Pulse Detector HDL Workflow Tutorial R2019b

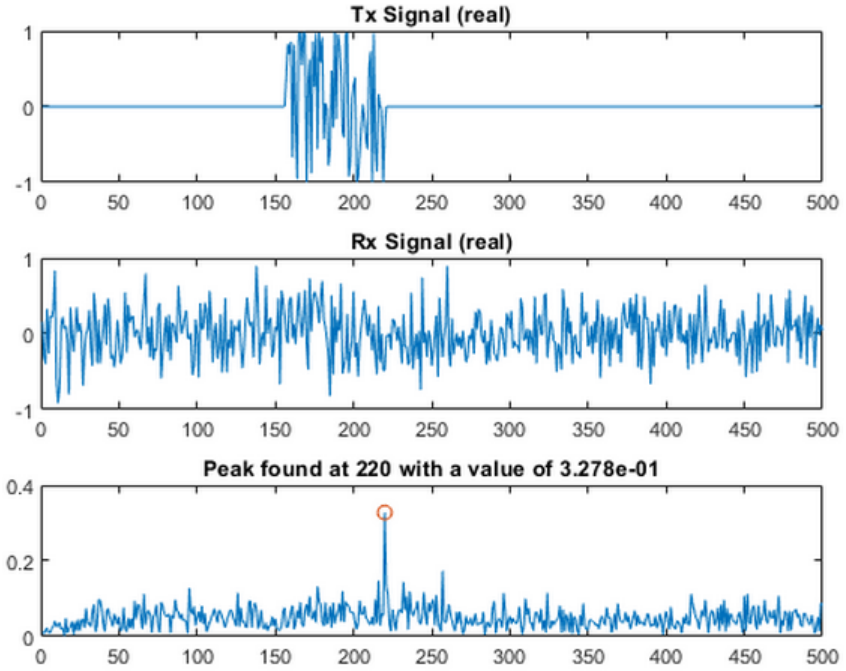
MathWorks Application Engineering

Example Overview

In this tutorial, you are provided the MATLAB® reference of a pulse detection algorithm. The algorithm detects a known waveform in a received signal using a matched filter, and finding the resulting peak. It is a commonly used technique in radar or wireless communication systems.

MATLAB golden reference

```
% Create matched filter coefficients
CorrFilter = conj(flip(pulse))/PulseLen;

% Correlate Rx signal against matched filter
FilterOut = filter(CorrFilter,1,RxSignal);

% Find peak magnitude & location
[peak, location] = max(abs(FilterOut));

% Print results
figure(1)
subplot(311); plot(real(TxSignal)); title('Tx Signal (real)');
subplot(312); plot(real(RxSignal)); title('Rx Signal (real)');

t = 1:length(FilterOut);
str = sprintf('Peak found at %d with a value of %.3d',location,peak);
subplot(313); plot(t,abs(FilterOut),location,peak,'o'); title(str);
```

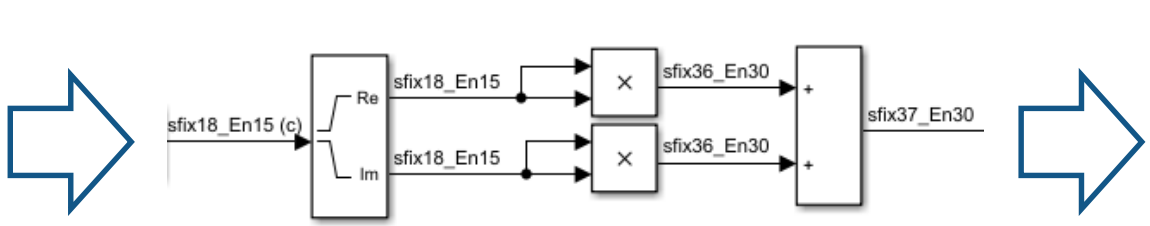


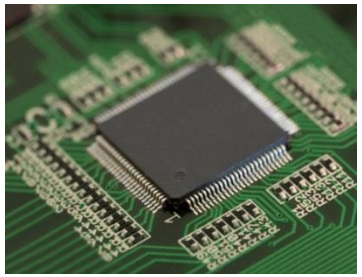
Example Overview

This tutorial will guide you through the steps necessary to implement the algorithm in FPGA hardware, including:

- Create a Simulink® model for the algorithm
- Implement the hardware architecture
- Convert the design to fixed-point
- Generate and synthesize the HDL code

```
% Create matched filter coefficients
CorrFilter = conj(flip(pulse))/PulseLen;

% Correlate Rx signal against matched filter
FilterOut = filter(CorrFilter,1,RxSignal);

% Find peak magnitude & location
[peak, location] = max(abs(FilterOut));
```





Software Requirements

- You will need the following MathWorks products:
  - MATLAB® (R2019b)
  - Simulink®
  - Fixed-Point Designer™
  - MATLAB Coder™
  - HDL Coder™
  - Signal Processing Toolbox™
  - DSP System Toolbox™

- Xilinx® Vivado® 2018.3 is required for the last step (step 4)
  - Other Vivado versions typically work, but synthesis results may be different.

Preparation

- You should be familiar with MATLAB and have basic knowledge of Simulink such as:
  - Creating Simulink models (adding & connecting blocks, creating subsystem, etc)
  - Setting block and model-level parameters
  - Updating, simulating model and viewing simulation results
  - Interacting with MATLAB using workspace variables and signal logging
  - Using fixed-point data type in Simulink

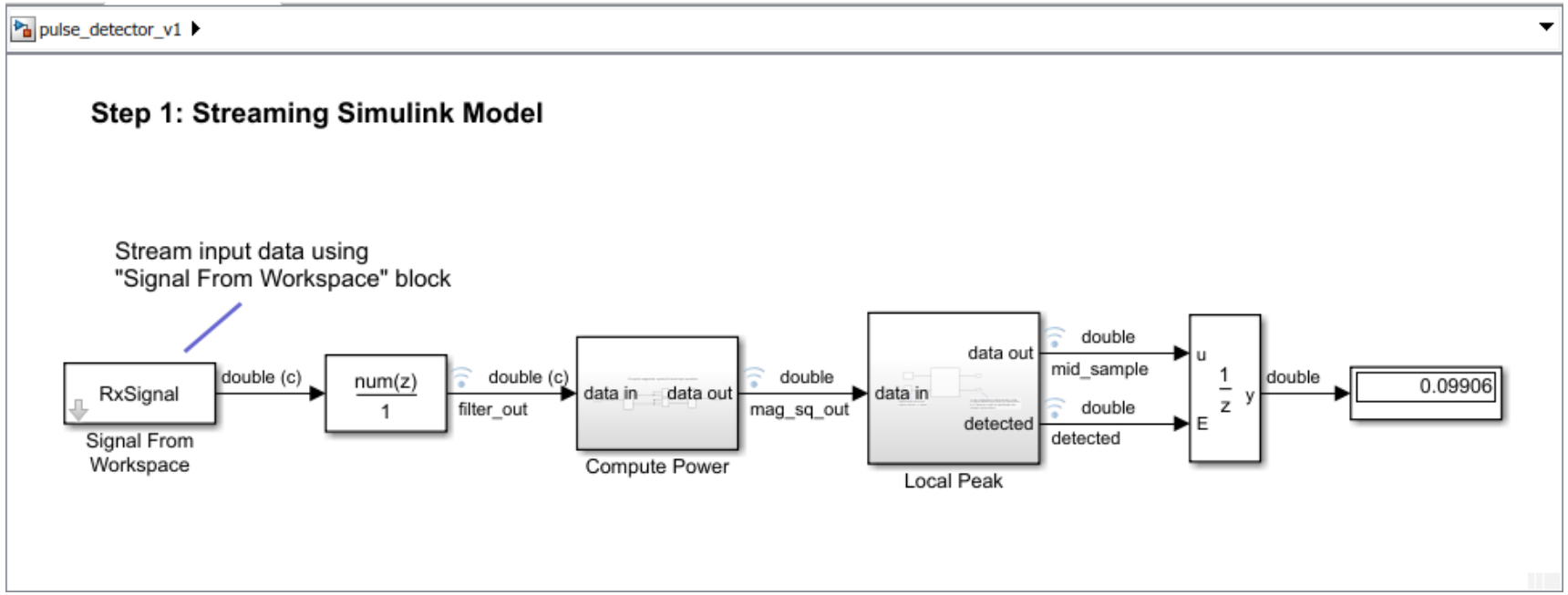
- Use the following resources as needed to prepare for this tutorial:
  - MATLAB Onramp
  - Simulink Onramp
  - HDL Coder Tutorial Overview (video series)
  - HDL Coder Evaluation Reference Guide

- Locate the example files in the folder: **/pulse_detector/work**

Step 1: Streaming Simulink model

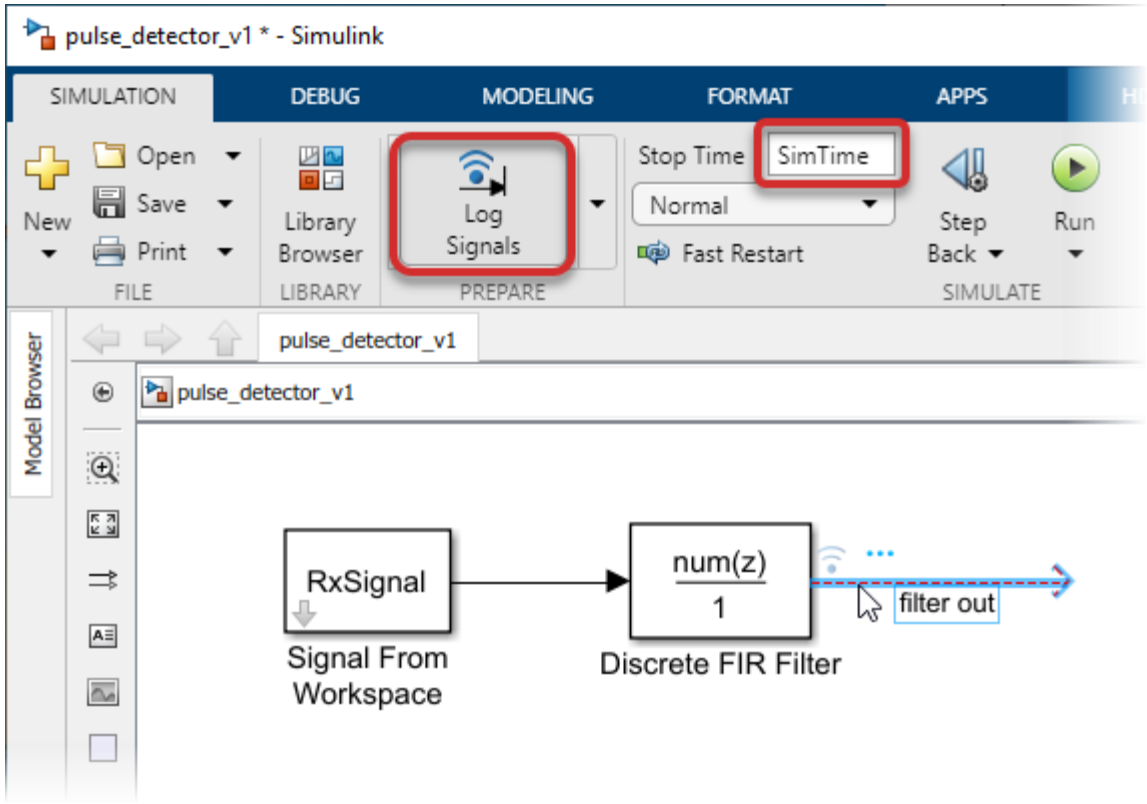
In this step, you will:

- Create a Simulink model with streaming input
- Implement a hardware-friendly peak finder
- Compare the Simulink pulse detector to the MATLAB golden reference



Step 1.1: Stream and filter input signal

1. Run **pulse_detector_reference.mlx** to initialize parameters needed in this step.
2. Create a new Simulink model and name it pulse_detector_v1.slx
3. Add a **Signal From Workspace** block, and enter RxSignal for `Signal` on the block dialog. This will stream the vector RxSignal one sample at a time.
4. Implement the filter function using a **Discrete FIR Filter** block. Enter CorrFilter for `Coefficients`.
5. Name the filter block output signal filter_out, and enable data logging for the signal.
6. Enter SimTime for simulation stop time.



Step 1.1: Stream and filter input signal

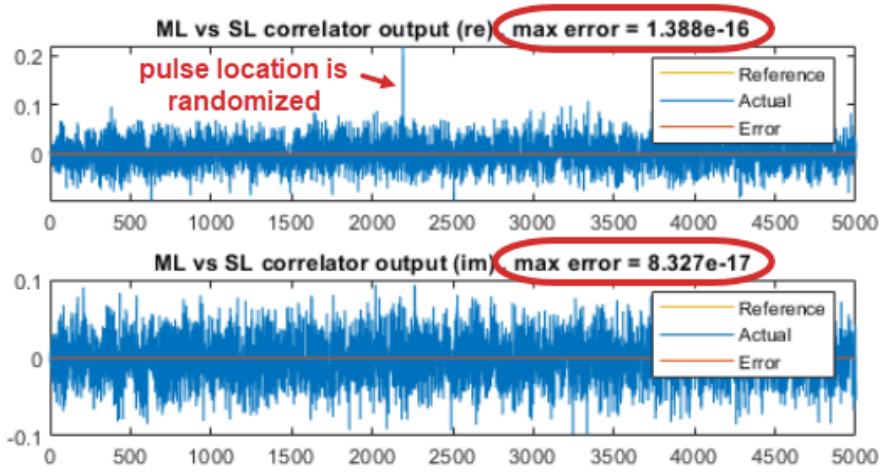
7. Compare the Simulink filter output against the MATLAB reference. Verifying your Simulink model incrementally helps catch mistakes early on.
   - In the test bench script **pulse_detector_v1_tb.mlx**, click **Run to Here** on line 16.
   - Verify the Simulink filter output matches the MATLAB reference (max error ~ $1e^{-16}$).
   - Click **Stop** to un-pause the test bench.

Simulate model and compare results to reference

```
2  if iscolumn(CorrFilter)
3      CorrFilter = transpose(CorrFilter); % need row vector for filter block
4  end
5  SimTime = length(RxSignal) + WindowLen;
6
7  % Simulate model
8  slout = sim('pulse_detector_v1');
9
10 % Correlation filter output
11 FilterOutSL = getLogged(slout,'filter_out');
12 compareData(real(FilterOut),real(FilterOutSL),{2 3 1},'ML vs SL correlator
13 compareData(imag(FilterOut),imag(FilterOutSL),{2 3 2},'ML vs SL correlator
14
15 % Magnitude squared output
16 MagSqSL = getLogged(slout,'mag_sq_out');
17 compareData(MagSqOut,MagSqSL,{2 3 3},'ML vs SL mag-squared output');
18
19
```

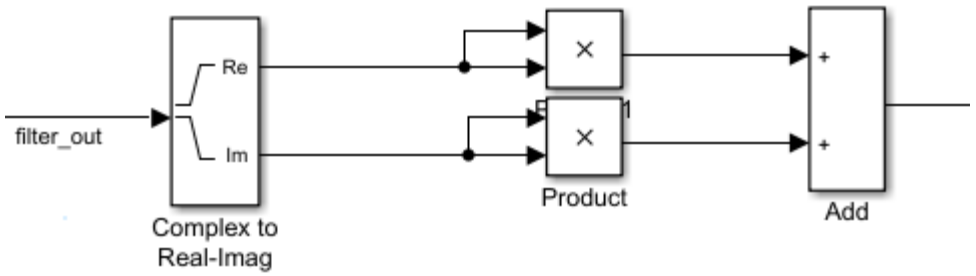
Run to Here
Run up to this line and pause

```
Warning: Unconnected output line found on 'pulse_detector_v1/Discrete FIR Filter'
(output port: 1)
Maximum error for ML vs SL correlator output (re) out of 5000 values
 1.387779e-16 (absolute), 6.288352e-14 (percentage)
Maximum error for ML vs SL correlator output (im) out of 5000 values
 8.326673e-17 (absolute), 8.414273e-14 (percentage)
```

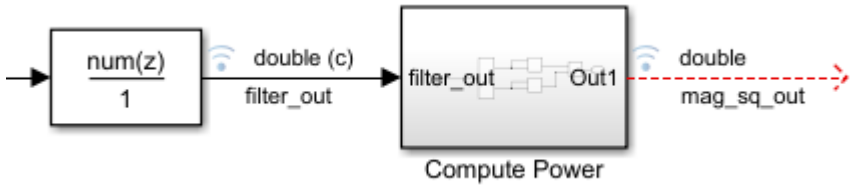


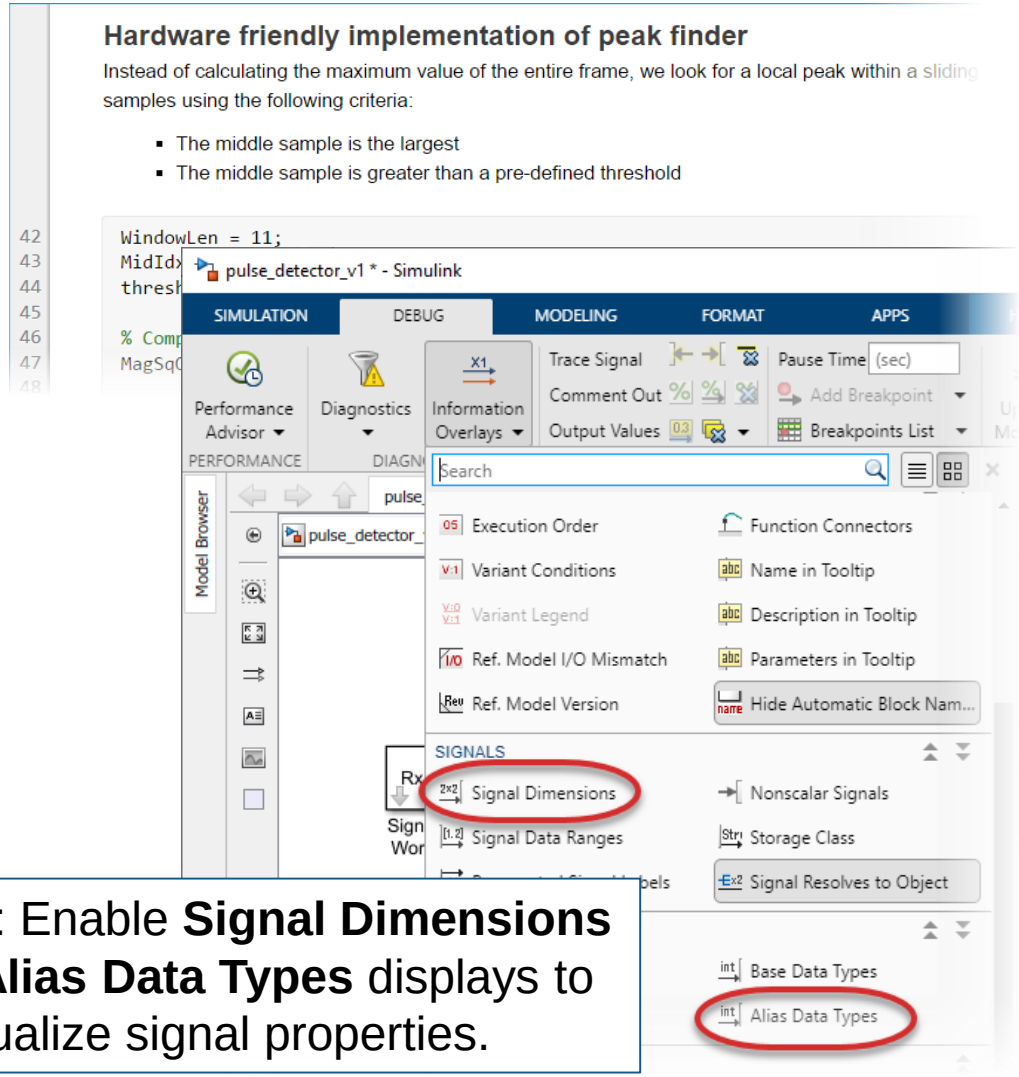
Step 1.2: Hardware-friendly peak finder (magnitude-squared)

1. Review the section *Hardware friendly implementation of peak finder* in **pulse_detector_reference.mlx**.
2. At the filter block output, implement magnitude-squared using the following blocks:



3. Group the blocks in a subsystem and name it Compute Power. Log the subsystem output as mag_sq_out.

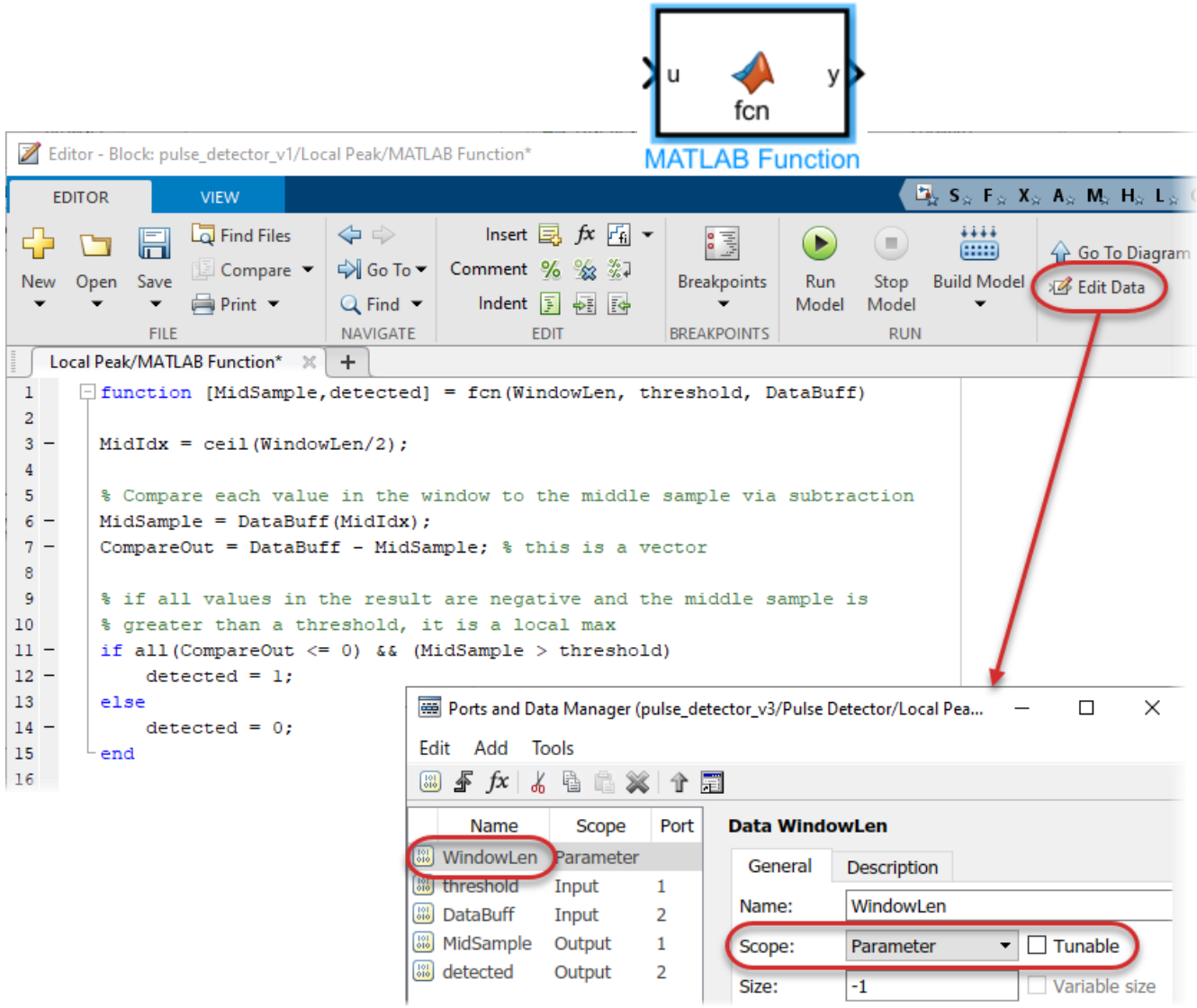




Tip: Enable **Signal Dimensions** & **Alias Data Types** displays to visualize signal properties.

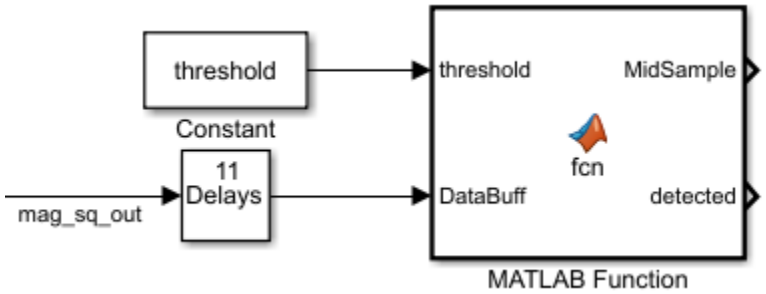
Step 1.3: Hardware-friendly peak finder (local peak)

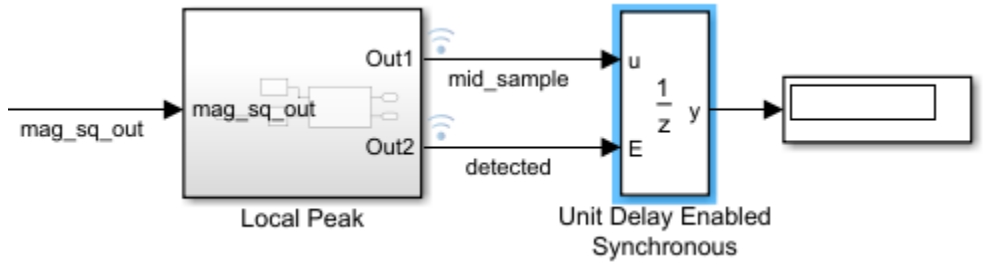
1. Connect mag_sq_out to a **Tapped Delay** block for the sliding window buffer. Set `Number of delays` to WindowLen.
2. Implement the rest of the hardware-friendly peak finder using a **MATLAB Function** block. Copy and paste the code from **pulse_detector_reference.mlx** (line 42 to the end) to the block, and modify it to match the screenshot.
3. Define WindowLen as a `Parameter` (uncheck `Tunable`) in the **Ports and Data Manager**.
4. Close the MATLAB Function block editor.



Step 1.3: Hardware-friendly peak finder (local peak)

5. Add a **Constant** block and enter threshold for `Constant value`.
6. Connect the constant, tapped delay and MATLAB function blocks as shown. Group the 3 blocks in a subsystem named Local Peak, and log the subsystem outputs as mid_sample and detected.
7. Connect a **Unit Delay Enabled Synchronous** block to the subsystem output to “latch” the peak value when detected == 1. Connect the unit delay block to a **Display** block for visualization.
8. Save the model.



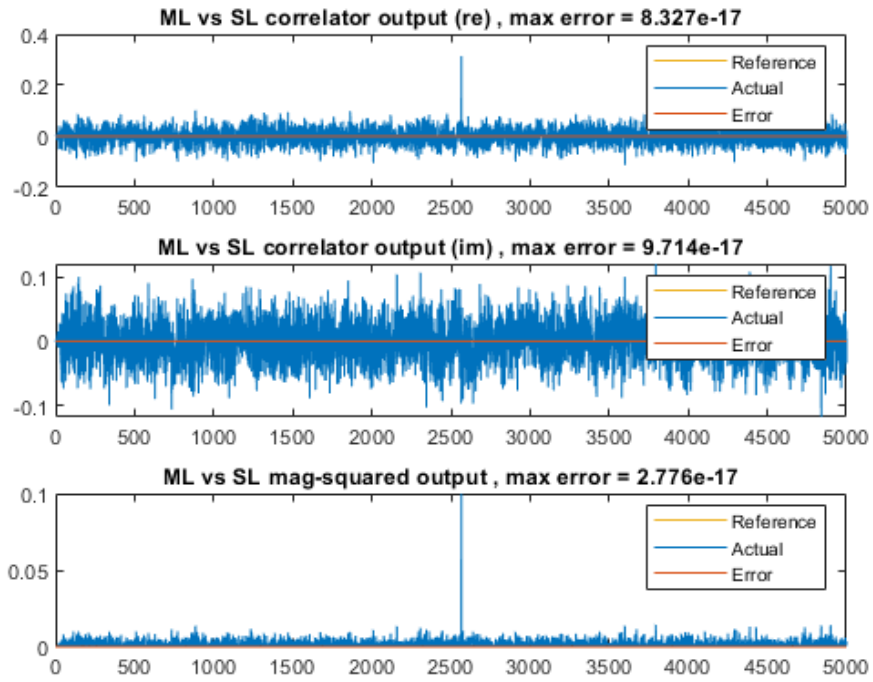


Step 1.4: Compare model to MATLAB reference

Run **pulse_detector_v1_tb.mlx** to simulate the model, and compare the Simulink outputs to the MATLAB reference.

- Maximum error for the correlator, magnitude-squared and peak value should be in the range of floating-point eps (e^{-16}).
- Peak location is randomized for each run.
- The Simulink model implements a *detected* output instead of an index for the peak location, as the algorithm is often used to determine the beginning of a data frame, where a detected signal is sufficient.

```
Maximum error for ML vs SL correlator output (re) out of 5000 values
 8.326673e-17 (absolute), 2.648455e-14 (percentage)
Maximum error for ML vs SL correlator output (im) out of 5000 values
 9.714451e-17 (absolute), 8.099696e-14 (percentage)
Maximum error for ML vs SL mag-squared output out of 5000 values
 2.775558e-17 (absolute), 2.772557e-14 (percentage)
```

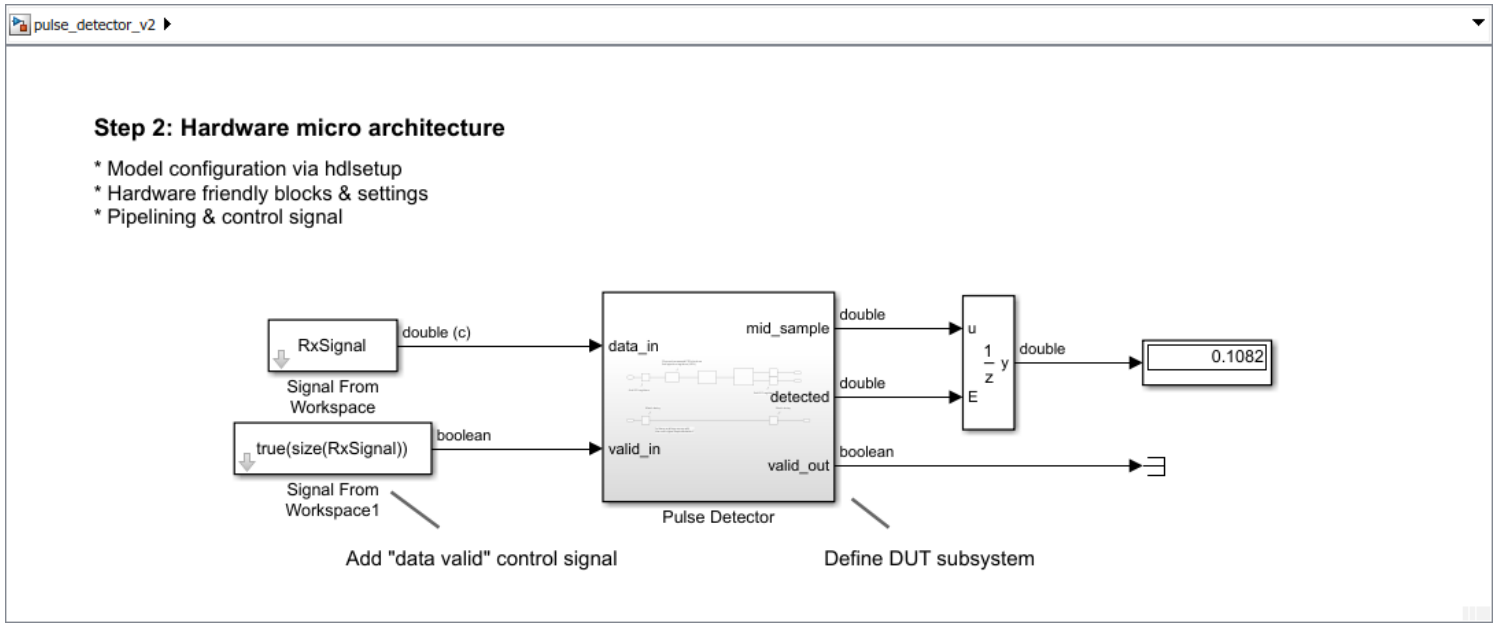



```
Peak location = 2566, magnitude = 3.164e-01 using global max
Peak location = 2566, mag-squared = 1.001e-01 using local max
Peak mag-squared from Simulink = 1.001e-01, error = 2.776e-17
```

Step 2: Hardware micro architecture

In this step, you will:

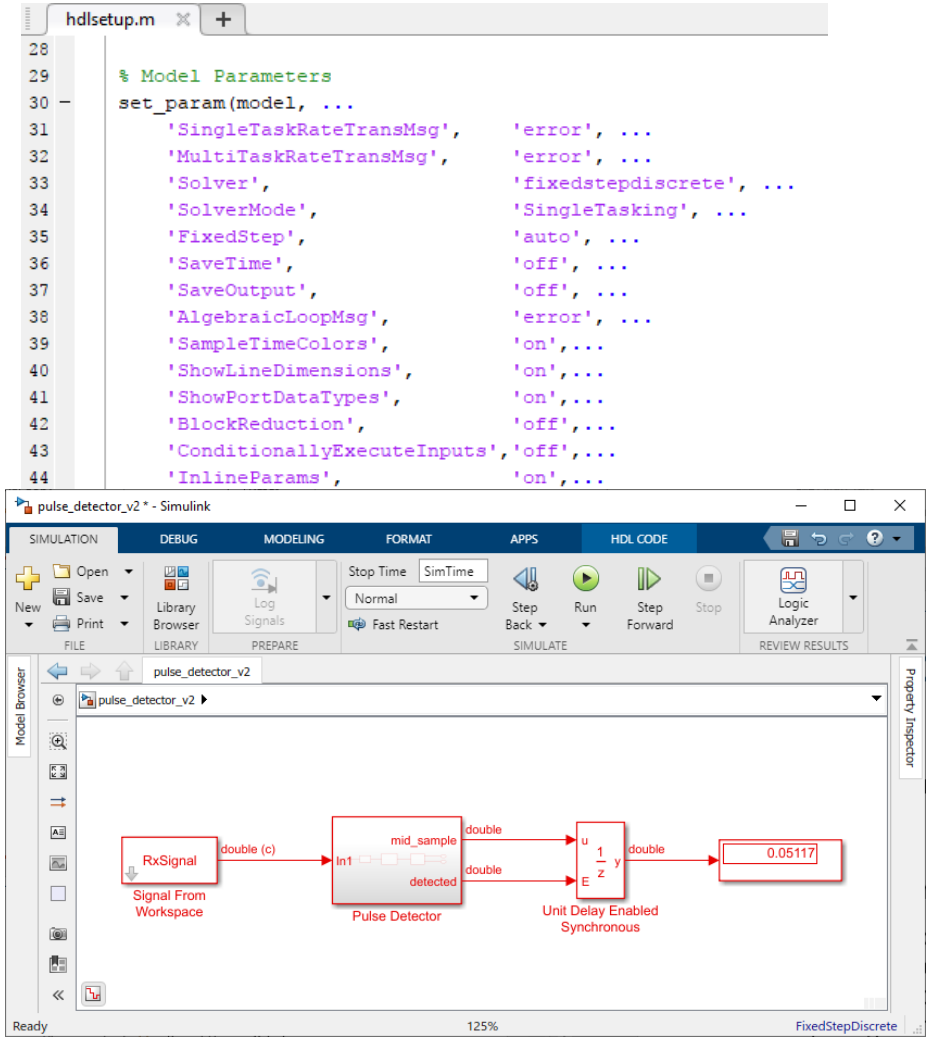
- Prepare the model for HDL code generation
- Add data valid control signal
- Use hardware-efficient blocks and pipeline the data path
- Compare the Simulink architecture model to the MATLAB golden reference



Step 2.1: HDL model configurations

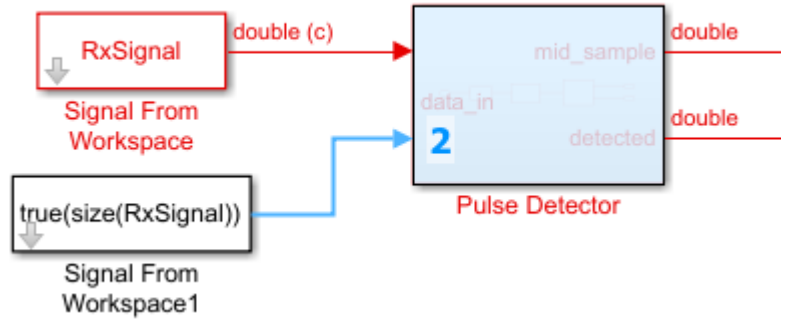
1. Save the Simulink model as pulse_detector_v2.slx.
2. Run the MATLAB command hdlsetup('pulse_detector_v2'). This will configure the model settings to be compatible with HDL code generation.
3. Group the filter block, the *Compute Power* and *Local Peak* subsystems into a top-level subsystem named Pulse Detector. This top-level subsystem will be referred to as the **DUT** (Design Under Test) – i.e. the portion of the model that will generate HDL.

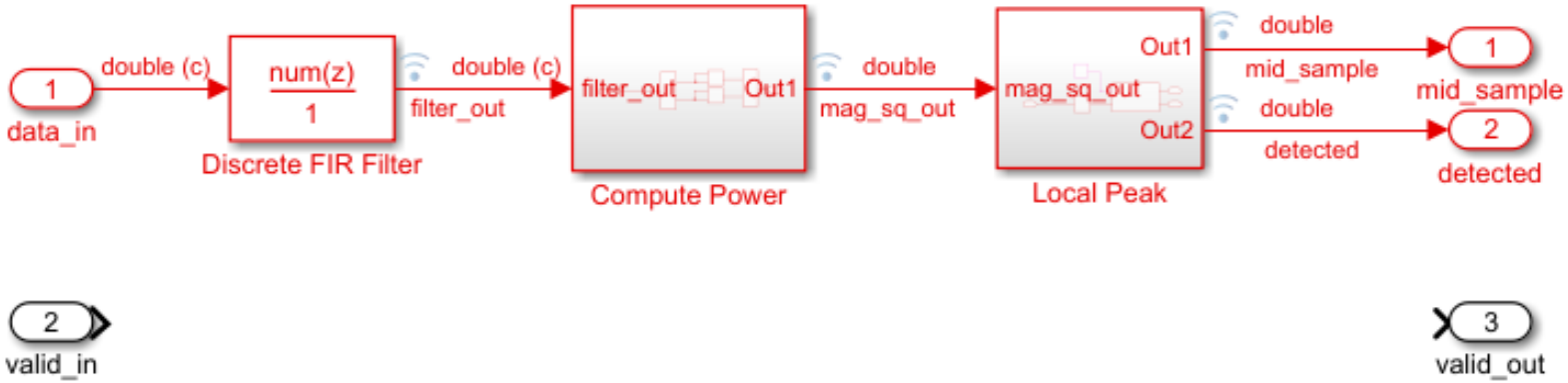
Note: **hdlsetup** enables sample time color display for the model. The next time you update/simulate the model, blocks will appear red as they operate at the fastest sample rate.



Step 2.2: Implement data valid interface

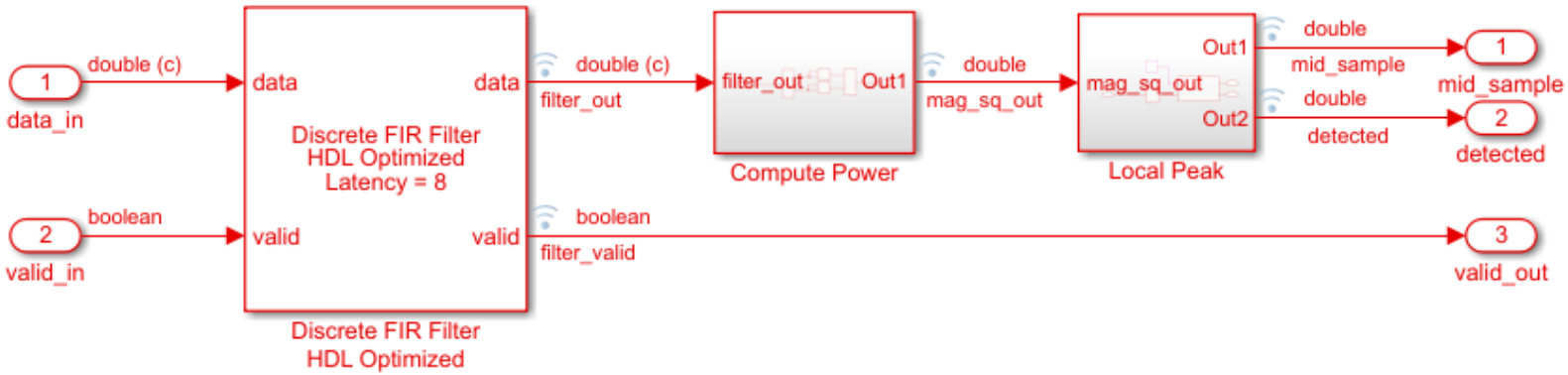
1. Add a data valid input using a **Signal From Workspace** block, and enter true(size(RxSignal)) for `Signal`. Data valid is commonly used in hardware to interface with a non-continuous data source.
2. Connect the block to the DUT subsystem as a second input.
3. Inside the DUT subsystem, name the first input port data_in, and the new input port valid_in.
4. Add a new **Output** port and name it valid_out.





Step 2.3: HDL optimized FIR filter

1. Replace the FIR filter block with a **Discrete FIR Filter HDL Optimized** block. This block offers systolic filter architectures and pipeline register placements that are designed to use FPGA DSP resources efficiently. It also provides control signals for common data interfaces.
2. Again, enter CorrFilter for `Coefficients`, name the data output signal filter_out, and log the signal.
3. Connect the *valid_in* port to the FIR valid input, and the FIR valid output to the *valid_out* port. Log the FIR valid out as filter_valid.



FIR systolic cells

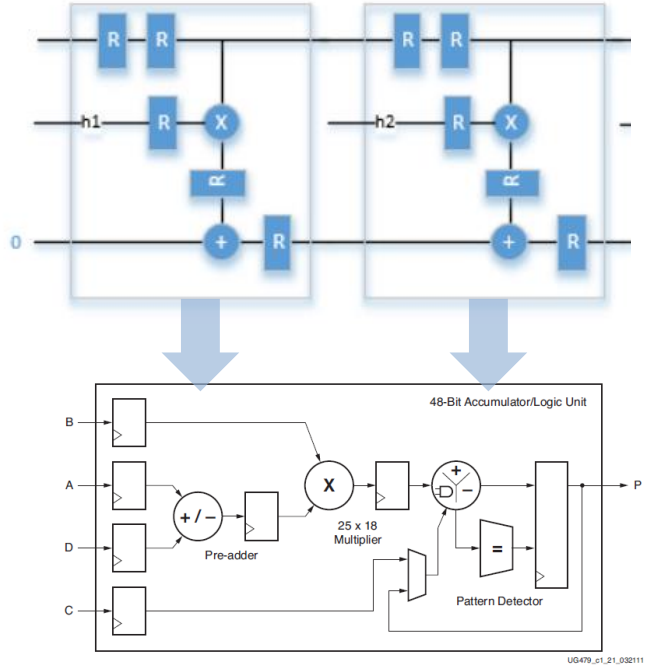
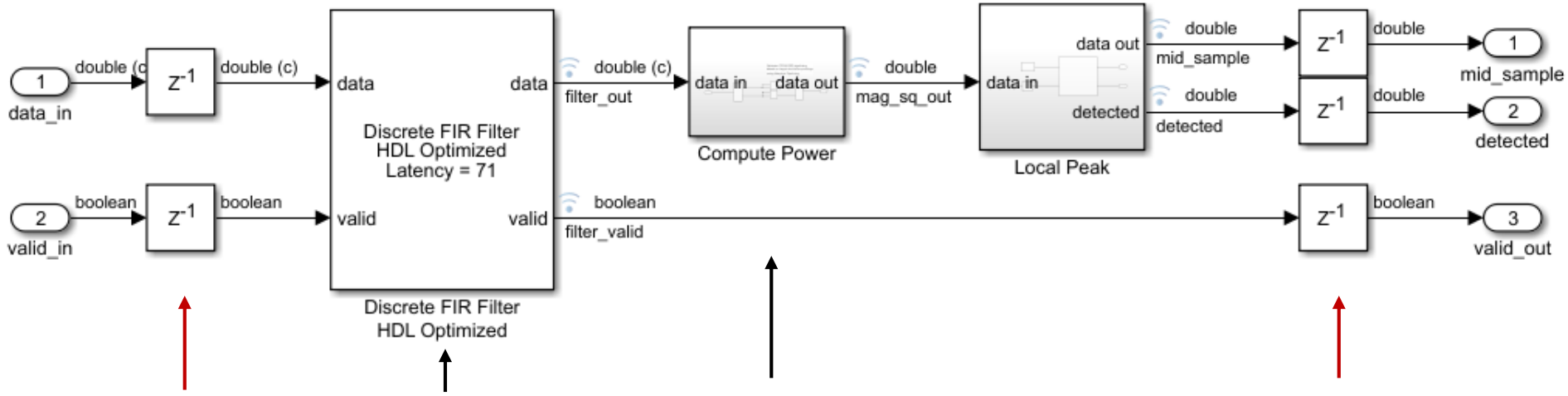


Figure 1-1: **Basic DSP48E1 Slice Functionality**

Step 2.4: Insert pipeline registers

Pipeline the data input & output using **Delay** blocks. Add matching delays to the valid signal.

- The HDL optimized FIR filter already contains pipeline registers. No action is required.
- The *Compute Power* subsystem will gain pipeline registers in the generated HDL via *Adaptive Pipelining.* This feature automatically inserts pipeline registers for certain blocks such as multipliers, based on hardware target device and frequency settings. It is enabled by default, so again no action is required.



Pipelining options:

Explicitly modeled

Provided by library block

Automatically inserted and balanced in HDL

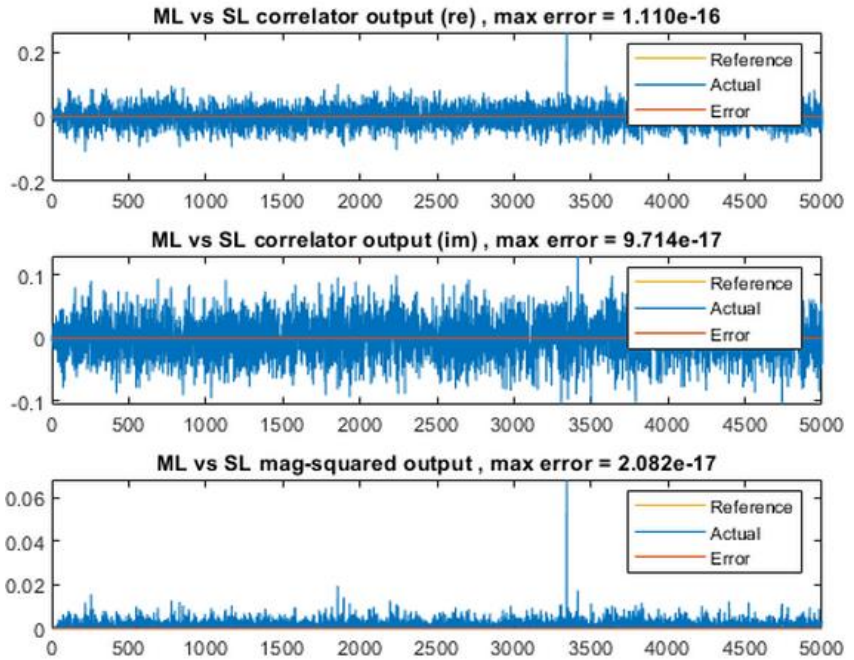
Explicitly modeled

Step 2.5: Compare architecture model to MATLAB reference

1. In the top-level model, terminate the DUT output port *valid_out* by connecting it to a **Terminator** block.
2. Run **pulse_detector_v2_tb.mlx** to simulate the model, and compare the Simulink outputs to the MATLAB reference.
   - This test bench uses the new *filter_valid* signal to qualify the logged filter & magnitude-squared outputs:

```
% Correlation filter output
FilterOutSL = getLogged(slout,'filter_out');
FilterValid = getLogged(slout,'filter_valid');
FilterOutSL = FilterOutSL(FilterValid);
```

   - Maximum error for all outputs should be similar to that of step 1.



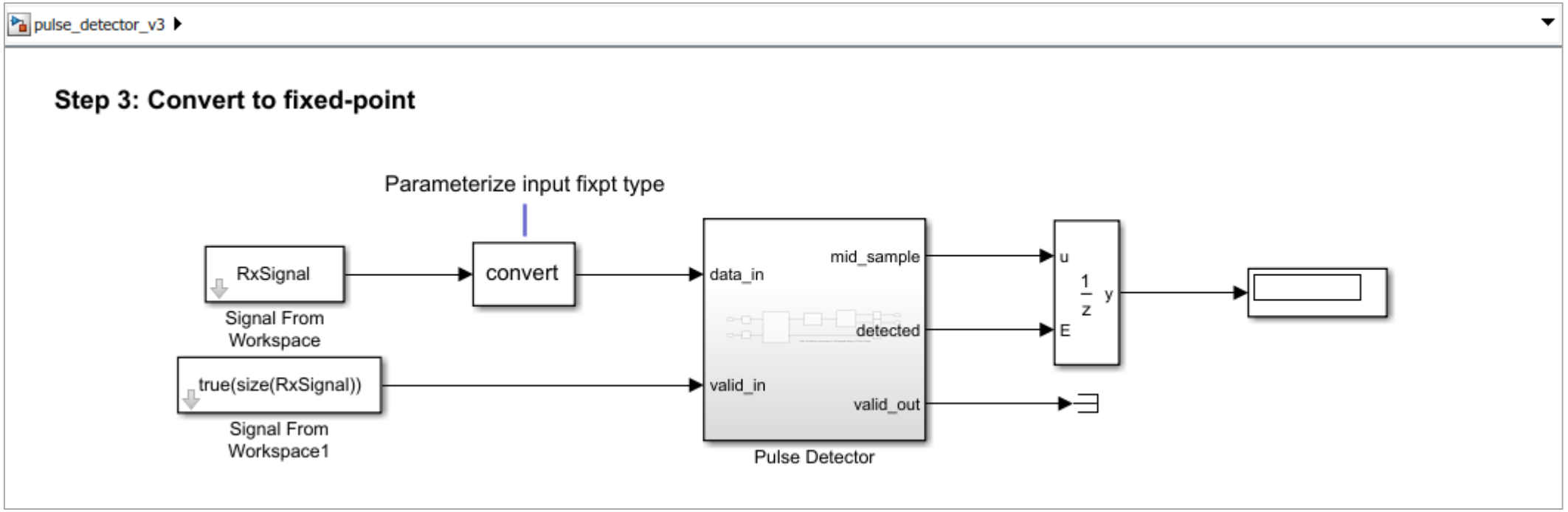
Question: What's wrong with the valid signal implementation?

Answer: The valid signal should also be connected to the sliding window buffer, to prevent invalid samples from entering the delay line. The simulation is only correct in this example because *valid_in* is always high.

Step 3: Fixed-point conversion

In this step, you will:

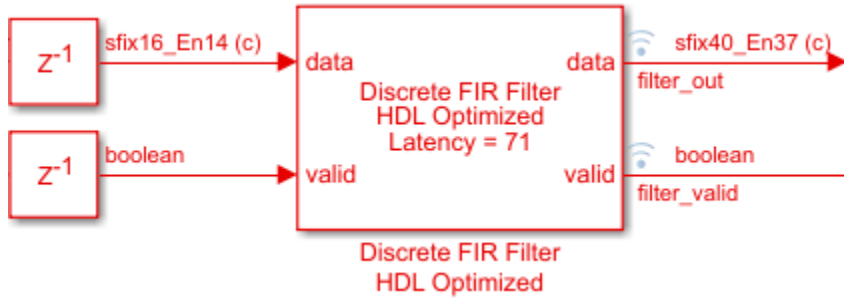
- Convert the model to fixed-point
- Compare the Simulink fixed-point model to the MATLAB golden reference

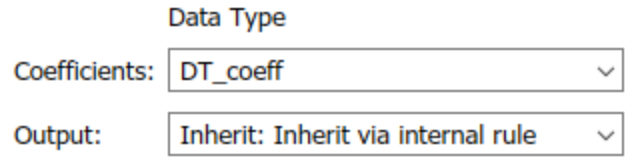


Step 3.1: Define input and filter fixed-point data types

1. Save the Simulink model as pulse_detector_v3.slx.
2. At the top-level model, add a **Data Type Conversion** block at the output of *RxSignal*. Set `Output data type` to DT_input, which is defined as `fixdt(1,16,14)` in the test bench script to fully represent -1 to 1.
3. Inside the DUT, set `Coefficients Data Type` on the filter block to DT_coeff, which is defined as `fixdt(1,18)`. Fraction length of a constant can be automatically determined when it is left unspecified.
4. The filter is set up to perform multiply & add using full-precision fixed-point. Run **pulse_detector_v3_tb.mlx** to update the filter output data type (error is expected due to temporary downstream issues).

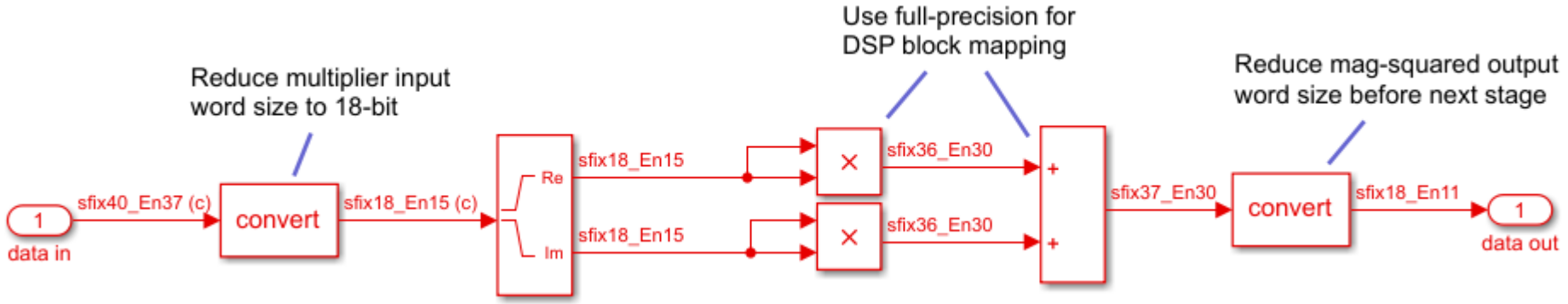
```
>> fixdt(1,16,14)

ans =

  NumericType with properties:

     DataTypeMode: 'Fixed-point: binary point scaling'
       Signedness: 'Signed'
       WordLength: 16
   FractionLength: 14
```





Step 3.2: Define magnitude-squared data types

1. In the *Compute Power* subsystem, add a **Data Type Conversion** block at the input and set it to DT_filter. This reduces the data word length to 18-bit while maintaining the integer range.
2. Update the model to examine the data types through the subsystem (error is still okay).
3. Reduce the final adder output to DT_power using another **Data Type Conversion** block, and update the model once more.



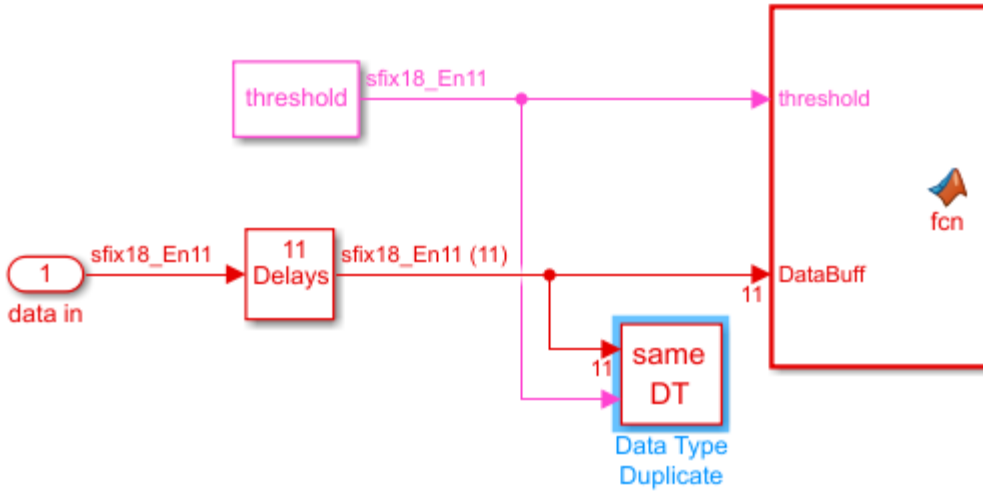
Tip: 18-bit inputs and full-precision fixed-point allow multiply-add operations (including those within the FIR filter in the previous step) to be implemented using DSP blocks in most FPGAs.

Step 3.3: Define peak picker data types

1. In the *Local Peak* subsystem, add a **Data Type Duplicate** block. Connect one port to the threshold constant block output, and another to the tapped delay block output.
2. Set `Output data type` of the threshold constant to **Inherit via back propagation**.

   These 2 steps allow the threshold constant to take on the same data type used by the input data.

3. In the MATLAB function block, change the *detected* value to true/false.
4. Update the model. It should be error-free at this point.




```
22 -    if all(CompareOut <= 0) && (MidSample > threshold)
23 -        detected = true;
24      else
25 -        detected = false;
26      end
```

```
Warning: Parameter precision loss occurred for 'Value' of
'pulse_detector_v3/Pulse Detector/Local Peak/Constant'. The parameter's value
cannot be represented exactly using the run-time data type. A small quantization
error has occurred. To disable this warning or error, in the Configuration
Parameters > Diagnostics > Data Validity pane, set the 'Detect precision loss'
option in the Parameters group to 'none'.
Suggested Actions:
    • - Suppress
```

This warning message can be safely ignored or suppressed.

Step 3.4: Compare fixed-point model to MATLAB reference

Run **pulse_detector_v3_tb.mlx** to simulate the model, and compare the Simulink fixed-point outputs to the MATLAB floating-point reference.

- Note the increase in error due to quantization.
- You may switch between fixed-point and floating-point using the following flag in the test bench.

```
% Simulate model in fixed-point or floating-point
fxpt_mode = ☑;
if fxpt_mode  % fixed-point
    DT_input = fixdt(1,16,14);
    DT_filter = fixdt(1,18,15);
    DT_power = fixdt(1,18,11);
else  % floating-point
    DT_input = 'double';
    DT_filter = 'double';
    DT_power = 'double';
end
DT_coeff = fixdt(1,18); % coeff is treated as double
```

```
Maximum error for ML vs SL correlator output (re) out of 5000 values
 8.713091e-06 (absolute), 4.352322e-03 (percentage)
Maximum error for ML vs SL correlator output (im) out of 5000 values
 8.120594e-06 (absolute), 8.318296e-03 (percentage)
Maximum error for ML vs SL mag-squared output out of 5000 values
 4.907380e-04 (absolute), 1.224467e+00 (percentage)
```

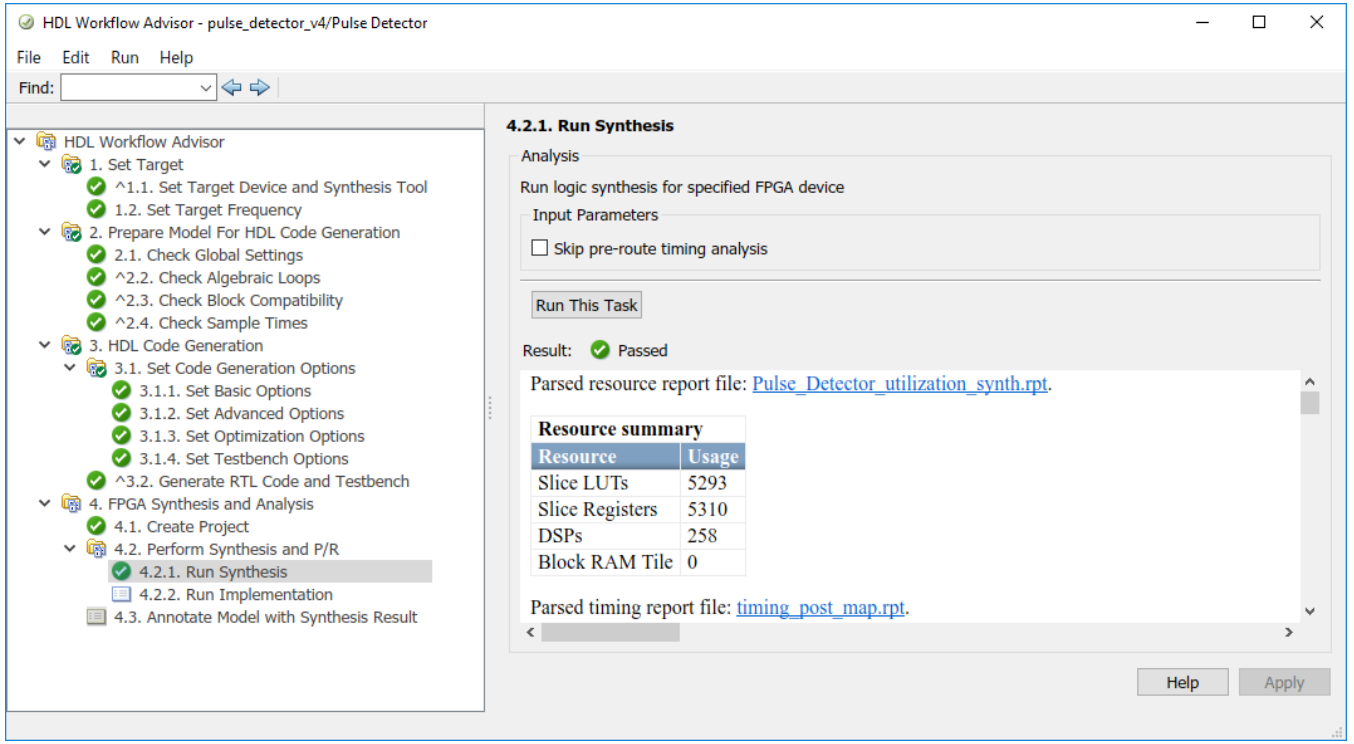
```
Peak location = 1268, magnitude = 2.002e-01 using global max
Peak location = 1268, mag-squared = 4.008e-02 using local max
Peak mag-squared from Simulink = 4.004e-02, error = 3.862e-05
```

Tip: Doing hardware architecture design before fixed-point conversion prevents quantization noise from obscuring design errors.

Step 4: HDL code generation & synthesis

In this step, you will:

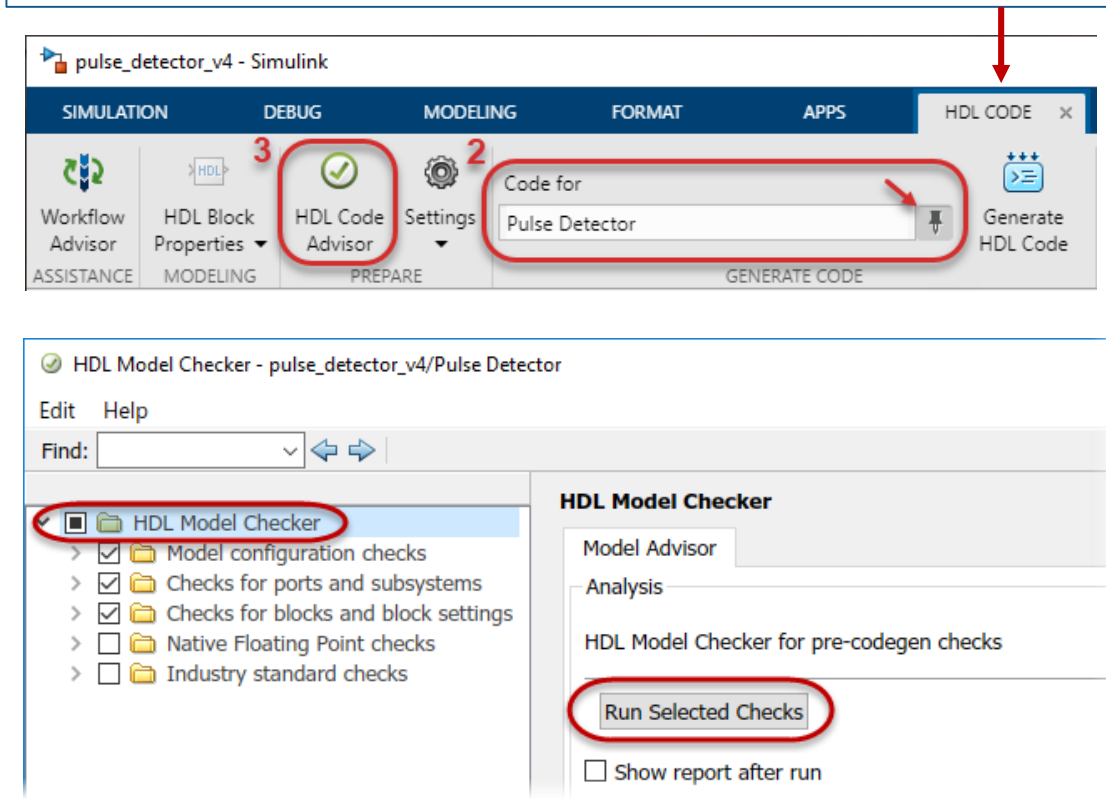
- Check the model for HDL compatibility
- Generate HDL code and reports
- Synthesize the generated design using Xilinx Vivado



Step 4.1: Check model for HDL compatibility

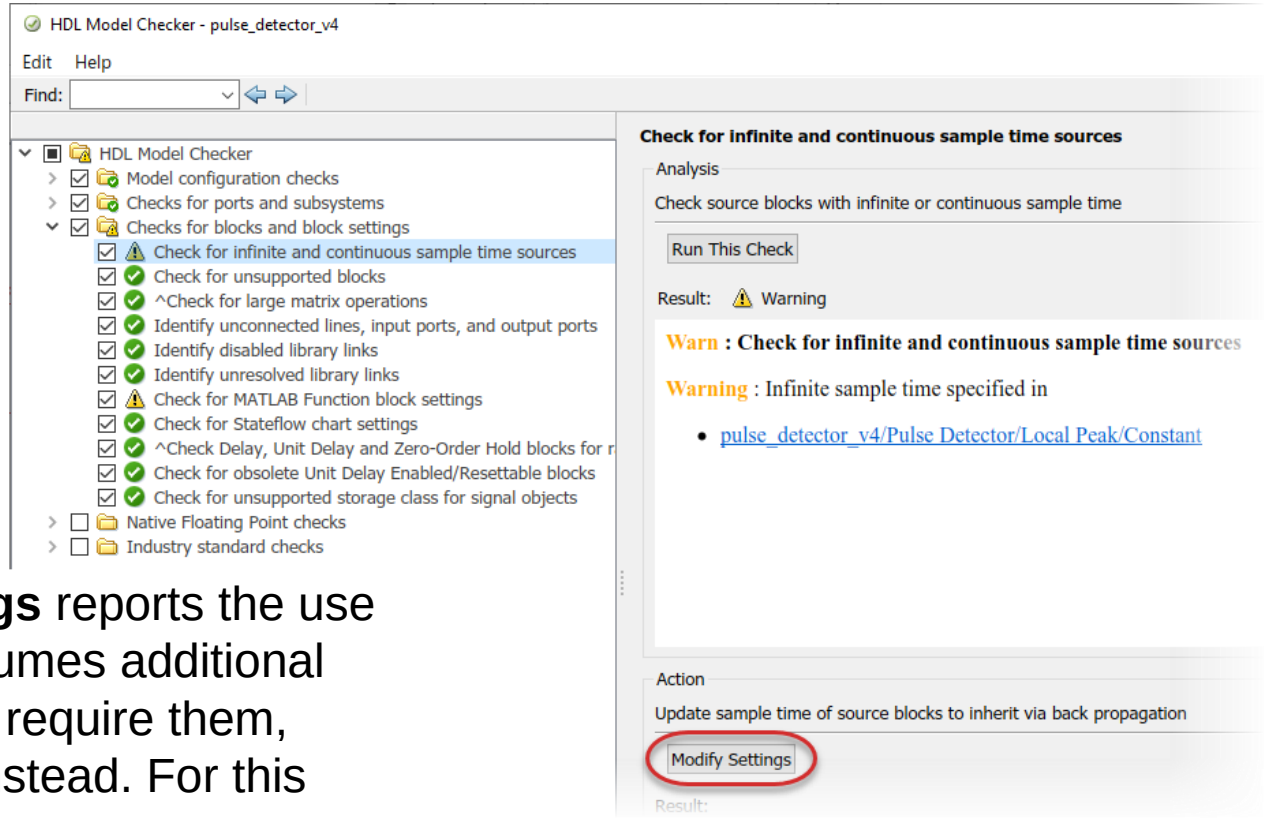
1. Save the Simulink model as pulse_detector_v4.slx.
2. Set the top-level subsystem *Pulse Detector* as DUT (HDL Code Toolstrip > unpin > click on subsystem > pin).
3. Check HDL Compatibility for the DUT subsystem using **HDL Code Advisor**. Besides incompatibility, the tool also checks for settings that may result in inefficient hardware. In many cases, a shortcut is provided to modify those settings.
4. De-select the last 2 groups of checks – they are not applicable for this example. Highlight the top folder named **HDL Model Checker**, then click **Run Selected Checks**.

Tip: If the **HDL Code** tab is not visible, you can open it under Apps > Code Generation > HDL Coder.



Step 4.1: Check model for HDL compatibility

5. HDL Model Checker returns 2 warnings. **Check for infinite and continuous sample time sources** reports the use of *inf* sample time by the threshold constant block. Replacing it with back propagation (-1) will allow optimization features such as delay balancing to function properly. Click **Modify Settings** to apply the fix automatically.



6. **Check for MATLAB Function block settings** reports the use of saturation and rounding logic, which consumes additional hardware resources. If your design does not require them, click Modify Settings to use floor and wrap instead. For this exercise, you may ignore the warning.

7. Close the HDL Model Checker.

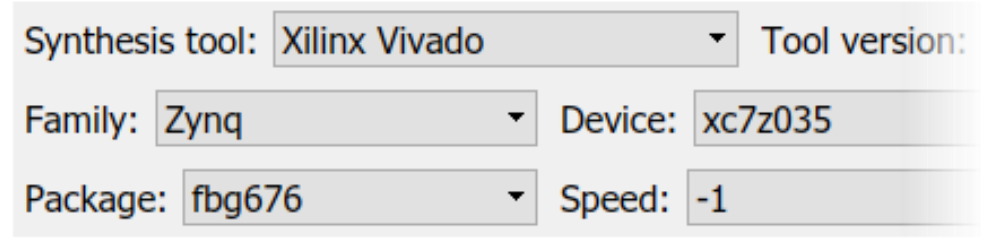
Step 4.2: Generate HDL code and reports

1. In MATLAB, run the following command to add Xilinx Vivado 2018.3 to the system path:

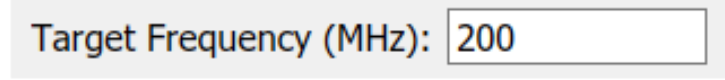
```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2018.3\bin');
```

2. Launch **HDL Workflow Advisor** for the DUT subsystem (HDL Code Toolstrip > Workflow Advisor).
3. Configure the properties as shown on the right, clicking **Apply** at each step to save the changes.
4. Run through steps 1 to 3 (right-click on step 3.2 > **Run to Selected Task**).

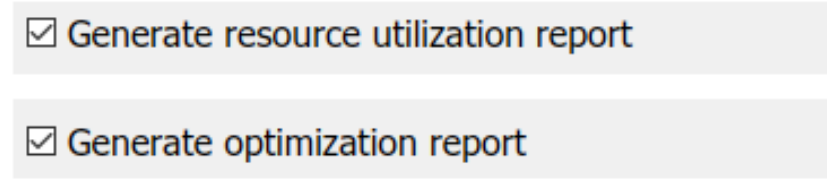
1.1:



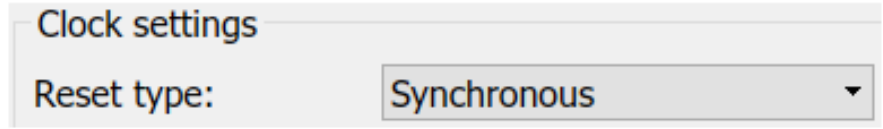
1.2:



3.1.2:



3.1.3:

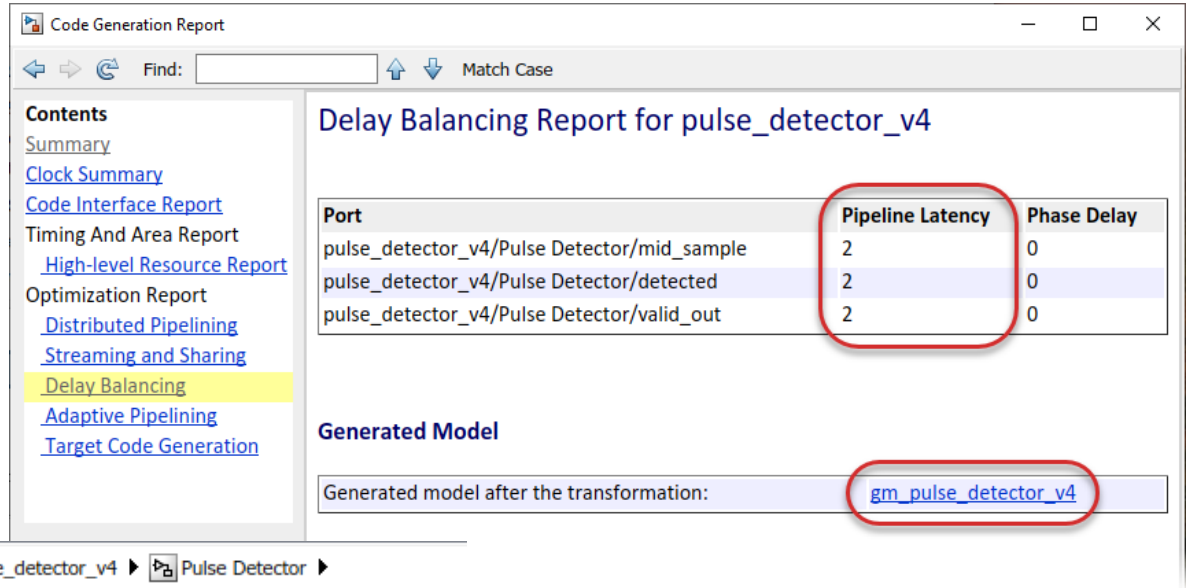


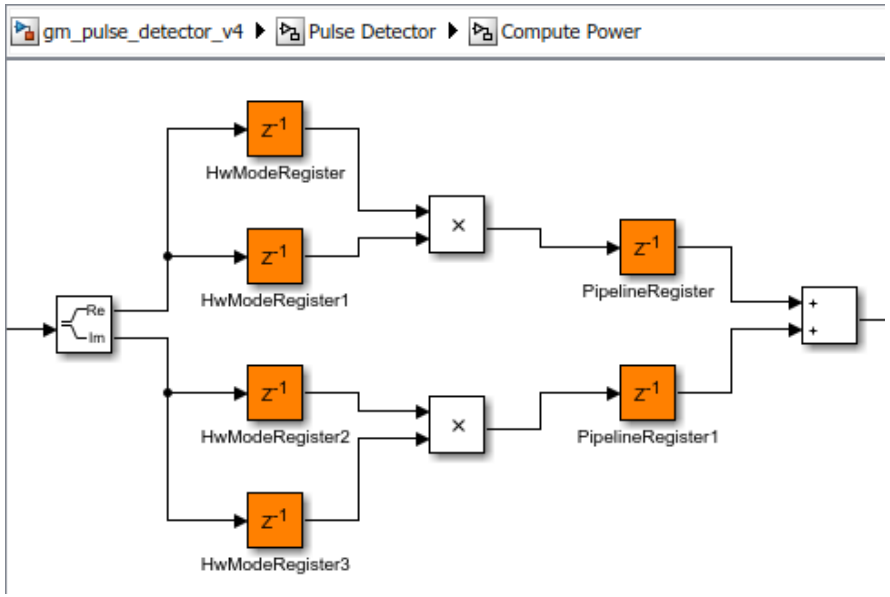
Tip: Choose the appropriate global reset type for your target FPGA device to ensure multipliers and multiply-add operators are mapped to DSP blocks.

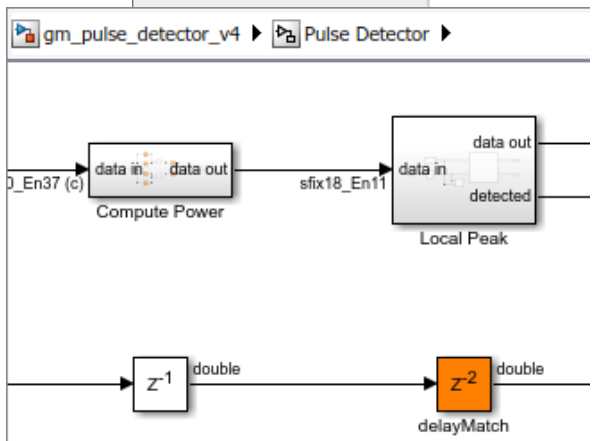
Step 4.2: Generate HDL code and reports

5. Review the results of *Adaptive Pipelining* and *Delay Balancing* in the generated optimization report. Open the **generated model** using the provided link to visualize the added latency.

6. Review resource usage in the high-level resource report.



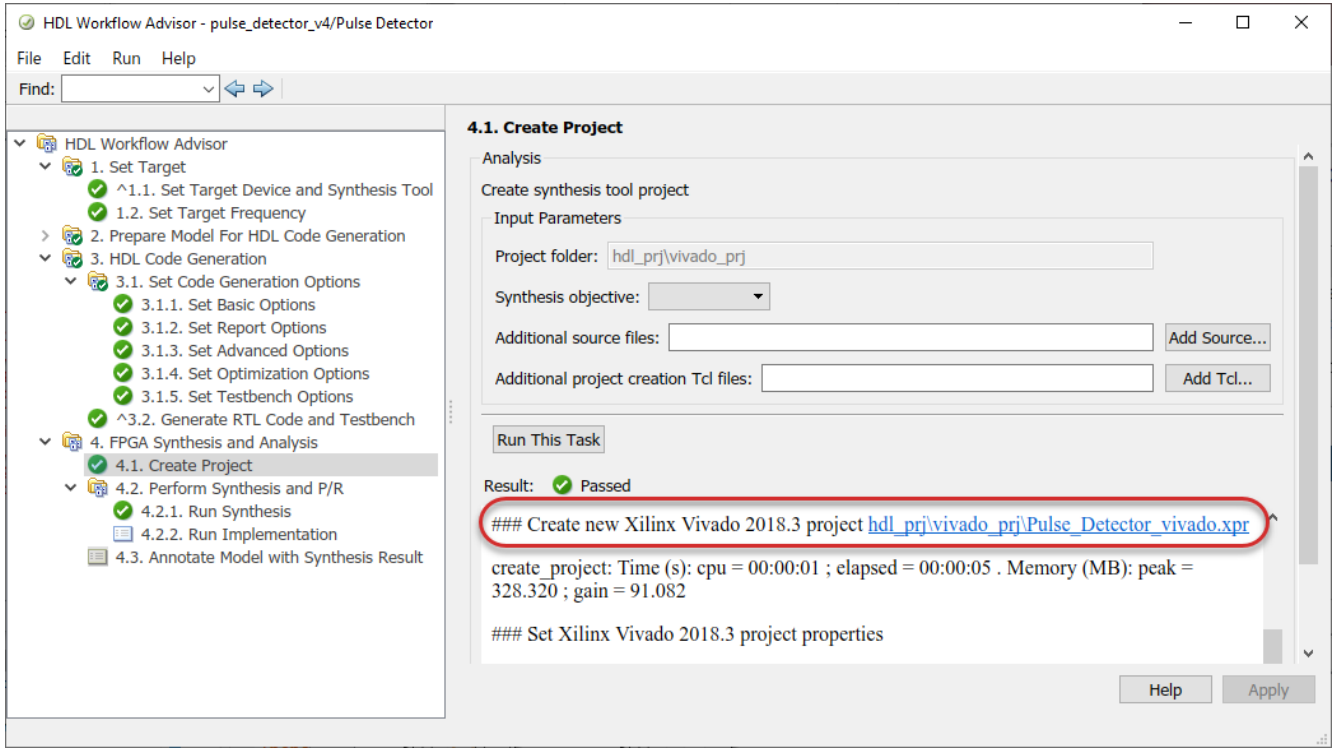


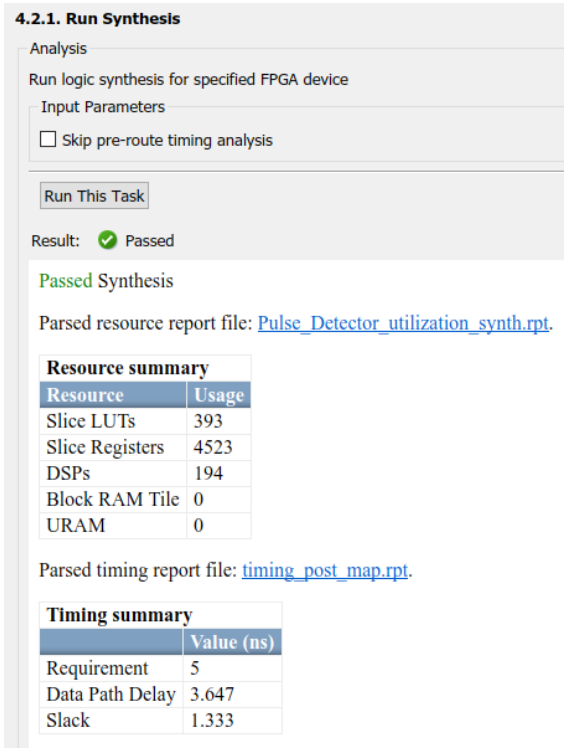


Note: The generated model is bit-true & cycle-accurate to the generated HDL. It is used to verify the generated code.

Step 4.3: Synthesis the generated design

1. Run step 4.1 to create a Xilinx Vivado project. You may open the project using the provided link and continue synthesis in Vivado, especially if you expect it to take a long time.
2. Run step 4.2.1 to synthesize the design without launching Xilinx Vivado. MATLAB is blocked while synthesis is run in the background.





Summary

This completes the tutorial. Compare your work to the solution models in the folder **/pulse_detector/solution**.

For more information:

- Visit our FPGA and SoC design website
- Explore MathWorks Training on HDL code generation and FPGA signal processing
- Contact us on File Exchange for additional questions