapping.



```
LOGIC_FUNC
ports
  clk                 0
  rst                 0
  addr1               1111011110
  Din1                334
  Dout1               334
signals
  b0                  902
  b0                  0
  b0                  1009
  b0                  -334
  b0                  778
  b0                  235
  b0                  990
  b0                  -3259
  b1                  1302
  b1                  2128
  b1                  -794
  b1                  -6791
  b1                  -1248
  b1                  -659
  b1                  -829
  b1                  -190
  b2                  -742
  b2                  0
  b2                  -829
  b2                  190
  b2                  -1248
  b2                  659
  b2                  -794
  b2                  6791
  b3                  1302
  b3                  -2128
  b3                  990
  b3                  3259
  b3                  778
  b3                  -235
  b3                  1009
  b3                  334
  state               217
  sample              30
  k                   1
  number_of_samples   30
  fl                  16
  fl_int              16
  u                   299
  fixedpointmult      10000
  ef                  741
  a_mat1              {{161} {1537} {-566} {-311} {691} {-682} {-339} {1399}}
  a_mat2              {{67} {-1430} {37} {765} {-228} {-78} {257} {-378}}

                Now   5005421875 ps
           Cursor 1   4552466445 ps

0 ps to 4541891610 ps      Now: 5,005,421,875 ps   Delta: 1
```

This slide displays the results of my VHDL simulation. Not much detail can be seen here but I wanted to show a general idea of what is happening. To the left you will see the names of some of the signals I decided to display on this simulation. They are divided in a "ports" section and a "signals" section. The "ports" section shows the signals that are used to connect with other blocks in my design. Here I have shown the ports used to connect with the block RAM, which contain all of the samples and will contain the PSD results. The clock signal is also shown, and is designated by "clk". The "signals" section displays events happening within my logic block. The signals with the orange diamond shape are the 256-bit registers showing the intermediate results of the FFT computations. They have been broken down into 32-bit real and 32-bit imaginary parts for clarity. Below the registers you can see the signal that keeps track of the current state. Other signals such as "number of samples", "fl", "u", and fixedpointmult" are parameters that are read in at the beginning of the execution and do not change throughout the code. The changing values of the AR parameters can be seen in "a_mat1", and "a_mat2".

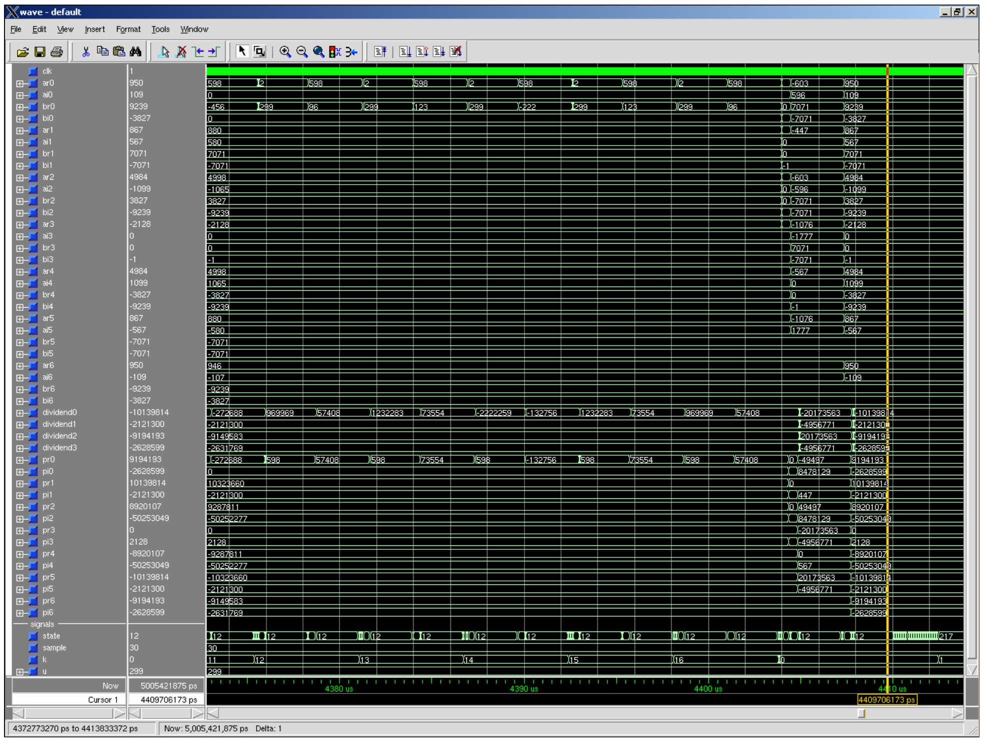
The clock speed is designated in my test bench. During development I chose a clock period of 100 nano seconds. After I found that my design worked on the board I used the clock period determined by the routing software in my test bench which was always less than 100 nano seconds. This was the best way I found to measure the execution time of the VHDL.



LOGIC_FUNC	
ports	
clk	1
rst	0
addr1	1111011110
Din1	334
Dout1	334
signals	
b0	902
b0	0
b0	1009
b0	-334
b0	778
b0	235
b0	990
b0	-3259
b1	1302
b1	2128
b1	-794
b1	-6791
b1	-1248
b1	-659
b1	-829
b1	-190
b2	-742
b2	0
b2	-829
b2	190
b2	-1248
b2	659
b2	-794
b2	6791
b3	1302
b3	-2128
b3	990
b3	3259
b3	778
b3	-235
b3	1009
b3	334
state	217
sample	30
k	1
number_of_samples	30
fl	16
fl_int	16
u	299
fixedpointmult	10000
ef	741
a_mat1	{{161} {1537}
a_mat2	{{67} {-1430}
Now	5421875 ps
Cursor 1	2485838 ps

This figure shows the FFT calculation for sample 30. You can see how the registers look like they have three different times for calculations. Those are for stages 2, 3 ,and 4 in the FFT. The first stage requires no complex multiplication and therefore can be seen within the same column as stage 2. Closer to the bottom you can see the state changes. States ten and twelve can clearly be seen repeating. State ten is used to meet the latency requirement of the multipliers. State 12 is used to meet the latency requirement of the dividers. The divider latency can be reduced if I use a Xilinx divider with smaller inputs, but I do not want to sacrifice that precision. To the left of the three columns in the FFT registers the results of the previous FFT from sample 29 can be seen. This portion is where the LMS routine is still calculating the AR parameters for sample 30. In fact you can see how the last few numbers in a_mat2 are updated just before the FFT begins. The k signal is used as a loop index in the LMS routine. Remember that my routine contains two loops whose number of iterations match the filter length. So each of the two loops use k from 1 to 16.

Notice at the left of the diagram there is a portion after the FFT where the state changes very rapidly. The signals labeled, “addr1”, “Din1”, and “Dout1” also change rapidly. This is where the VHDL writes the results of the FFT to the dual port block RAM, so they can eventually be transferred to the SDRAM on the Amirix board.

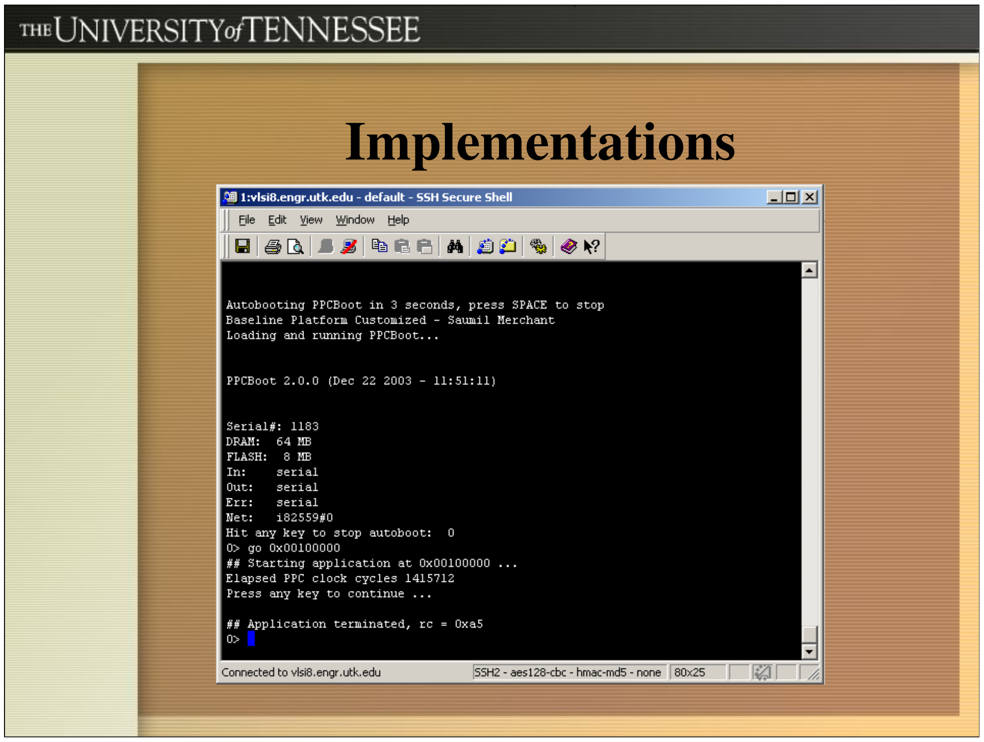


clk	1
ar0	950
ai0	109
br0	9239
bi0	-3827
ar1	867
ai1	567
br1	7071
bi1	-7071
ar2	4984
ai2	-1099
br2	3827
bi2	-9239
ar3	-2128
ai3	0
br3	0
bi3	-1
ar4	4984
ai4	1099
br4	-3827
bi4	-9239
ar5	867
ai5	-567
br5	-7071
bi5	-7071
ar6	950
ai6	-109
br6	-9239
bi6	-3827
dividend0	-10139814
dividend1	-2121300
dividend2	-9194193
dividend3	-2628599
pr0	9194193
pi0	-2628599
pr1	10139814
pi1	-2121300
pr2	8920107
pi2	-50253049
pr3	0
pi3	2128
pr4	-8920107
pi4	-50253049
pr5	-10139814
pi5	-2121300
pr6	-9194193
pi6	-2628599
signals	
state	12
sample	30
k	0
u	299
Now	5005421875 ps
Cursor 1	4409706173 ps

4372773270 ps to 4413833372 ps   Now: 5,005,421,875 ps Delta: 1

This diagram illustrates the operation of the multipliers and dividers. It is located at nearly the same time as the last slide, where the FFT of sample 30 is shown as well as some of the LMS calculation for sample 30. The upper part of this diagram shows the inputs to the dividers and multipliers. The “ar#” and “br#” signals are the multiplier inputs and the “dividend#” signals are the divider inputs. The divisors to the dividers are not shown because they are always the value of “fixedpointmult” or 10,000. The bottom shows the outputs of the multipliers.

Notice that the LMS calculations on the left side only seem to be using one multiplier and one divider. That is because so many of the LMS calculations are dependant on previous calculations, and thus cannot be done in parallel. The FFT portion is quickly noticed. But, it appears that only two stages are represented because only two columns are readily visible. This is because stage one requires no complex multiplication and is done in one clock cycle. Stage two uses complex multiplication, but the twiddle factors can be represented with whole numbers, so they are not initially multiplied by 10,000 and the result does not need to be divided by 10,000. Stages three, and four, on the other hand, require all of these extra fixed point calculation. The division in each stage, done in state 12, makes these stages readily visible in the simulation.



THE UNIVERSITY of TENNESSEE

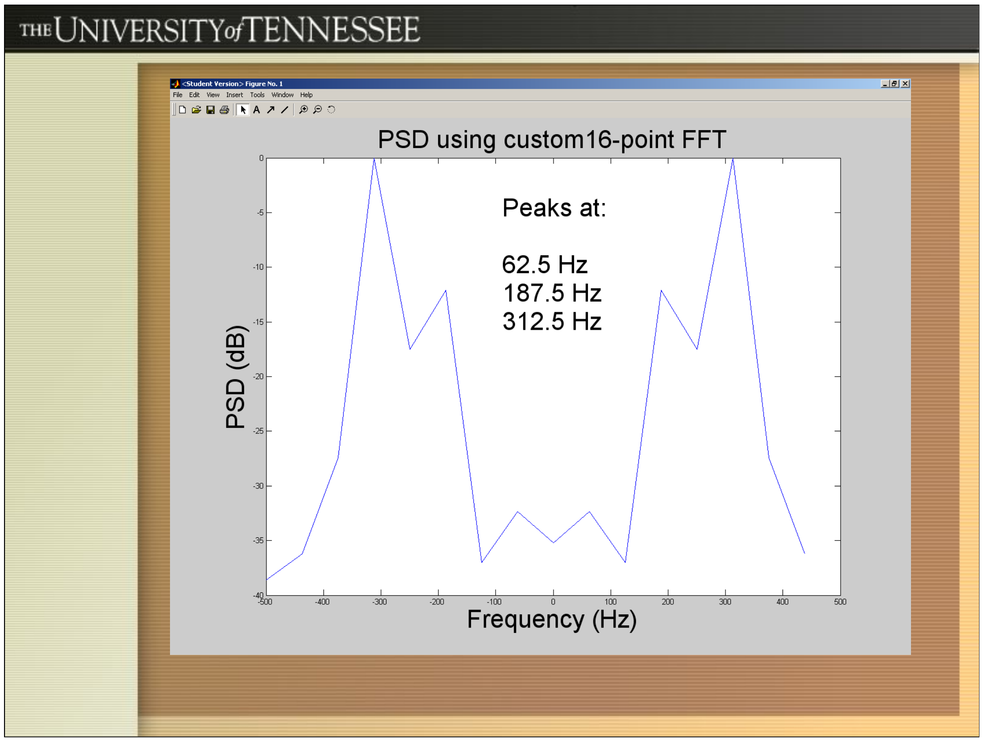
Implementations

1:vlsi8.engr.utk.edu - default - SSH Secure Shell
File Edit View Window Help

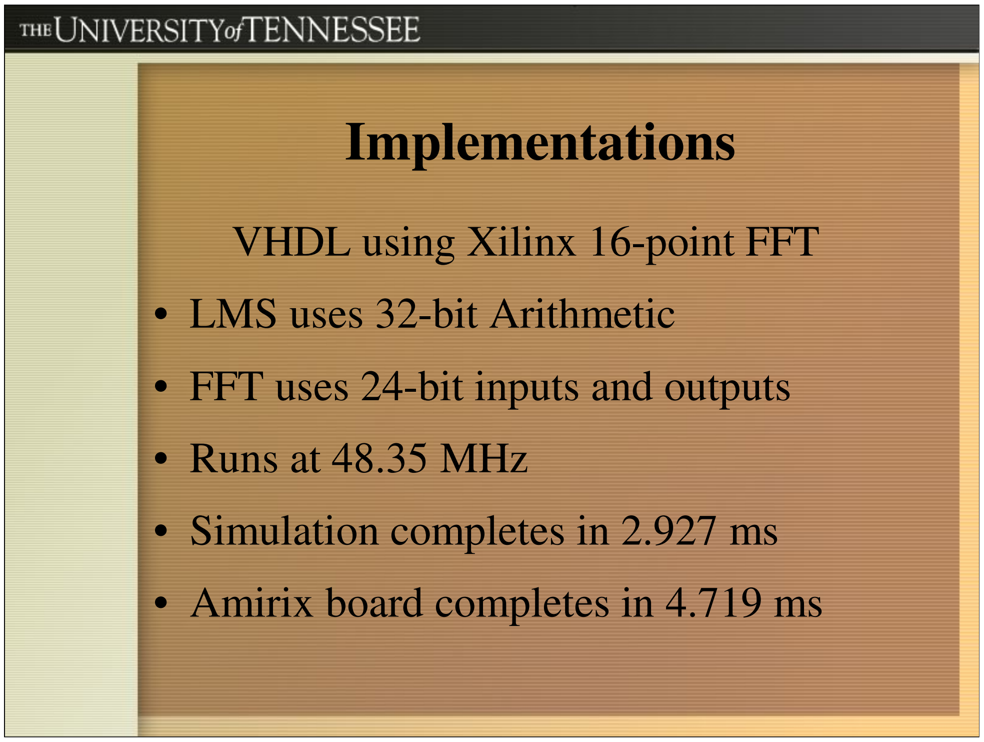
```
Autobooting PPCBoot in 3 seconds, press SPACE to stop
Baseline Platform Customized - Saumil Merchant
Loading and running PPCBoot...


PPCBoot 2.0.0 (Dec 22 2003 - 11:51:11)


Serial#: 1183
DRAM:  64 MB
FLASH:  8 MB
In:    serial
Out:   serial
Err:   serial
Net:   i82559#0
Hit any key to stop autoboot:  0
0> go 0x00100000
## Starting application at 0x00100000 ...
Elapsed PPC clock cycles 1415712
Press any key to continue ...

## Application terminated, rc = 0xa5
0>
```

Connected to vlsi8.engr.utk.edu
SSH2 - aes128-cbc - hmac-md5 - none
80x25

This slide shows the actual use of the design on the board. The C code location on the SDRAM is at hex address 0x00100000. The Amirix board time measurement begins when data is moved from SDRAM into block RAM and ends when the results are transferred from block RAM to SDRAM. The number of elapsed PPC clock cycles here was 1415712. With the PPC clock frequency set at 300 MHz, the calculation is completing in 4.719 milliseconds on the board. This is slightly higher than the number obtained in simulation, 2.927 milliseconds, but is still much faster than the 125 millisecond run time for floating point Matlab.



This figure shows the PSD estimation for sample 30 produced by the Amirix board. The data here is identical to the fixed point Matlab data. That is because all calculations were done in a similar way.


THE UNIVERSITY of TENNESSEE

Implementations

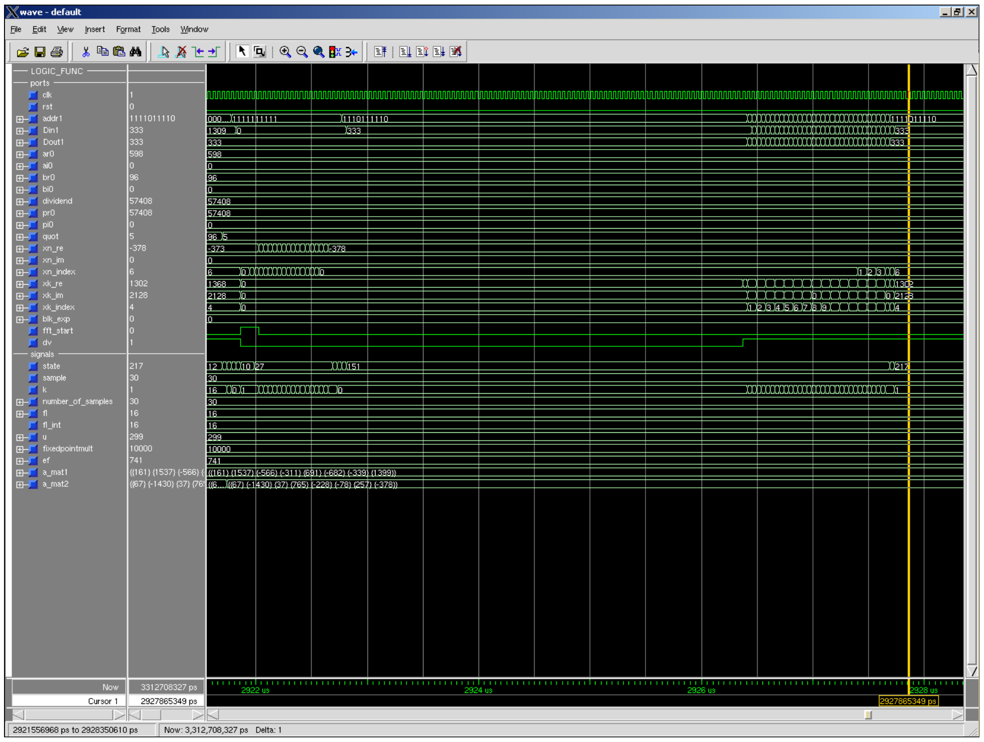
VHDL using Xilinx 16-point FFT

- LMS uses 32-bit Arithmetic
- FFT uses 24-bit inputs and outputs
- Runs at 48.35 MHz
- Simulation completes in 2.927 ms
- Amirix board completes in 4.719 ms

After creating a version with my custom FFT code, I decided to compare its performance with a version that uses a Xilinx 16-point FFT block. This version uses the same LMS code, but all of the FFT code is replaced with an instantiation of a Xilinx FFT block. I also removed all of the FFT registers I had used. One thing I noticed before I inserted the block was that the maximum precision for the input and output for one of these blocks is 24-bits. This gives the Xilinx code a slight disadvantage over my FFT code.

Obviously, for my LMS routine to use this FFT, I had to remove bits from the AR parameters so that they would fit into the FFT. I read the data sheet to learn what I could about this FFT. It uses Radix-2, and a method called block floating point. Block floating point calculates how much to scale the results of each stage to get the most precision. Whereas my algorithm scaled each output by 10,000.

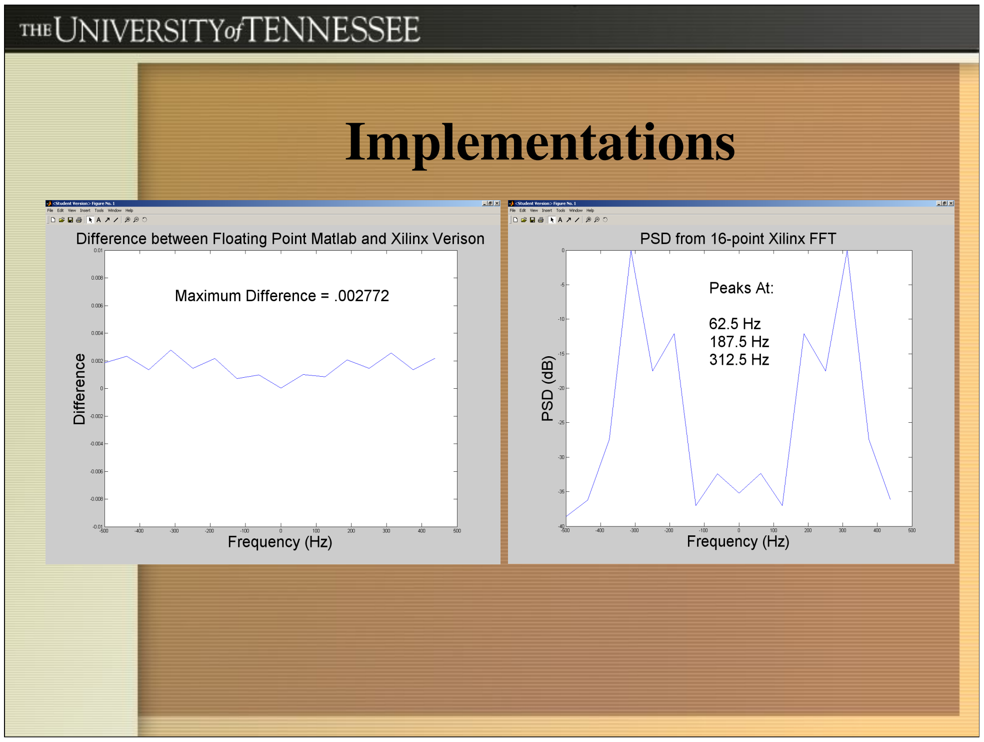
After running the completed design through place and route, a faster clock speed was automatically assigned. So this version completes faster than my design. The 24-bit restriction could pose a problem though. If the LMS results contain numbers requiring more than 24-bits to represent then they cannot be sent to the Xilinx FFT.



LOGIC_FUNC	
ports	
clk	1
rst	0
addr1	1111011110
Din1	333
Dout1	333
ar0	598
ai0	0
br0	96
bi0	0
dividend	57408
pr0	57408
pi0	0
quot	5
xn_re	-378
xn_im	0
xn_index	6
xk_re	1302
xk_im	2128
xk_index	4
blk_exp	0
fft_start	0
dv	1
signals	
state	217
sample	30
k	1
number_of_samples	30
fl	16
fl_int	16
u	299
fixedpointmult	10000
ef	741
a_mat1	{{161} {1537} {-566} {
a_mat2	{{67} {-1430} {37} {76
Now	3312708327 ps
Cursor 1	2927865349 ps

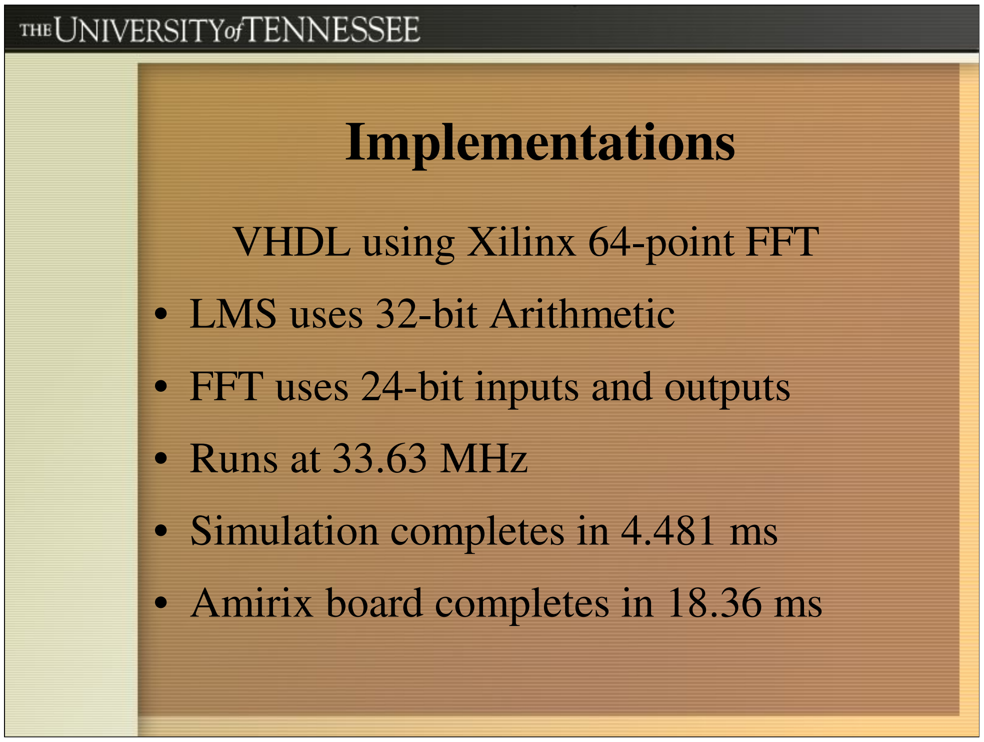
2921556968 ps to 2928350610 ps   Now: 3,312,708,327 ps Delta: 1

Since the LMS routine is the same I have not shown it in this slide. The FFT is shown for sample 30. The signals “xn_re” and “xn_im” are the inputs to the FFT. Notice they are entered in a serial fashion. The signal xn_index shows when data should be sent to the FFT. There is a latency of three involved in this process. The outputs are also produced in a serial fashion in the signals “xk_re” and “xk_im”. As these outputs are produced they are immediately sent to the dual port block RAM, indicated by all of the activity seen on “Dout”. Interestingly my FFT design requires fewer clock cycles to complete, but the Xilinx place and route software assigns a faster clock speed to the version containing the Xilinx FFT, possibly because the design is less complex.



Here are plots of the data that the board produces with this new implementation, I knew nothing of the internal calculations of the Xilinx FFT, and therefore could not produce Matlab code which would create the exact same results. On the left I have made a plot which is the result of subtracting the Matlab floating point PSD estimation for sample 30 from the VHDL PSD estimation for sample 30. Here, the maximum difference is 0.002772. The maximum difference associated with the VHDL using my custom FFT code was 0.00280. So, it has slightly more accuracy than my version. Which is surprising given the limited bit-width. This leads me to believe that it uses very small scaling on certain intermediate results and very large scaling on other results. I truncated all of my decimals, Xilinx may be rounding with their decimals.

The left shows a plot of the PSD from this implementation. The peaks are located at the exact same frequencies as my VHDL and the Matlab version. Again, the only differences would be in the height of the peaks.


THE UNIVERSITY of TENNESSEE

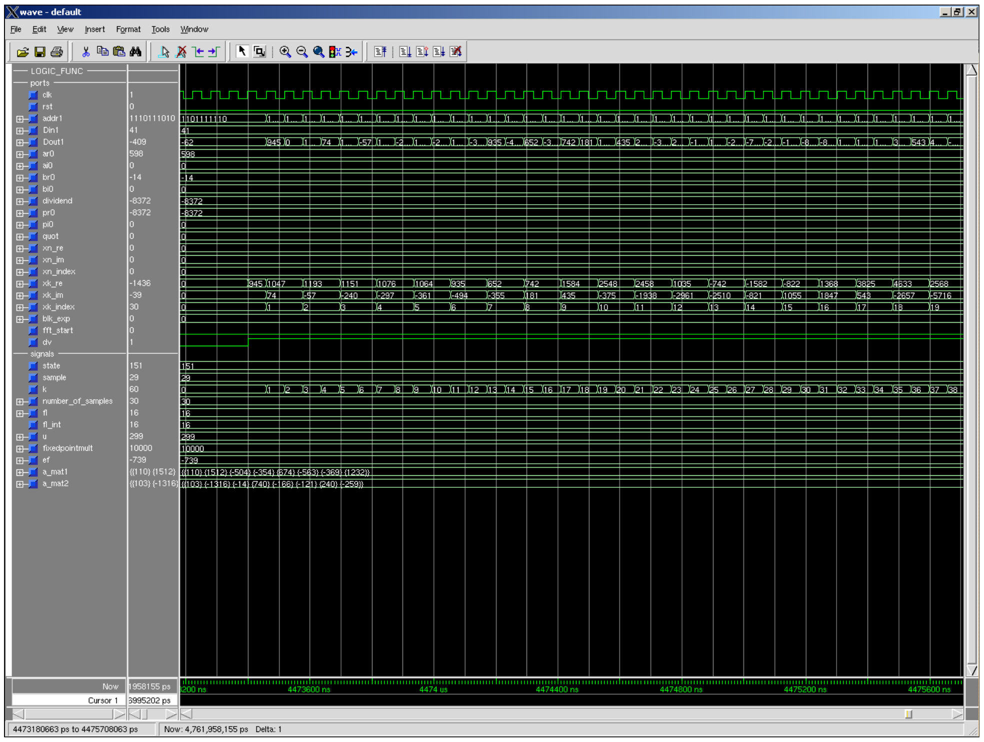
Implementations

VHDL using Xilinx 64-point FFT

- LMS uses 32-bit Arithmetic
- FFT uses 24-bit inputs and outputs
- Runs at 33.63 MHz
- Simulation completes in 4.481 ms
- Amirix board completes in 18.36 ms

I wanted to attempt using a larger FFT size from Xilinx. This also meant that the output vectors would be larger. In my other designs the output occupied the entire dual port block RAM. But, with this design, since more output will be produced, extra block RAMs would be necessary. This design uses four block RAMs and does PSD estimations for 29 samples. I would have needed one more block ram to fit the results from the last sample and chose to omit it.

As with the other Xilinx FFT blocks this one has the same restriction on bit width. After running the design through place and route a frequency of 33.63 Hz is assigned. This is a lower frequency than the design containing the 16-point Xilinx FFT, which is expected because the design is more complex. Using this clock frequency in the simulator determines that the code completes execution in 4.481 milliseconds. This is still much faster than the 125 milliseconds for the floating point Matlab doing only a 16-point FFT.

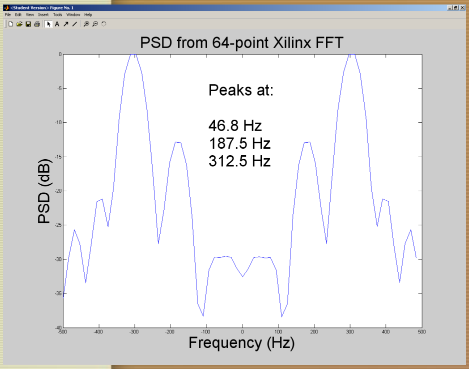


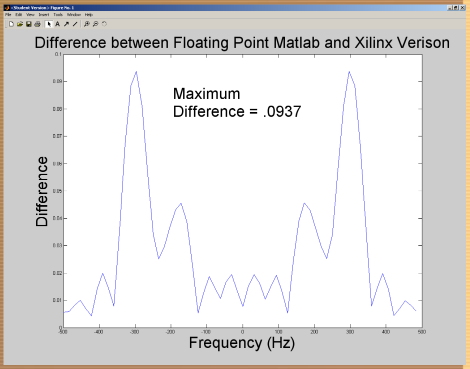
LOGIC_FUNC	
ports	
clk	1
rst	0
addr1	1110111010
Din1	41
Dout1	-409
ar0	598
ai0	0
br0	-14
bi0	0
dividend	-8372
pr0	-8372
pi0	0
quot	0
xn_re	0
xn_im	0
xn_index	0
xk_re	-1436
xk_im	-39
xk_index	30
blk_exp	0
fft_start	0
dv	1
signals	
state	151
sample	29
k	60
number_of_samples	30
fl	16
fl_int	16
u	299
fixedpointmult	10000
ef	-739
a_mat1	{{110} {1512}
a_mat2	{{103} {-1316}
Now	1958155 ps
Cursor 1	3995202 ps

This slide shows the results from the FFT at sample 29. The signal xk_index goes to 63 for each FFT computation. As does the signal xn_index. So obviously the data loading and unloading take more time than the Xilinx 16-pint FFT. Even with the added time the code completes faster than Matlab.

THE UNIVERSITY of TENNESSEE

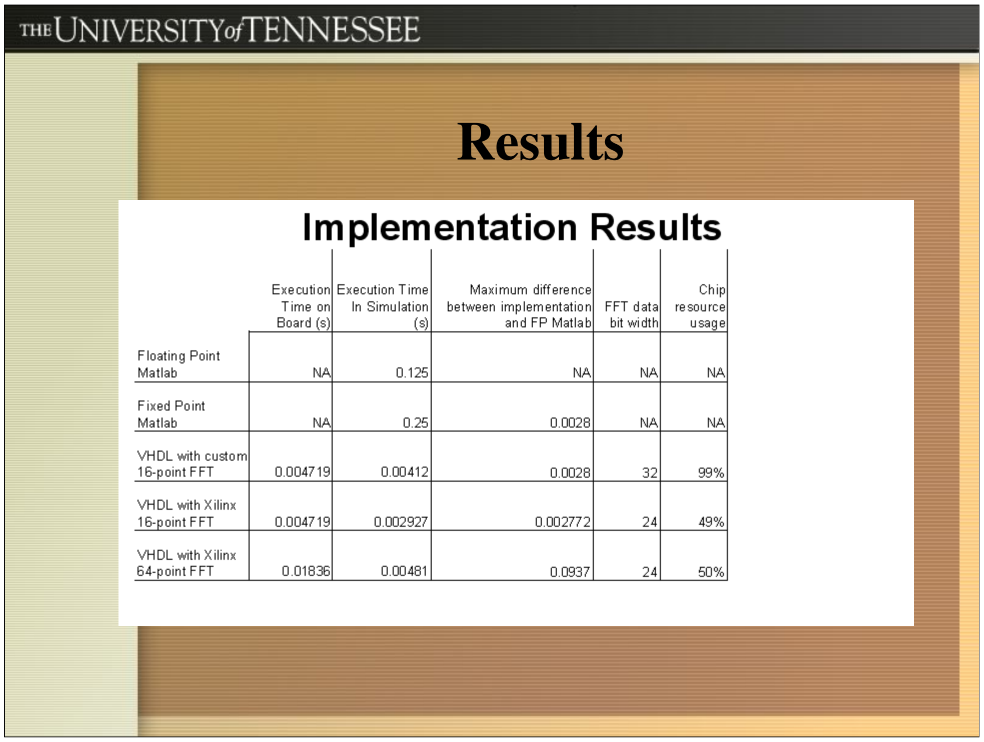
Implementations





With the larger FFT size, more detail can be seen in the PSD plot. On the left is the PSD data from sample 29 produced by the Amirix board. I looked at the three main peaks in the plot and found the highest point in each. The peaks located at 187.5 Hz and 312 Hz, are the same as the results form the other implementations. But, the frequency that was at 62.5 Hz now seems to be closer to 46.8 Hz. I would attribute this to the low amplitude assigned to the 100 Hz signal and my choice of filter length. I used a much higher filter length on the data than should have been used. This could also be the cause of the addition of the smaller peaks near the outer frequencies on the graph. Again, I chose to use this high filter length to show the code's performance, under maximum demands.

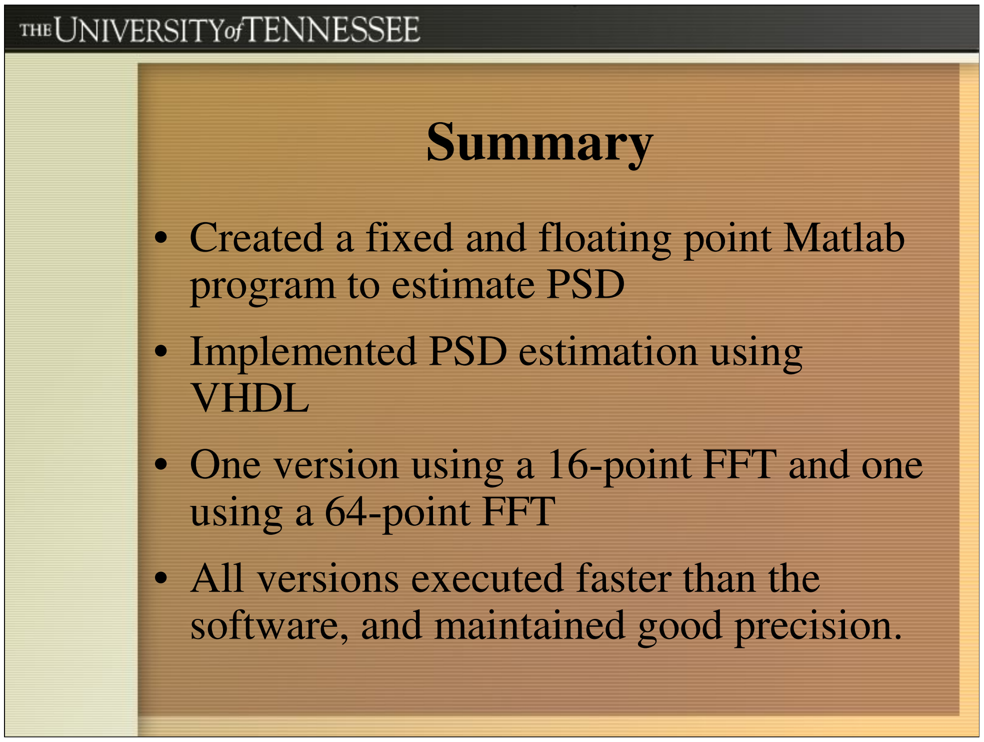
The plot on the right shows the difference between the PSD of sample 29 on the Amirix board and the same PSD from Matlab. I changed the Matlab code to do a 64-point FFT to make the comparison. Apparently at such large FFT computation, the Xilinx FFT begins to loose some precision. The maximum difference in the previous designs was 0.002772 and here is 0.0937.



Results

Implementation Results

	Execution Time on Board (s)	Execution Time In Simulation (s)	Maximum difference between implementation and FP Matlab	FFT data bit width	Chip resource usage
Floating Point Matlab	NA	0.125	NA	NA	NA
Fixed Point Matlab	NA	0.25	0.0028	NA	NA
VHDL with custom 16-point FFT	0.004719	0.00412	0.0028	32	99%
VHDL with Xilinx 16-point FFT	0.004719	0.002927	0.002772	24	49%
VHDL with Xilinx 64-point FFT	0.01836	0.00481	0.0937	24	50%


THE UNIVERSITY of TENNESSEE

Summary

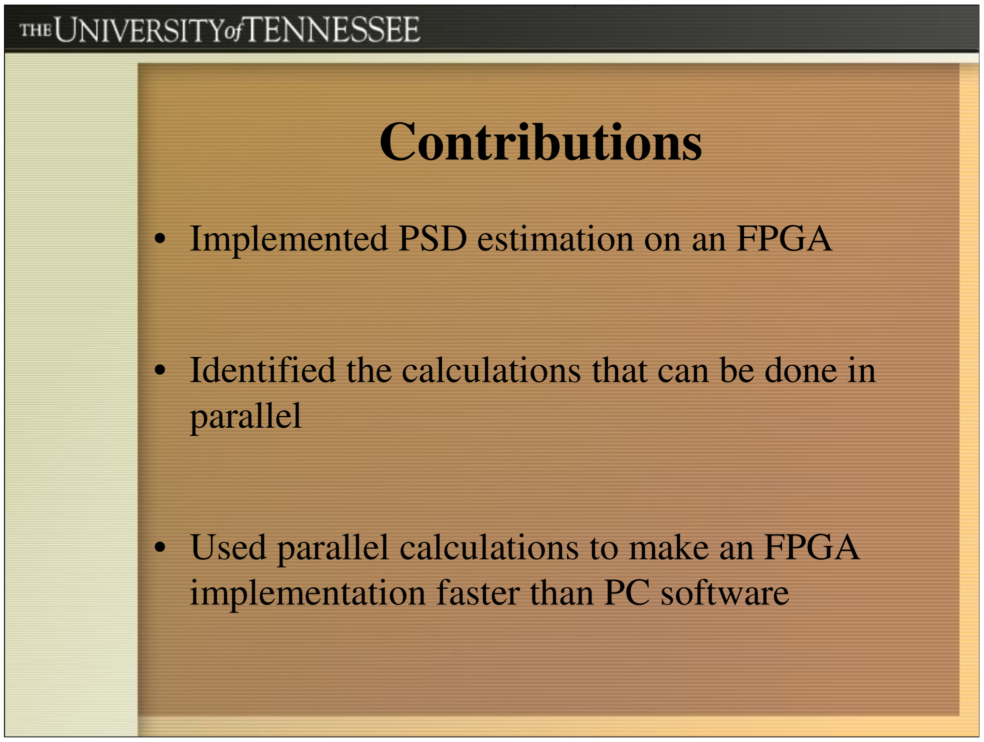
- Created a fixed and floating point Matlab program to estimate PSD
- Implemented PSD estimation using VHDL
- One version using a 16-point FFT and one using a 64-point FFT
- All versions executed faster than the software, and maintained good precision.

In summary I began with a PSD estimator written in Matlab. I made a floating point version and a fixed point version. The execution time for the floating point version was measured and the fixed point version was used to aid in the development of a VHDL implementation. My first VHDL implementation that worked on the Amirix board contained custom 16-point FFT code that I wrote myself. By doing several butterfly operations simultaneously, it was able to execute faster than the floating point Matlab version, and precision was preserved.

For the sake of possibly increasing the FFT size I replaced my custom 16-point FFT code with Xilinx 16-point FFT block. In comparison with my version, this one was slightly faster, and used less resources on the chip. But, the bit width of the inputs and outputs are limited to 24.

Since that design worked I tried a Xilinx 64-point FFT, this design also fit, and still only uses half of the chip.

Each of the VHDL implementations execute faster than the Matlab version and reasonable precision is maintained, at least up until the 64-point FFT.

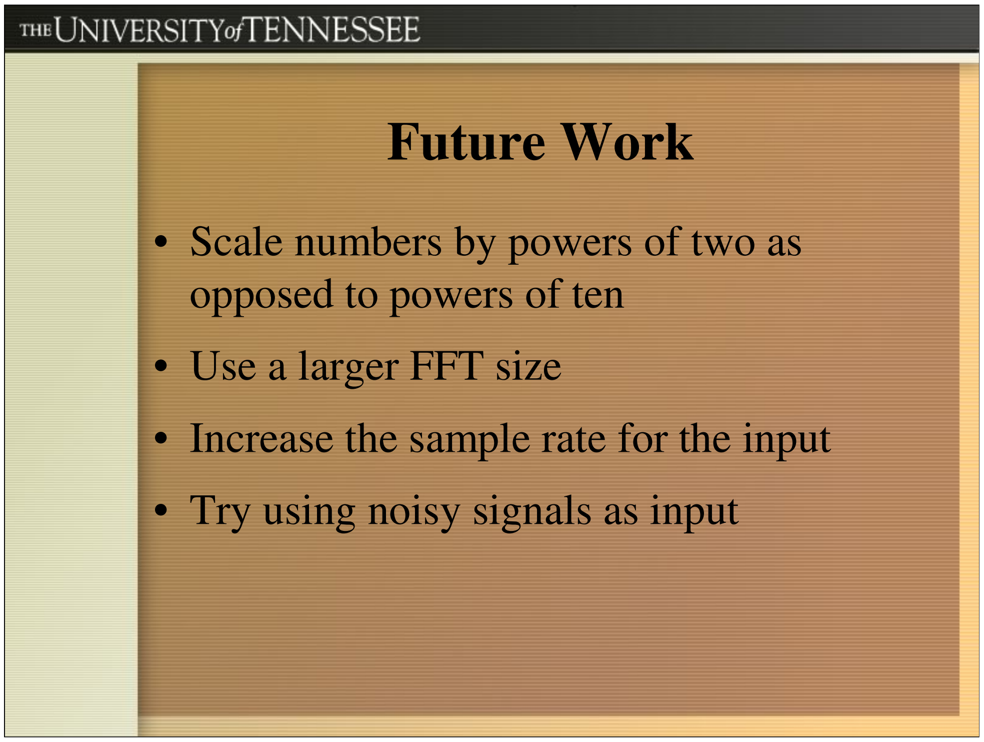


THE UNIVERSITY of TENNESSEE

Contributions

- Implemented PSD estimation on an FPGA
- Identified the calculations that can be done in parallel
- Used parallel calculations to make an FPGA implementation faster than PC software

I have successfully implemented PSD estimation on an FPGA. I have also used parallel calculations on the FPGA to make the implementation faster than software. I have also improved the design by increasing the FFT size from 16-points to 64-points.


THE UNIVERSITY of TENNESSEE

Future Work

- Scale numbers by powers of two as opposed to powers of ten
- Use a larger FFT size
- Increase the sample rate for the input
- Try using noisy signals as input

It has become apparent that scaling the data by powers of two instead of powers of ten would yield a more efficient design. This would eliminate the need for the dividers, since division by a power of two involves simply shifting the bits. This would be s very significant improvement since the dividers occupy a very large amount of resources on the FPGA. With this change it may be possible to expand the custom FFT code to 32 points.

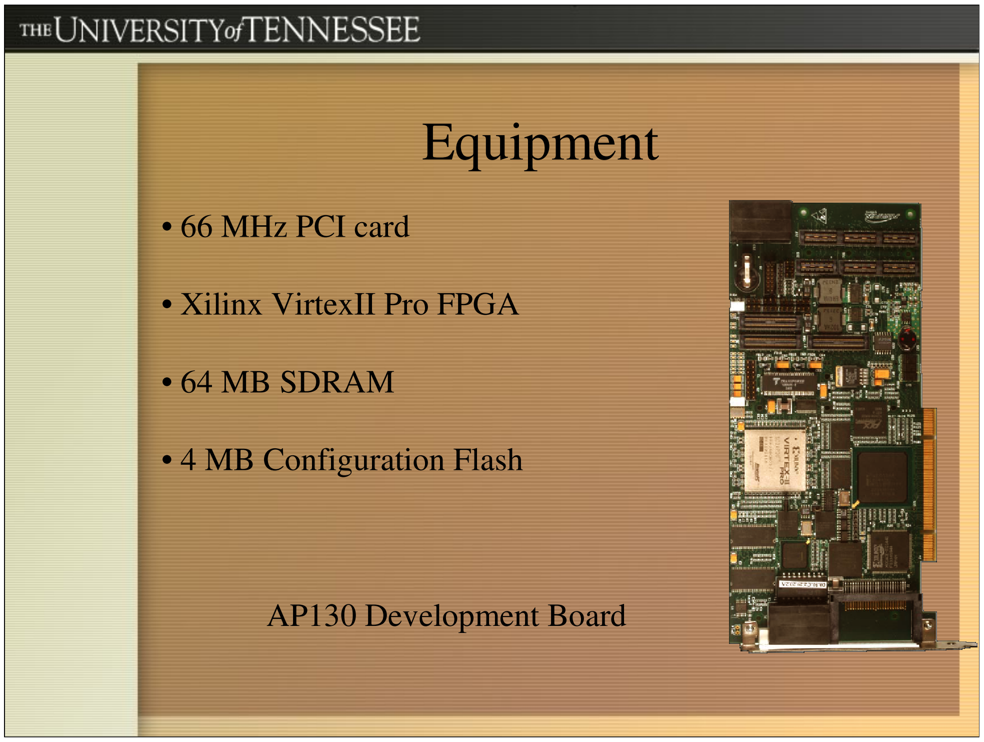
The PSD results could have also been improved if the sample rate for the input were increased.

Also, since PSD is typically used for determining the frequency content of a noisy signal, this type of signal should be tried on the VHDL implementation. In my project I only mixed sinusoids together and did not add noise.

THE UNIVERSITY of TENNESSEE

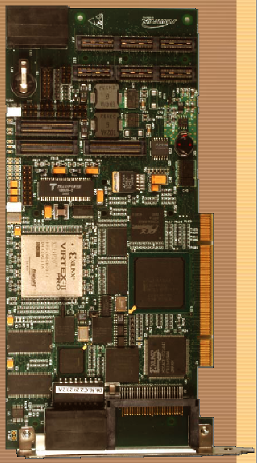
Extra Slides

This slide marks the begging of the “extra slides” section.


THE UNIVERSITY of TENNESSEE

Equipment

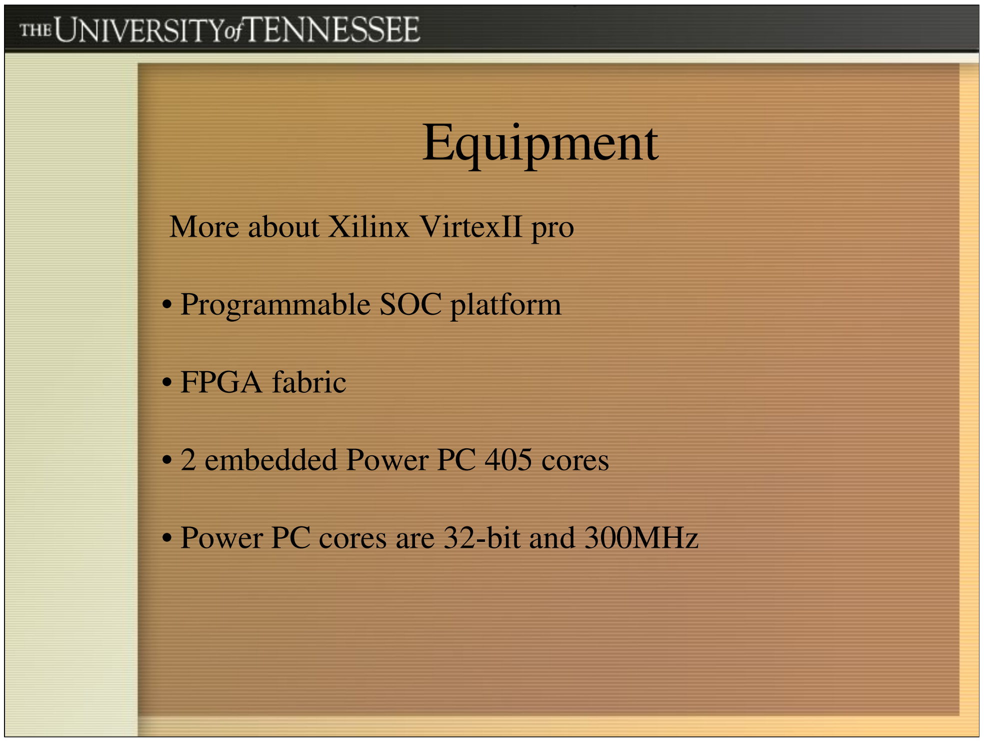
- 66 MHz PCI card
- Xilinx VirtexII Pro FPGA
- 64 MB SDRAM
- 4 MB Configuration Flash



AP130 Development Board

The AP130 Development Board is a 66MHz PCI card that can be inserted and used in a computer that runs Unix as an operating system. It contains one Xilinx VirtexII Pro FPGA. This chip contains two embedded PPC405 CPU cores. The board also contains 64 MB of SDRAM and a 4MB configuration flash.

A typical design for this board would include C code that transfers data from the SDRAM to a user defined memory structure in the VirtexII Pro. The C code is executed by a PPC. The user defined memory structure is synthesized on the FPGA fabric using VHDL. VHDL would further be used to define a logic function to be performed on the data. Results would then be transferred back to SDRAM through the PPC.

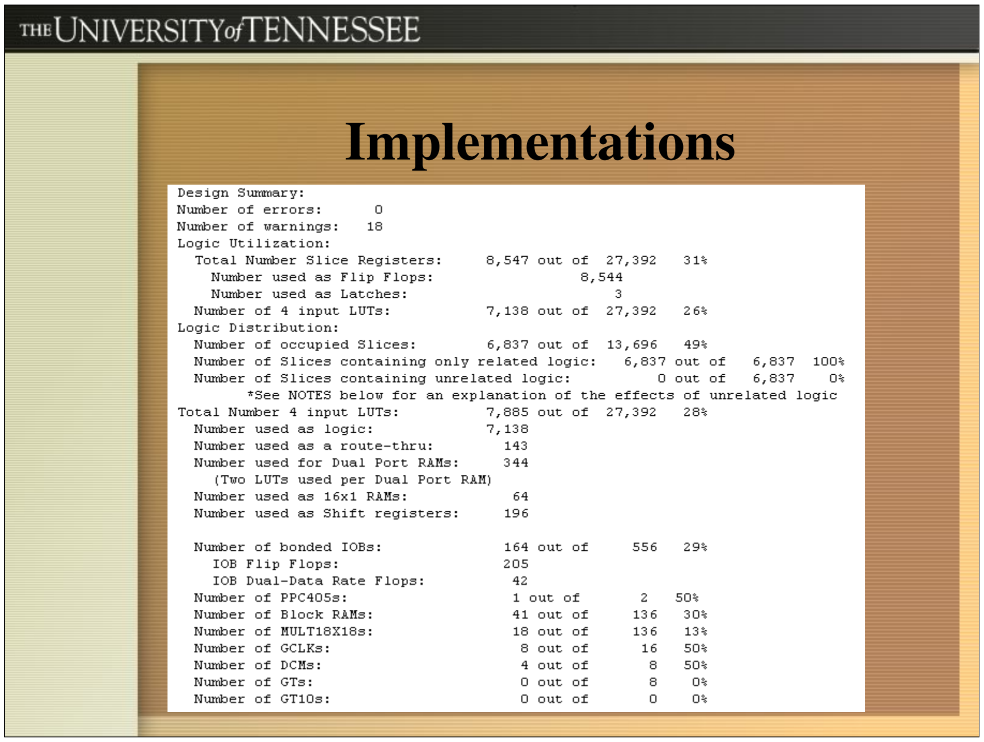


THE UNIVERSITY of TENNESSEE

Equipment

More about Xilinx VirtexII pro

- Programmable SOC platform
- FPGA fabric
- 2 embedded Power PC 405 cores
- Power PC cores are 32-bit and 300MHz

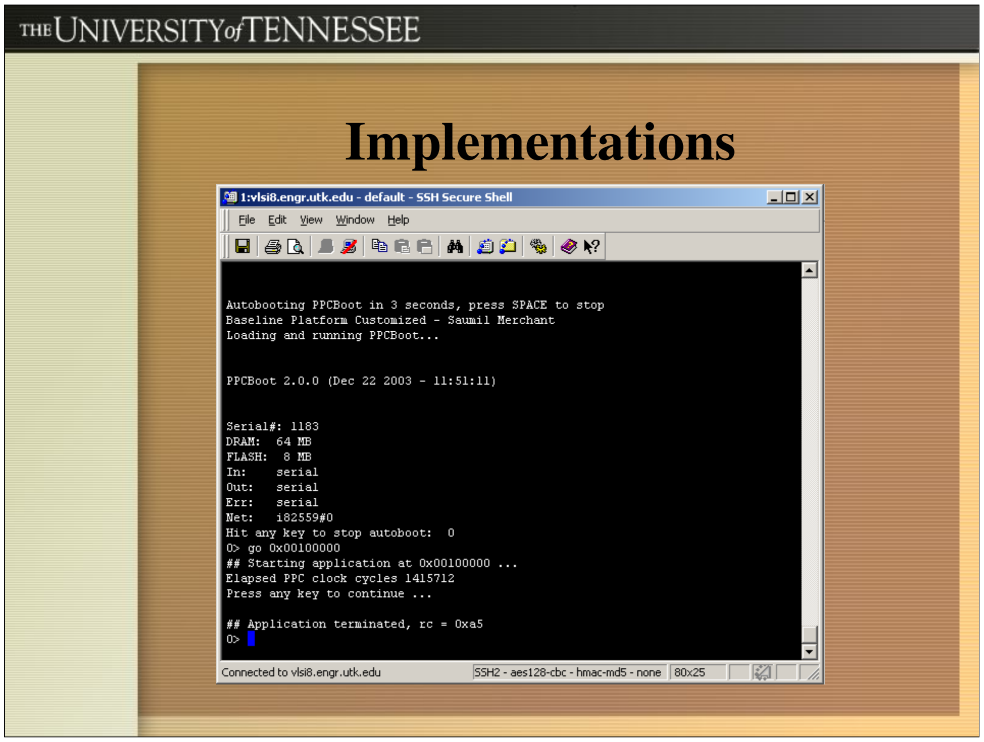

THE UNIVERSITY of TENNESSEE

Implementations

```
Design Summary:
Number of errors:      0
Number of warnings:   18
Logic Utilization:
  Total Number Slice Registers:     8,547 out of  27,392   31%
    Number used as Flip Flops:                 8,544
    Number used as Latches:                        3
  Number of 4 input LUTs:           7,138 out of  27,392   26%
Logic Distribution:
  Number of occupied Slices:        6,837 out of  13,696   49%
  Number of Slices containing only related logic:   6,837 out of   6,837  100%
  Number of Slices containing unrelated logic:          0 out of   6,837    0%
        *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:          7,885 out of  27,392   28%
  Number used as logic:             7,138
  Number used as a route-thru:        143
  Number used for Dual Port RAMs:     344
    (Two LUTs used per Dual Port RAM)
  Number used as 16x1 RAMs:            64
  Number used as Shift registers:     196

  Number of bonded IOBs:              164 out of     556   29%
    IOB Flip Flops:                   205
    IOB Dual-Data Rate Flops:          42
  Number of PPC405s:                   1 out of       2   50%
  Number of Block RAMs:               41 out of     136   30%
  Number of MULT18X18s:               18 out of     136   13%
  Number of GCLKs:                     8 out of      16   50%
  Number of DCMs:                      4 out of       8   50%
  Number of GTs:                       0 out of       8    0%
  Number of GT10s:                     0 out of       0    0%
```

Xilinx 16-point FFT implementation

This design used less logic slices than my design. This may be due to the variable scaling and the reduced bit width. But, since the usage is so low it means that a larger Xilinx FFT block could possibly be used.


THE UNIVERSITY of TENNESSEE

Implementations

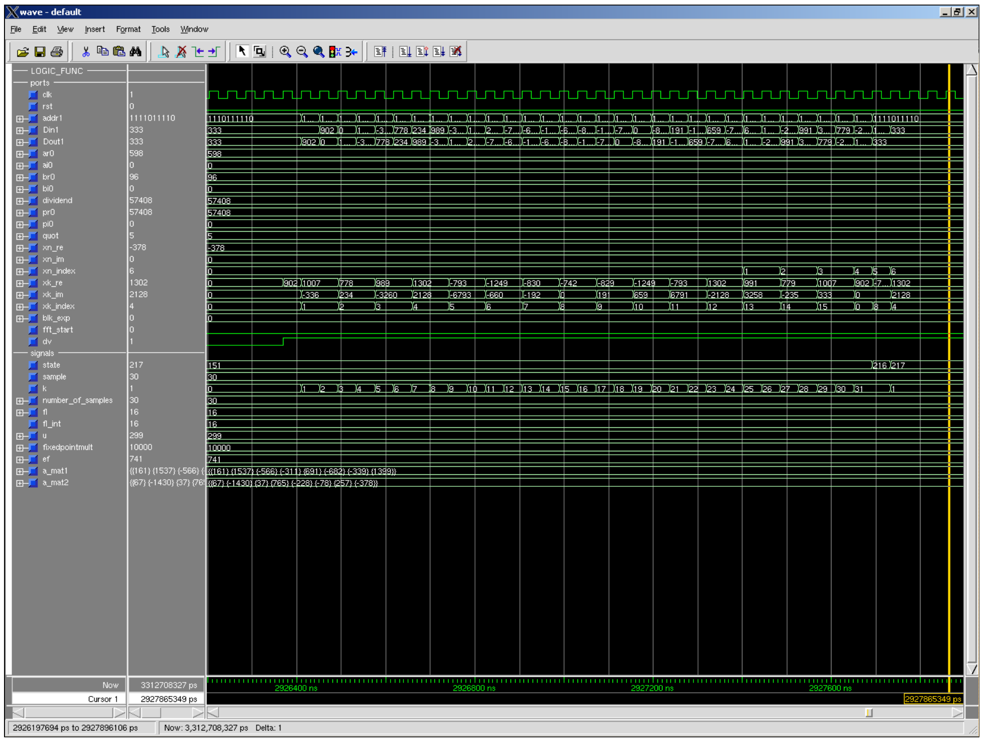
1:vlsi8.engr.utk.edu - default - SSH Secure Shell
File Edit View Window Help

```
Autobooting PPCBoot in 3 seconds, press SPACE to stop
Baseline Platform Customized - Saumil Merchant
Loading and running PPCBoot...


PPCBoot 2.0.0 (Dec 22 2003 - 11:51:11)


Serial#: 1183
DRAM:  64 MB
FLASH:  8 MB
In:    serial
Out:   serial
Err:   serial
Net:   i82559#0
Hit any key to stop autoboot:  0
0> go 0x00100000
## Starting application at 0x00100000 ...
Elapsed PPC clock cycles 1415712
Press any key to continue ...

## Application terminated, rc = 0xa5
0>
```

Connected to vlsi8.engr.utk.edu
SSH2 - aes128-cbc - hmac-md5 - none
80x25

Xilinx 16-point FFT implementation

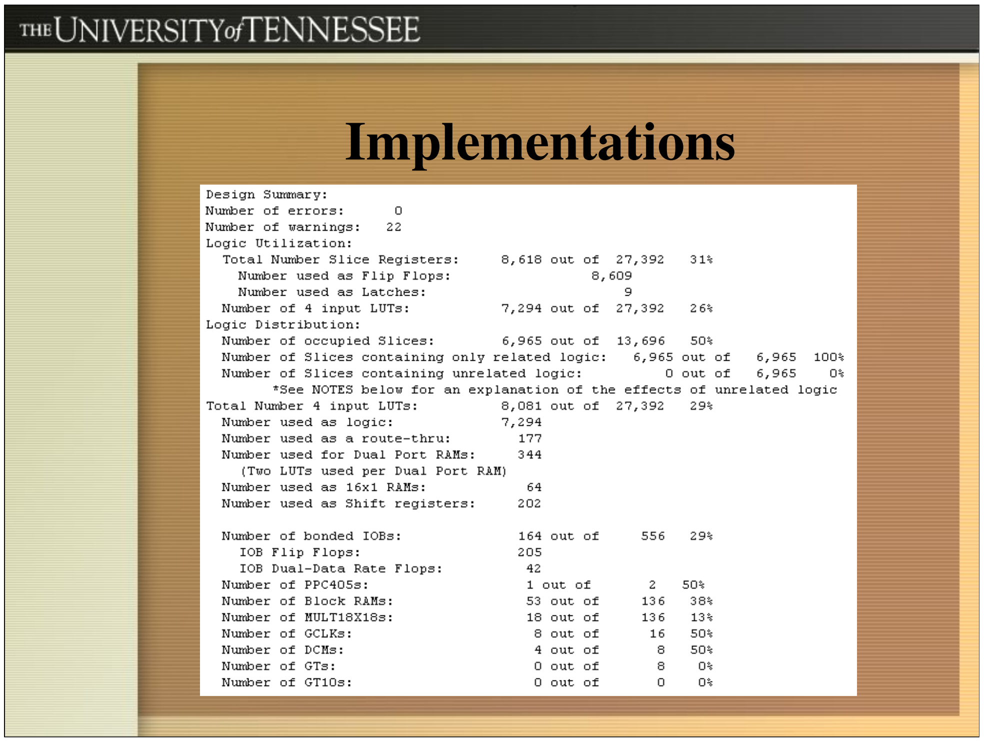
This slide shows the actual use of the design on the board. The C code location on the SDRAM is again at hex address 0x00100000. The Amirix board time measurement begins and ends at the same time the implementation with my custom FFT code did. The number of elapsed PPC clock cycles here was 1415712. This is the same as the previous implementation. With the PPC clock frequency set at 300 MHz, the calculation is completing in 4.719 milliseconds on the board. This is slightly higher than the number obtained in VHDL simulation, 2.927 milliseconds, but is still much faster than the 125 millisecond run time for floating point Matlab.



```
LOGIC_FUNC
ports
clk                 1
rst                 0
addr1               1111011110
Din1                333
Dout1               333
ar0                 598
ai0                 0
br0                 96
bi0                 0
dividend            57408
pr0                 57408
pi0                 0
quot                5
xn_re               -378
xn_im               0
xn_index            6
xk_re               1302
xk_im               2128
xk_index            4
blk_exp             0
fft_start           0
dv                  1
signals
state               217
sample              30
k                   1
number_of_samples   30
fl                  16
fl_int              16
u                   299
fixedpointmult      10000
ef                  741
a_mat1              {(161} {1537} {-566} {
a_mat2              {(67} {-1430} {37} {76!

Now        3312708327 ps
Cursor 1   2927865349 ps

2926197694 ps to 2927898106 ps    Now: 3,312,708,327 ps  Delta: 1
```

Implementation with Xilinx 16-point FFT.

This slide shows the output of the FFT for sample 30 in detail. The numbers can actually be seen.

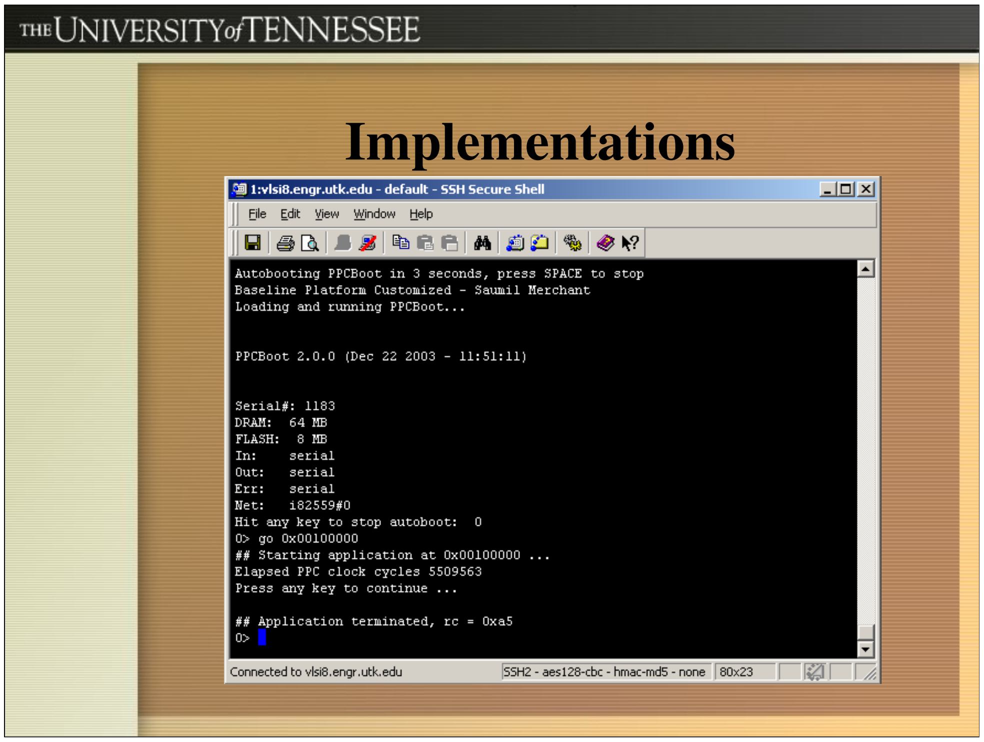

THE UNIVERSITY of TENNESSEE

Implementations

```
Design Summary:
Number of errors:      0
Number of warnings:   22
Logic Utilization:
  Total Number Slice Registers:     8,618 out of  27,392   31%
    Number used as Flip Flops:                 8,609
    Number used as Latches:                        9
  Number of 4 input LUTs:           7,294 out of  27,392   26%
Logic Distribution:
  Number of occupied Slices:        6,965 out of  13,696   50%
  Number of Slices containing only related logic:   6,965 out of   6,965  100%
  Number of Slices containing unrelated logic:          0 out of   6,965    0%
        *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:          8,081 out of  27,392   29%
  Number used as logic:             7,294
  Number used as a route-thru:        177
  Number used for Dual Port RAMs:     344
    (Two LUTs used per Dual Port RAM)
  Number used as 16x1 RAMs:            64
  Number used as Shift registers:     202

  Number of bonded IOBs:              164 out of     556   29%
    IOB Flip Flops:                   205
    IOB Dual-Data Rate Flops:          42
  Number of PPC405s:                   1 out of       2   50%
  Number of Block RAMs:                53 out of     136   38%
  Number of MULT18X18s:                18 out of     136   13%
  Number of GCLKs:                      8 out of      16   50%
  Number of DCMs:                       4 out of       8   50%
  Number of GTs:                        0 out of       8    0%
  Number of GT10s:                      0 out of       0    0%
```

Implementation with Xilinx 64-point FFT.

It is very interesting how this design does not use up much more space on the chip than the design with the 16-point FFT. That design used 49% of the logic slices, this design uses 50%. This means that the FFT size could be increased even more. But, also more dual port block RAMs would need to be added to store the results.



THE UNIVERSITY of TENNESSEE
1:vlsi8.engr.utk.edu - default - SSH Secure Shell
File Edit View Window Help
Connected to vlsi8.engr.utk.edu
SSH2 - aes128-cbc - hmac-md5 - none
80x23

Implementations

```
Autobooting PPCBoot in 3 seconds, press SPACE to stop
Baseline Platform Customized - Saumil Merchant
Loading and running PPCBoot...


PPCBoot 2.0.0 (Dec 22 2003 - 11:51:11)


Serial#: 1183
DRAM:   64 MB
FLASH:   8 MB
In:     serial
Out:    serial
Err:    serial
Net:    i82559#0
Hit any key to stop autoboot:  0
0> go 0x00100000
## Starting application at 0x00100000 ...
Elapsed PPC clock cycles 5509563
Press any key to continue ...

## Application terminated, rc = 0xa5
0>
```

Implementation with Xilinx 64-point FFT.

For this design, since more time is required to load and unload the block RAMs with the extra data, more time is required on the board execution. The number of elapsed clock cycles for this one was 5509563. Again, if the PPC clock frequency is assumed to be 300MHz then execution time was 18.36 milliseconds. Also much faster the Matlab's time of 125 milliseconds.