

Pulse Detector HDL Workflow Tutorial

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

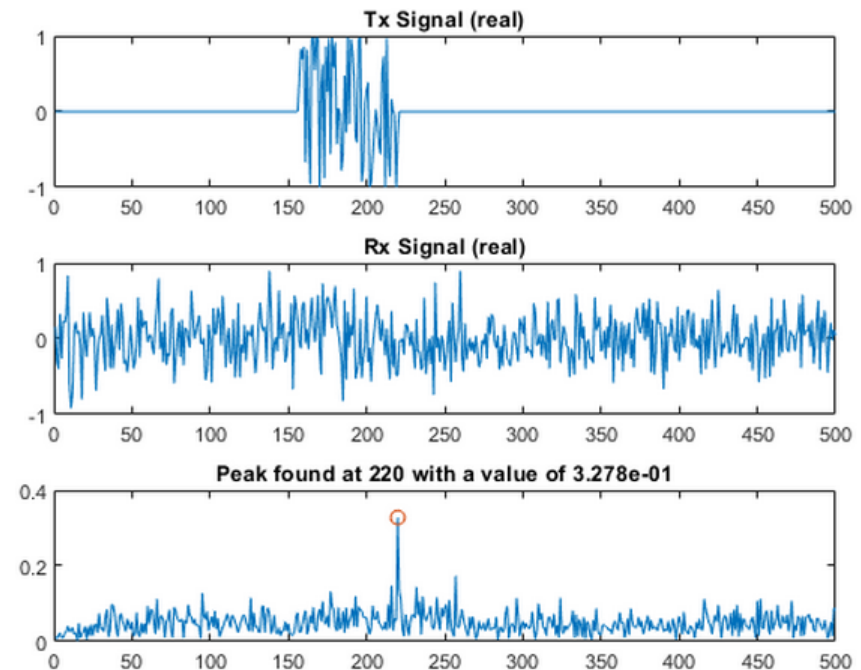
```
% Create matched filter coefficients
CorrFilter = conj(flip(pulse))/PulseLen;

% 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

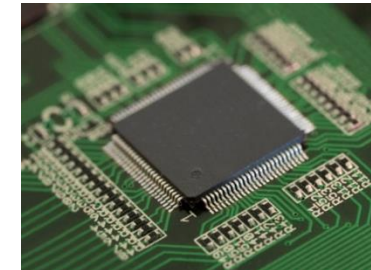
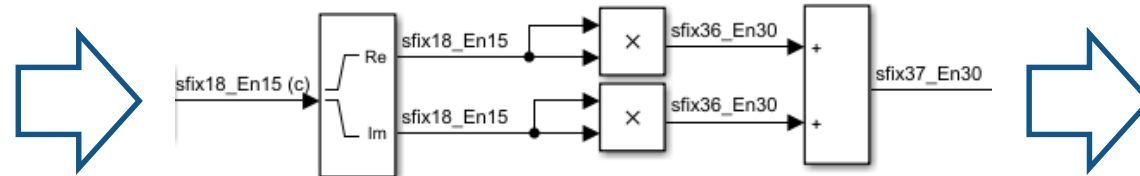
This tutorial will guide you through the steps necessary to implement the algorithm in FPGA hardware, including:

- 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® (R2019a)
 - Simulink®
 - Fixed-Point Designer™
 - MATLAB Coder™
 - HDL Coder™
 - Signal Processing Toolbox™
 - DSP System Toolbox™

- Xilinx® Vivado® 2018.2 is required for the last step (step 4)
 - Other Vivado versions typically work, but synthesis results may be different.

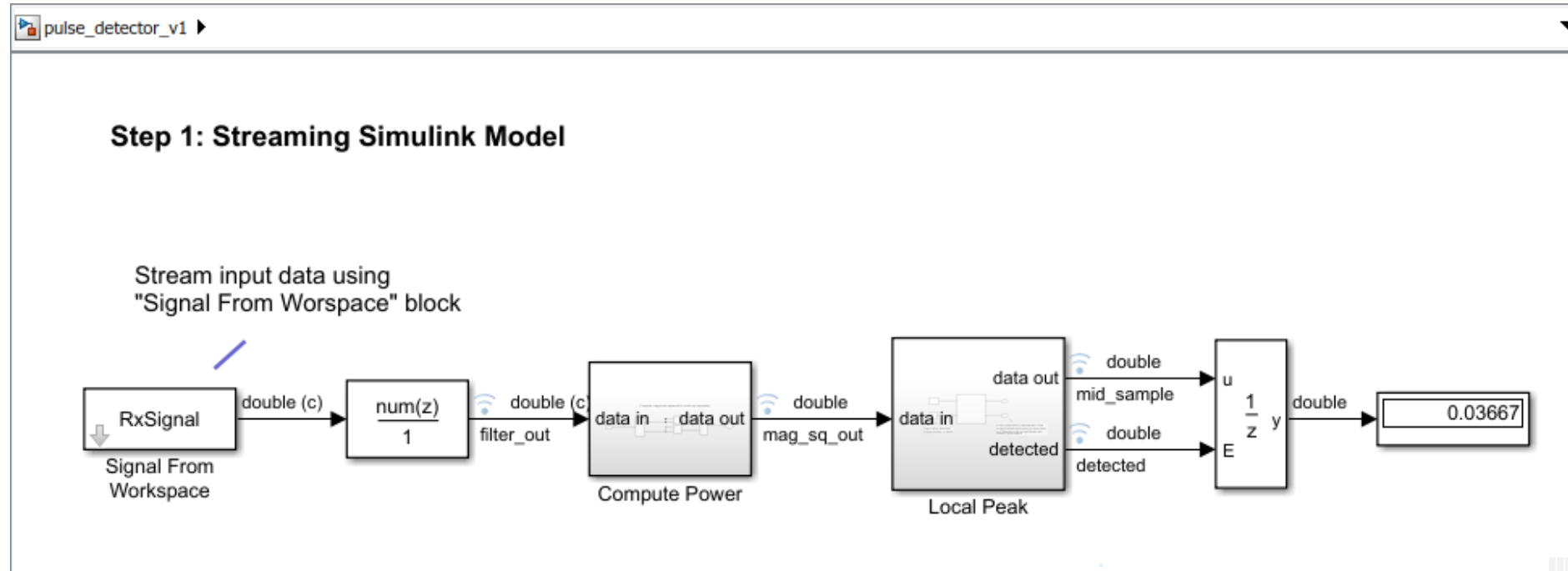
Preparation

- 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 Evaluation Reference Guide](#)
- Locate the example files in the folder: **/pulse_detector/work**

Step 1: Streaming Simulink model

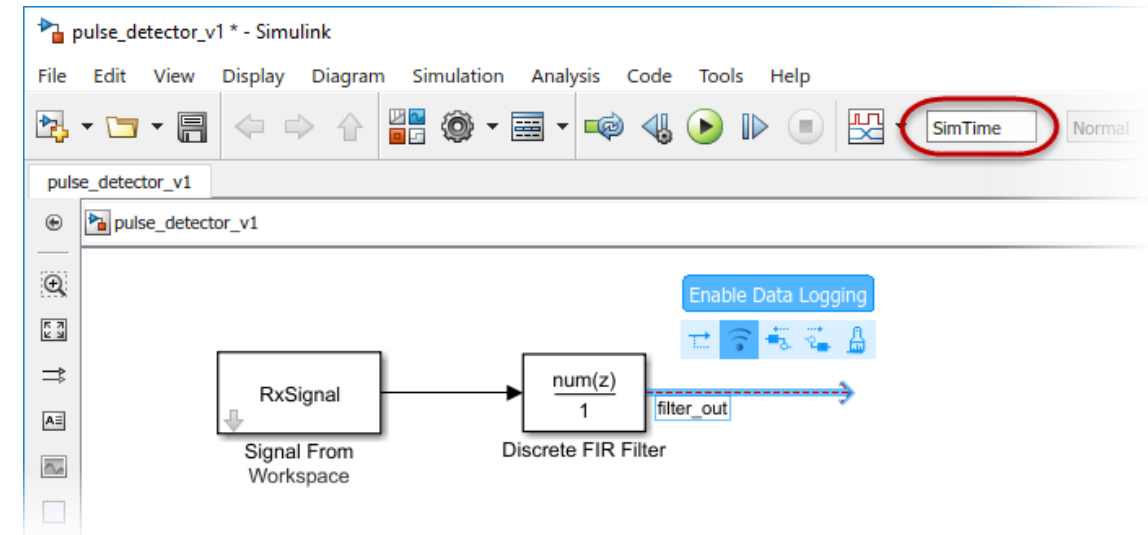
In this step, you will:

- 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 mistake early on.
 - In the test bench script **pulse_detector_v1_tb.mlx**, add a breakpoint on line 16.
 - Run the test bench. Verify the Simulink filter output matches the MATLAB reference.
 - Stop the test bench and remove the breakpoint.

Simulate model and compare results to reference

```

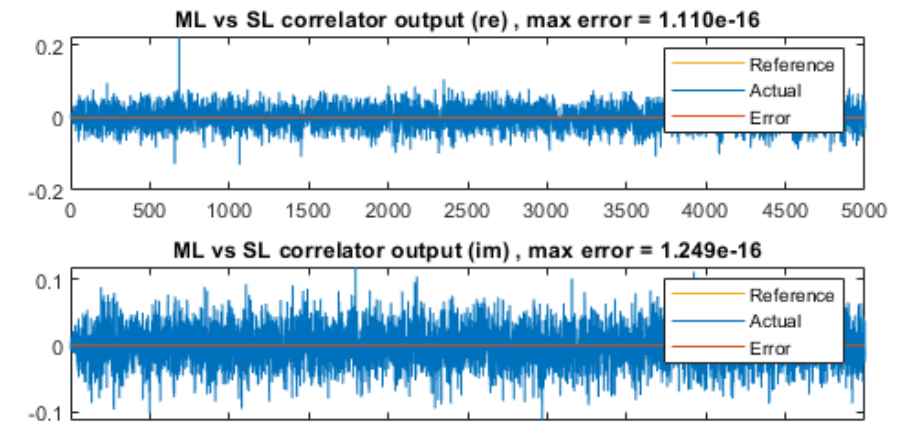
2  if iscolumn(CorrFilter)
3      CorrFilter = transpose(CorrFilter); % need row vector for filter block
4  end
5  SimTime = length(RxSignal) + WindowLen;
6
7  % Simulate model
8  slout = sim('pulse_detector_v1');
9
10 % Correlation filter output
11 FilterOutSL = squeeze(slout.logout.getElement('filter_out').Values.Data);
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 = squeeze(slout.logout.getElement('mag_sq_out').Values.Data);
17 compareData(MagSqOut,MagSqSL,{2 3 3},'ML vs SL mag-squared output');

```

Warning: Unconnected output line found on '[pulse_detector v1/Discrete FIR Filter](#)'

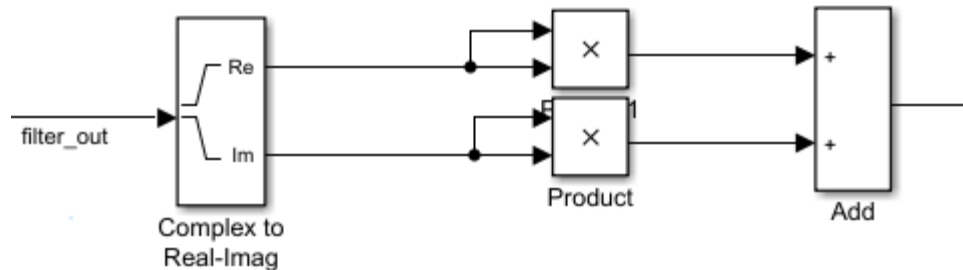
Maximum error for ML vs SL correlator output (re) out of 5000 values
1.110223e-16 (absolute), 4.961869e-14 (percentage)

Maximum error for ML vs SL correlator output (im) out of 5000 values
1.249001e-16 (absolute), 1.057727e-13 (percentage)

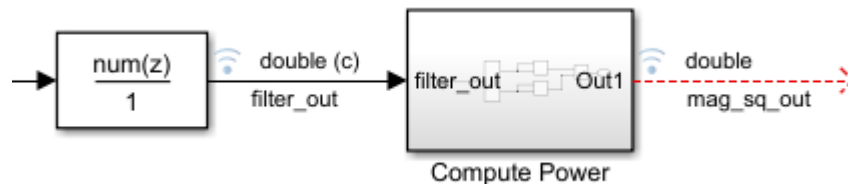


Step 1.2: Hardware-friendly peak finder (magnitude-squared)

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**.

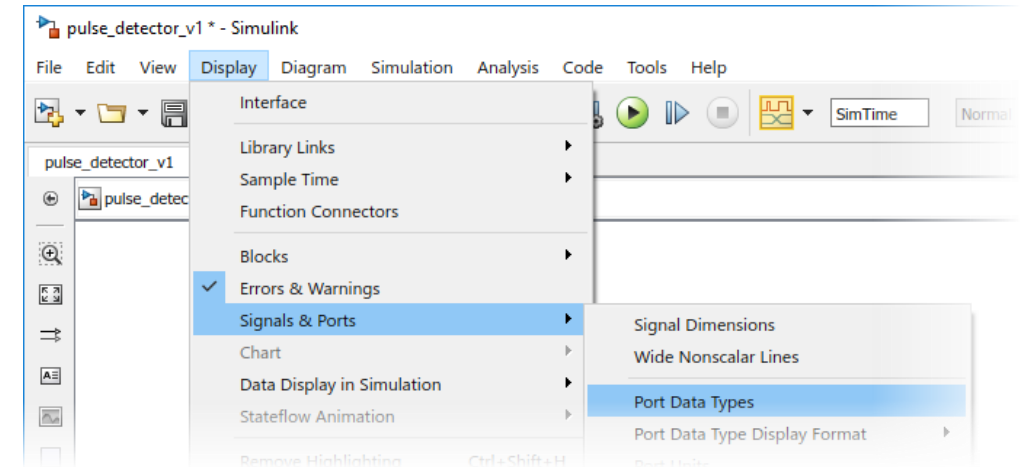


Hardware friendly implementation of peak finder

Instead of calculating the maximum value of the entire frame, we look for a local peak within a sliding samples using the following criteria:

- The middle sample is the largest
- The middle sample is greater than a pre-defined threshold

```
42 WindowLen = 11;
43 MidIdx = ceil(WindowLen/2);
44 threshold = 0.03;
45
46 % Compute magnitude squared to avoid sqrt operation
47 MagSqOut = abs(FilterOut).^2;
48
```



Tip: Enable **Signal Dimensions & Port 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.

The screenshot shows the MATLAB Function block editor for 'Block: pulse_detector_v1/MATLAB Function'. The editor displays the following code:

```

1 function [MidSample,detected] = fcn(threshold,DataBuff, WindowLen)
2
3 MidIdx = ceil(WindowLen/2);
4
5 % Compare each value in the window to the middle sample via subtraction
6 MidSample = DataBuff(MidIdx);
7 CompareOut = DataBuff - MidSample; % this is a vector
8
9 % if all values in the result are negative and the middle sample is
10 % greater than a threshold, it is a local max
11 if all(CompareOut <= 0) && (MidSample > threshold)
12     detected = 1;
13 else
14     detected = 0;
15 end
16

```

The 'Edit Data' button in the top right of the editor is circled in red. A red arrow points from this button to the 'Ports and Data Manager' window below.

The 'Ports and Data Manager' window for 'pulse_detector_v1/MATLAB Function' shows the following table:

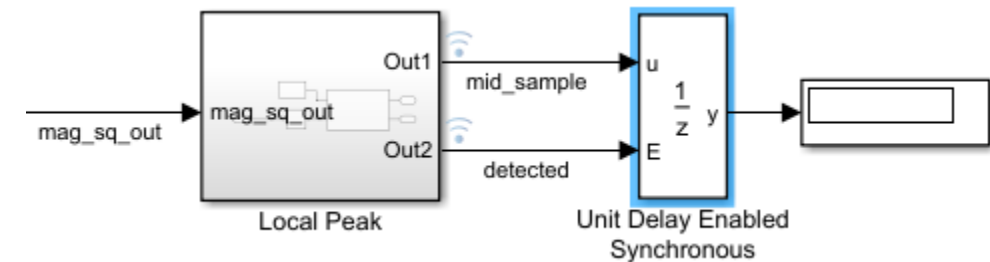
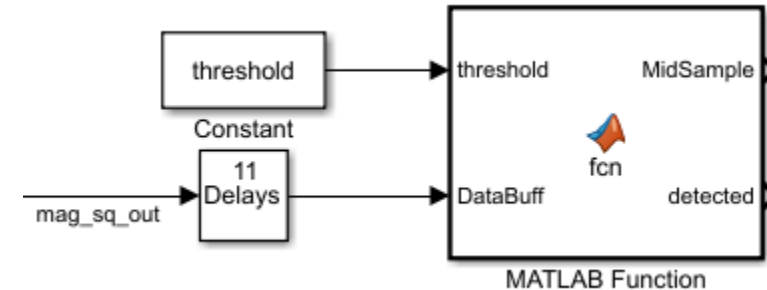
Name	Scope	Port
WindowLen	Parameter	
MidSample	Output	1
threshold	Input	1
DataBuff	Input	2
detected	Output	2

The 'Data WindowLen' section on the right shows the 'General' tab with the following settings:

- Name: WindowLen
- Scope: Parameter (selected)
- Tunable: ☐ (unchecked)
- Size: -1

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.

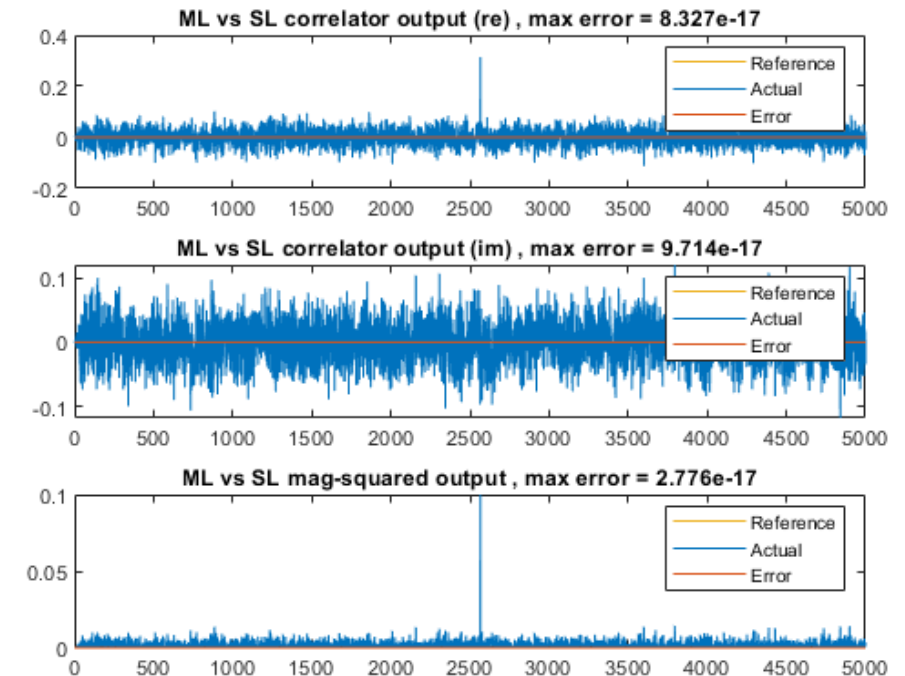


Step 1.4: Compare model to MATLAB reference

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)

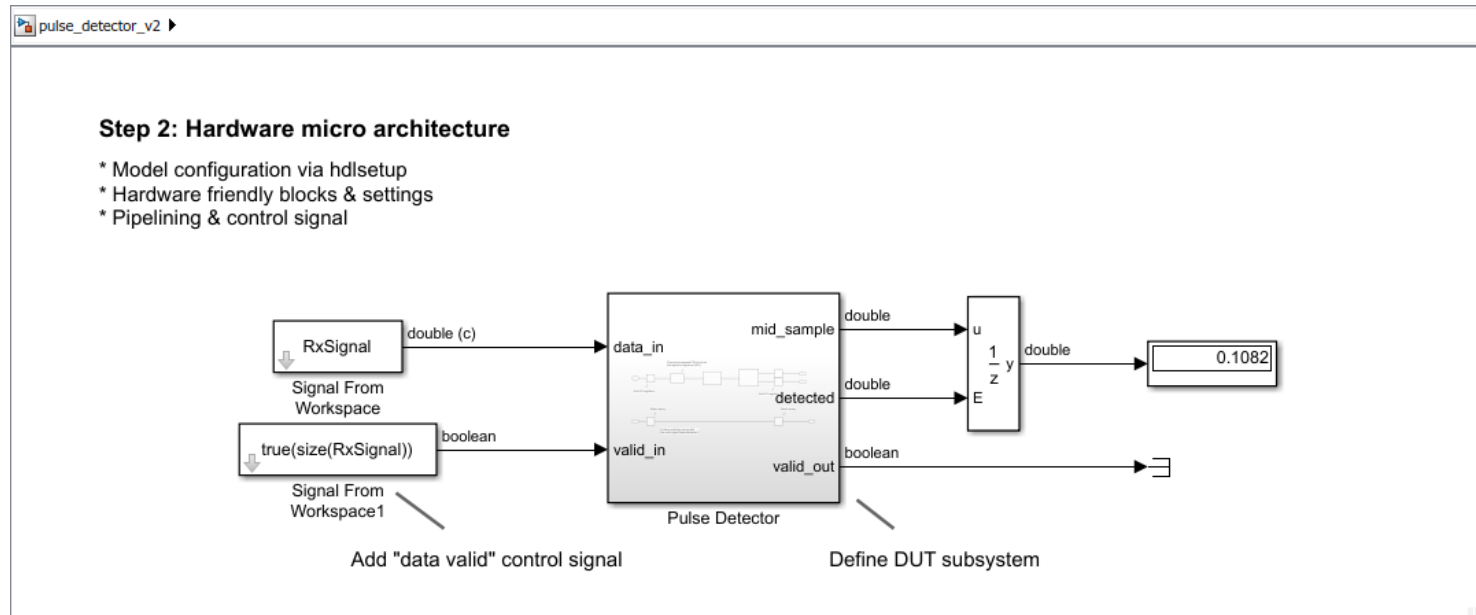


Peak location = 2566, magnitude = 3.164e-01 using global max
 Peak location = 2566, mag-squared = 1.001e-01 using local max
 Peak mag-squared from Simulink = 1.001e-01, error = 2.776e-17

Step 2: Hardware micro architecture

In this step, you will:

- Prepare the model for HDL code generation
- Pipeline the data path using various techniques
- Add data valid control signal
- Compare the Simulink architecture model to the MATLAB golden reference



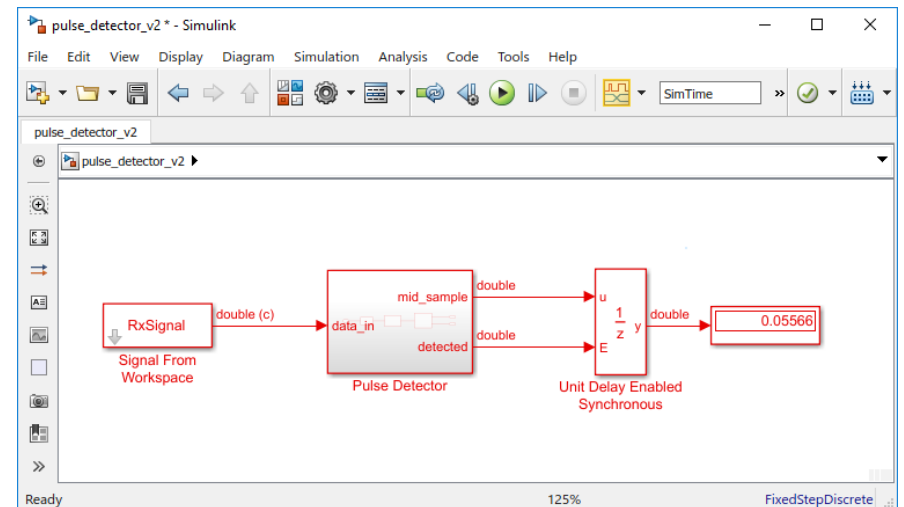
Step 2.1: HDL model configurations

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.
4. Inside the DUT subsystem, name the input port **data_in**.

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.

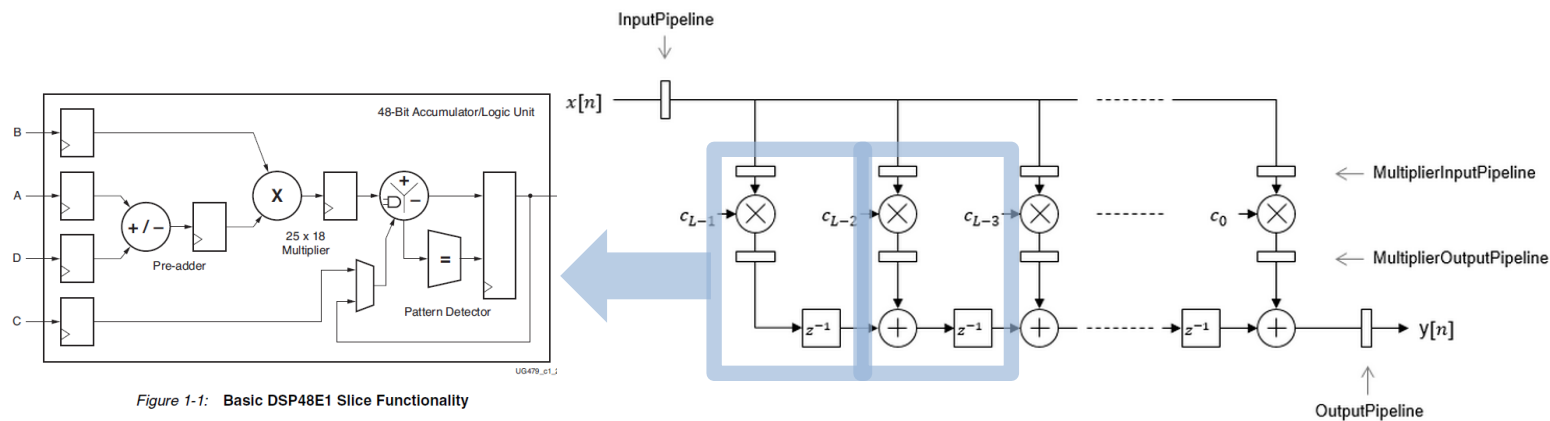
```

hdlsetup.m
28
29 % Model Parameters
30 set_param(model, ...
31     'SingleTaskRateTransMsg', 'error', ...
32     'MultiTaskRateTransMsg', 'error', ...
33     'Solver', 'fixedstepdiscrete', ...
34     'SolverMode', 'SingleTasking', ...
35     'FixedStep', 'auto', ...
36     'SaveTime', 'off', ...
37     'SaveOutput', 'off', ...
38     'AlgebraicLoopMsg', 'error', ...
39     'SampleTimeColors', 'on', ...
40     'ShowLineDimensions', 'on', ...
41     'ShowPortDataTypes', 'on', ...
42     'BlockReduction', 'off', ...
43     'ConditionallyExecuteInputs', 'off', ...
44     'InlineParams', 'on', ...
45     'ProdHWDeviceType', 'ASIC/FPGA->ASIC/FPGA', ...
46     'DataTypeOverride', 'ForceOff', ...
47     'SignalLoggingSaveFormat', 'Dataset');
  
```

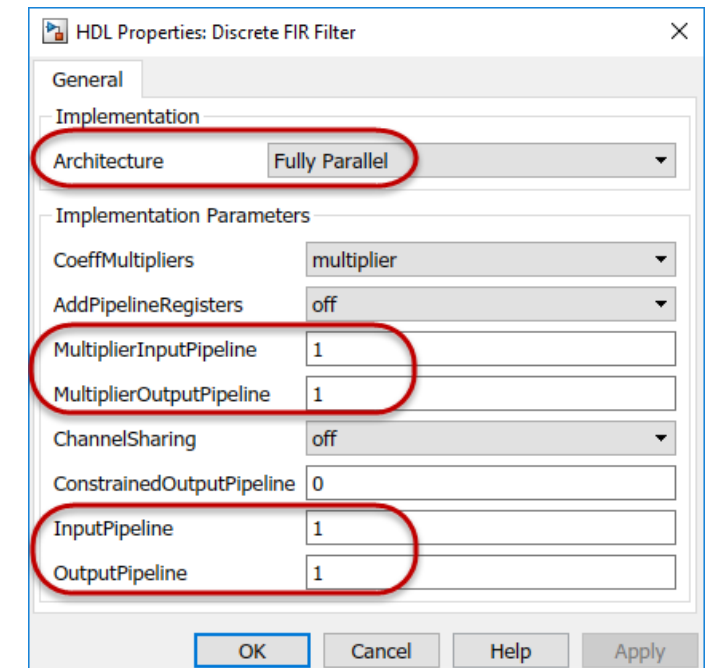


Step 2.2: FIR structure and pipelining

1. Inside the DUT, open the FIR filter block dialog, and change Filter structure to **Direct form transposed**. The transposed structure has a shorter critical path, resulting in better timing performance.



2. Insert additional pipeline registers as shown in the figure to the right, using the filter block's **HDL Block Properties** dialog (right-click on the block > HDL Code). This leverages the built-in registers within the FPGA DSP core, and further improves timing.

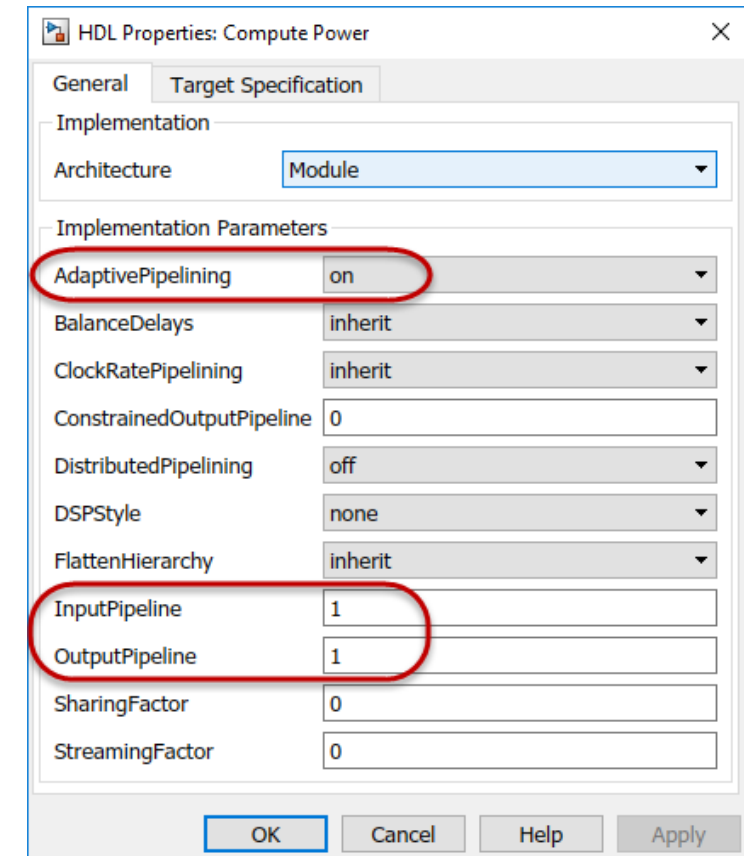


Step 2.3: Magnitude-squared pipelining

1. Open the **HDL Block Properties** dialog for the *Compute Power* subsystem. Add 1 level of input & output pipeline registers, and set Adaptive Pipelining to **on**.

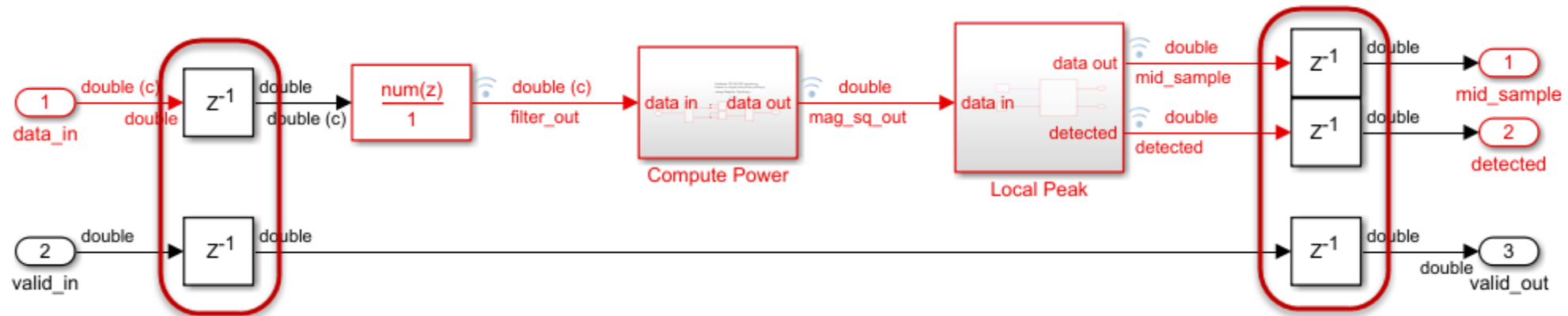
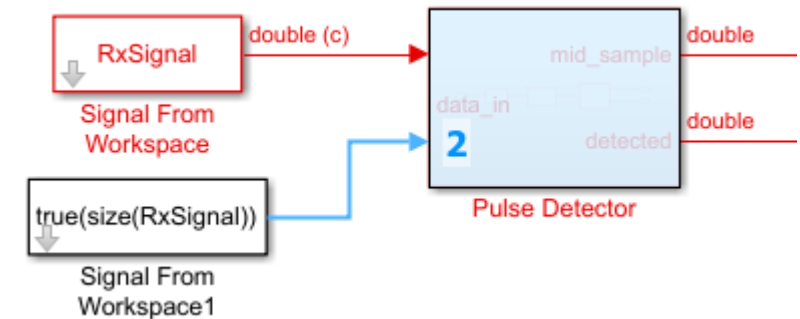
Adaptive Pipelining automatically inserts pipeline registers for certain operations, and adjusts the amount based on hardware target device and frequency settings. In this case, the multipliers will be pipelined for optimal DSP block mapping.

Note: Pipelines added using HDL block properties, such as those performed in the last 2 steps, do not affect simulation in the current model. You will be able to examine and simulate them during the code generation step.



Step 2.4: Data valid and I/O pipelining

1. In the top-level model, add a data valid input using a **Signal From Workspace** block, and enter `true(size(RxSignal))` for Signal1. Data valid is commonly used in hardware to indicate whether the input data is a valid sample.
2. Connect the block to the DUT subsystem as a second input.
3. Inside the DUT subsystem, name the new input port `valid_in`. Connect it to a new **Output** port and name it `valid_out`.
4. Pipeline the data input & output using **Delay** blocks. Add matching delays to the valid signal.

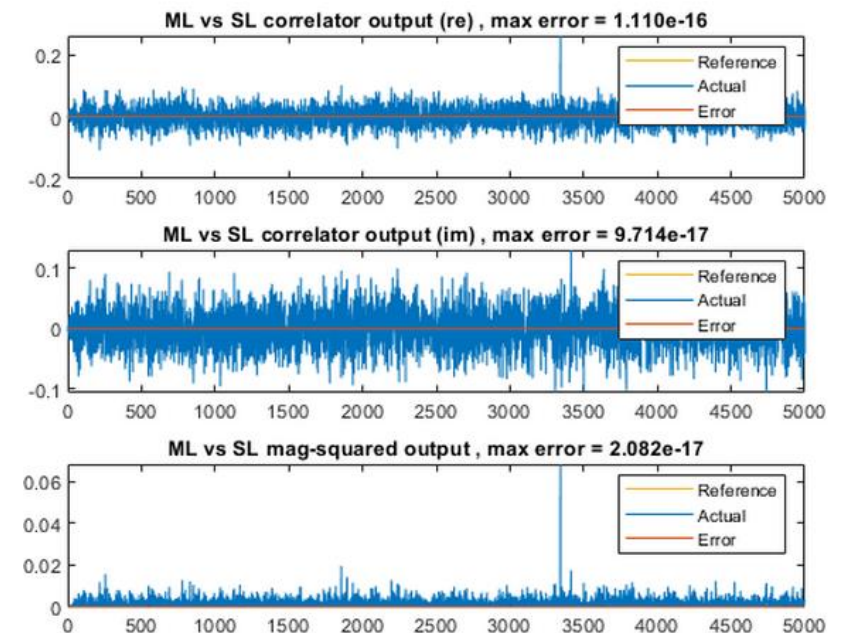
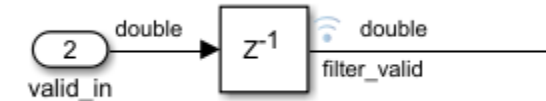


Step 2.5: Compare model to MATLAB reference

1. Log the `valid_in` delay block output as `filter_valid`. This is used to qualify the logged filter & magnitude-squared outputs in the test bench.

```
% Correlation filter output
FilterOutSL = squeeze(logsout.getElement('filter_out').Values.Data);
FilterValid = squeeze(logsout.getElement('filter_valid').Values.Data);
FilterOutSL = FilterOutSL(FilterValid);
```

2. In the top-level model, terminate the DUT output port `valid_out` by connecting it to a **Terminator** block.
3. Run `pulse_detector_v2_tb.mlx` to simulate the model, and compare the Simulink outputs to the MATLAB reference. Maximum error for all outputs should be similar to that of step 1.



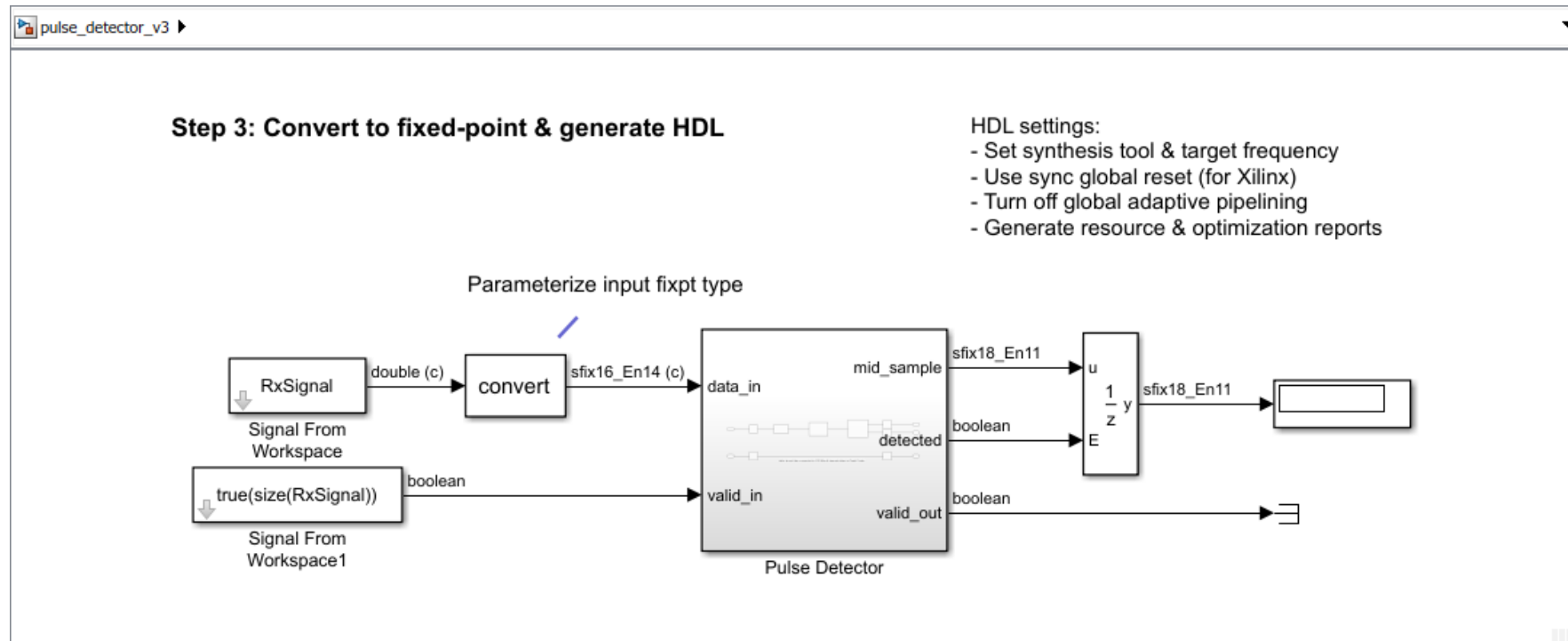
Question: What's wrong with the valid signal implementation?

The valid signal should be connected to the correlation filter and the sliding window buffer, to prevent invalid samples from entering the delay lines in those blocks. The simulation is only correct in this example because `valid_in` is always high.

Step 3: Fixed-point conversion

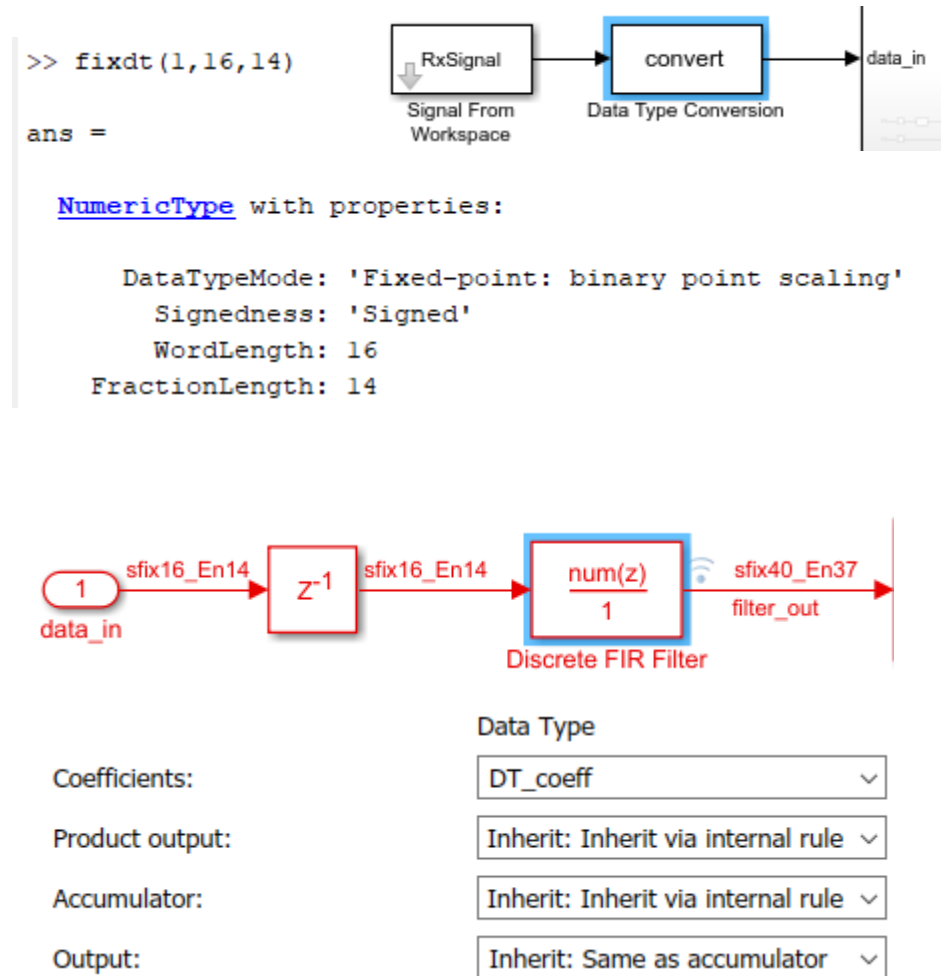
In this step, you will:

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



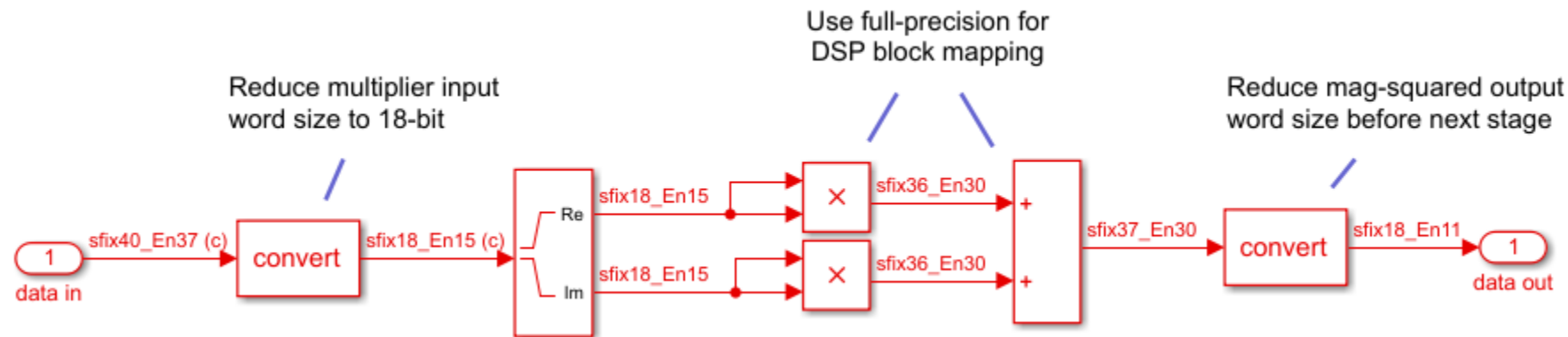
Step 3.1: Define input and filter fixed-point data types

1. Save the Simulink model as **pulse_detector_v3.slx**.
2. 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. 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).



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.

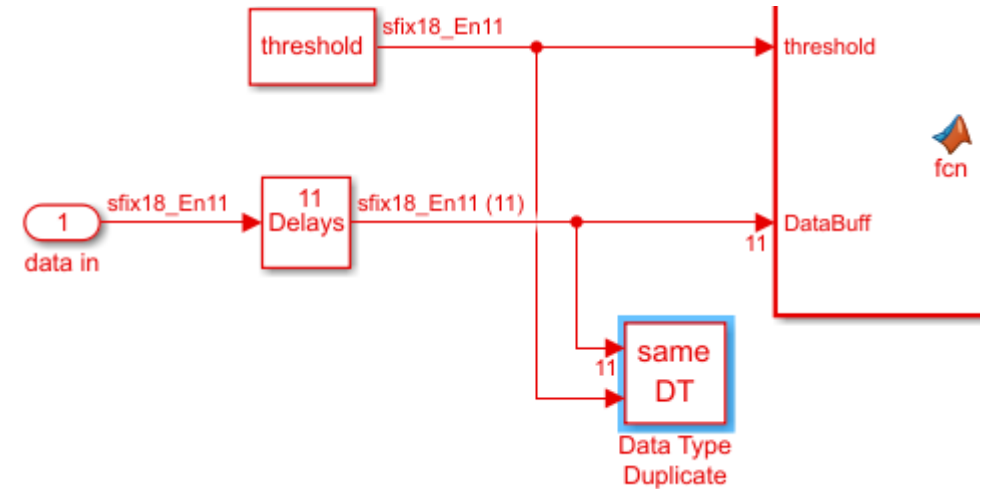


Tip: 18-bit inputs and full-precision fixed-point allow multiply-add operations (including those within the FIR filter in the previous step) to be implemented using DSP blocks in most FPGAs.

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

```

Step 3.4: Compare 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 increased in error due to quantization.
- You may switch between fixed-point and floating-point using the following flag in the test bench.

```

2  % Simulate model in fixed-point or floating-point
3  fxpt_mode = ☒ ;
4  if fxpt_mode % fixed-point
5      DT_input = fixdt(1,16,14);
6      DT_coeff = fixdt(1,18);
7      DT_filter = fixdt(1,18,15);
8      DT_power = fixdt(1,18,11);
9  else % floating-point
10     DT_input = 'double';
11     DT_coeff = 'double';
12     DT_filter = 'double';
13     DT_power = 'double';
14 end

```

```

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

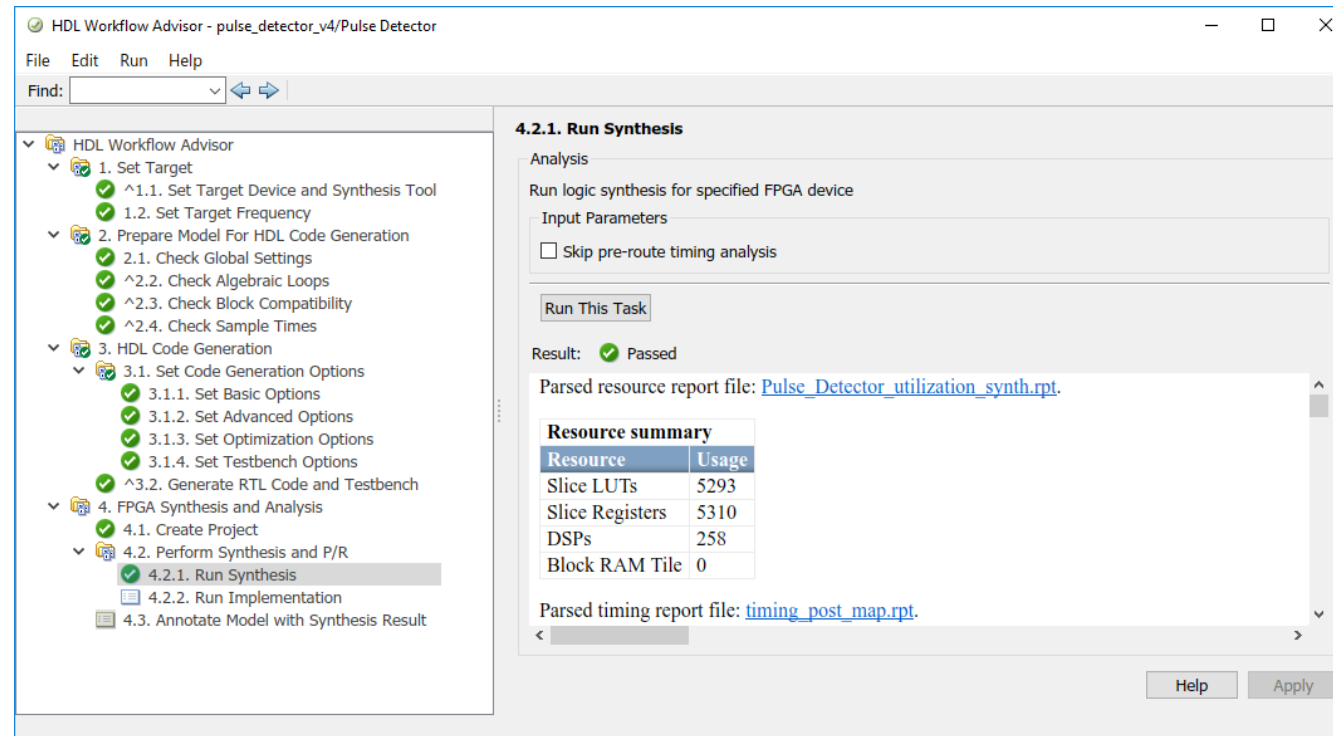
```

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



The screenshot shows the HDL Workflow Advisor window for a project named 'pulse_detector_v4/Pulse Detector'. The left pane displays a tree view of the workflow steps, with '4.2.1. Run Synthesis' highlighted. The right pane shows the details for this step, including a 'Run This Task' button, a 'Result: Passed' status, and links to generated reports.

4.2.1. Run Synthesis

Analysis

Run logic synthesis for specified FPGA device

Input Parameters

☐ Skip pre-route timing analysis

Run This Task

Result: ✔ Passed

Parsed resource report file: [Pulse_Detector_utilization_synth.rpt.](#)

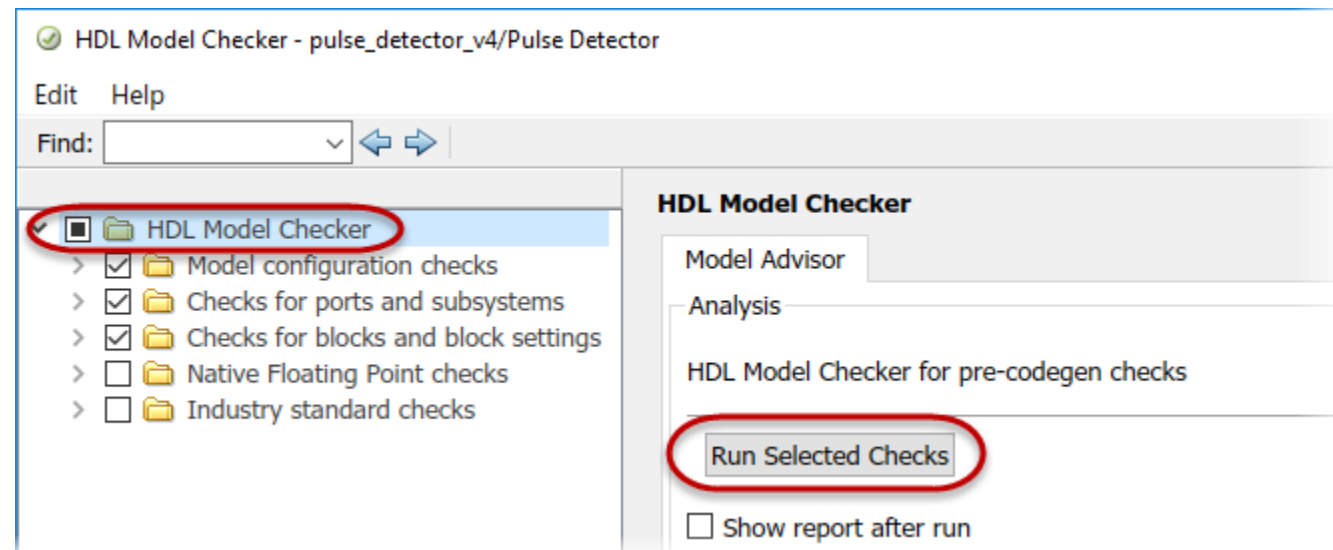
Resource summary	
Resource	Usage
Slice LUTs	5293
Slice Registers	5310
DSPs	258
Block RAM Tile	0

Parsed timing report file: [timing_post_map.rpt.](#)

Help Apply

Step 4.1: Check compatibility using HDL Model Checker

1. Save the Simulink model as **pulse_detector_v4.slx**.
2. **Check Model Compatibility** for the DUT subsystem using HDL Model Checker (right-click the subsystem > HDL Code). 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.
3. 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**.



Step 4.1: Check compatibility using HDL Model Checker

4. 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.

5. **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.

6. Close the HDL Model Checker.

The screenshot displays the HDL Model Checker interface. On the left, a tree view shows the 'HDL Model Checker' folder expanded, with 'Check for infinite and continuous sample time sources' selected. The main panel on the right shows the analysis results for this check. It indicates a 'Warning' and lists the issue: 'Infinite sample time specified in' followed by a link to 'pulse_detector_v3/DUT/Subsystem/Constant'. At the bottom of the panel, the 'Action' section shows the option 'Update sample time of source blocks to inherit via back propagation', with a 'Modify Settings' button circled in red.

Step 4.2: Generate HDL code and reports

1. In MATLAB, run the following command to add Xilinx Vivado 2018.2 to the system path:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2018.2\bin');
```
2. Launch **HDL Workflow Advisor** for the DUT subsystem (right-click the subsystem > HDL Code).
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**).

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

1.1:	SynthesisTool	Xilinx Vivado
	SynthesisToolChipFamily	Zynq
	SynthesisToolDeviceName	xc7z035
	SynthesisToolPackageName	fbg676
	SynthesisToolSpeedValue	-1
1.2:	TargetFrequency	200
3.1.2:	ResourceReport	on
	OptimizationReport	on
3.1.3:	ResetType	Synchronous
3.1.4:	AdaptivePipelining	off

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.

Code Generation Report

Find: Match Case

Contents

- [Summary](#)
- [Clock Summary](#)
- [Code Interface Report](#)
- [Timing And Area Report](#)
- [High-level Resource Report](#)
- [Optimization Report](#)
- [Distributed Pipelining](#)
- [Streaming and Sharing](#)
- [Delay Balancing](#)**
- [Adaptive Pipelining](#)
- [Target Code Generation](#)

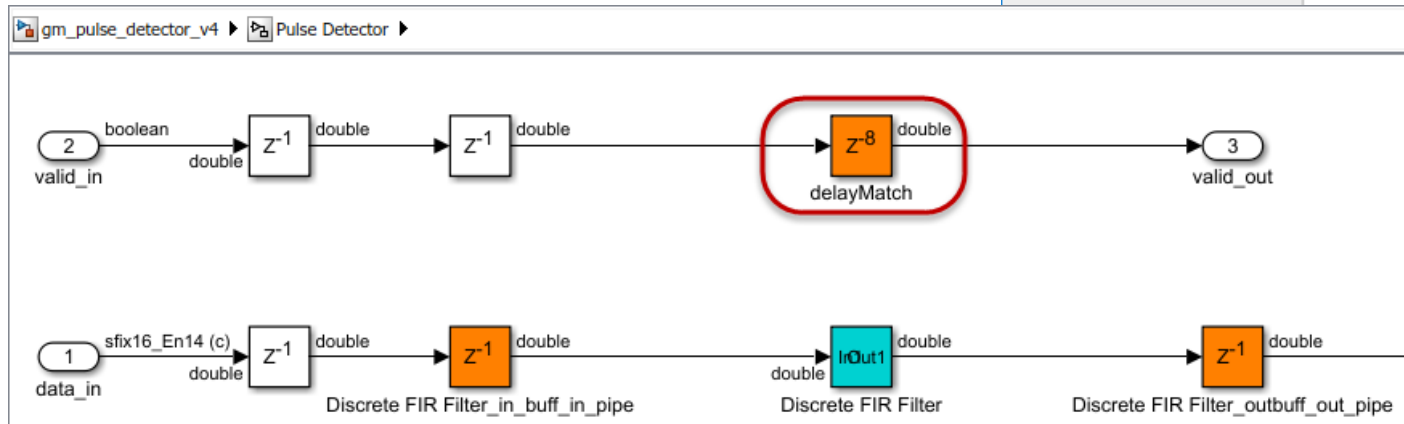
Delay Balancing Report for pulse_detector_v4

Port	Pipeline Latency	Phase Delay
pulse_detector_v4/Pulse Detector/mid_sample	8	0
pulse_detector_v4/Pulse Detector/detected	8	0
pulse_detector_v4/Pulse Detector/valid_out	8	0

Generated Model

Generated model after the transformation: [gm_pulse_detector_v4](#)

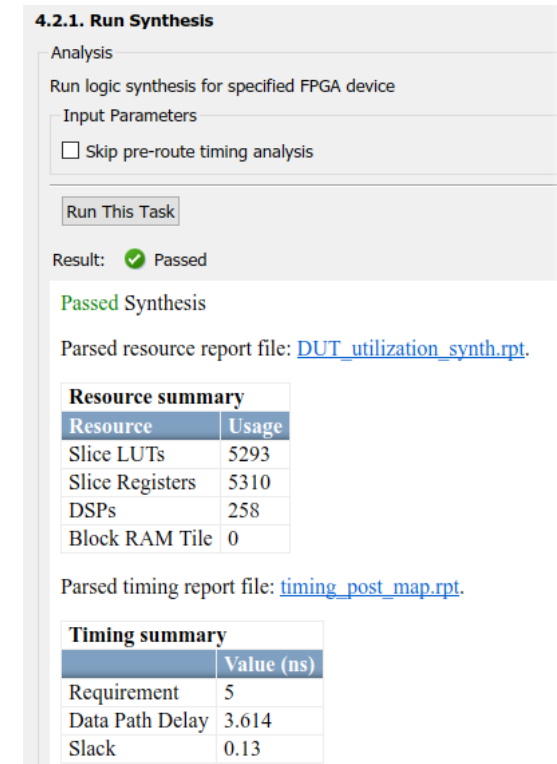
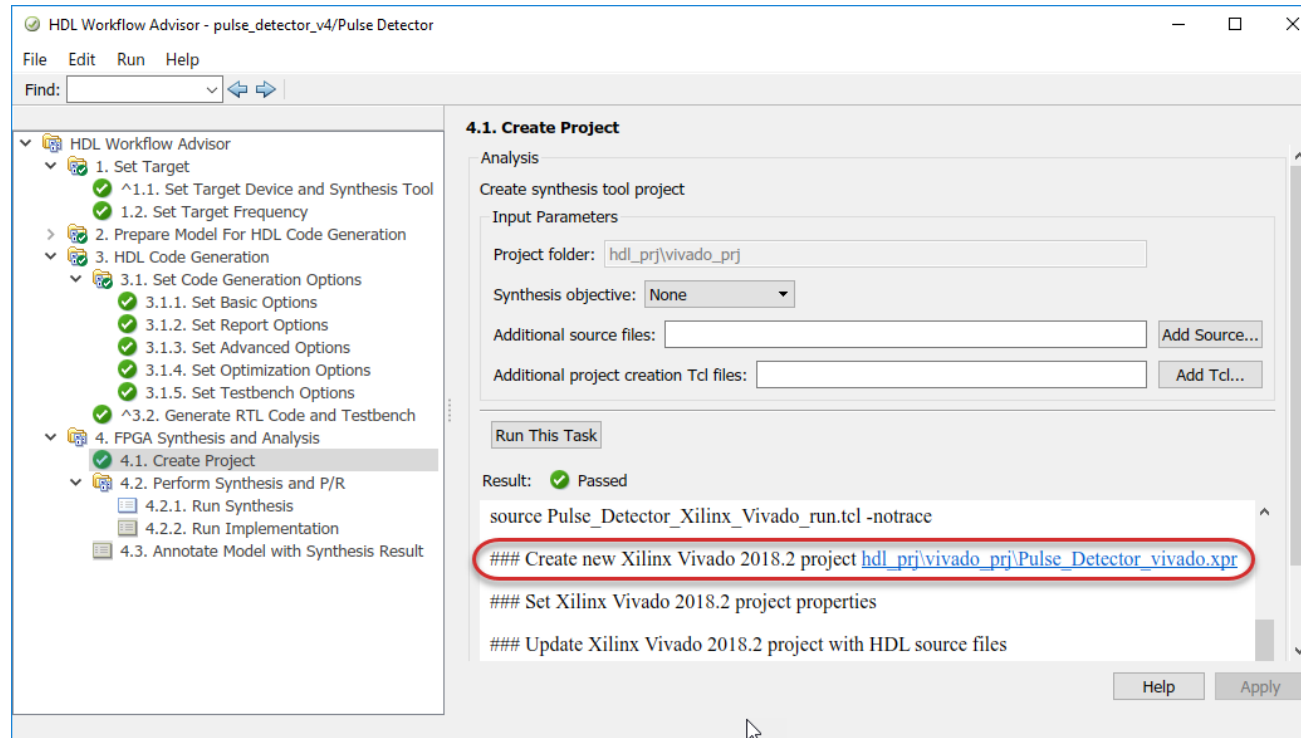
OK Help



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