# Pulse Detector HDL Workflow Tutorial
# R2025a - R2025b

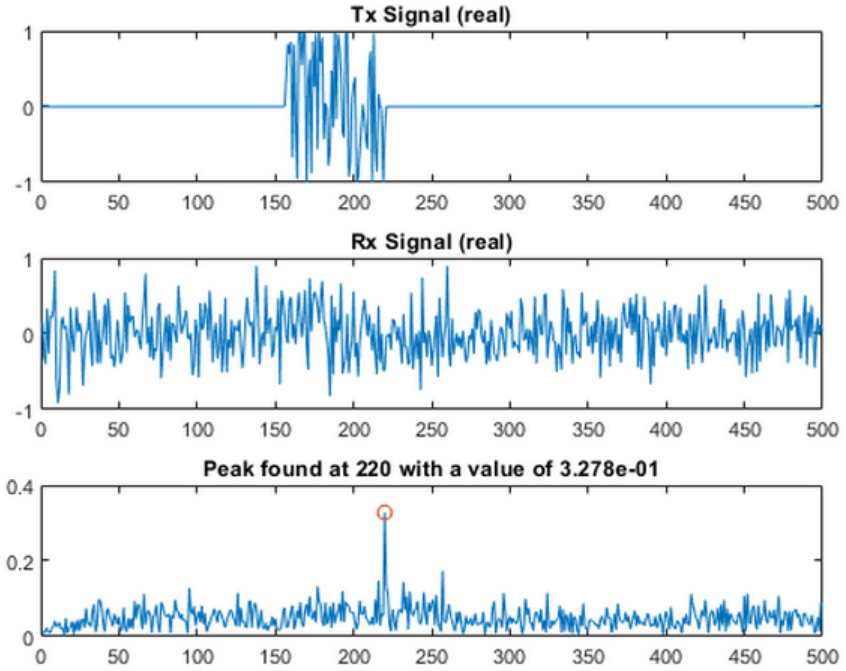
MathWorks Application Engineering

# 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);
```

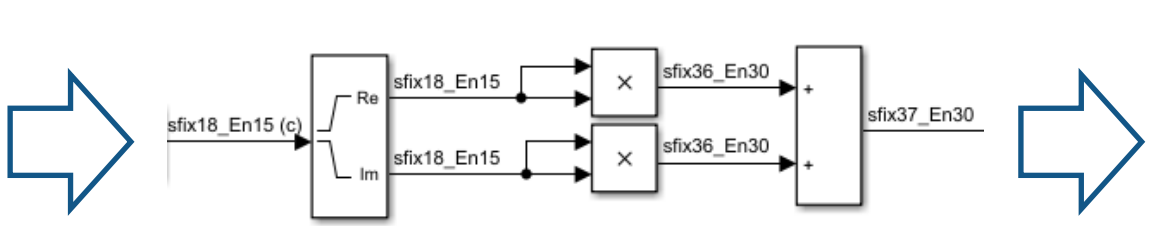


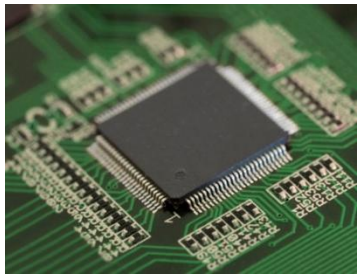
# Example Overview

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 (R2025a or R2025b)
  - Simulink
  - Fixed-Point Designer™
  - MATLAB Coder™
  - HDL Coder™
  - Signal Processing Toolbox™
  - DSP System Toolbox™
  - DSP HDL Toolbox™
- AMD Vivado™ 2024.1 is used 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

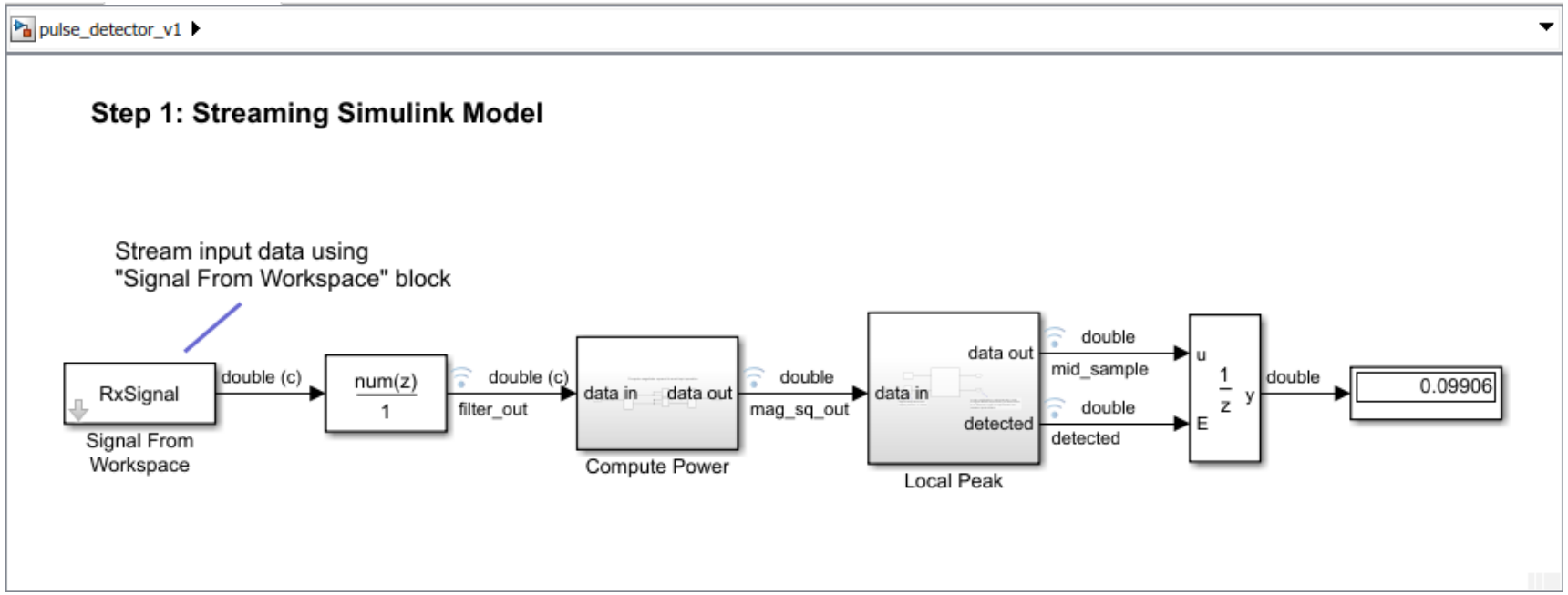
- Use the following resources as needed to prepare for this tutorial:
  - MATLAB Onramp
  - Simulink Onramp
  - HDL Coder Tutorial Overview (video series)
  - HDL Coder Evaluation Reference Guide

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

# Step 1: Streaming Simulink model

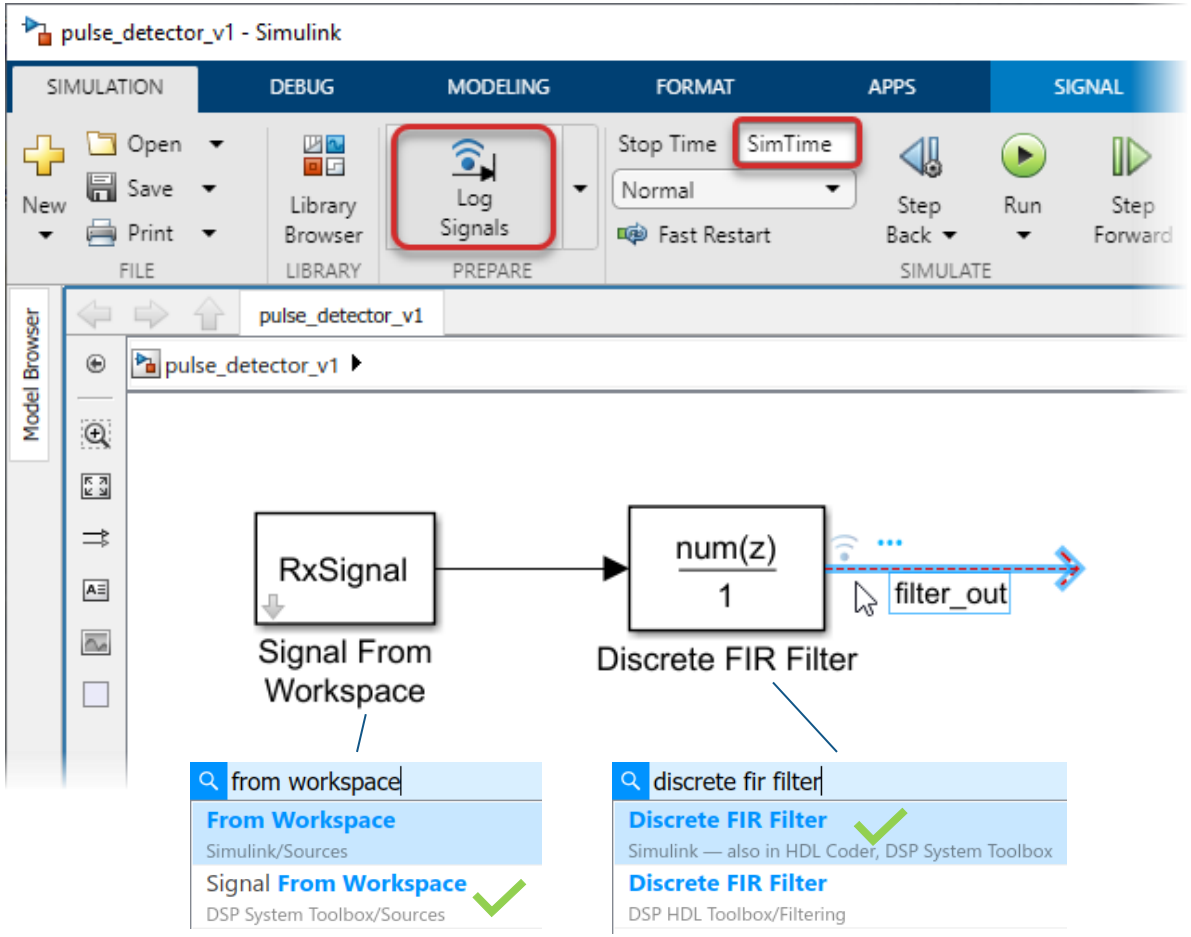
In this step, you will:

- 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 from the **Simulink** library. 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.




Tip: Make sure to choose the right blocks

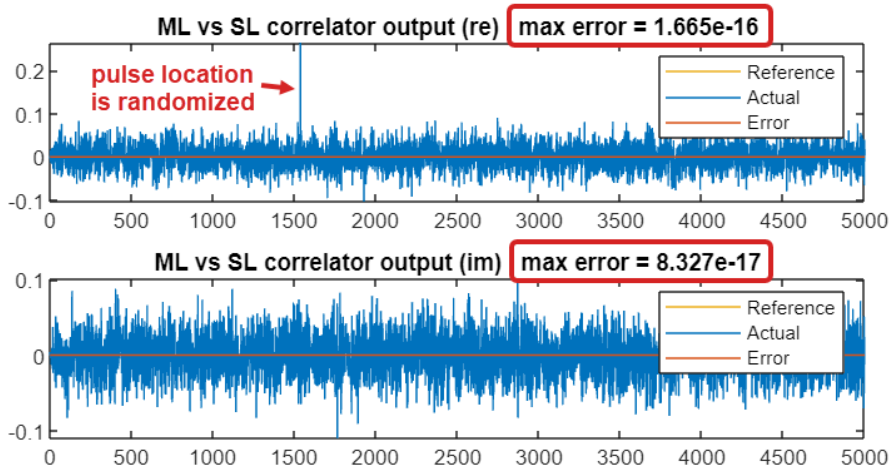
# Step 1.1: Stream and filter input signal

7. Compare the Simulink filter output against the MATLAB reference. Verifying your Simulink model incrementally helps catch mistakes early on.
   - Run the test bench script **pulse_detector_v1_tb.mlx**. An error is expected at line 16 as the model is incomplete.
   - Verify the Simulink filter output matches the MATLAB reference (max error ~ $1e^{-16}$ ).

**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 output
13  compareData(imag(FilterOut),imag(FilterOutSL),{2 3 2},'ML vs SL correlator output
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  % Peak value
20  MidSampleSL = getLogged(slout,'mid_sample');
```

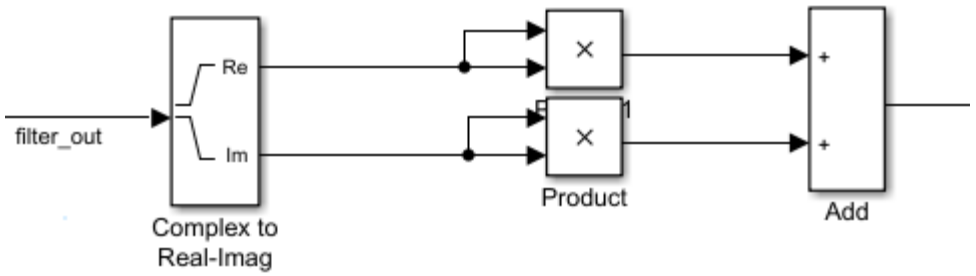
```
Maximum error for ML vs SL correlator output (re) out of 5000 values
 1.665335e-16 (absolute), 6.301458e-14 (percentage)
Maximum error for ML vs SL correlator output (im) out of 5000 values
 8.326673e-17 (absolute), 7.535410e-14 (percentage)
```



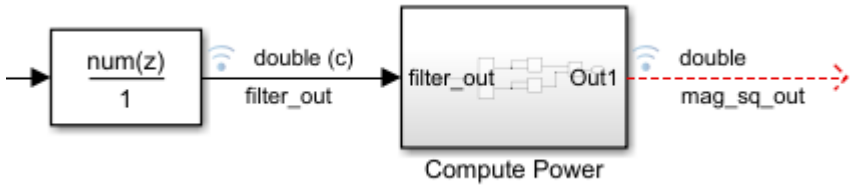

```
Warning: Did not find any Dataset element using 'mag_sq_out'.

Error using getLogged
Signal 'mag_sq_out' not found. Make sure it is logged and named correctly.
```

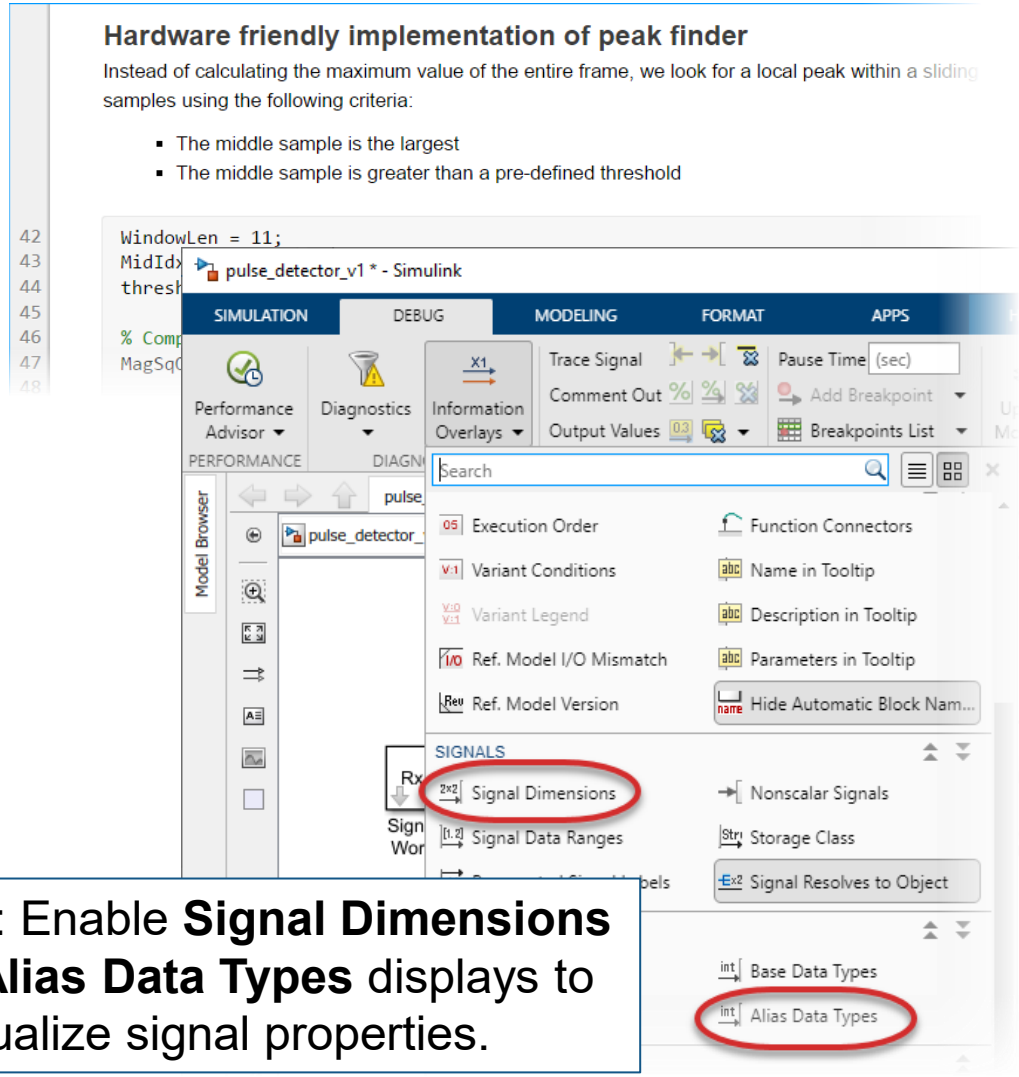
# Step 1.2: Hardware-friendly peak finder (magnitude-squared)

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.

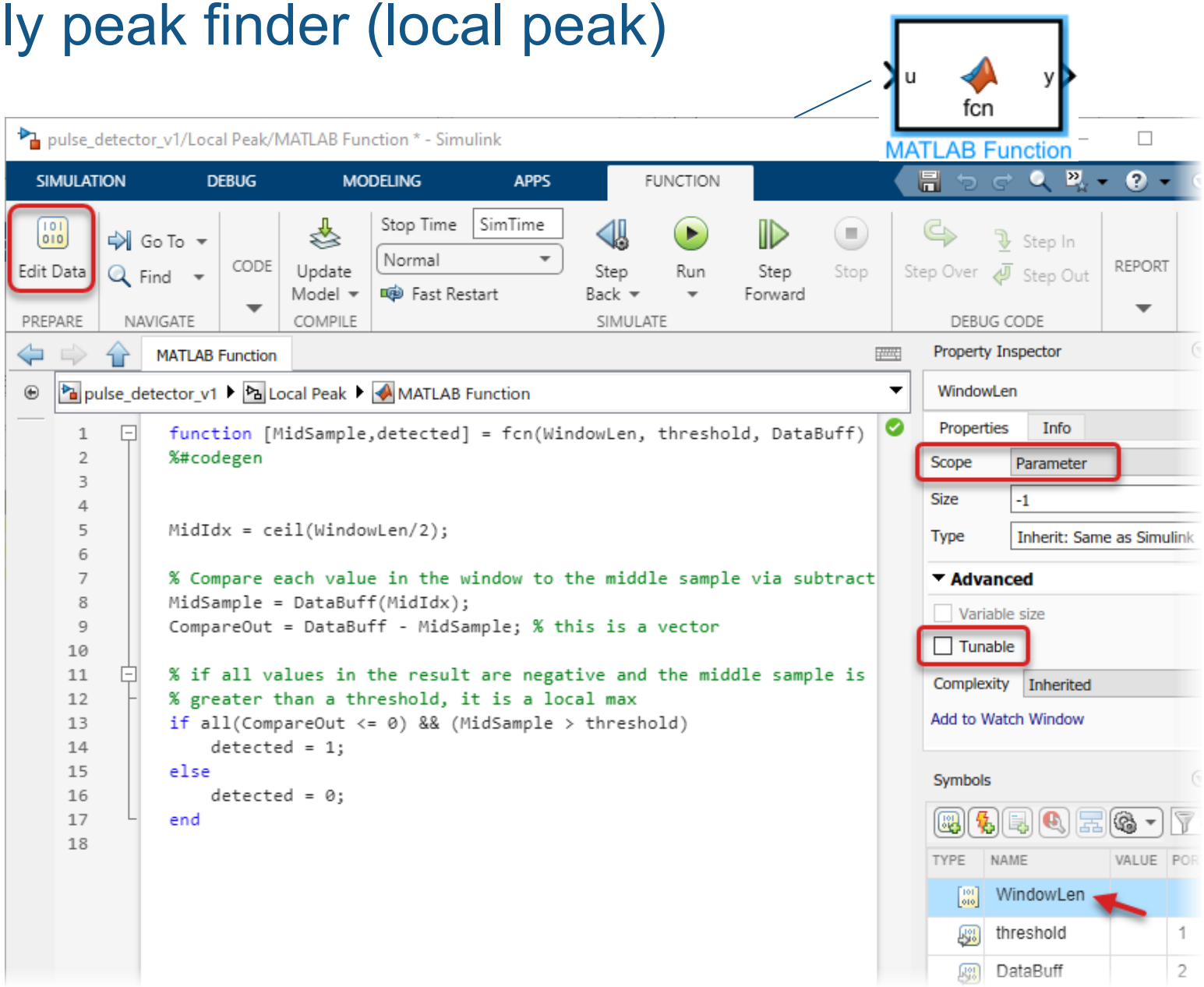



Tip: Enable **Signal Dimensions** & **Alias Data Types** displays to visualize signal properties.

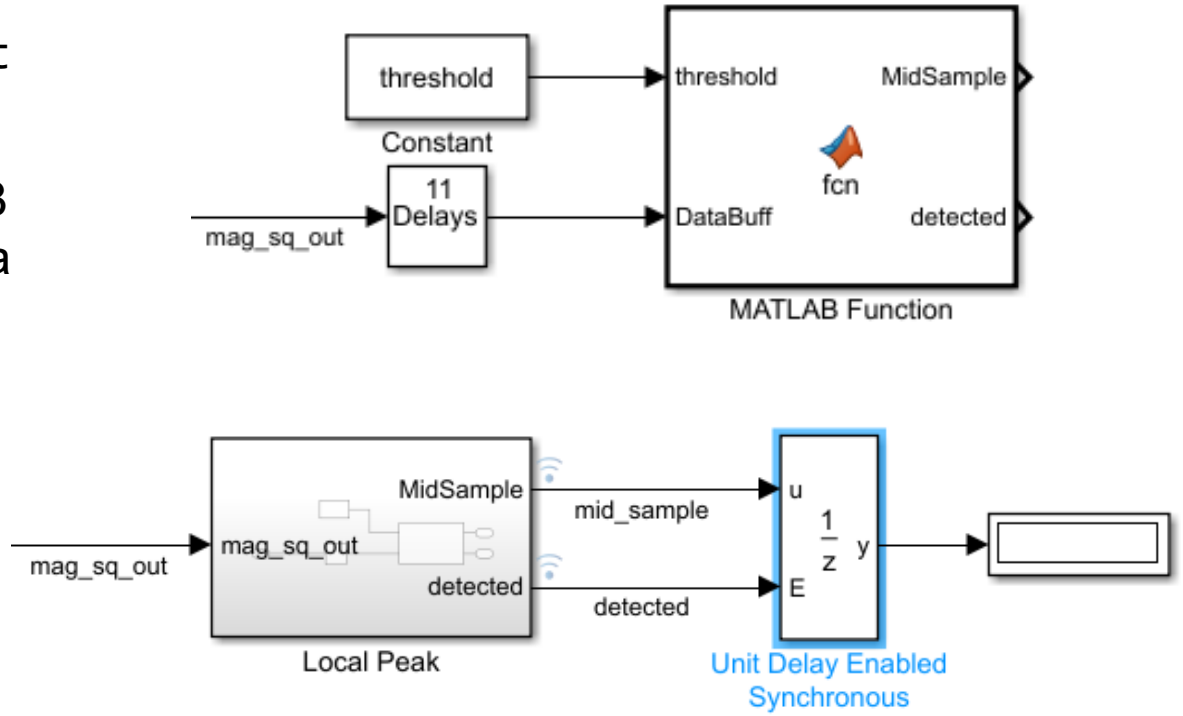
# Step 1.3: Hardware-friendly peak finder (local peak)

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. Click on **Edit Data** to open the **Symbols** pane and the **Property Inspector**. Select WindowLen, change its `Scope` to `Parameter` and uncheck `Tunable`.



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

4. Navigate back to the top-level model. Add a **Constant** block and enter threshold for `Constant value`.
5. Connect the constant, tapped delay and MATLAB function blocks as shown. Group the 3 blocks in a subsystem named Local Peak.
6. Connect Local Peak to a **Unit Delay Enabled Synchronous** (UDES) block; log and name the connecting signals mid_sample and detected as shown.
7. Finally, connect the UDES block to a **Display** block. The UDES block will keep the detected peak value on display at the end of the simulation.
8. Save the model.

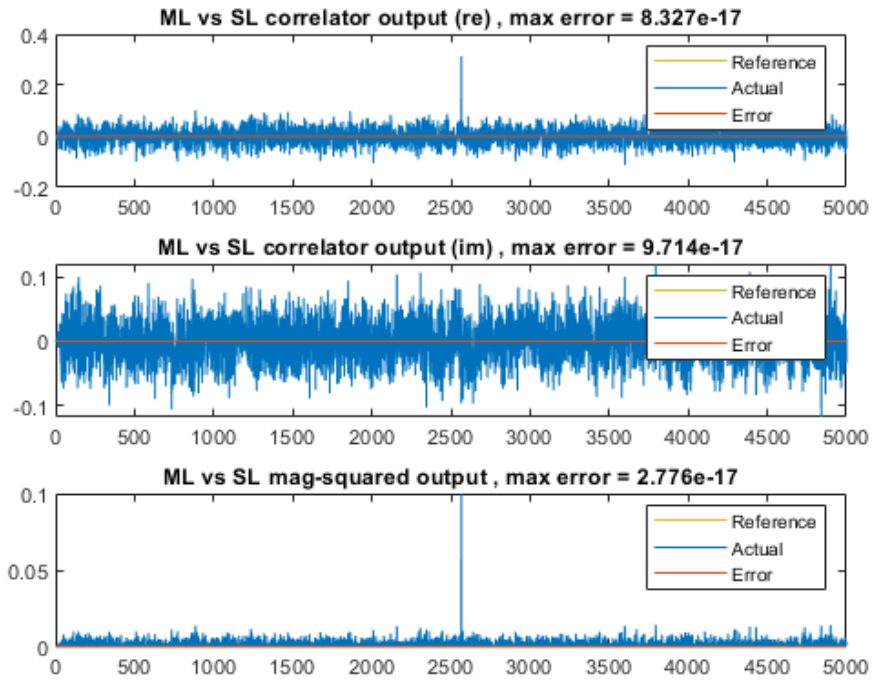


# Step 1.4: Compare model to MATLAB reference

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)
```

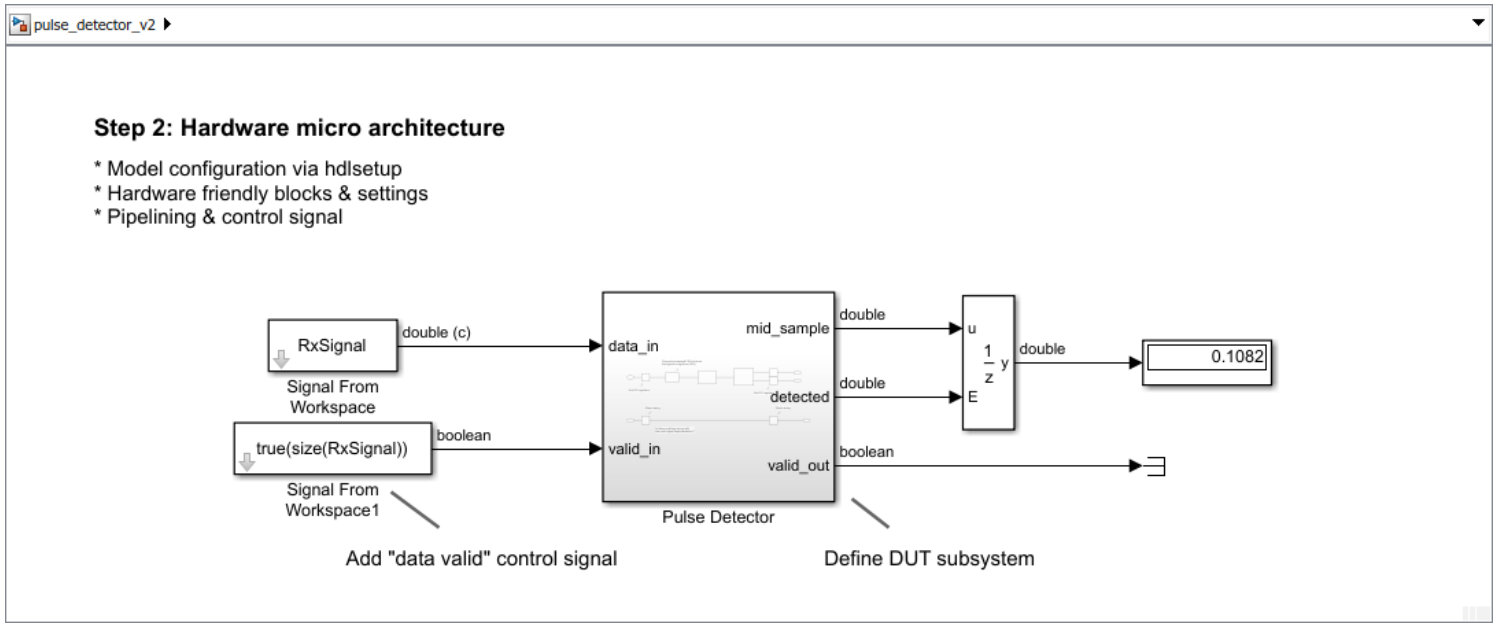



```
Peak location = 2566, magnitude = 3.164e-01 using global max
Peak location = 2566, mag-squared = 1.001e-01 using local max
Peak mag-squared from Simulink = 1.001e-01, error = 2.776e-17
```

# Step 2: Hardware micro architecture

In this step, you will:

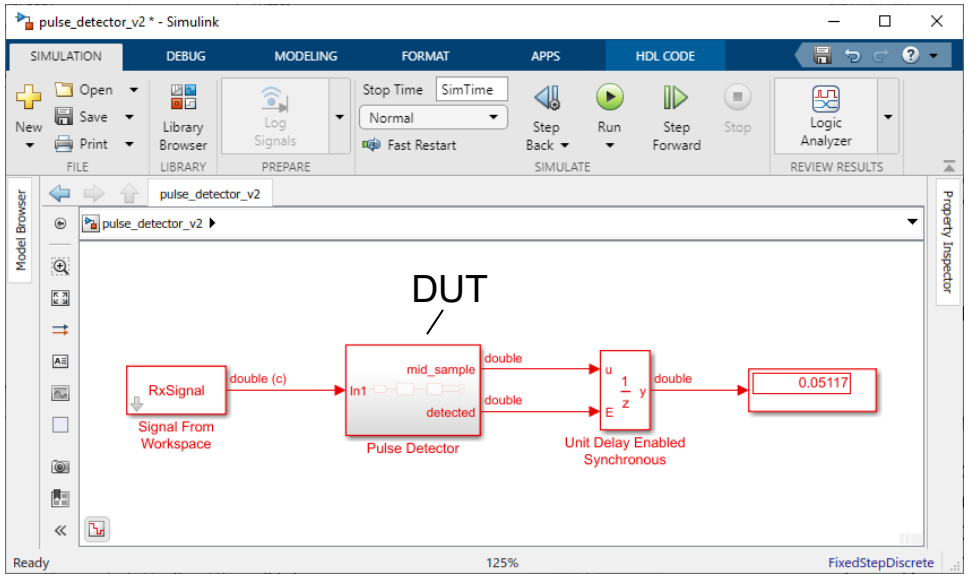
- Prepare the model for HDL code generation
- Add data valid control signal
- Use hardware-efficient blocks and pipeline the data path
- Compare the Simulink architecture model to the MATLAB golden reference



# 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.

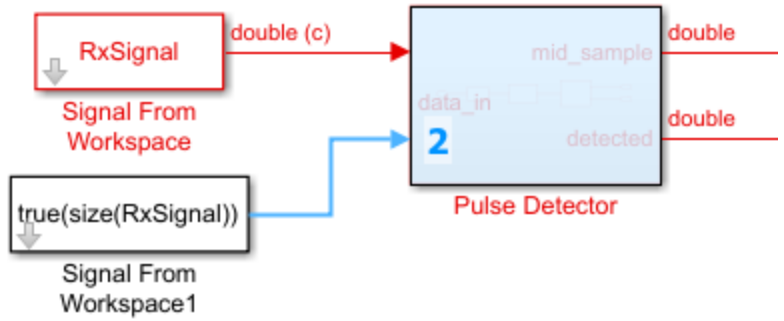
```
Command Window
>> hdlsetup('pulse_detector_v2')
### AlgebraicLoopMsg value is set from 'warning' to 'error' (revert).
### BlockReduction value is set from 'on' to 'off' (revert).
### ConditionallyExecuteInputs value is set from 'on' to 'off' (revert).
### DefaultParameterBehavior value is set from 'Tunable' to 'Inlined' (revert).
### InheritOutputTypeSmallerThanSingle value is set from 'off' to 'on' (revert).
### ProdHWDeviceType value is set from 'Intel->x86-64 (Windows64)' to 'ASIC/FPGA-
### SingleTaskRateTransMsg value is set from 'none' to 'error' (revert).
### Solver value is set from 'VariableStepAuto' to 'FixedStepDiscrete' (revert).
### The listed configuration parameter values are modified as a part of hdlsetup.
```

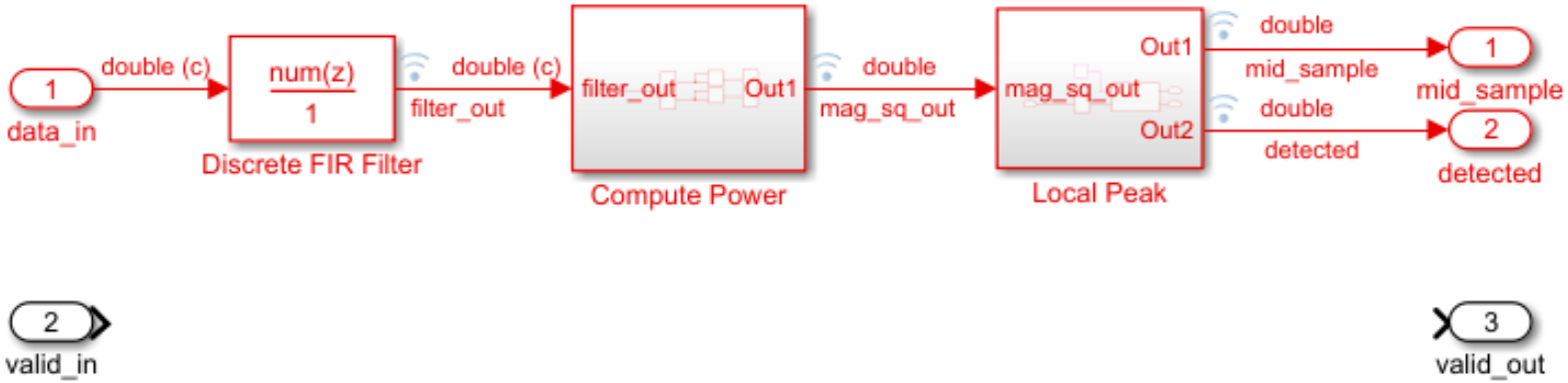



Note: **hdlsetup** enables sample time color display for the model. The next time you update/simulate the model, blocks will appear red as they operate at the fastest sample rate.

# Step 2.2: Implement data valid interface

1. Add a data valid input using a **Signal From Workspace** block; enter true(size(RxSignal)) for the `Signal` parameter. This enables the design to interface with a non-continuous data source, although the example uses continuous input data for simplicity.
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 current FIR filter with a **Discrete FIR Filter** block from the **DSP HDL Toolbox** library. 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

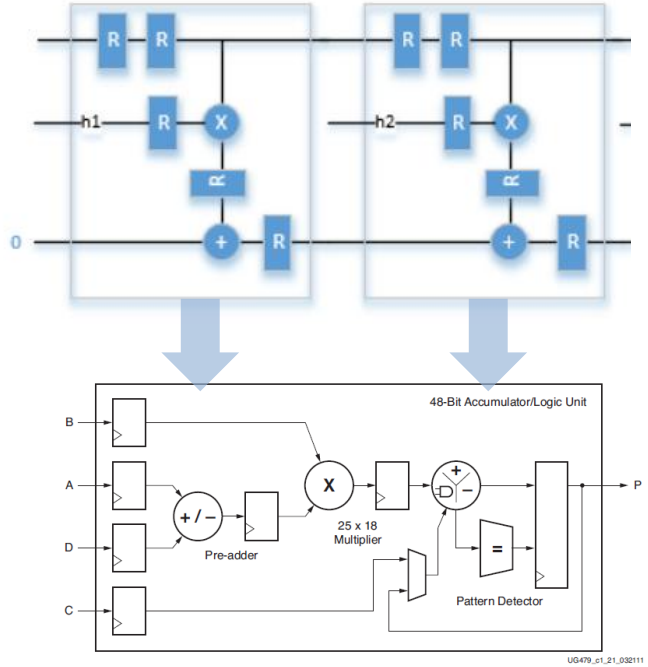
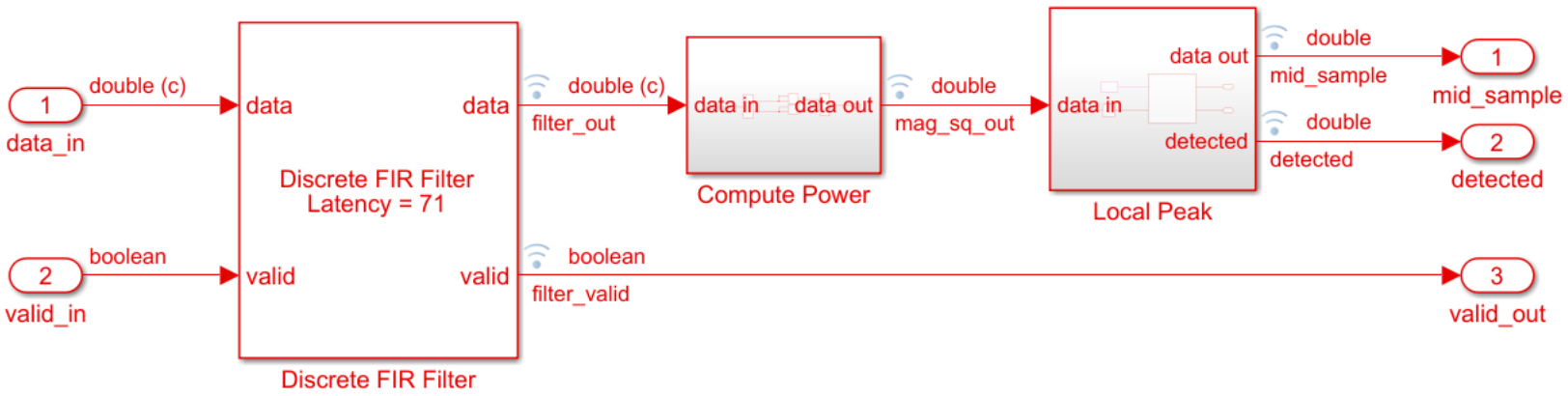


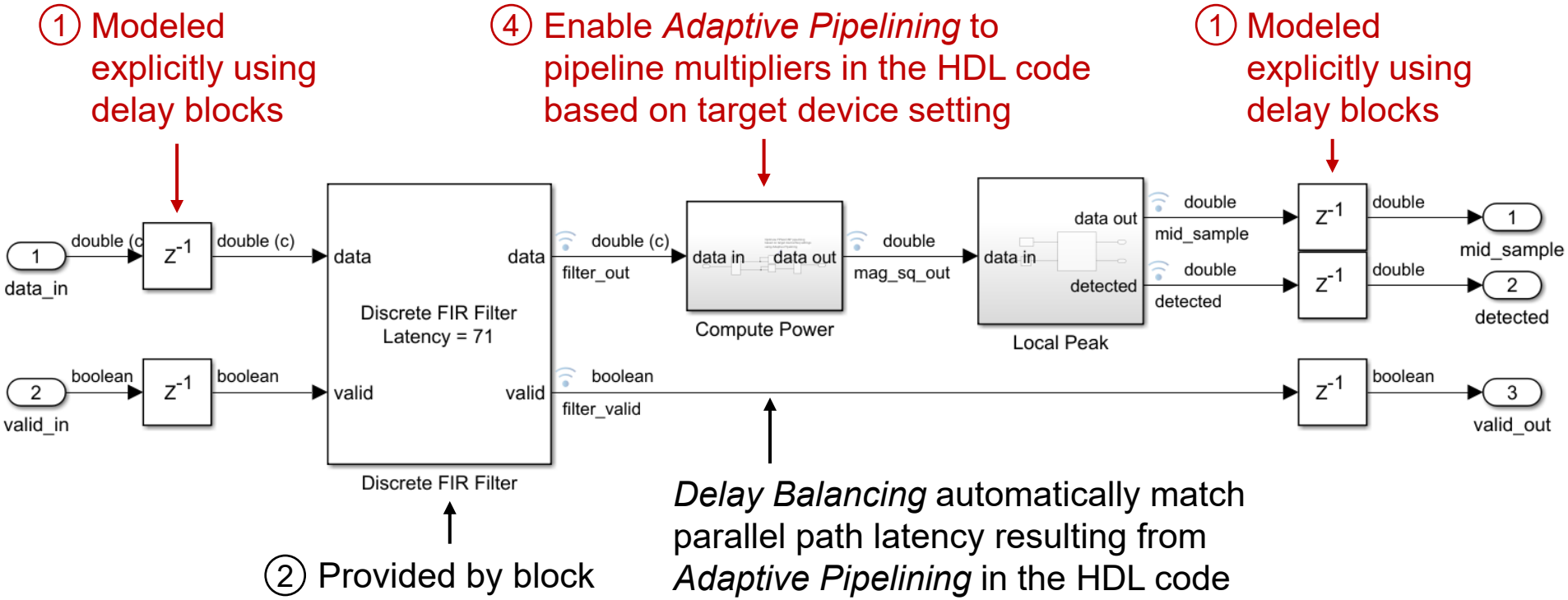

*Figure 1-1:* **Basic DSP48E1 Slice Functionality**



# Step 2.4: Insert pipeline registers

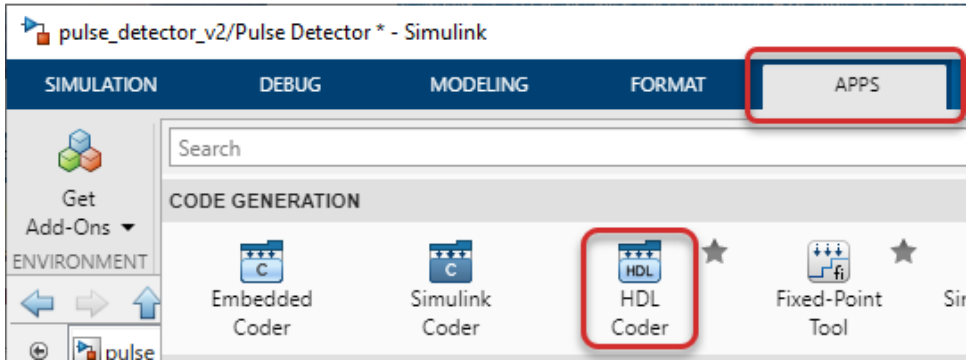
1. Pipeline the data input & output using **Delay** blocks as shown. Add matching delays to the valid signal path.
2. The FIR filter from DSP HDL Toolbox already contains pipeline registers. No action is required.

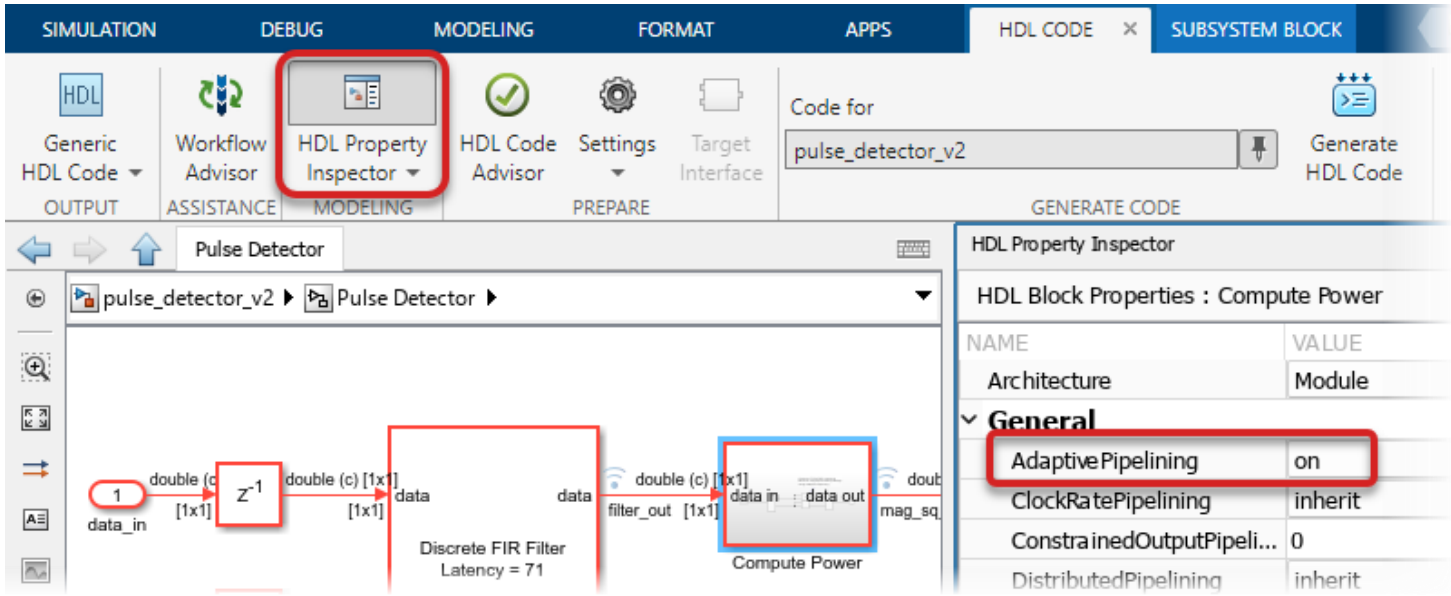
**Pipelining options:**



# Step 2.4: Insert pipeline registers

3. Open the **HDL Code** Toolstrip under Apps > Code Generation > HDL Coder.
4. Select the *Compute Power* subsystem and click **HDL Property Inspector** on the HDL Code Toolstrip. This opens its HDL Block Properties dialog**.**
5. Set `AdaptivePipelining` to on, then close or minimize the dialog.



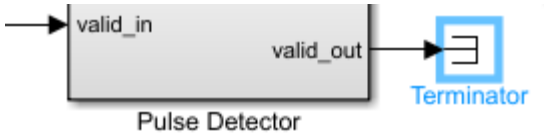

Tip: HDL Block Properties do not change simulation behavior in Simulink; they are applied to the generated HDL code.

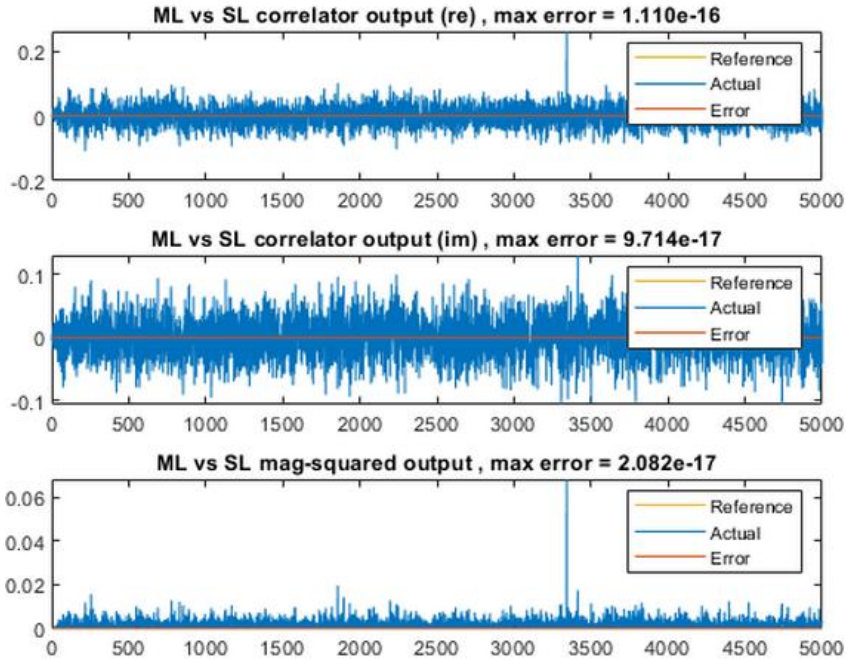
# Step 2.5: Compare architecture model to MATLAB reference

1. In the top-level model, terminate the DUT output port *valid_out* by connecting it to a **Terminator** block. Save the model.
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.



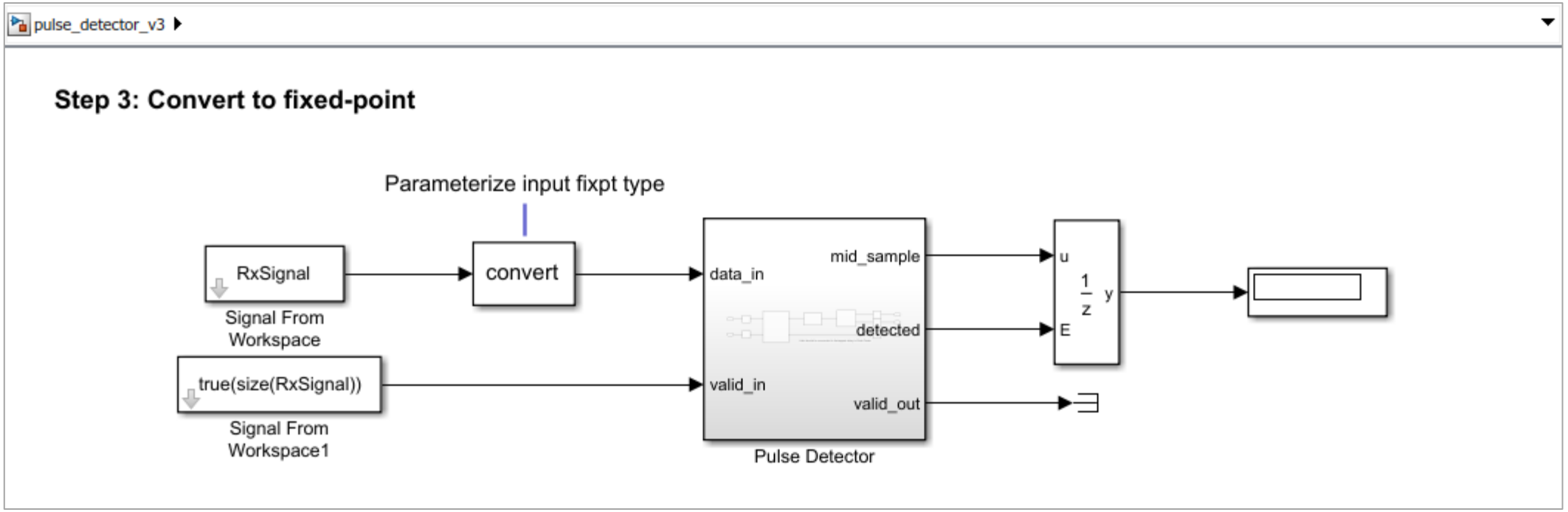

*Question: What's wrong with the valid signal implementation?*

*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:

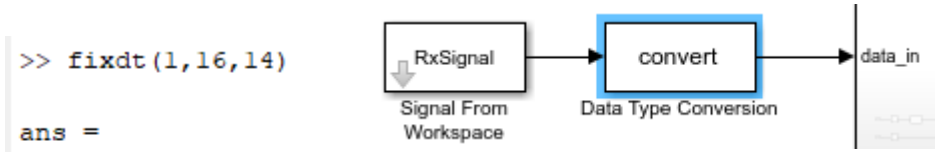
- Convert the model to fixed-point
- Compare the Simulink fixed-point model to the MATLAB golden reference

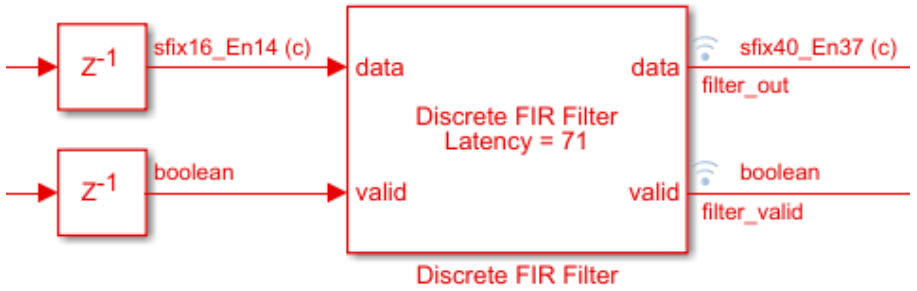


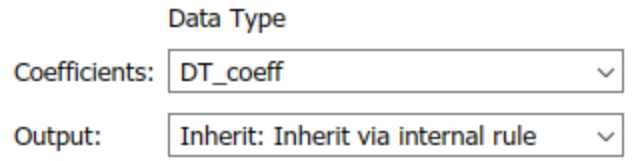
# Step 3.1: Define input and filter fixed-point data types

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 FIR 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 as the model is partially converted to fixed-point).

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

ans =

  NumericType with properties:

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

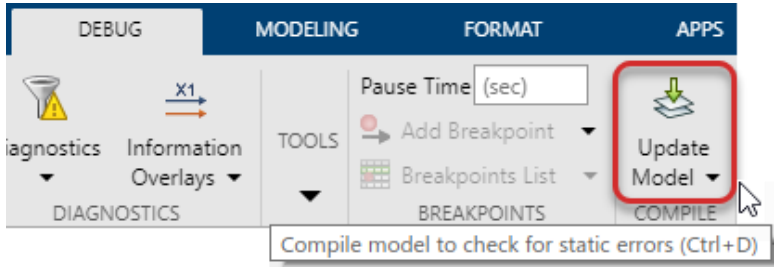


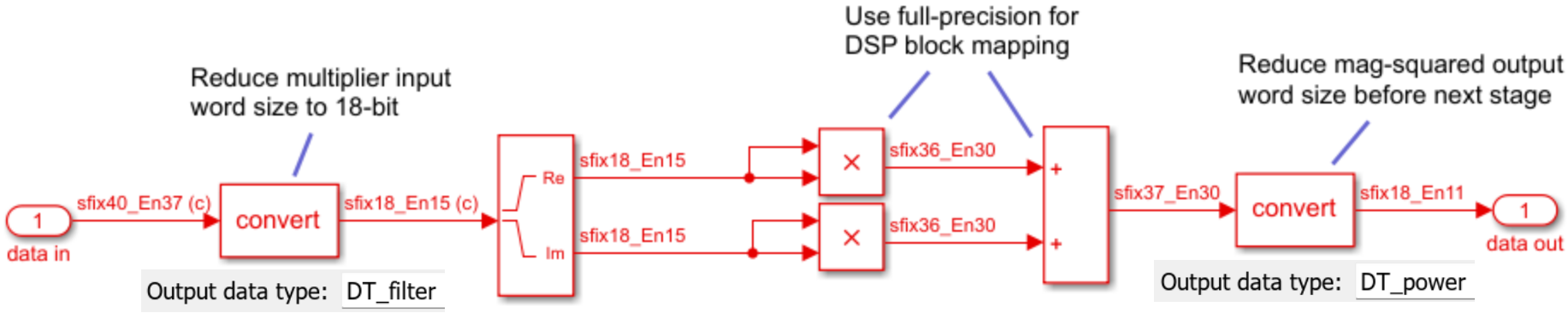



# Step 3.2: Define magnitude-squared data types

1. In the *Compute Power* subsystem, add a **Data Type Conversion** block after the input port; set `Output data type` 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 (error is still okay).




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

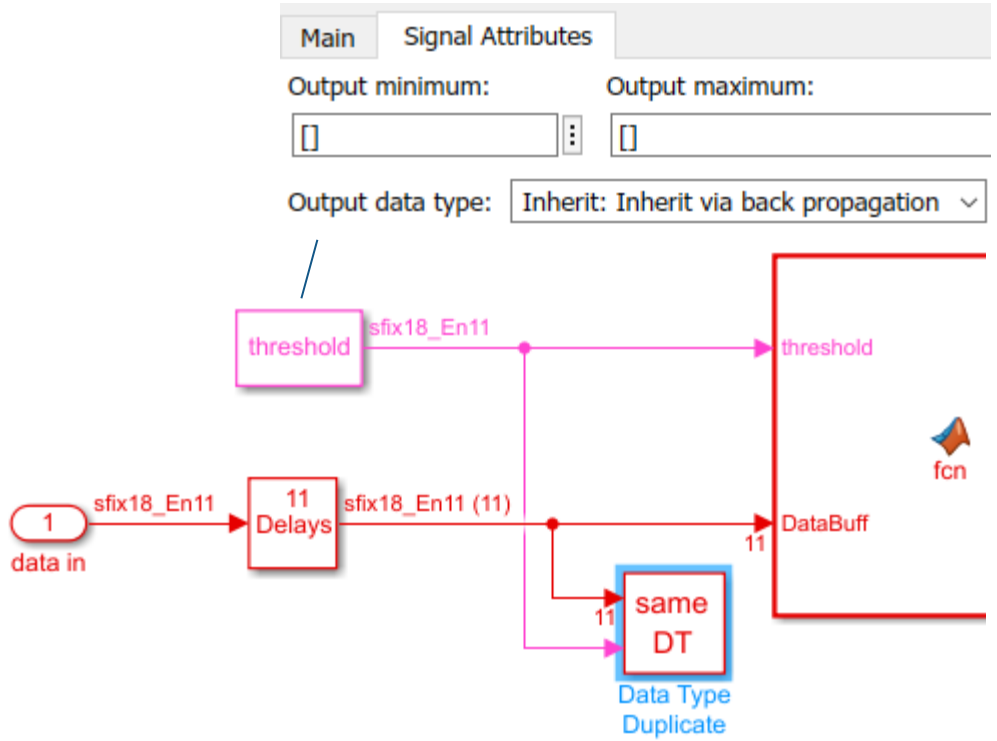
1. In the *Local Peak* subsystem, add a **Data Type Duplicate** block. Connect one port to the threshold constant block output, and another to the tapped delay block output.
2. Set `Output data type` of the threshold constant to **Inherit via back propagation**.

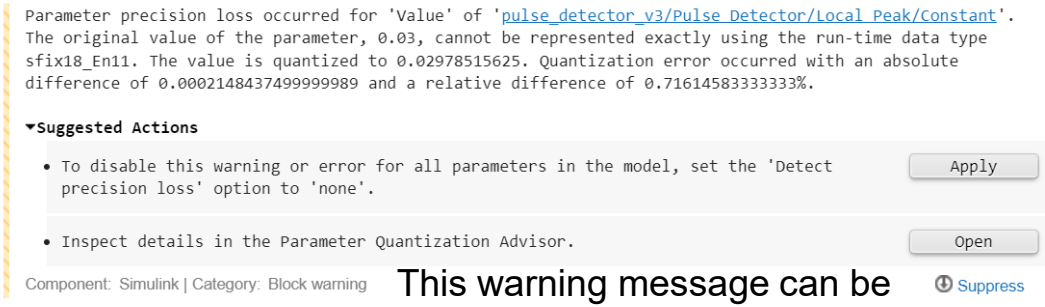
   These 2 steps allow the threshold constant to take on the same data type used by the input data.

3. In the MATLAB function block, change the *detected* value to true/false.

```
11 -    if all(CompareOut <= 0) && (MidSample > threshold)
12 -        detected = true;
13      else
14 -        detected = false;
15      end
```

4. Update and save the model. It should be error-free at this point.




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.

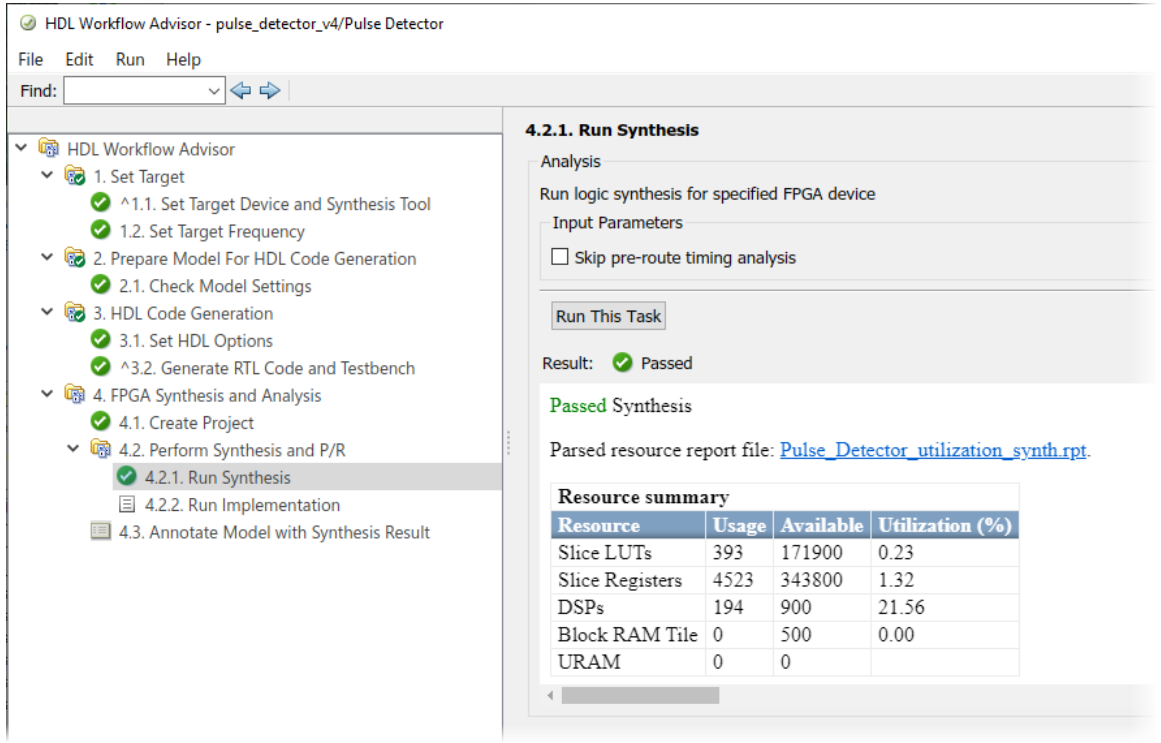
# Step 4: HDL code generation & synthesis

In this step, you will:

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

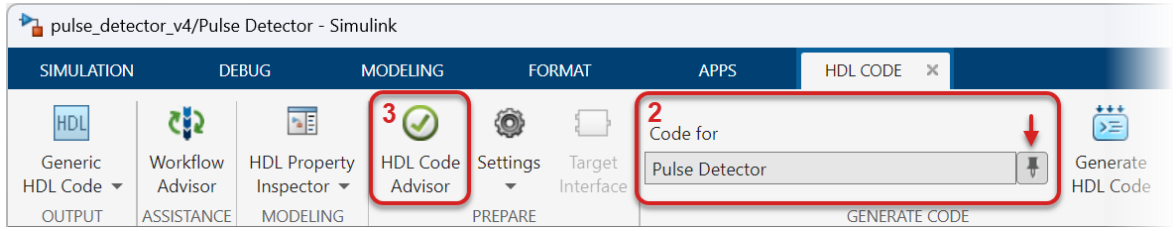
Compute_Power.vhd

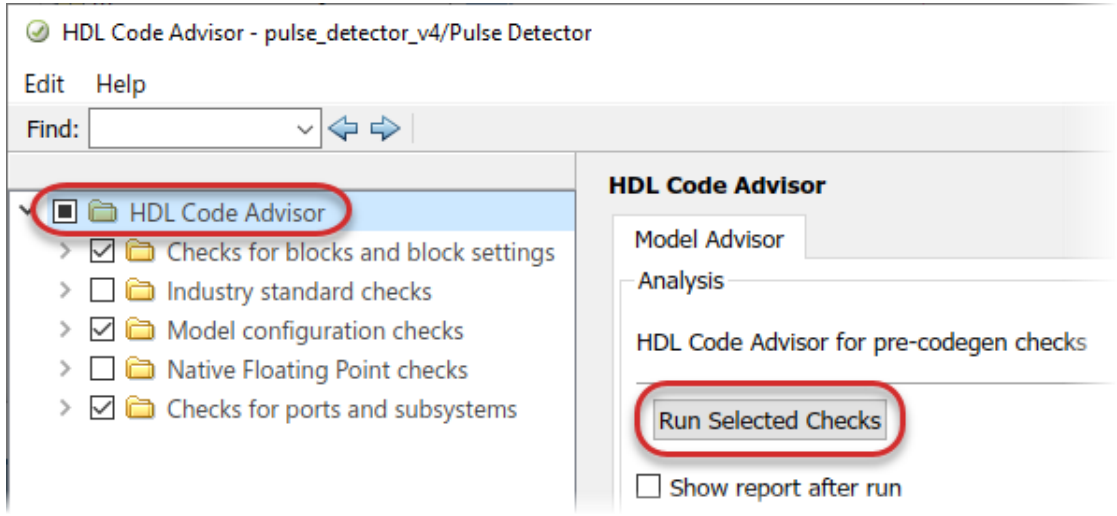
```
96
97      Product1_out1 <= Complex_to_Real_Imag_out2 * Complex_to_Real_Imag_out2;
98
99      PipelineRegister1_process : PROCESS (clk)
100     BEGIN
101       IF clk'EVENT AND clk = '1' THEN
102         IF reset = '1' THEN
103           Product1_out1_1 <= to_signed(0, 36);
104         ELSIF enb = '1' THEN
105           Product1_out1_1 <= Product1_out1;
106         END IF;
107       END IF;
108     END PROCESS PipelineRegister1_process;
109
110
111     Add_add_cast <= resize(Product_out1_1, 37);
112     Add_add_cast_1 <= resize(Product1_out1_1, 37);
113     Add_out1 <= Add_add_cast + Add_add_cast_1;
114
115     Data_Type_Conversion_out1 <= Add_out1(36 DOWNTO 19);
116
117     data_out <= std_logic_vector(Data_Type_Conversion_out1);
118
119   END rtl;
120
```



# Step 4.1: Check model for HDL compatibility

1. Save the Simulink model as pulse_detector_v4.slx.
2. Set the *Pulse Detector* subsystem as DUT using the HDL Code Toolstrip (click  to unlock; select the subsystem in the model; click  again to lock).
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 groups **Industry standard checks** and **Native Floating Point checks** – they are not applicable for this example. Highlight the top folder named **HDL Code Advisor**, then click **Run Selected Checks**.

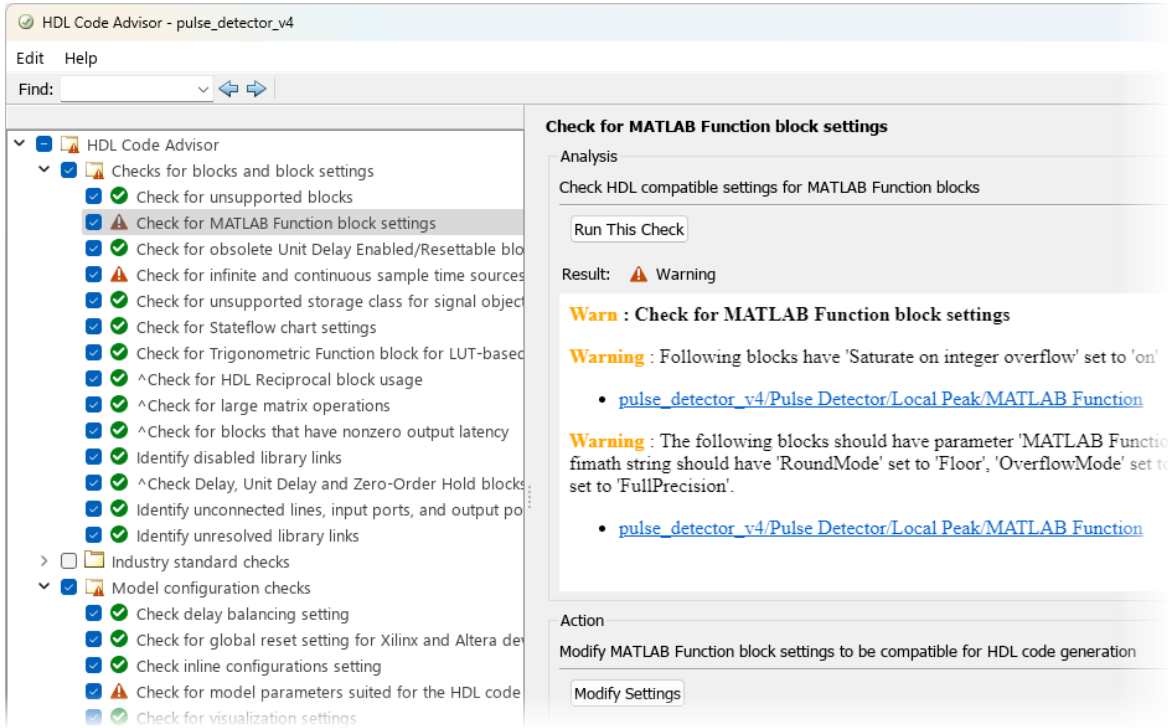


# Step 4.1: Check model for HDL compatibility

5. HDL Code Advisor identifies 3 warnings that are explained below. For this exercise, you may ignore the warnings and close the UI.

- **Check for MATLAB Function block settings** reports the use of saturation and rounding logic, which consumes extra hardware resources. If your design works with floor rounding and wrap on overflow, you may click Modify Settings to apply them automatically.
- **Check for infinite and continuous sample time sources** reports the use of *inf* sample time in the model, which may inhibit some optimization features. If that is the case, you may click Modify Settings to replace all such instances with back propagation (-1).
- **Check for model parameters suited for HDL code generation** recommends Simulink diagnostics settings that may help identify code generation issues, such as warnings for unconnected signals.



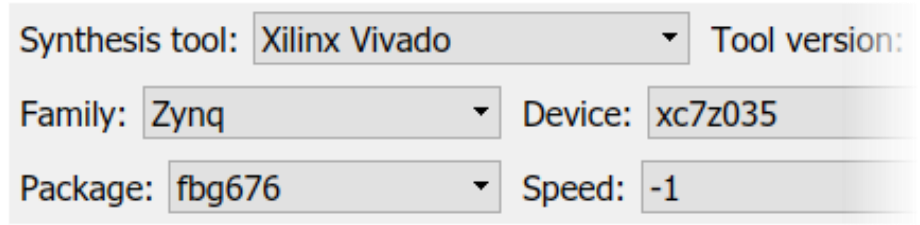
# Step 4.2: Generate HDL code and reports

1. In MATLAB, run the following command to add AMD Vivado 2024.1 to the system path:

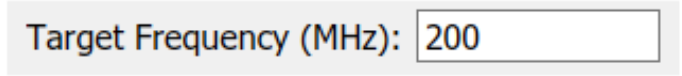
```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2024.1\bin');
```

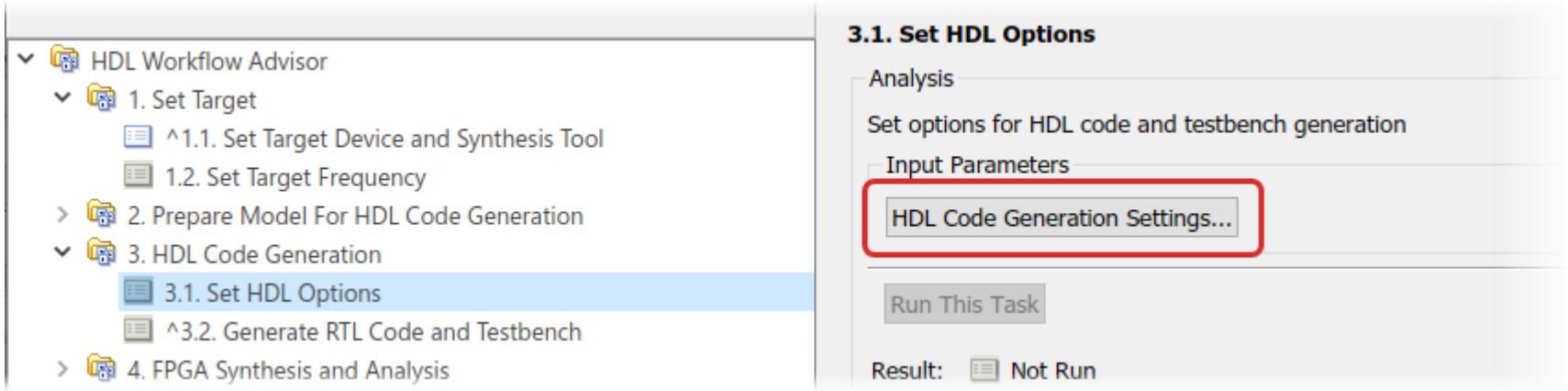
2. Launch **HDL Workflow Advisor** for the DUT subsystem (HDL Code Toolstrip > Workflow Advisor).
3. Configure the target settings as shown on the right in step 1. Click **Apply** at each step to save the changes.
4. Open **HDL Code Generation Settings…** in step 3.1.

1.1:



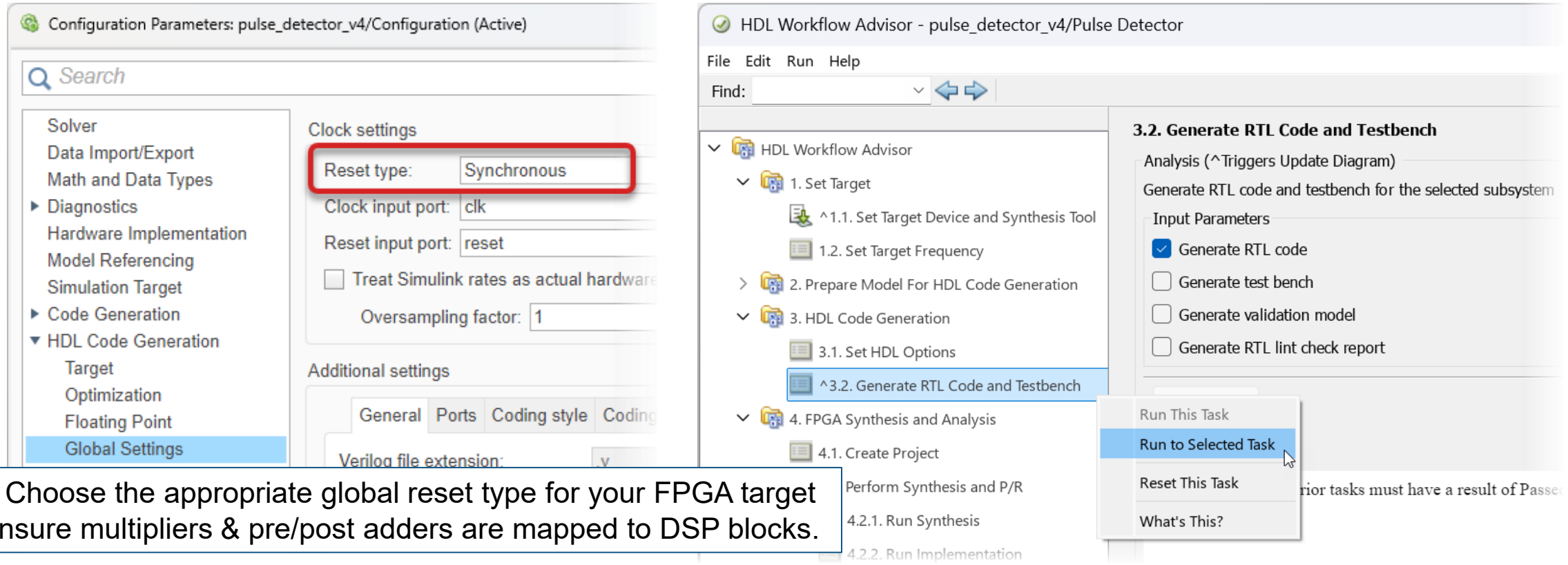

1.2:



# Step 4.2: Generate HDL code and reports

5. Change `Reset type` to `Synchronous` (under **HDL Code Generation** > **Global Settings**), then click **OK** to close the Configuration Parameters dialog.
6. Back in HDL Workflow Advisor, run through steps 1 to 3 (right-click on step 3.2 > **Run to Selected Task**).

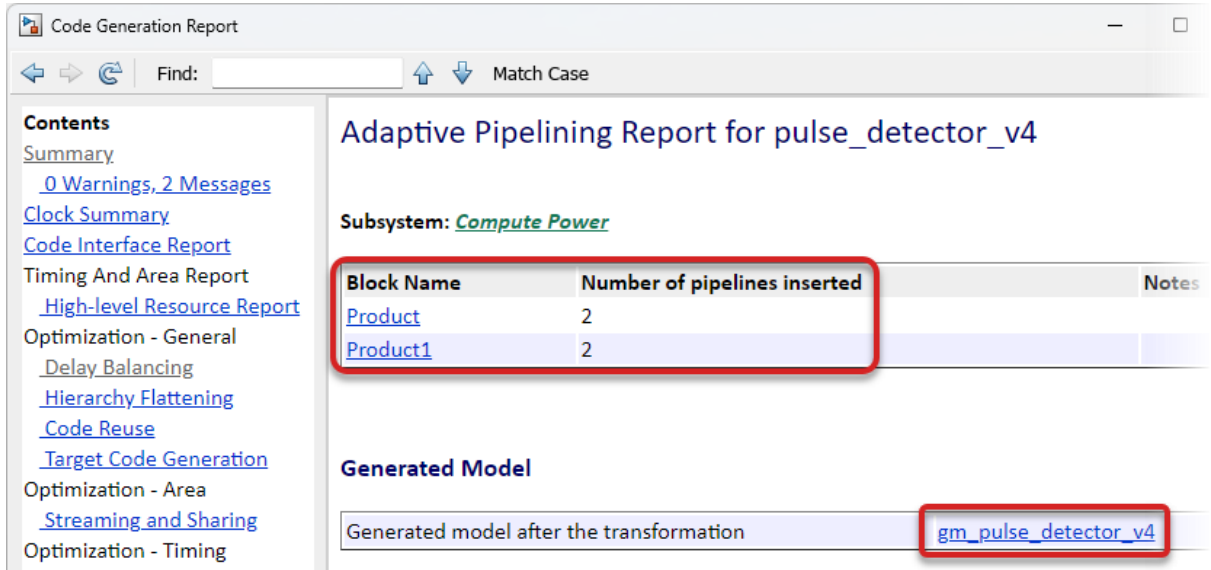


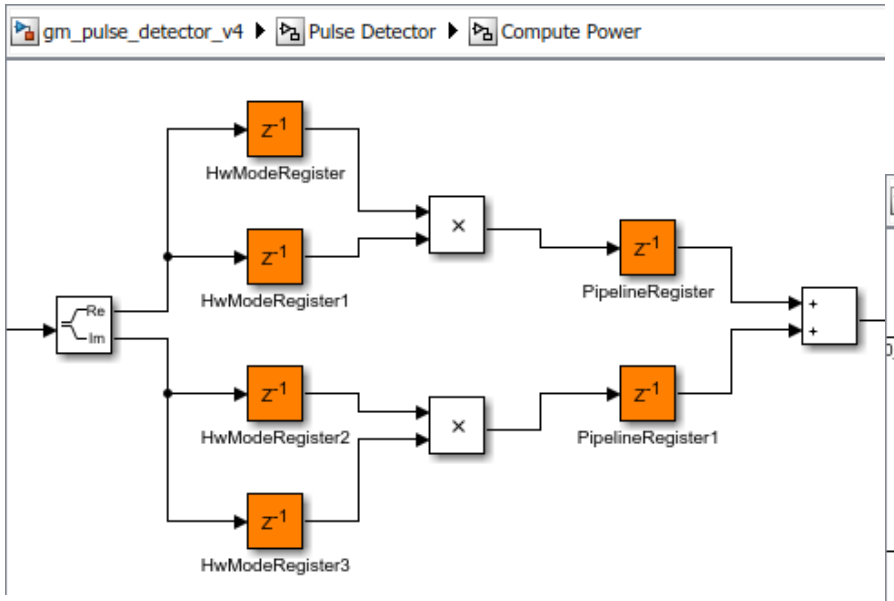

Tip: Choose the appropriate global reset type for your FPGA target to ensure multipliers & pre/post adders are mapped to DSP blocks.

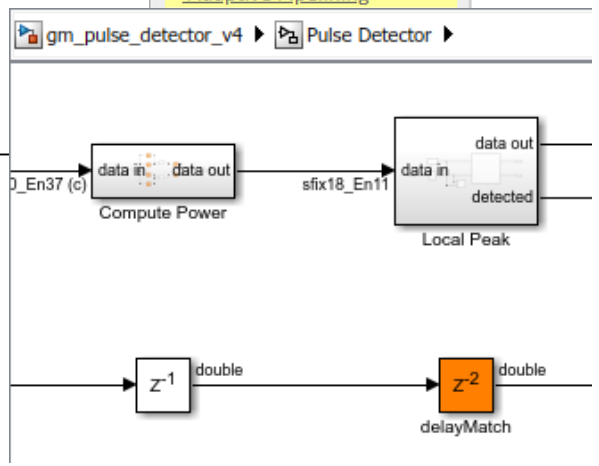
# Step 4.2: Generate HDL code and reports

7. 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.

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



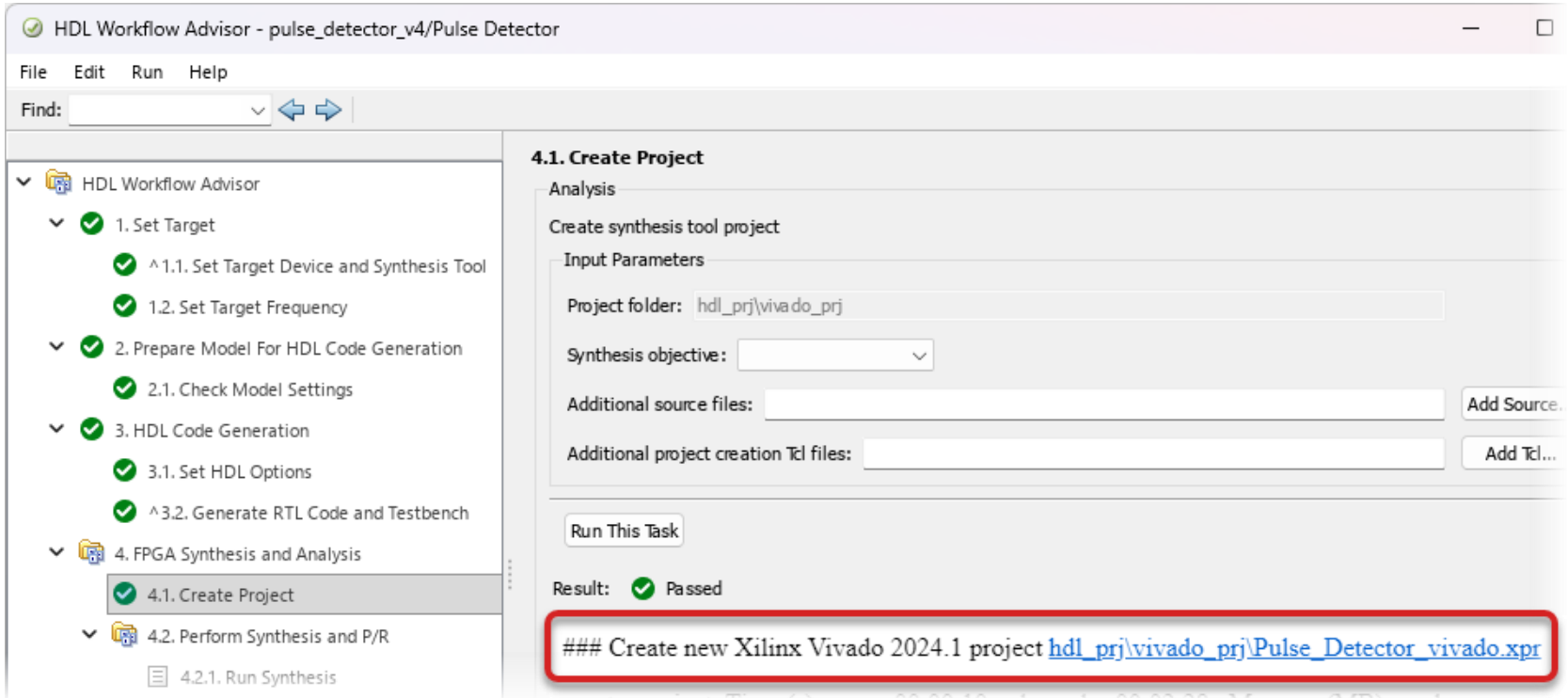



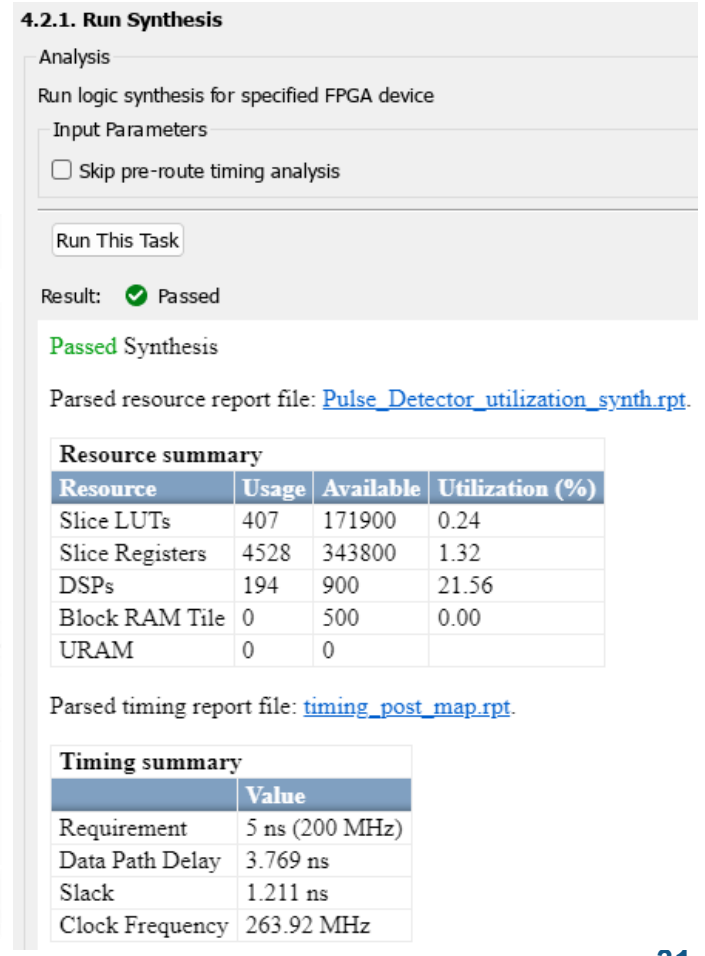




Note: The generated model is bit-true & cycle-accurate to the generated HDL. It is used to verify the generated code.

# Step 4.3: Synthesize the generated design

1. Run step 4.1 to create a 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 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