

Fixed-Point Conversion

In this lab we will examine the fixed-point conversion process of a segment of a Simulink model. Since this can be a time-consuming process, we have segmented a section of the receiver design model and provided a test vector of data transmitted between two PlutoSDR devices. The section of interest here is the frequency recovery loop (Carrier Phase Estimation), which is the third stage in the recovery process as outlined in Figure 1.

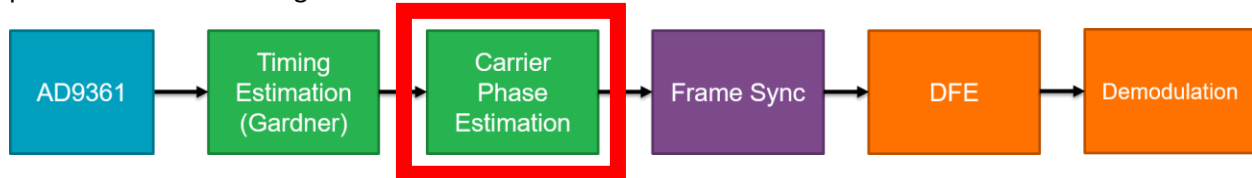


Figure 1

A simple block diagram of the PLL-based frequency corrector is provided in Figure 2. This is a common structure used for feedback-based techniques for carrier recovery in a transceiver. However, it is not vital in this lab to understand every detail of the design. We just need to know that this loop operates by measuring the instantaneous phase of a signal, and uses that phase information to rotate successive symbols to correct for phase errors.

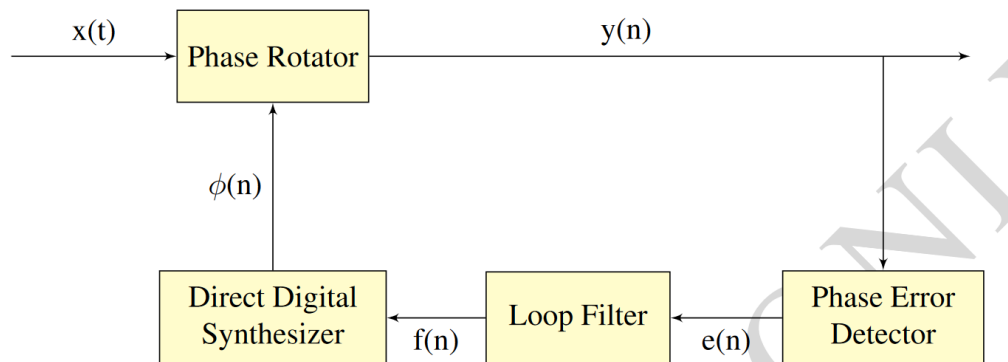
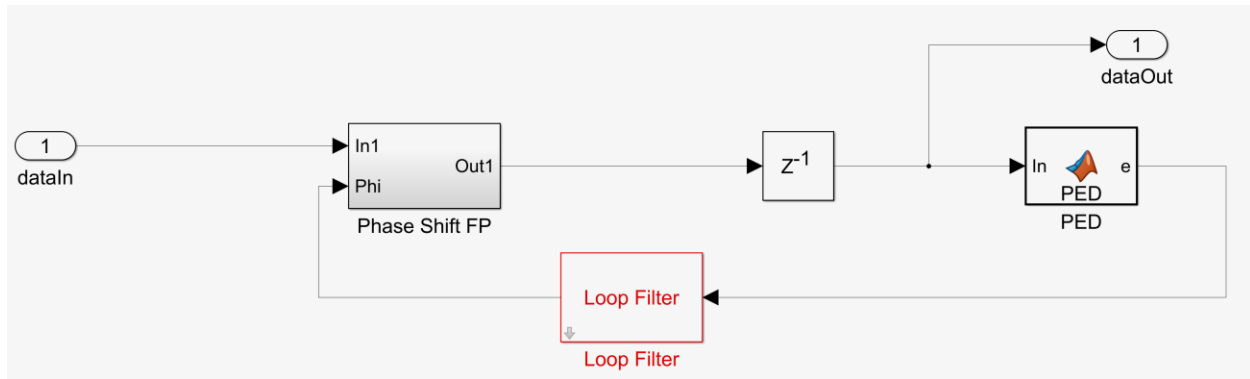


Figure 2 Structure of PLL based frequency recovery algorithm

This was reimplemented in the Simulink floating point model (Receiver.slx) with a near identical structure which we can see in Figure 3, except the Phase Rotator and Direct Digital Synthesizer are implemented in the same block.



It may take a few seconds for Simulink to start and load the model. This only happens when we initially launch Simulink. This will bring up a window which should look like Figure 4. This top-level model contains two main blocks: “Receiver HDL” and “Check Receiver”. “Receiver HDL” is a subsystem which contains all the necessary signal processing for packet recovery, which is what we will be editing for Fixed Point support.

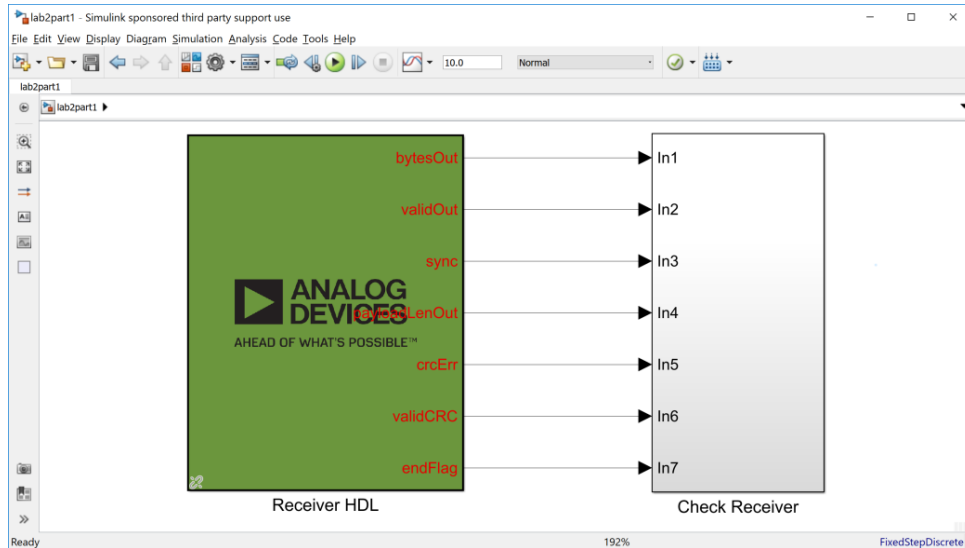


Figure 4 Lab2part1.slx model top level view

For the sake of time we have already processed the data for timing correction. We will be only converting the “Frequency Recovery” block within the top-level “Receiver HDL” block to fixed point in this lab. We have already converted everything after the “Frequency Recovery” block to fixed point, which we will use to validate packets which pass through this block based on their recovered CRC values.

Navigate into the “Receiver HDL” block by double-clicking on the block’s mask outlined in red in Figure 5.

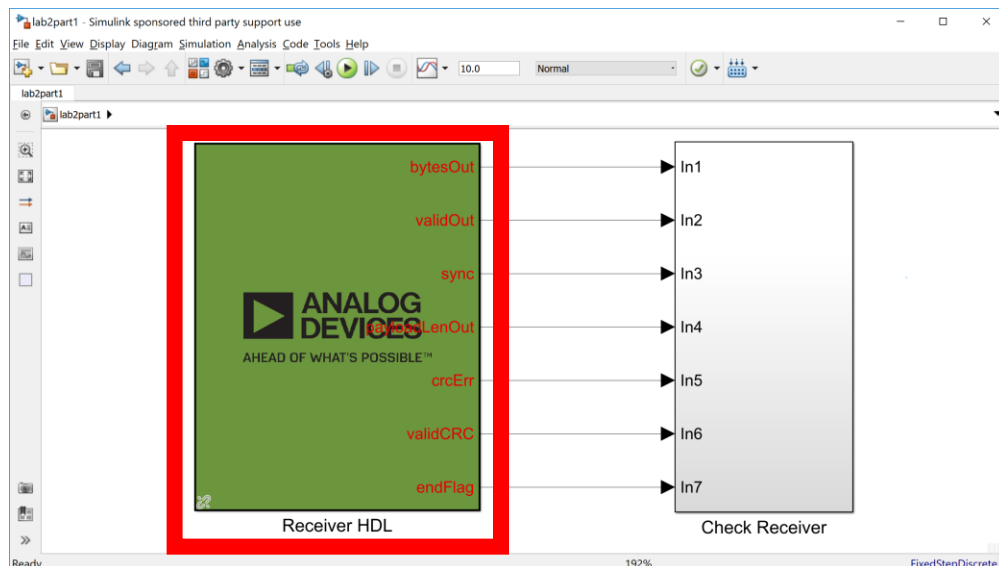


Figure 5 Lab2part1.slx model top level view

Within this subsystem there are three primary blocks: “Frame Recover”, “Viterbi Decode”, and “CRC Check and DMA Pack”. **Since we are only interested in converting the “Frequency Recovery” block**

we will go deeper into the model by double-clicking on the “Frame Recover” block which is highlighted in Figure 6.

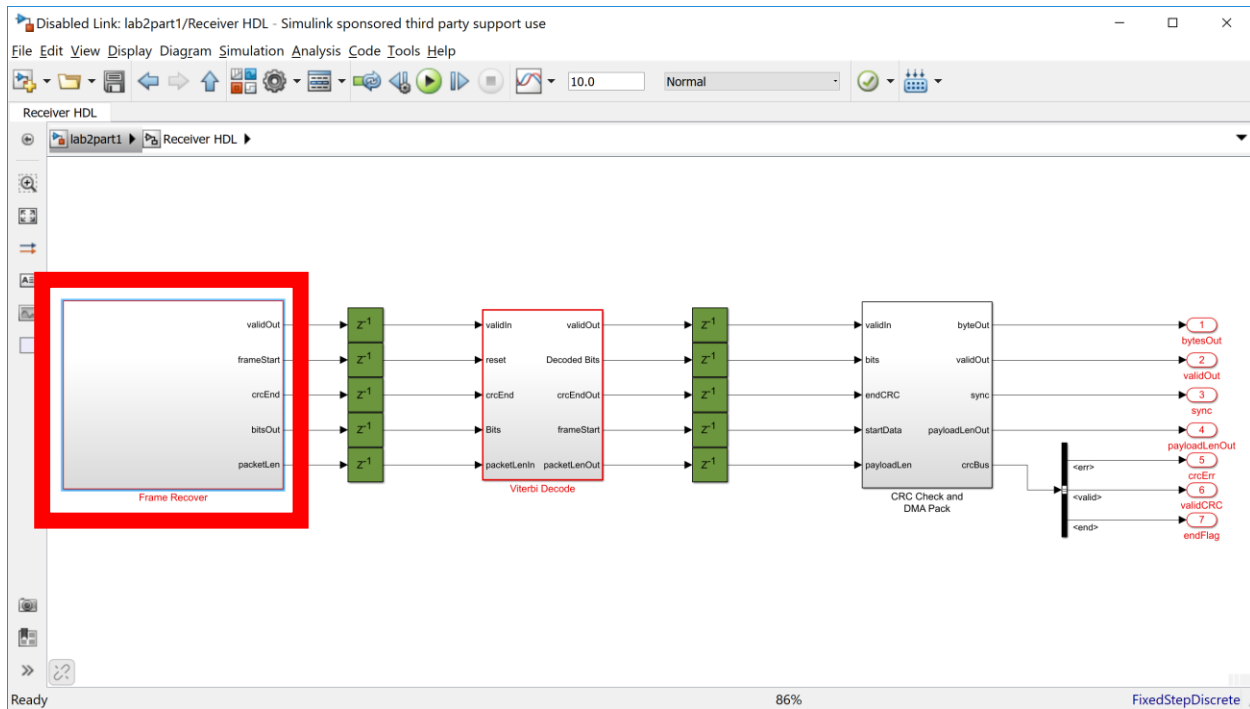


Figure 6 Receiver HDL Subsystem

The “Frame Recover” subsystem contains our “Frequency Recover” block of interest, which we will be editing. It is important to note here that the left-most blocks in this system are our test vector sources, which are the necessary IQ data and validation signals from the timing recovery process.

Double click on the “Frequency Recover” block in your model which is outlined in Figure 7.

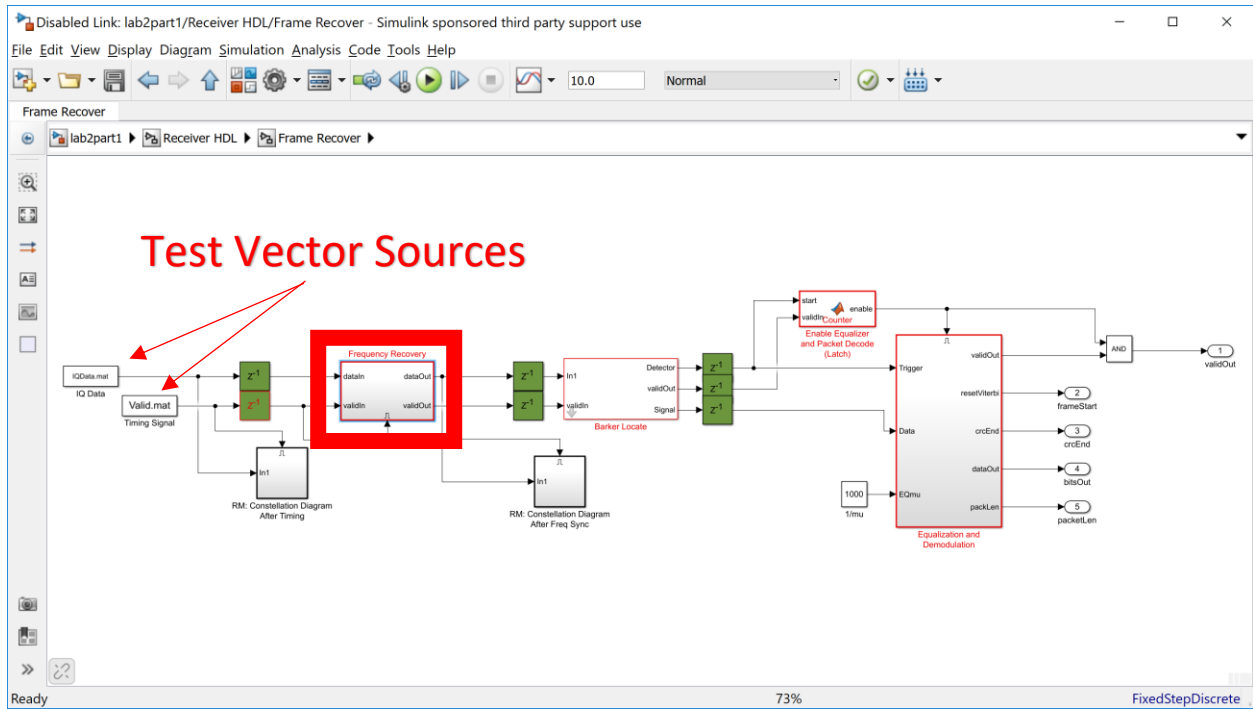


Figure 7 Frame Recover Subsystem

In the “Frequency Recovery” subsystem, show in Figure 8, we will notice that the “dataIn” and “dataOut” ports connect into the inner blocks through “Data Type Conversion” blocks. These gateway-like blocks allow us to utilize floating-point data types between these blocks, and allow us to convert single blocks at a time to fixed-point by using these blocks as boundaries.

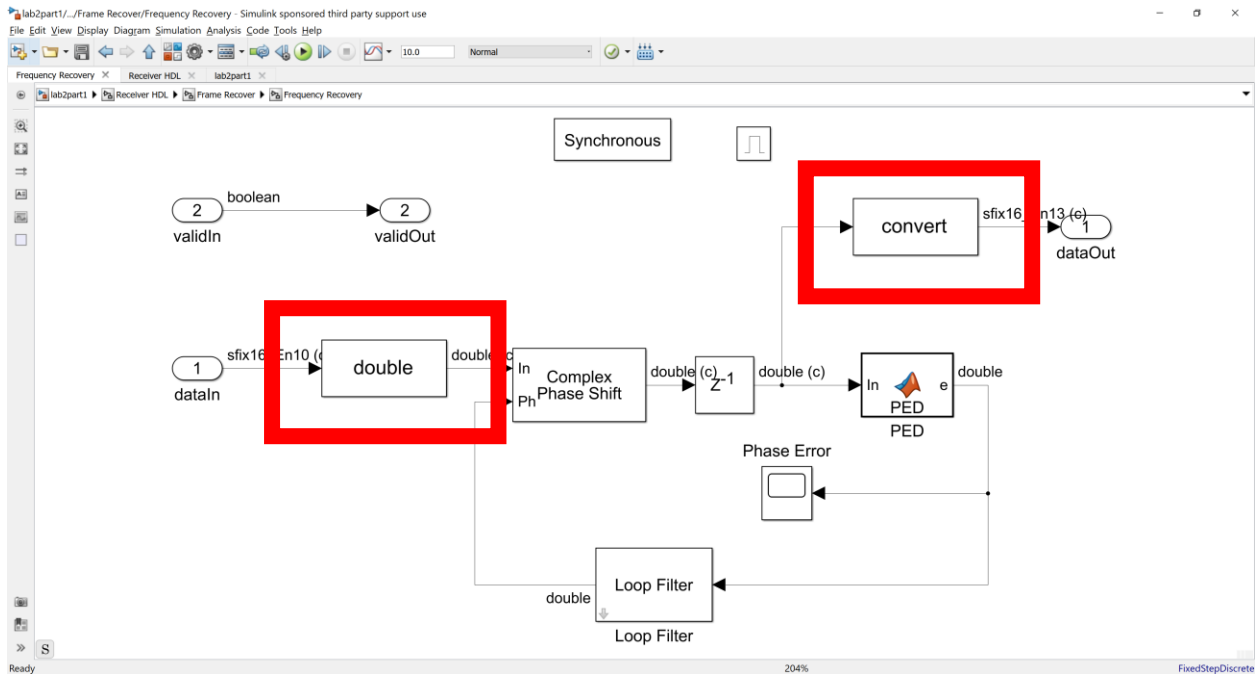
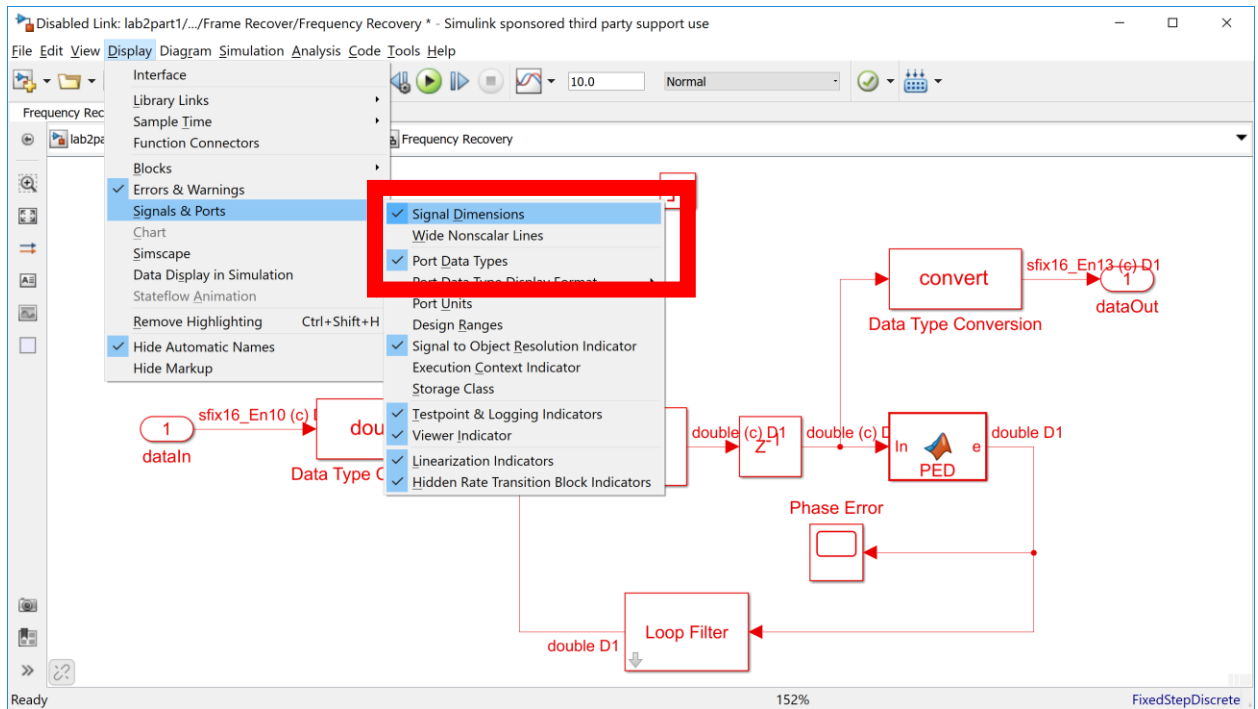


Figure 8

To make the conversion process easier, we will enable some additional information on the blocks and their connections. **From the top menu go to Display->Signals & Ports and enable both “Signal Dimensions” and “Port Data Types”.** Make sure both are checked. Next, compile the model by pressing **Ctrl+D**.

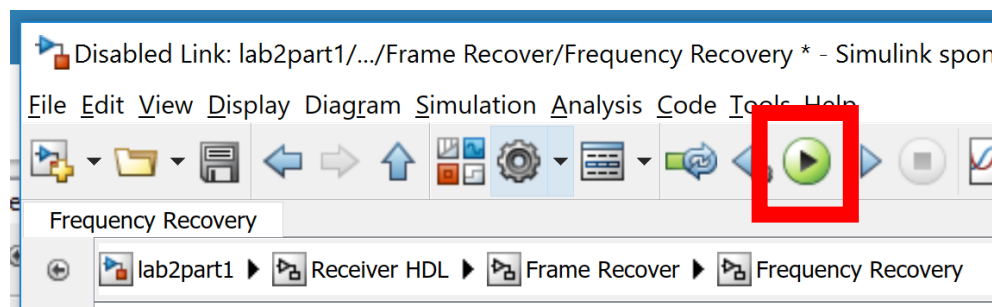


This will enable type information to be visible between blocks, which we will be modifying for fixed-point conversion. As an example, the first block after the “dataIn” port is shown in Figure 99. The data type of the incoming data is sfixed16_En10 (c). This means that the data is signed fixed point (the sfix), is a 16-bit number, has 10 fractional bits, and is complex. The output is simply a complex double.



Figure 9 Data type conversion block in Frequency Recovery Subsystem

Next let us run the model and view the diagnostic plots and messages. **Click the green play button on the top bar.**



A series of Constellation Diagrams will appear which will provide us information on the recovery process before and after the “Frequency Recovery” block, alongside a time scope for the estimation error of the “Phase Error Detector”, as shown in Figure 1010. There will be noise in the data since this data was captured after being transmitted over the air.

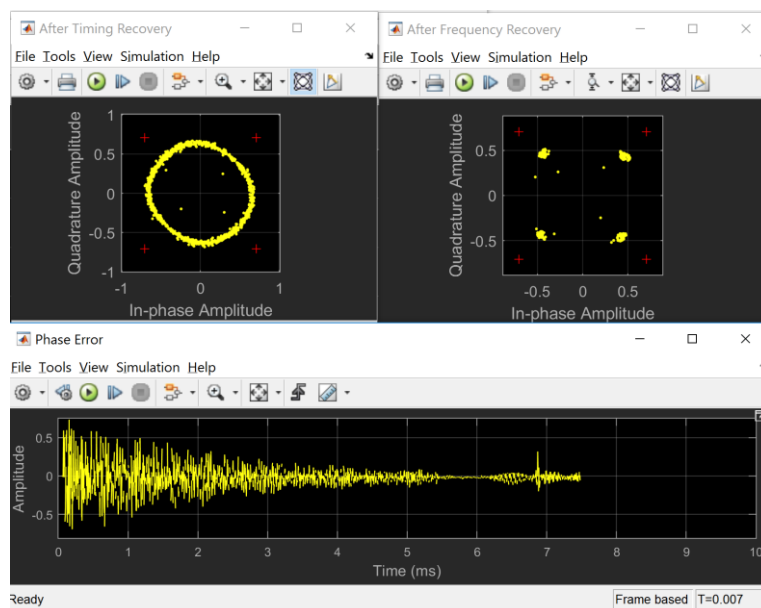


Figure 10 Diagnostic scope for Frequency Recovery Block

Since we are using double precision for our frequency synchronization, we will have optimal performance. However, this cannot be deployed onto the FPGA at our target speeds. Therefore, we must convert the blocks between our “Data Type Conversion” blocks to be fixed-point compatible.

Adding blocks in Simulink (The Fast Way)

Throughout this lab, you will be asked to insert blocks into the model. The easiest way to do this is by clicking on any open whitespace within the model, and then start typing the block name. This is demonstrated in Figure 11, where a Constellation Diagram block is being added.

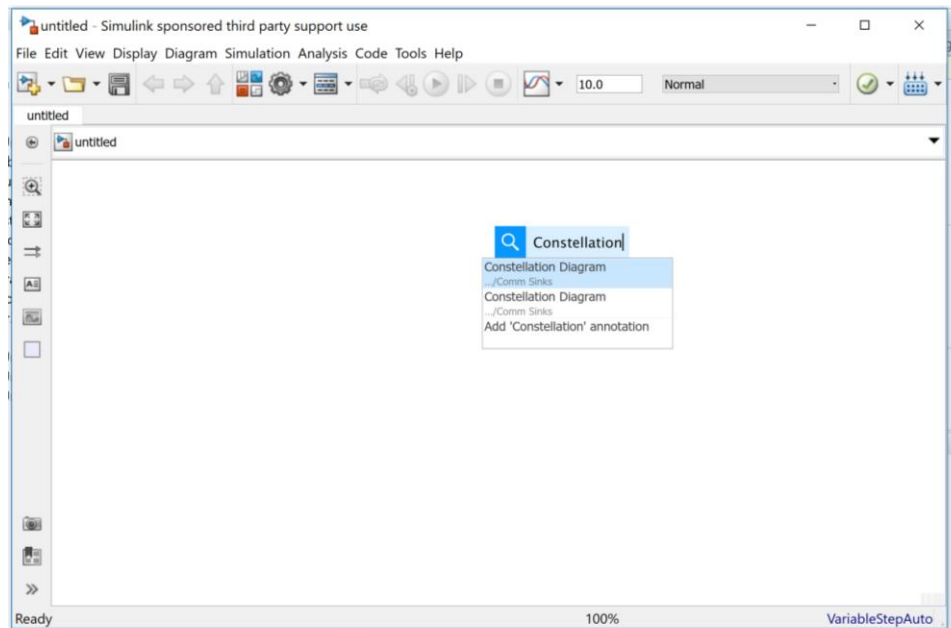


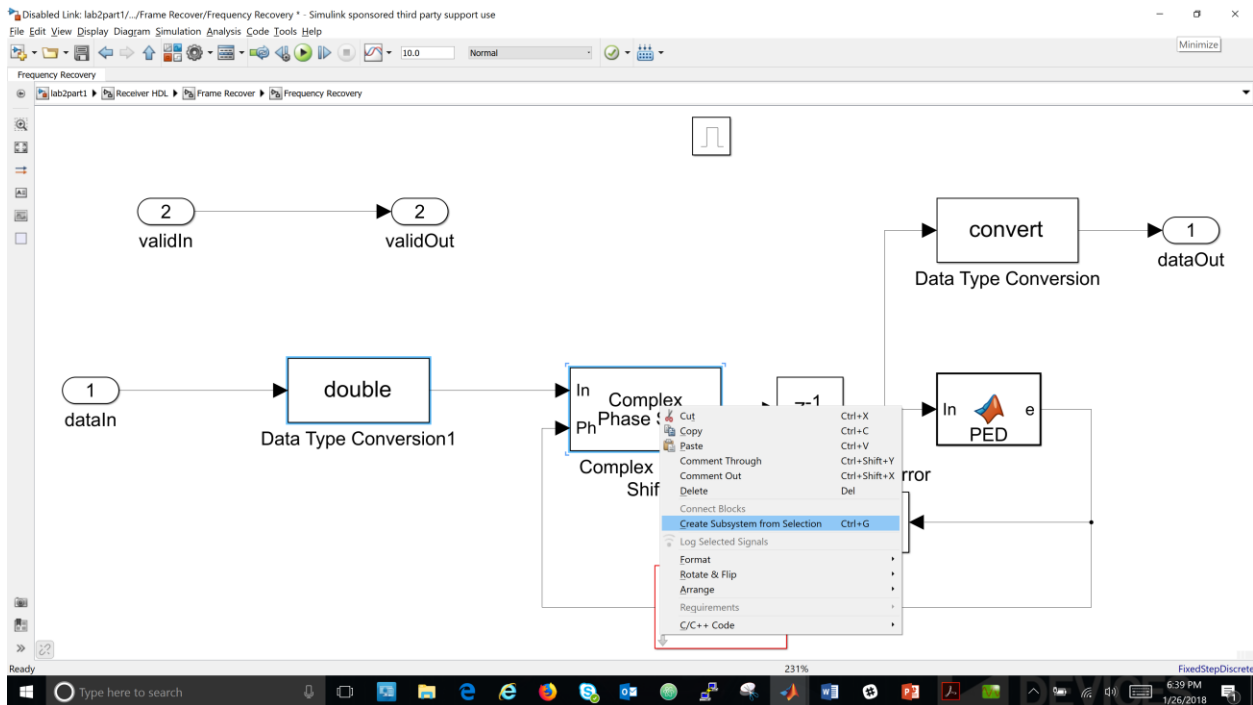
Figure 11

Converting the Design

In this section of the lab we will be working through the “Frequency Recovery” block to convert the design section-by-section to fixed-point. Sometimes we can automate this process by using the Fixed-Point Tool and sometimes we may want to manually tune the fixed-point configuration. We will be performing this task here in order to get a better understanding of how fixed-point data types work in MATLAB and Simulink, and how we can perform the conversion manually. Generally, this is a very time-consuming process, but we have predetermined the design path in order to fit within the lab session. Nonetheless, once completely converted, users are free to modify the implementation in any way they wish.

When converting floating-point sections to fixed-point, we always move from the source data forward. This is done because bit propagation can be easily managed by starting with our input data, whose range we know, and working through the signal chain, rather than assuming an output data type and working backwards.

Start by selecting the “Data Type Conversion1” block and the “Complex Phase Shift” block (Shift and click to do this). Once both are highlighted, right click on either block and select “Create Subsystem from Selection” in the popup menu. This will create a subsystem from these blocks for easier manipulation moving forward. **Now double-click on this new subsystem which should be labeled “Subsystem”.** We created this subsystem to better organize our work since the conversion of this section can become a mess to visualize and manipulate. Using generic subsystem blocks has no effect on the performance of the generated HDL code.



Now that we are within our newly created subsystem we can start adding and replacing blocks. In this Subsystem we need to replace the “Complex Phase Shift” block, which is a transcendental mathematical operation. Simulink has many of these functions implemented in HDL-friendly implementations within the base HDL library. However, before we can replace the “Complex Phase Shift” block with a fixed-point version we will add some additional “Data Type Conversion” blocks into the subsystem surrounding the “Complex Phase Shift” block. Place those additional blocks first as in Figure 12 and Figure 13. Be sure to select the “Output data type” parameter correctly. These values we predetermined for the sake of time in this lab.

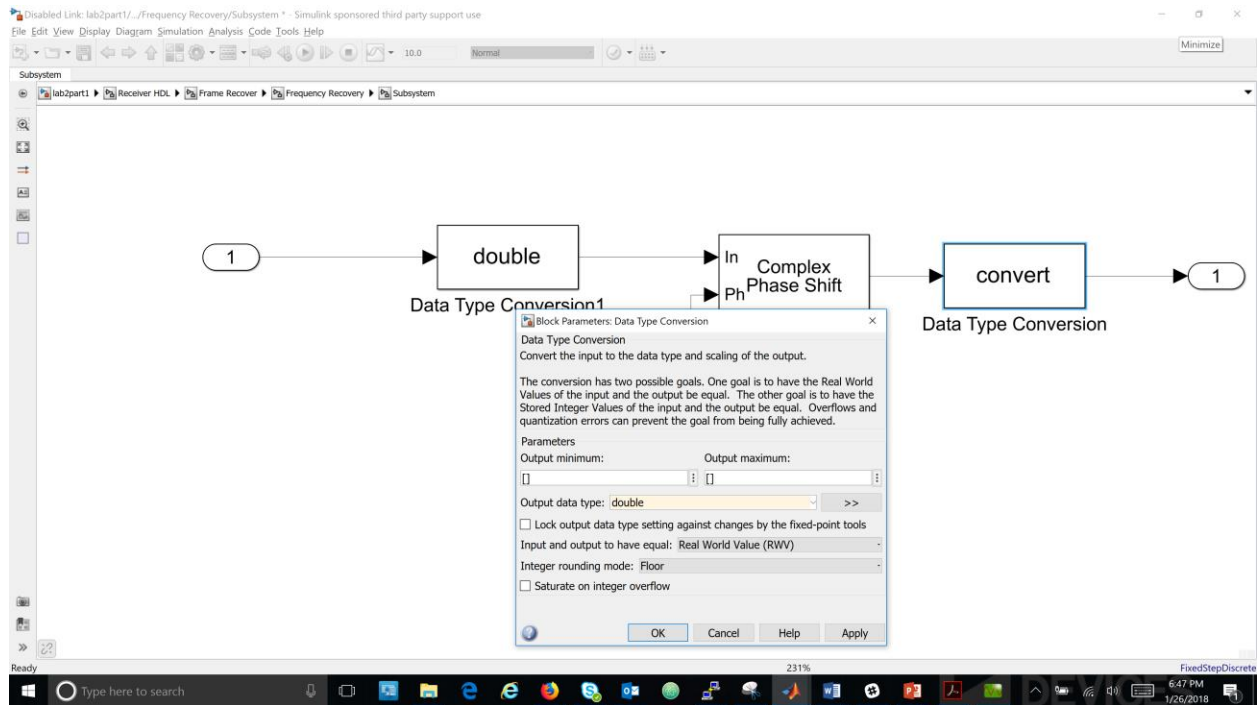


Figure 12

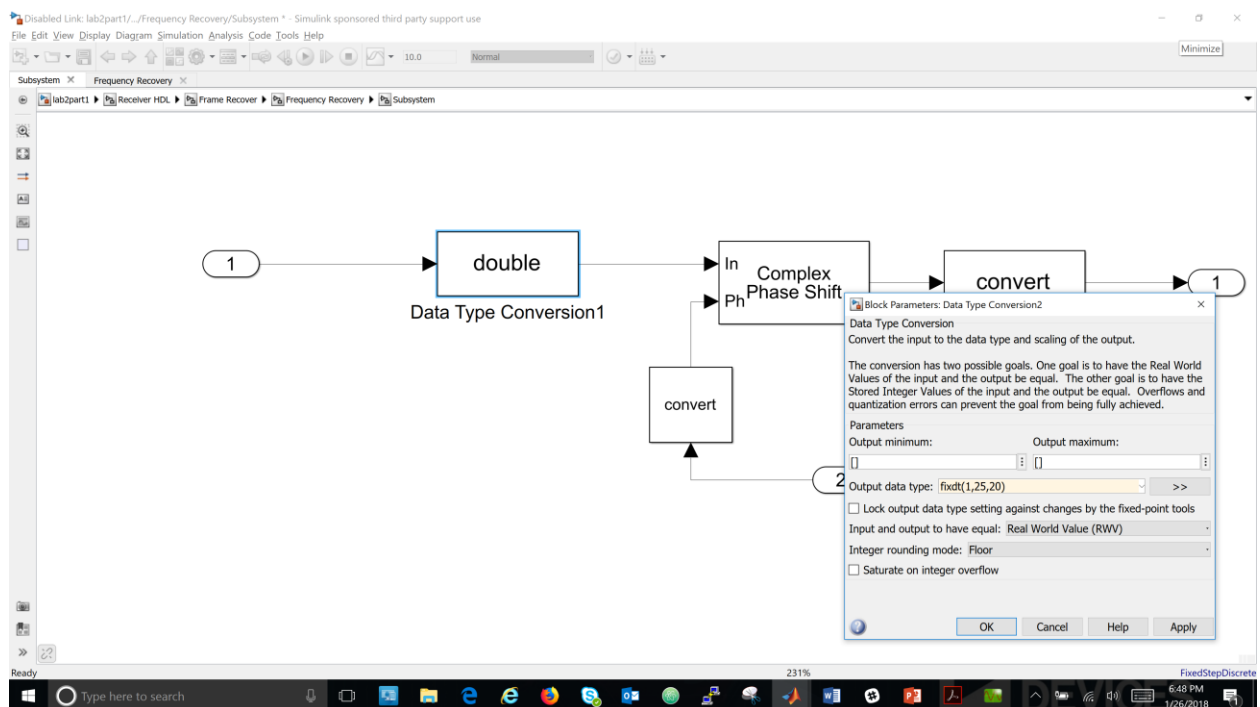


Figure 13

At this point we can replace the “Complex Phase Shift” block with two blocks: “Cosine HDL Optimized” and a “Product” block. The “Complex Phase Shift” block implements the following mathematical operation:

$$y = x * e^{-i\phi},$$

where x is our input signal and ϕ is our phase. The “Cosine HDL Optimized” block implements this block using a lookup table which is obvious based on the parameters available in the block. At this stage we will also remove the “Data Type Conversion” block connect to “in1” within our subsystem since we want all data to be of fixed-point type.

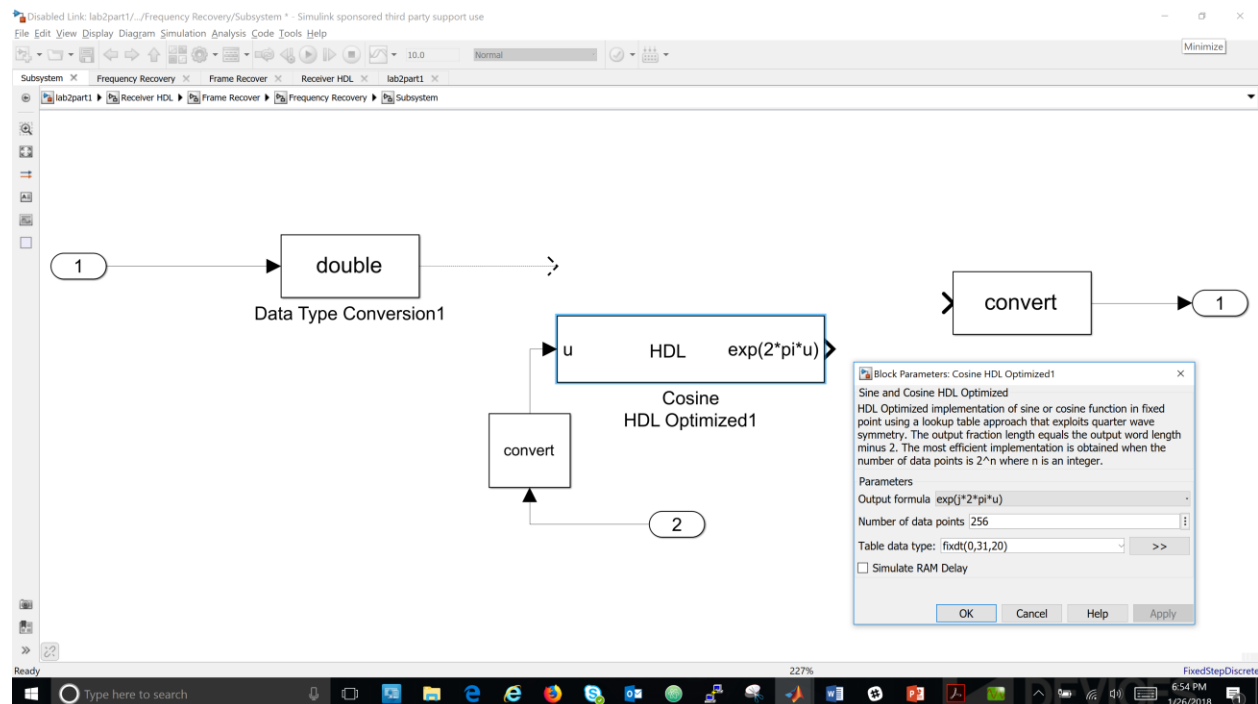


Figure 14

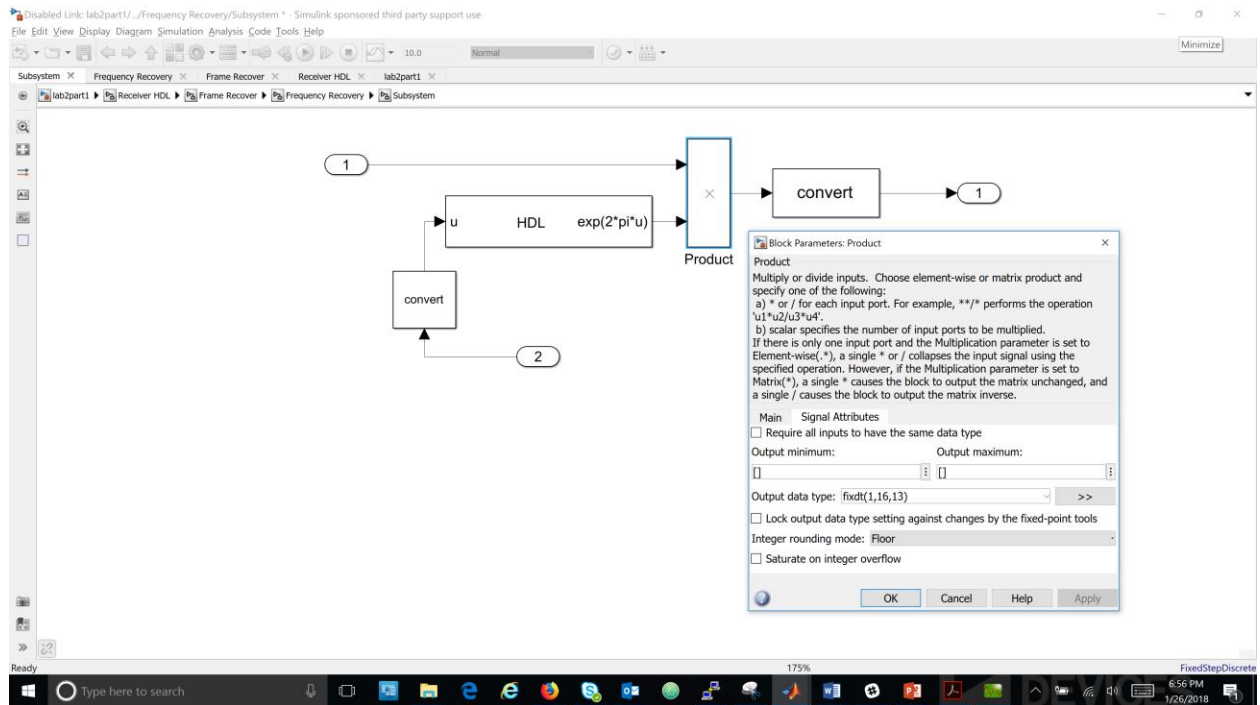


Figure 15

Since the “Cosine HDL Optimized” block scales the phase input by 2π , we need to fix this by scaling the input by $\frac{1}{2\pi}$. Do this by adding a “Gain” block with gain value $\frac{1}{2\pi}$ and with the “Output data type” shown in Figure 16.

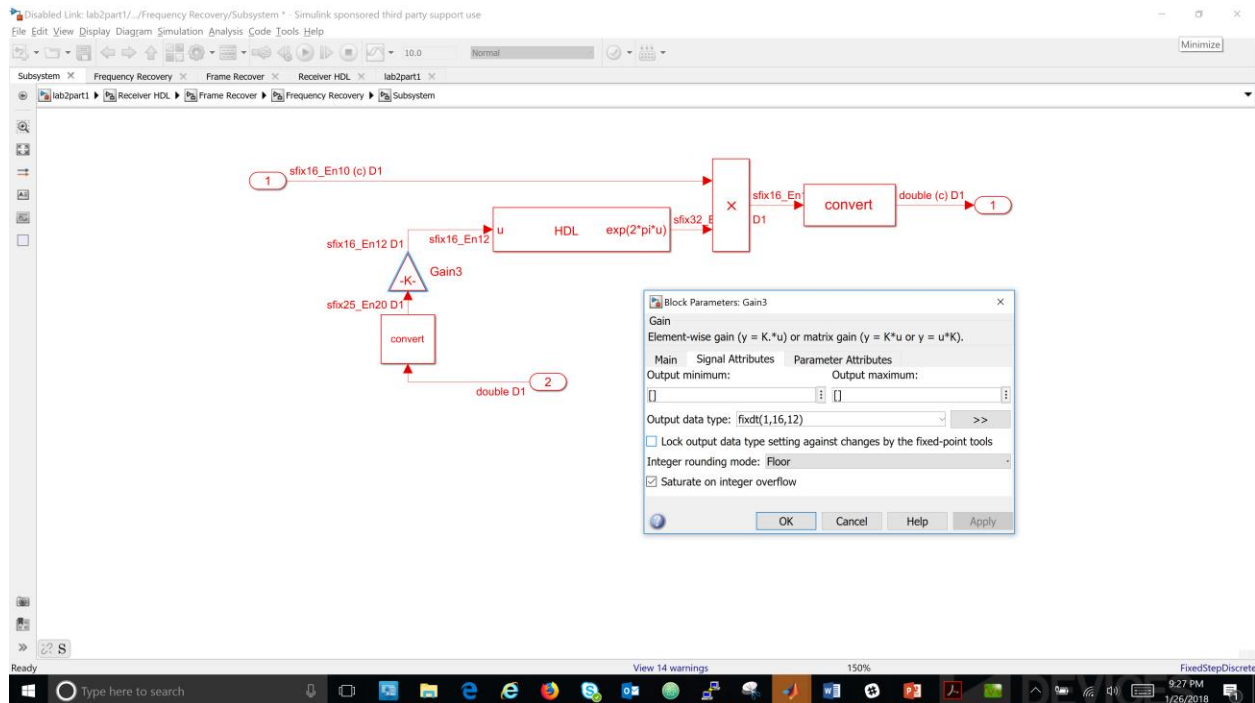

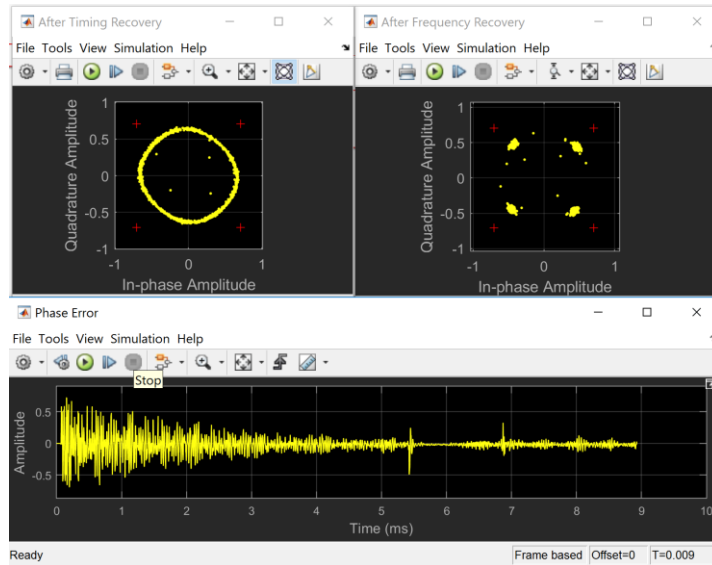


Figure 16

At this point we have successfully replaced the “Complex Phase Shift” block with the necessary fixed-point blocks and have migrated our “Date Type Conversion” blocks further into the design. To verify the

algorithmic operation of these blocks, press the play button  again and view the constellation plots for convergence of the constellation. Under the diagnostic panel, at a Simulation time of 9e-3 seconds, a packet should have been successfully detected (of size 3840 bits).



At this point we can go back up to the primary “Frequency Recovery” subsystem using the “Up to



parent” button on the top bar. Your view should look similar to **Error! Reference source not found.7.**

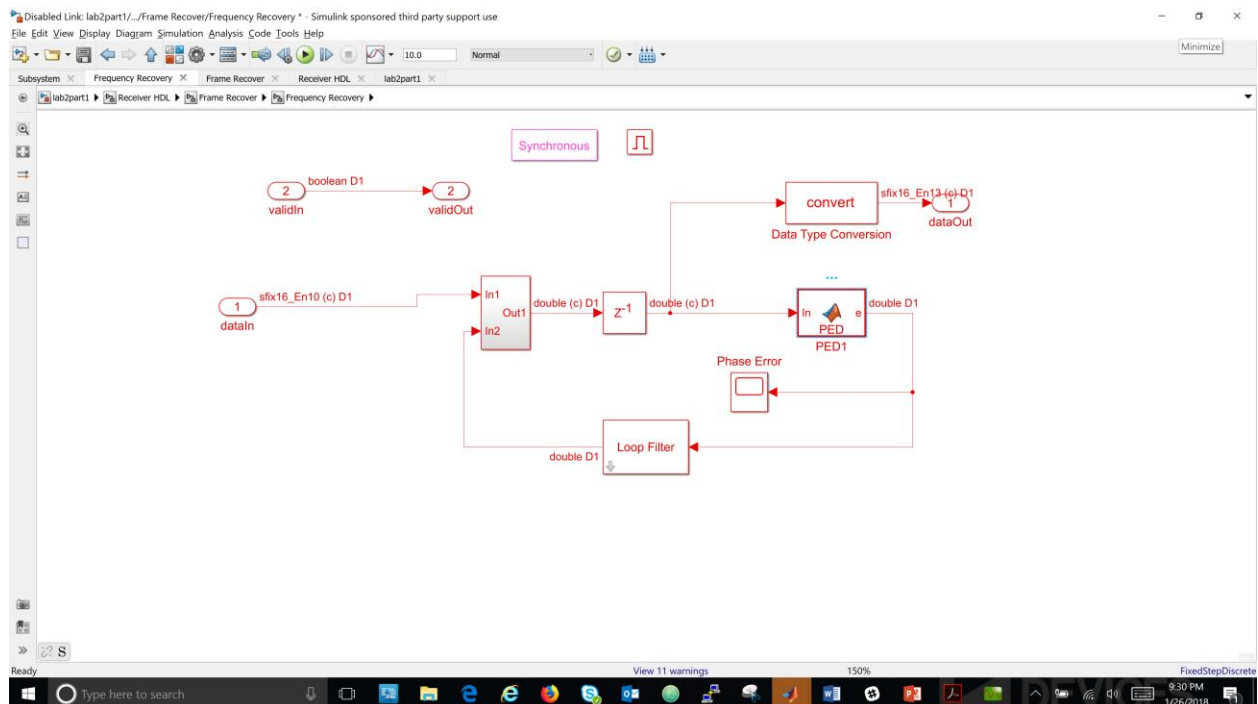


Figure 17

The next block we will address is the “PED” (Phase Error Detector) block. This block was implemented in a MATLAB function block. This example shows how MATLAB code can be used within our model and still generate HDL code. Start by double-clicking on the “PED” block, which will open the MATLAB editor as shown in Figure 18.

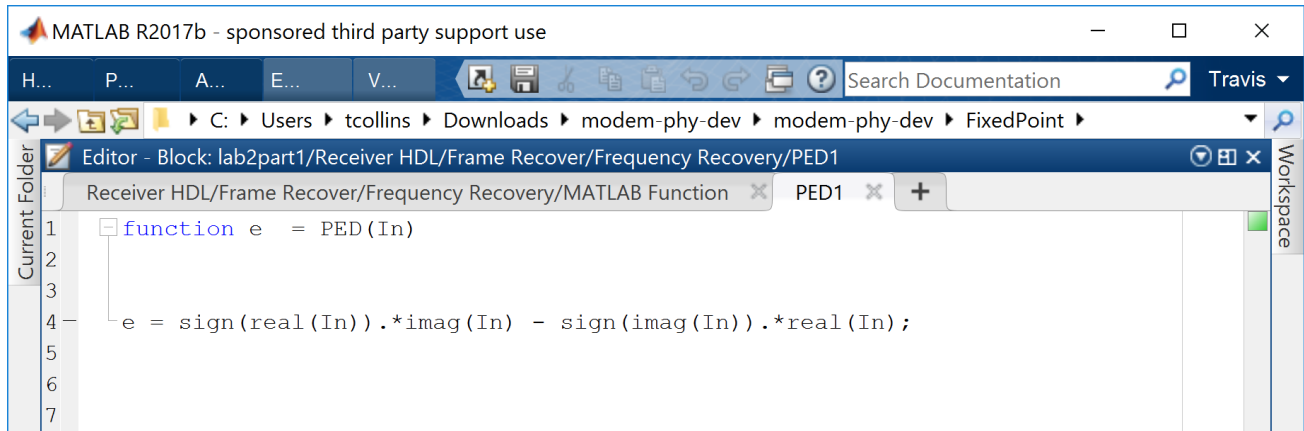


Figure 18

Next, update this code by copying from the code from the Code 1 window below, which should look similar to Figure 19. The PED measures the phase error of the rotated signal (assuming QPSK) through a simple equation. Since we only need a few lines of code to implement this process, we elected to keep it in MATLAB, rather than use a Simulink block for implementation. When using fixed-point data in MATLAB, our data becomes of type **fi.embedded**, which you can see through the use of **fi** casting. This function has the following API:

`fi(<Input Data>, <Signed>, <Word Length>, <Fraction Length>).`

```

function e = PED(In)

OP2 = real(In);
OP1 = imag(In);

Real = OP2;
Imag = OP1;

if (bitget(Real,16)==true) % real component is negative
    OP1 = fi(-Imag,1,16,13);
end
if (bitget(Imag,16)==true) % imaginary component is negative
    OP2 = fi(-Real,1,16,13);
end

e= fi(OP1-OP2,1,16,10);

%e = sign(real(In)).*imag(In) - sign(imag(In)).*real(In);

```

Code 1

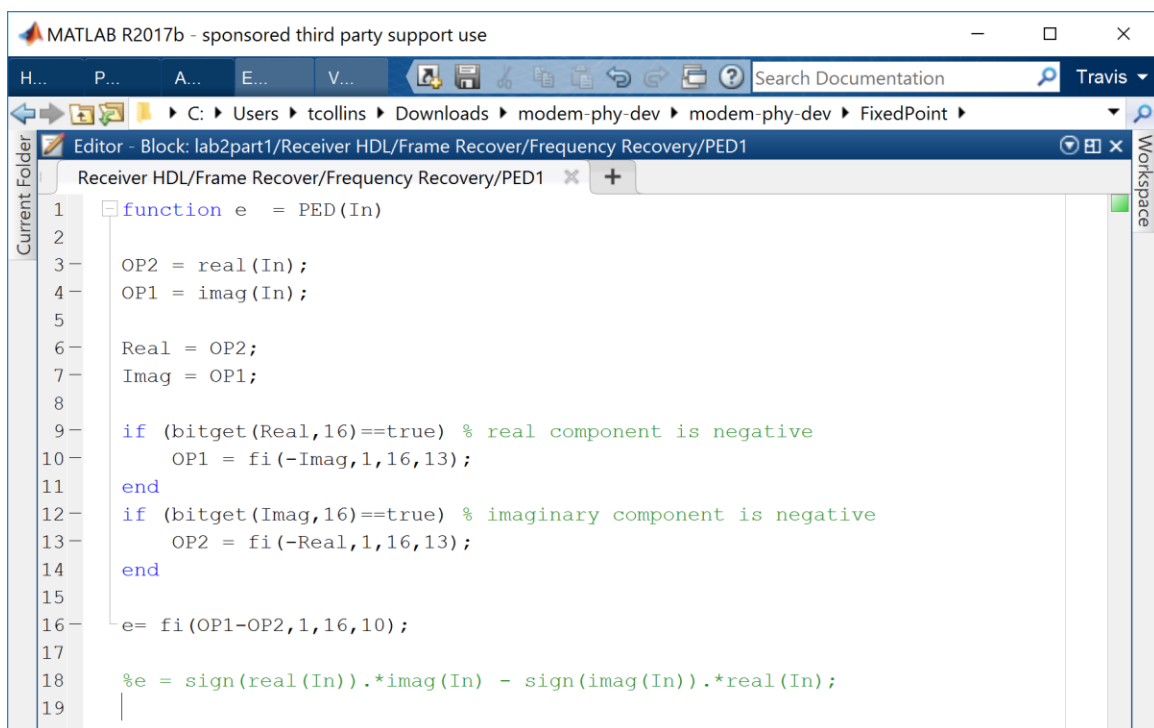


Figure 19

Next, we need to update the model topology again to pass fixed-point data into our updated PED block. ***Within the previously created Subsystem, delete the “Data Type Conversion” block connecting the “Product” block to the “out1” block.*** Your “Subsystem” block should now look like Figure 20.

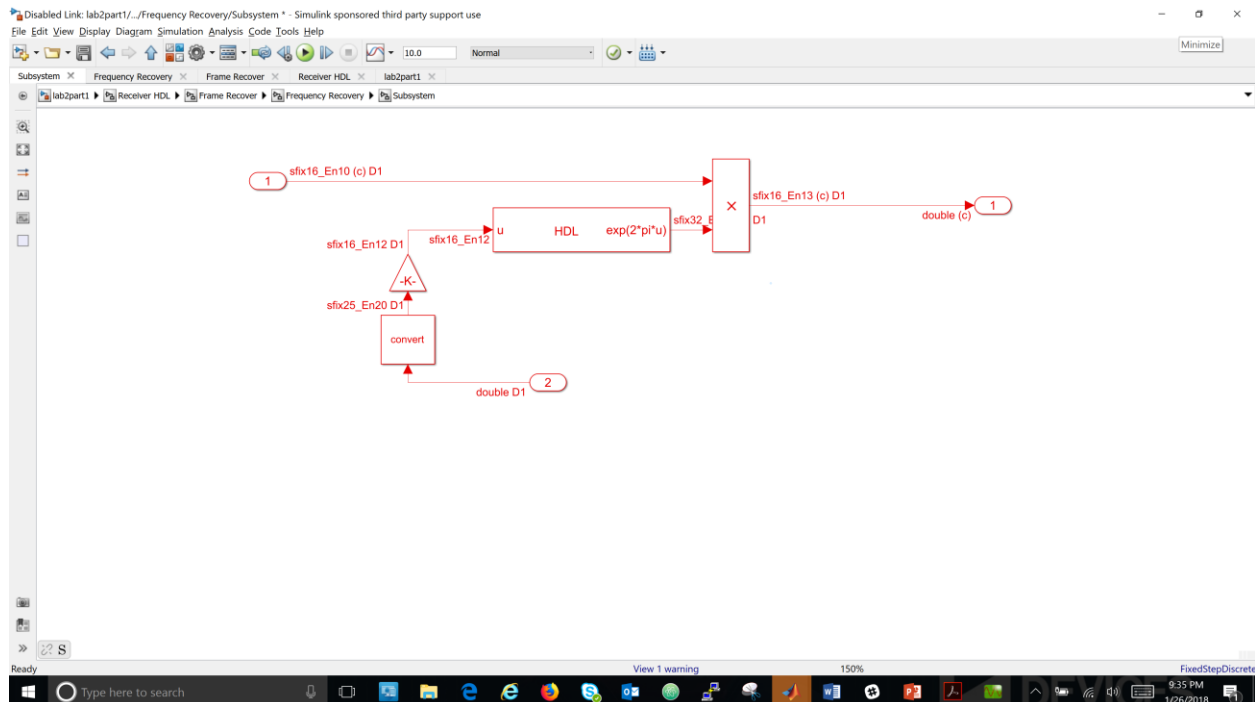


Figure 20

Now we can remove the “Data Type Conversion” block connected to the “dataOut” port, since this path is now represented in the desired Fixed-Point data type. The last remaining double precision blocks are located within the “Loop Filter” block, so we still need to have a “Data Type Conversion” block casting the output of the “PED” block into the “Loop Filter” block.

Insert a “Data Type Conversion” between the “PED” and “Loop Filter” block similar to Figure 21.

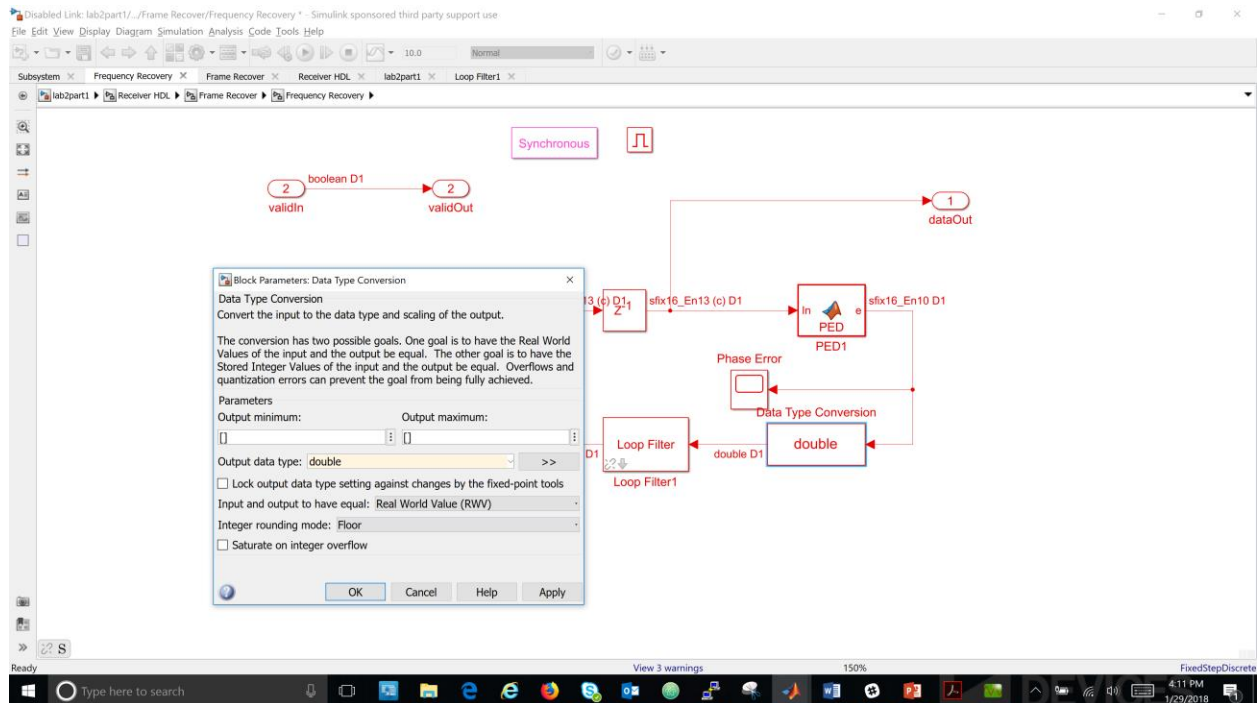

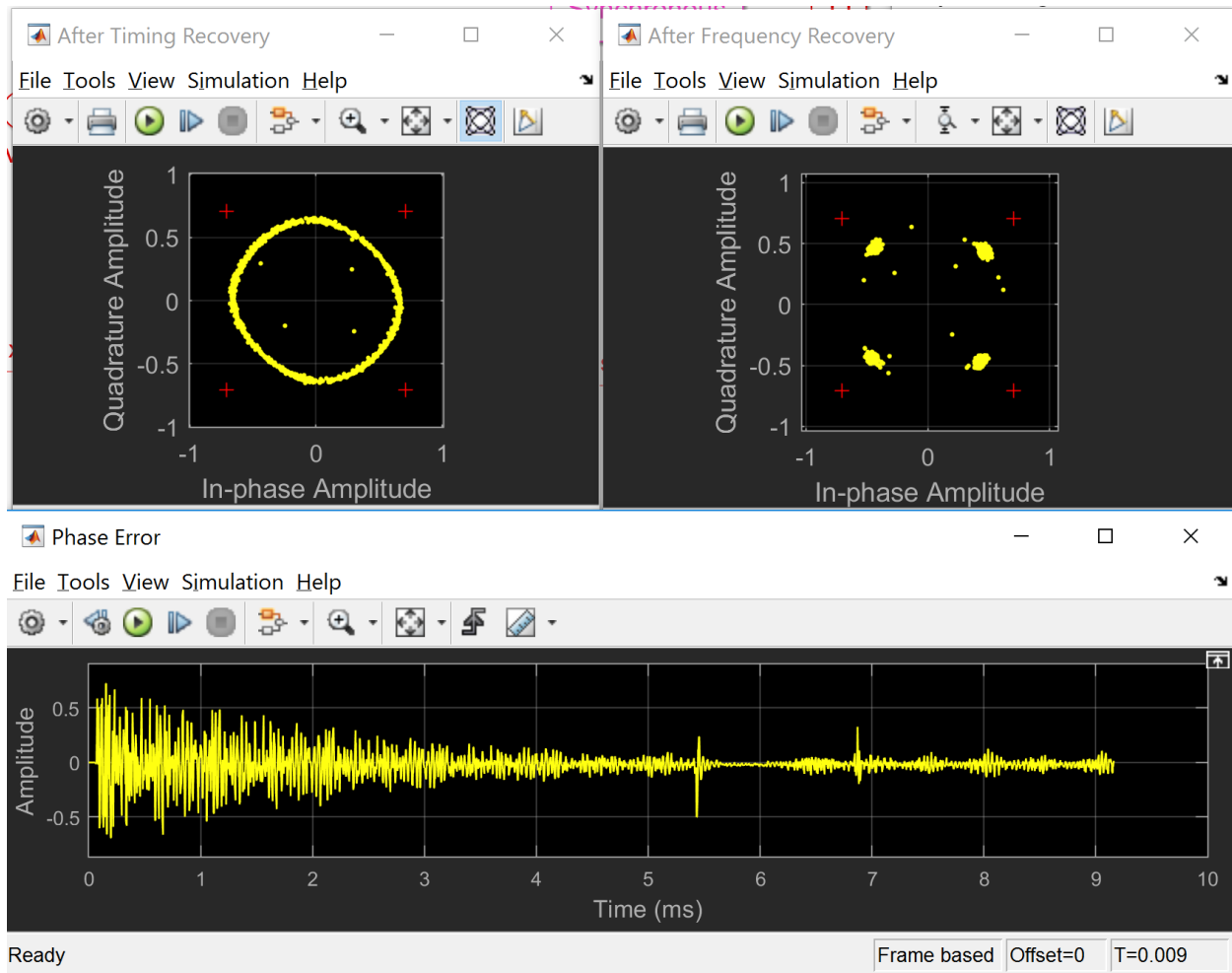


Figure 21

At this point we have successfully replaced the “PED” block and placed our “Date Type Conversion” blocks further into the design.

To verify the algorithmic operation of these blocks, press the run button  and view the constellation plots for convergence of the constellation. Under the diagnostic panel, at a Simulation time of 9e-3 seconds, a packet should have been successfully detected.

What is the message that was displayed?



Now we can consider our last Subsystem the “Loop Filter”. ***Since this block is made of atomic Simulink blocks we can simply remove the “Data Type Conversion” block feeding this subsystem and directly connect it to the “PED” block. Once removed, recompile the model with Ctrl+D.*** Your model should now look like Figure 22. Due to the default settings of the blocks within the “Loop Filter” subsystem, certain data types will propagate automatically. This is a very powerful feature of Simulink.

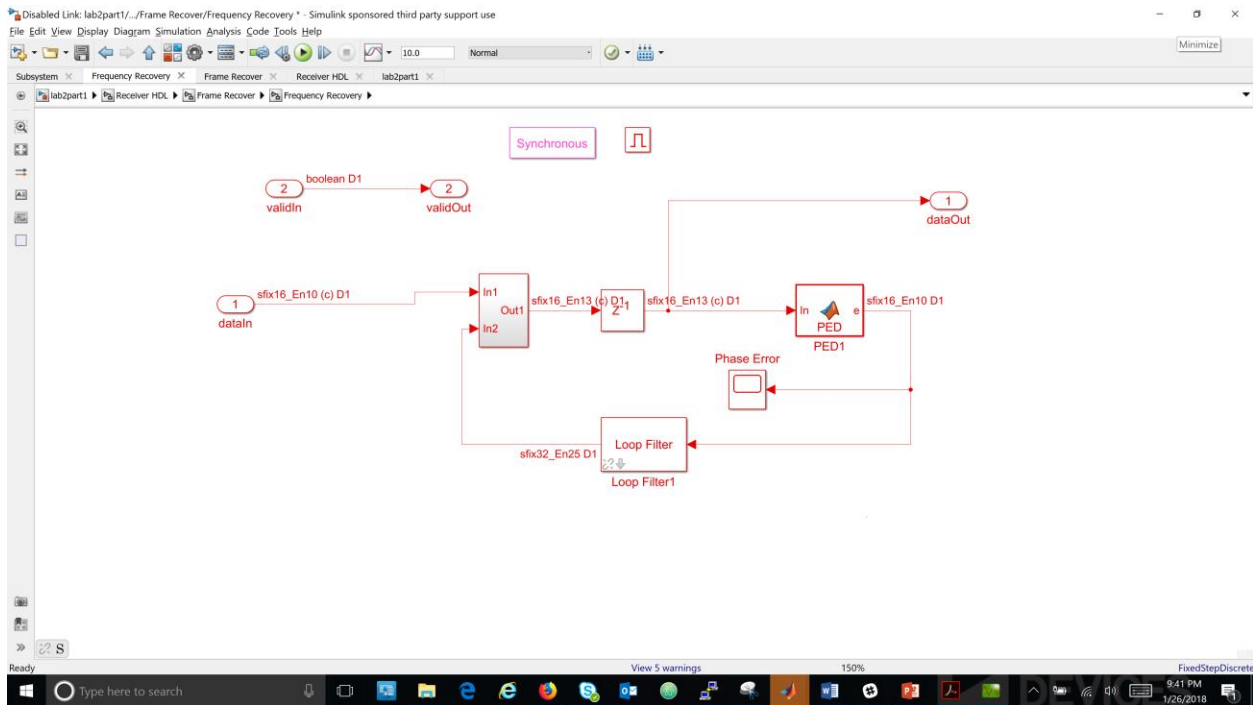


Figure 22

Next, let us look inside the “Loop Filter” subsystem by clicking on the “Look inside mask” icon on the “Loop Filter block” which is outlined in Figure 23.

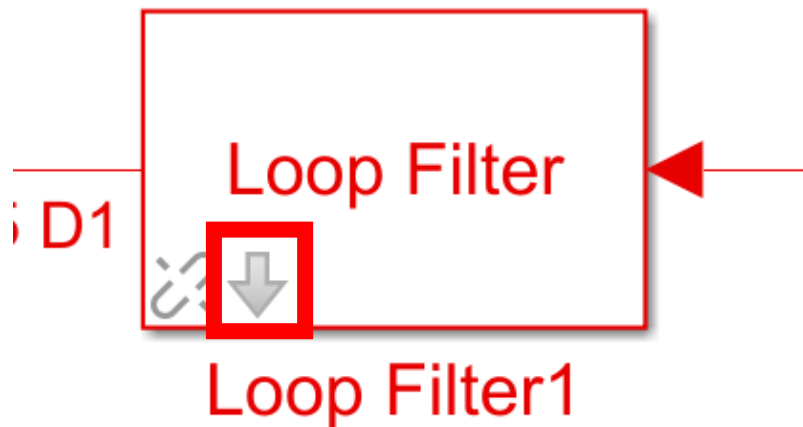


Figure 23

This block, shown in Figure 24, contains our filtering implementation with a small set of atomic blocks. If we consider the last mathematical block in the flow, the gain block with value -1.0, you will notice a rather large type inflation from an input of **sfix16_En10** to **sfix32_En25** for just a sign change of the data. This is very undesirable since it will require a 32-bit multiplication, requiring more resources and adding possible delay into the system.

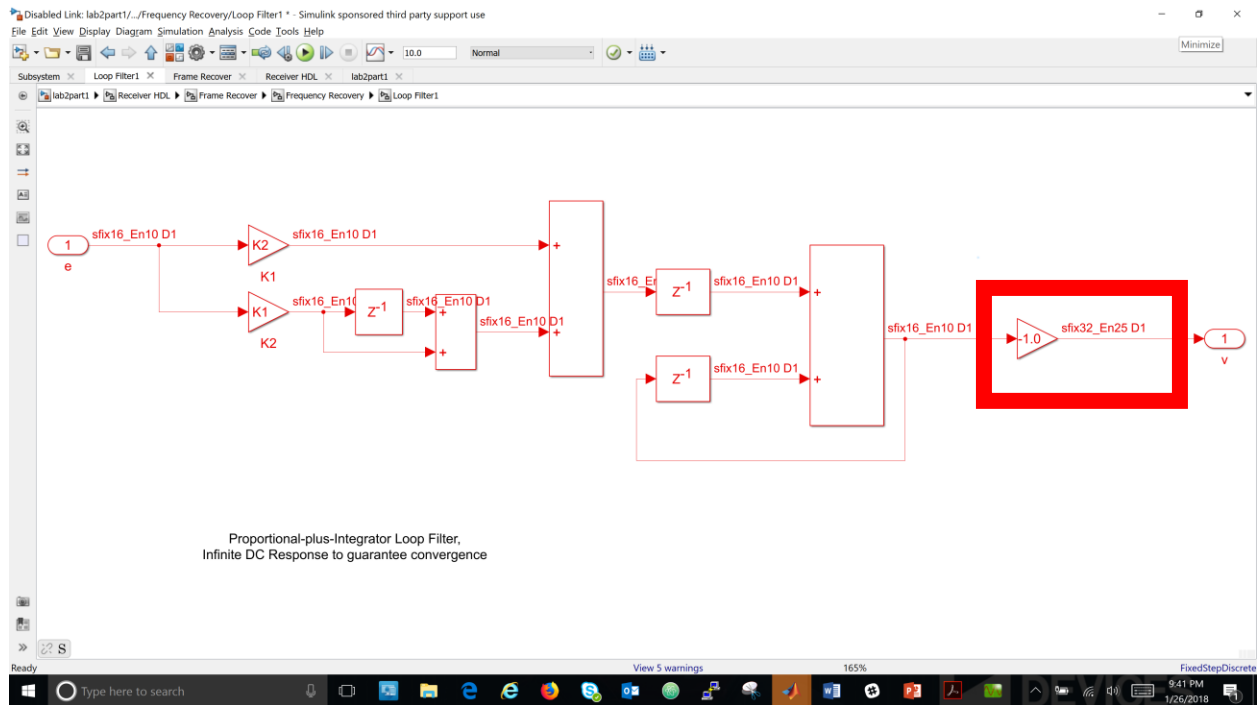


Figure 24

Let us start by removing the “Gain” block of interest and connecting the “Product” block directly to the subsystem’s output “v”. Your model now should look similar to Figure 25.

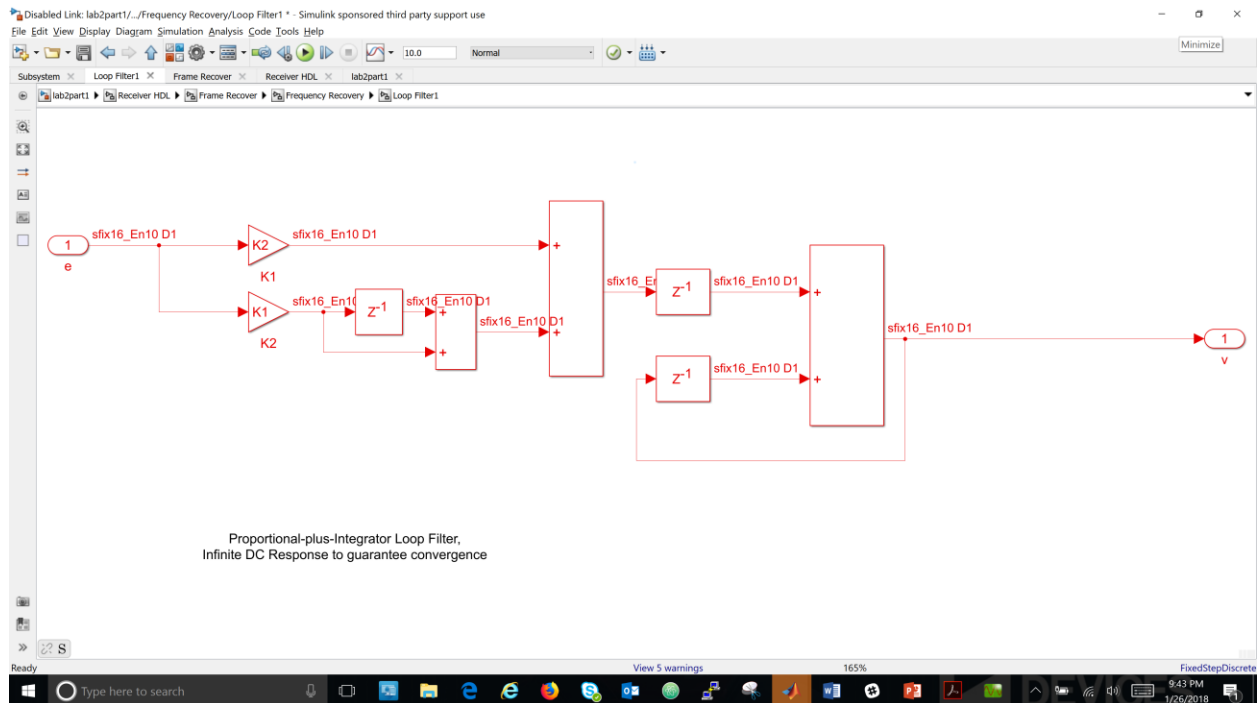


Figure 25

The negation we removed with the gain block needs to be added back into the signal chain, which we can do by updating the “Gain” block in the “Subsystem” we created earlier containing our “Cosine HDL Optimized” block. **Modify the gain block in this subsystem to include the negative sign. Update the mask of the “Gain” block as shown in Figure 26.**

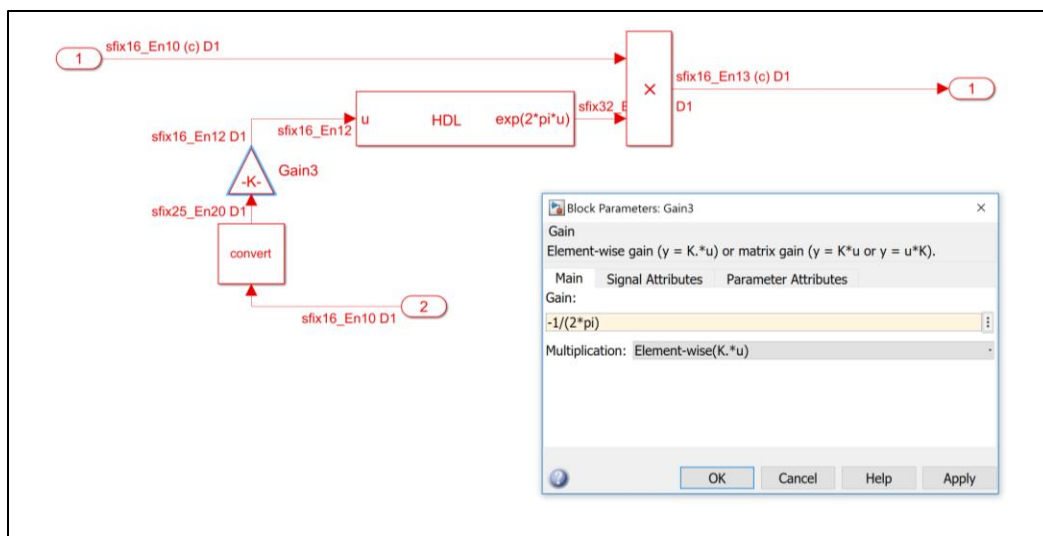
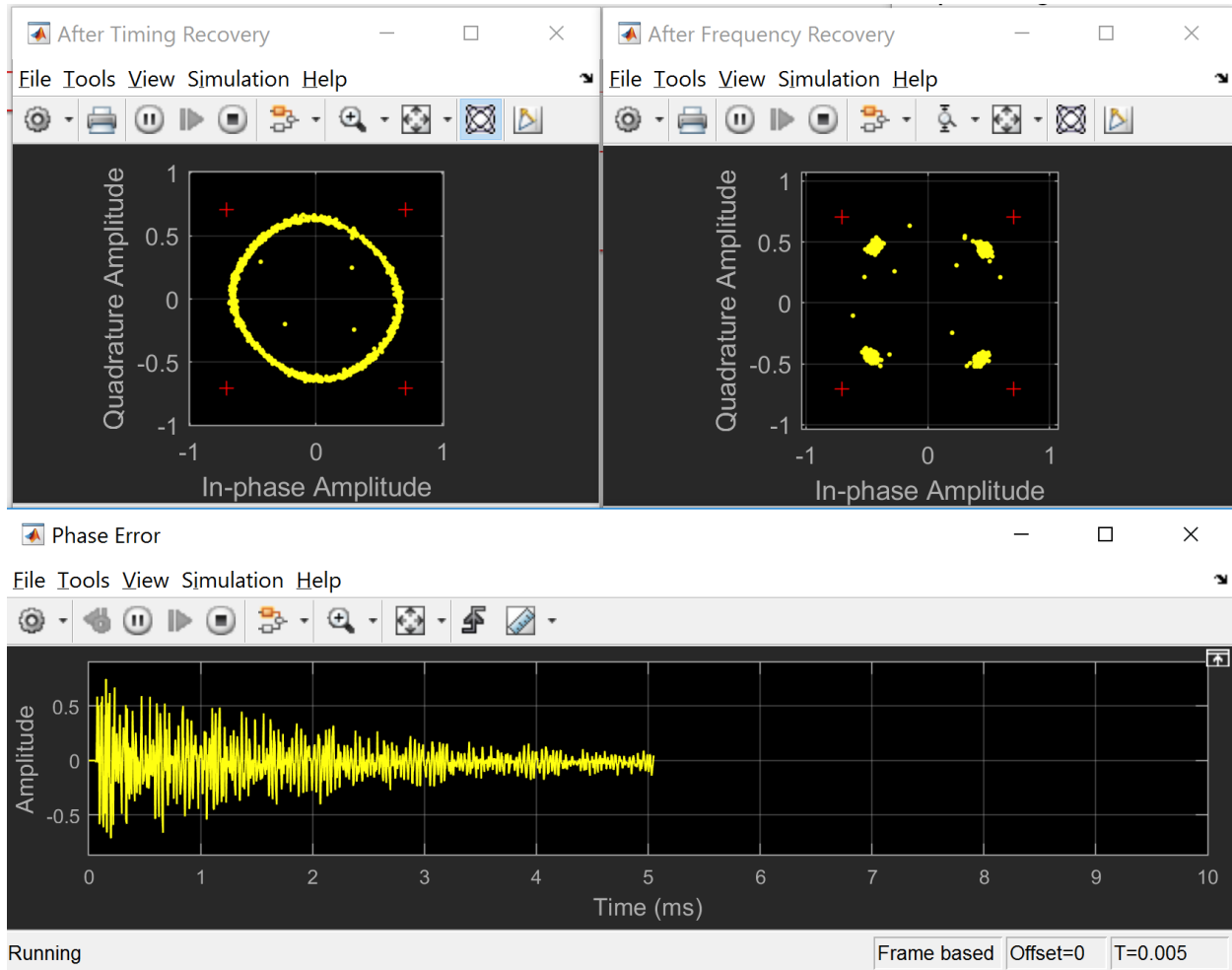


Figure 26

At this point we have successfully replaced all the double precision blocks in the design.



To verify the algorithmic operation of these blocks, press the run button and view the constellation plots for convergence of the constellation. Under the diagnostic panel, at a Simulation time of $9e-3$ seconds, a packet should have been successfully detected. Is this message identical to previous messages?



Extra Time

If you have extra time, try to reduce the output data type of the “Cosine HDL Optimized” block in the custom Subsystem we added. If possible try to reduce the output to utilize 16 bits, which would reduce the complexity of the downstream multiplications. The default, which is outlined in Figure 27, outputs a `ufix31_En20`. Each time you update the model, rerun the simulation to make sure the constellations converge and you can recover the same headers in the data.

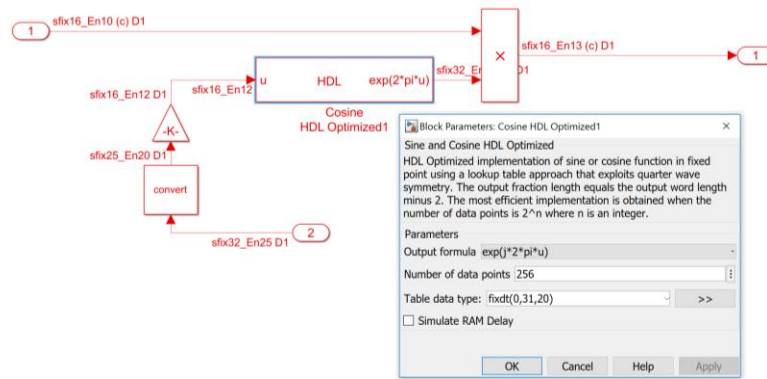


Figure 27