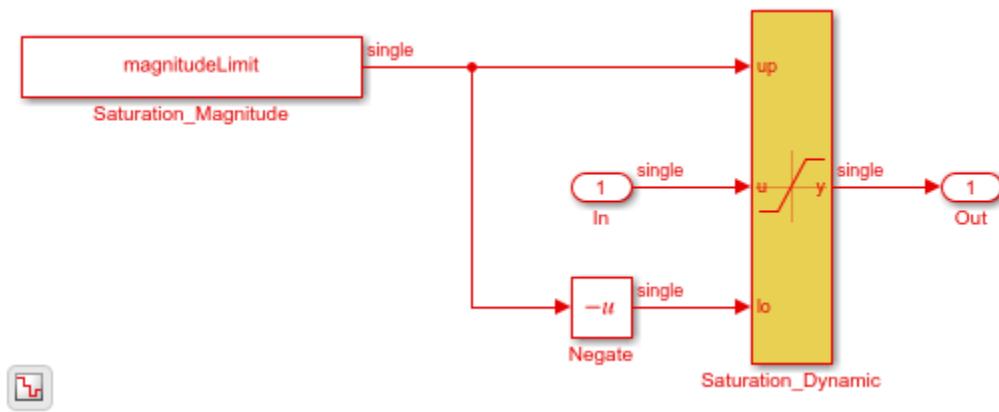


```
### Starting HDL check.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpe
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 1 errors, 0 warnings, and 0 n
```

For the block 'hdlcoderFocCurrentSingleTargetHdl/F0C\_Current\_Control/DQ\_Current\_Control/D\_Current' This block is not supported for Altera Megafunction mapping.

The error message indicates that the Dynamic Saturation block cannot map to floating-point library.

```
hilite_system('hdlcoderFocCurrentSingleTargetHdl/F0C_Current_Control/DQ_Current_Control/D_Current'
```

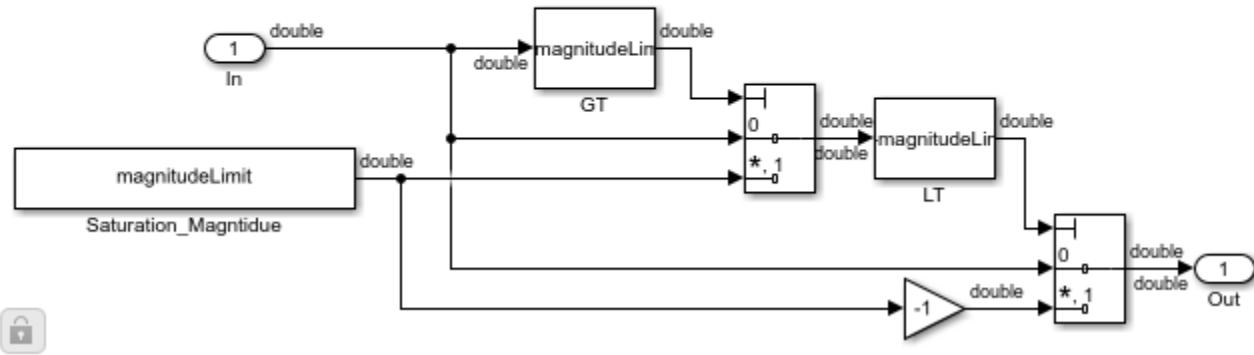


The blocks supported for floating-point library mapping is a subset of all HDL Coder supported blocks. Saturation dynamic is an example of a block that is supported for fixed-point, but not floating-point mapping. In these cases, the block may be described as a sub-graph of the supported subset. For a complete list of blocks and modes that can map to floating-point libraries, check "HDL Coder Support for FPGA Floating-Point Library Mapping" on page 31-47.

In this example, we will replace the Saturate\_Output subsystems that contains Dynamic Saturation blocks with an alternative implementation.

```
open_system('floatFocUtils');
blocksToReplace = {'hdlcoderFocCurrentSingleTargetHdl/F0C_Current_Control/DQ_Current_Control/D_Current';
    'hdlcoderFocCurrentSingleTargetHdl/F0C_Current_Control/DQ_Current_Control/Q_Current_Control/Q_Current';
    };
position1 = get_param(blocksToReplace{1}, 'Position');
delete_block(blocksToReplace{1});
add_block('floatFocUtils/Saturate_Output_Detailed', ...
    blocksToReplace{1}, 'Position', position1);
position2 = get_param(blocksToReplace{2}, 'Position');
delete_block(blocksToReplace{2});
add_block('floatFocUtils/Saturate_Output_Detailed', ...
    blocksToReplace{2}, 'Position', position2);
bdclose('floatFocUtils');
open_system(blocksToReplace{1}, 'force');
```

## Saturate Magnitude Library



### Applying clock rate pipelining to resolve IP latencies

Try to generate code again

```

try
    makehdl('hdlcoderFocCurrentSingleTargetHdl/F0C_Current_Control');
catch me
    disp(me.message);
end

### Generating HDL for 'hdlcoderFocCurrentSingleTargetHdl/F0C_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurrent...
### Starting HDL check.
### Using F:\hub\hub_share\share\apps\HDLTools\Altera\18.1-mw-0\Windows\quartus\bin64..\sopc_bu...
### Generating Altera(R) megafunction: alterafpf_add_single for target frequency of 250 MHz.
### alterafpf_add_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 250 MHz.
### alterafpf_mul_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_sub_single for target frequency of 250 MHz.
### alterafpf_sub_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_neq_single_NEQ for target frequency of 250 MHz.
### alterafpf_neq_single_NEQ takes 0 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_le_single_LE for target frequency of 250 MHz.
### alterafpf_le_single_LE takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_ge_single_GE for target frequency of 250 MHz.
### alterafpf_ge_single_GE takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_gt_single_GT for target frequency of 250 MHz.
### alterafpf_gt_single_GT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_lt_single_LT for target frequency of 250 MHz.
### alterafpf_lt_single_LT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_trig_single_SIN for target frequency of 250 MHz.

```

```

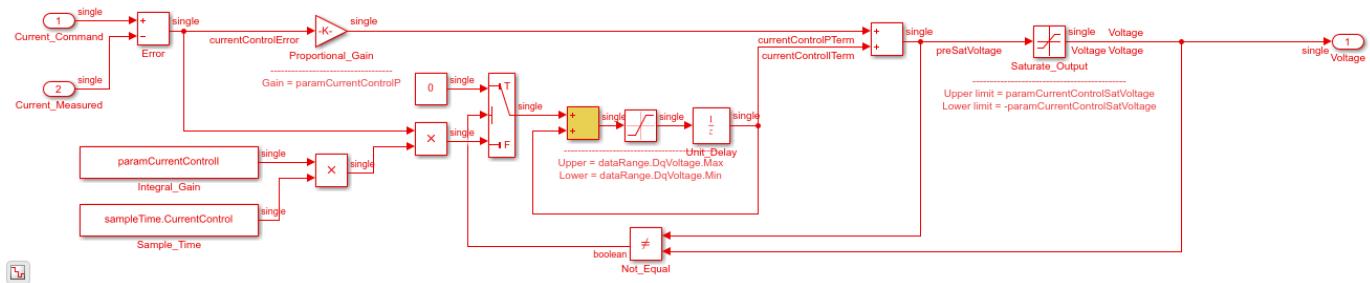
### alterafpf_trig_single_SIN takes 26 cycles.
### Done.
### Generating Altera(R) megafunction: alterafpf_trig_single_COS for target frequency of 250 MHz
### alterafpf_trig_single_COS takes 25 cycles.
### Done.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 5 errors, 0 warnings, and 0 n
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 5 errors, 0 warnings, and 0 n
Target-specific code generation cannot complete for the following reason(s): 'Cannot allocate 1 c

```

These error messages indicate that HDL Coder cannot replace operations in feedback loops with floating-point IP modules, because those loops are modeled with fewer delays than the latency of the equivalent floating-point IP modules to be replaced with. Floating-point IP modules are implemented as pipelined blocks. For some modules, there are minimum latency requirements. As changing the latency of the feedback loop generates an incorrect implementation, HDL Coder prevents the addition of such latency inside feedback loop.

The error indicates that the adder inside the feedback loop requires multiple cycles but the loop has only one delay.

```
hilite_system('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Current
```



There are a few options available in this situation:

- Reduce the target frequency may lower the pipelining depth requirement. But, this may also slow down all other IP modules in the design.
- Configure the IP modules used in the loop with a smaller latency. This also slows down the IP modules' operating frequency, but only for specified IP modules.
- Apply clock rate pipelining. When the data rate is slower than the FPGA clock rate, FPGA has multiple cycles at clock rate to finish operations and still retains the numerical consistency. For more information about clock rate pipelining, see "Clock-Rate Pipelining" on page 24-114.

Let us apply the clock-rate pipelining option to solve the feedback loop problem, since the sample time of 20  $\mu$ s and the FPGA target frequency of 250 MHz (or 4 ns). Thus, we define the Oversampling factor as the ratio of the two values, i.e. 5000, meaning that one unit delay, such as the one shown in the loop, with sample time of 20  $\mu$ s in the original model, is equivalent to 5000 clock rate cycles at sample time of 4 ns on the FPGA. They are sufficient for floating-point IP modules in the loops. Clock rate pipelining is an ideal option for this design.

Set oversampling to 5000.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'Oversampling', 5000);
```

Generate code

```

makehdl('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control');

### Generating HDL for 'hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control')>.
### Starting HDL check.
### Using F:\hub\hub_share\share\apps\HDLTools\Altera\18.1-mw-0\Windows\quartus\bin64..\sopc_bu...
### Generating Altera(R) megafunction: alteraafpf_add_single for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_add_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_mul_single for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_mul_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_gt_single_GT for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_gt_single_GT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_lt_single_LT for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_lt_single_LT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_sub_single for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_sub_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_trig_single_SIN for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_trig_single_SIN takes 26 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_trig_single_COS for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_trig_single_COS takes 25 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_le_single_LE for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_le_single_LE takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_ge_single_GE for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_ge_single_GE takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_neq_single_NEQ for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_neq_single_NEQ takes 0 cycles.
### Done.
### The code generation and optimization options you have chosen have introduced additional pipe...
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit...
### Output port 0: 2 cycles.
### Clock-rate pipelining results can be diagnosed by running this script: <a href="matlab:run('...
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode...
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocCurrentSingleTarg...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentSingleTargetHdl'.
### MESSAGE: The design requires 5000 times faster clock with respect to the base rate = 2e-05.
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Curren...
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Current...

```

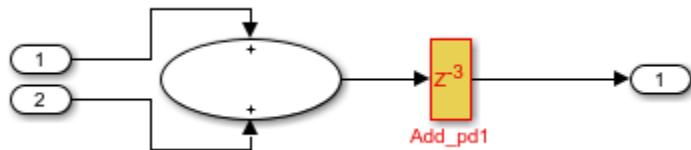
```
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control\Add
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control\Clarke_Transform
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control\Sine_Cosine
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Park_Transform as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control\Park_Transform
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Inverse_Park_Transform as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control\Inverse_Park_Transform
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Inverse_Clark_Transform as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control\Inverse_Clark_Transform
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Space_Vector_Modulation as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control\Space_Vector_Modulation
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control_tc
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control
### Generating package file hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control_pkg.vhd
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpd.html')>
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpd.html
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 0 errors, 0 warnings, and 2 notes
### HDL code generation complete.
```

Now, the entire design is mapped to floating-point IP modules. The target code generation report summarizes the floating IP module usage.

Inspect the generated model for implementation details. For example, subsystem gm\_hdlcoderFocCurrentSingleTargetHdl/FOC\_Current\_Control/DQ\_Current\_Control/D\_Current\_Control/Add corresponding to hdlcoderFocCurrentSingleTargetHdl/FOC\_Current\_Control/DQ\_Current\_Control/D\_Current\_Control/Add in the original model shows that this operation takes 3 cycles.

```
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Add')
get_param('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Add','DelayLength')
```

ans =  
'3'



Since floating-point IP modules introduce latencies across the design, HDL Coder automatically adds necessary matching delays to maintain data synchronization. See “Delay Balancing” on page 24-63 for more details.

### Share Floating-point IPs

Floating-point IP modules are suitable to share, because they are usually identical for the same kind. Floating point IPs are typically expensive operations and it is desirable to share these resources, if possible, to reduce the area footprint. HDL Coder shares resources in the same subsystem. In order to allow more resources to share, we flatten the subsystem hierarchy and set resource sharing factor on the top network.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control', 'FlattenHierarchy', 'on');
hdlset_param('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control', 'SharingFactor', 4);
```

Generate code

```

makehdl('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control');

### Generating HDL for 'hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurrent...
### Starting HDL check.
### Generating Altera(R) megafunction: alteraafpf_trig_single_COS for target frequency of 250 MHz
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_trig_single_COS takes 25 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_trig_single_SIN for target frequency of 250 MHz
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_trig_single_SIN takes 26 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_mul_single for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_mul_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_add_single for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_add_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_sub_single for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_sub_single takes 3 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_gt_single_GT for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_gt_single_GT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_lt_single_LT for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_lt_single_LT takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_neq_single_NEQ for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_neq_single_NEQ takes 0 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_ge_single_GE for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_ge_single_GE takes 1 cycles.
### Done.
### Generating Altera(R) megafunction: alteraafpf_le_single_LE for target frequency of 250 MHz.
### Found an existing generated file in a previous session: (C:\TEMP\Bdoc20b_1527579_10488\ib613...
### alteraafpf_le_single_LE takes 1 cycles.
### Done.
### Using F:\hub\hub_share\share\apps\HDLTools\Altera\18.1-mw-0\Windows\quartus\bin64..\sopc_bu...
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocCurrentSingleTarg...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentSingleTargetHdl'.
### MESSAGE: The design requires 5000 times faster clock with respect to the base rate = 2e-05.
### Working on crp_temp_shared as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared.vhd.
### Working on crp_temp_shared_block as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared...
### Working on crp_temp_shared_block1 as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared...
### Working on crp_temp_shared_block2 as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared...
### Working on crp_temp_shared_block3 as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared...
### Working on crp_temp_shared_block4 as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared...
### Working on crp_temp_shared_block5 as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared...
### Working on crp_temp_shared_block6 as hdsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared...

```

```

### Working on crp_temp_shared_block7 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on crp_temp_shared_block8 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on crp_temp_shared_block9 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on crp_temp_shared_block10 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on crp_temp_shared_block11 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on crp_temp_shared_block12 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on crp_temp_shared_block13 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on crp_temp_shared_block14 as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\crp_temp_shared
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Con
### Working on hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control as hdlsrc\hdlcoderFocCurrent
### Generating package file hdlsrc\hdlcoderFocCurrentSingleTargetHdl\FOC_Current_Control_pkg.vhd
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdlcoderFocCurrentSingleTargetHdl' complete with 0 errors, 0 warnings, and 1 n
### HDL code generation complete.

```

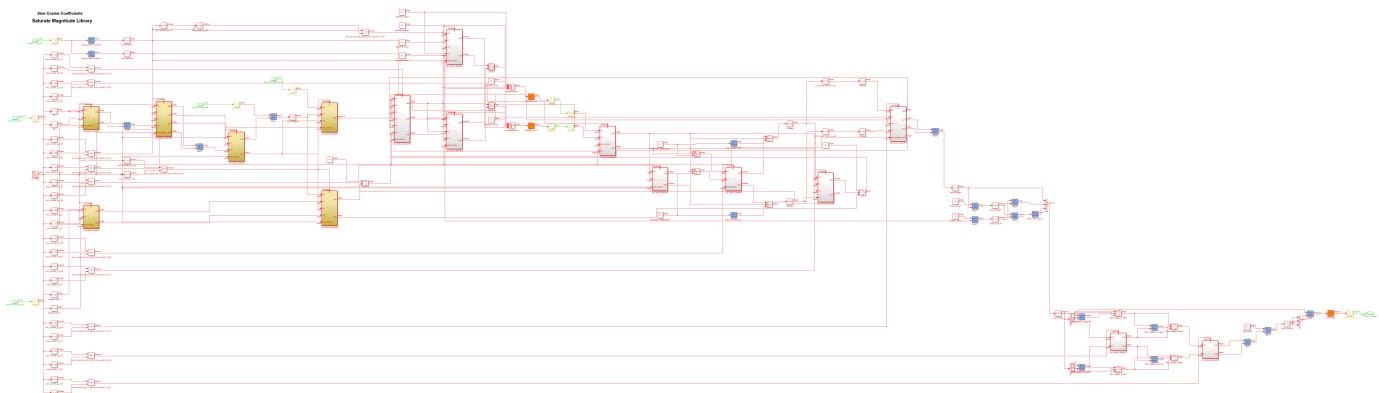
We can confirm that fewer IP modules are inferred from the Floating-point resource report now.

We can also observe the resource sharing results by inspecting the generated model.

```

open_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control');
set_param('gm_hdlcoderFocCurrentSingleTargetHdl', 'SimulationCommand', 'update');
set_param('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control', 'ZoomFactor', 'FitSystem')
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared1');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared2');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared3');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared4');
hilite_system('gm_hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/crp_temp_shared5');

```



## IP Library Configuration

Floating point IP libraries provide some customization options for their IP modules. In this section, we illustrate how to control this configuration in the HDL Coder workflow.

We will use XILINX LOGICORE and a simple model containing one add block for this section.

Create a floating-point target configuration object for XILINX LOGICORE.

```
fc = hdlcoder.createFloatingPointTargetConfig('XILINXLOGICORE');
```

In addition to library name, the configuration object has two other fields for library settings and individual IP module settings, respectively.

```
fc

fc =
    FloatingPointTargetConfig with properties:

        Library: 'XILINXLOGICORE'
        LibrarySettings: [1x1 fpconfig.LatencyDrivenMode]
        IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

LibrarySettings contains library-wide settings. Check the setting for XILINX LOGICORE library.

```
fc.LibrarySettings
```

```
ans =
    LatencyDrivenMode with properties:

        LatencyStrategy: 'MIN'
        Objective: 'SPEED'
```

These are the settings applicable to all IP modules from this library. For example Objective specifies the c\_optimization parameter to XILINX LOGICORE. We can switch it to 'AREA'.

```
fc.LibrarySettings.Objective = 'AREA';
fc.LibrarySettings
```

```
ans =
    LatencyDrivenMode with properties:

        LatencyStrategy: 'MIN'
        Objective: 'AREA'
```

Library settings are library specific. See “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20 for all settings for specific libraries.

IPConfig provides settings, such as Latency and ExtraArgs, for individual IP modules.

As shown in the previous section, latency is a critical property for IP mapping. HDL Coder infers latency based on library settings and target frequency, if applicable. We can also specify latency for individual IP module with the configuration object and HDL Coder uses them for code generation and optimizations.

```
fc.IPCfg.customize('ADDSUB', 'SINGLE', 'Latency', 11);
fc.IPCfg
```

```
ans =
```

Name	DataType	MinLatency	MaxLatency	Latency	ExtraArgs

{'ADDSub' }	{'DOUBLE' }	12	12	-1	{0x0 char}
{'ADDSub' }	{'SINGLE' }	12	12	11	{0x0 char}
{'CONVERT' }	{'DOUBLE_TO_NUMERICTYPE' }	6	6	-1	{0x0 char}
{'CONVERT' }	{'NUMERICTYPE_TO_DOUBLE' }	6	6	-1	{0x0 char}
{'CONVERT' }	{'NUMERICTYPE_TO_SINGLE' }	6	6	-1	{0x0 char}
{'CONVERT' }	{'SINGLE_TO_NUMERICTYPE' }	6	6	-1	{0x0 char}
{'DIV' }	{'DOUBLE' }	57	57	-1	{0x0 char}
{'DIV' }	{'SINGLE' }	28	28	-1	{0x0 char}
{'MUL' }	{'DOUBLE' }	9	9	-1	{0x0 char}
{'MUL' }	{'SINGLE' }	8	8	-1	{0x0 char}
{'RELOP' }	{'DOUBLE' }	2	2	-1	{0x0 char}
{'RELOP' }	{'SINGLE' }	2	2	-1	{0x0 char}
{'SQRT' }	{'DOUBLE' }	57	57	-1	{0x0 char}
{'SQRT' }	{'SINGLE' }	28	28	-1	{0x0 char}

The latency for the ADDSUB IP becomes 11 instead of the default value 12.

Other IP specific settings are specified with ExtraArgs. For example, HDL Coder calls XILINX LOGICORE to generate floating-point IP modules without using any DSP blocks by default. XILINX LOGICORE provides a parameter `c_mult_usage` to control DSP usage. In order to use DSP blocks, we can pass a different setting with ExtraArgs to override the default behavior. Because the ExtraArgs string is appended to the default IP module generation parameters, it must comply with library setting syntax. Check IP library documents for parameter usage and syntax.

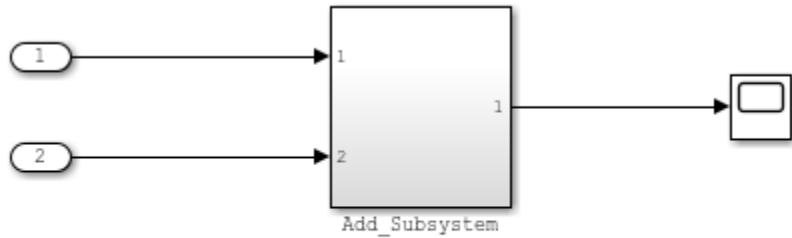
```
fc.IPCfg.customize('ADDSub', 'SINGLE', 'ExtraArgs', 'CSET c_mult_usage=Full_Usage');
fc.IPCfg
```

ans =

Name	DataType	MinLatency	MaxLatency	Latency	
{'ADDSub' }	{'DOUBLE' }	12	12	-1	{0x0 char}
{'ADDSub' }	{'SINGLE' }	12	12	11	{'CSET c_
{'CONVERT' }	{'DOUBLE_TO_NUMERICTYPE' }	6	6	-1	r
{'CONVERT' }	{'NUMERICTYPE_TO_DOUBLE' }	6	6	-1	{0x0 char}
{'CONVERT' }	{'NUMERICTYPE_TO_SINGLE' }	6	6	-1	{0x0 char}
{'CONVERT' }	{'SINGLE_TO_NUMERICTYPE' }	6	6	-1	{0x0 char}
{'DIV' }	{'DOUBLE' }	57	57	-1	{0x0 char}
{'DIV' }	{'SINGLE' }	28	28	-1	{0x0 char}
{'MUL' }	{'DOUBLE' }	9	9	-1	{0x0 char}
{'MUL' }	{'SINGLE' }	8	8	-1	{0x0 char}
{'RELOP' }	{'DOUBLE' }	2	2	-1	{0x0 char}
{'RELOP' }	{'SINGLE' }	2	2	-1	{0x0 char}
{'SQRT' }	{'DOUBLE' }	57	57	-1	{0x0 char}
{'SQRT' }	{'SINGLE' }	28	28	-1	{0x0 char}

Open the model and set the configuration object on it.

```
open_system('hdlcoder_targetIP_configuration');
hdlset_param('hdlcoder_targetIP_configuration', 'FloatingPointTargetConfiguration', fc);
```



Copyright 2016 The MathWorks, Inc.

Run synthesis and mapping to confirm the DSP block usage.

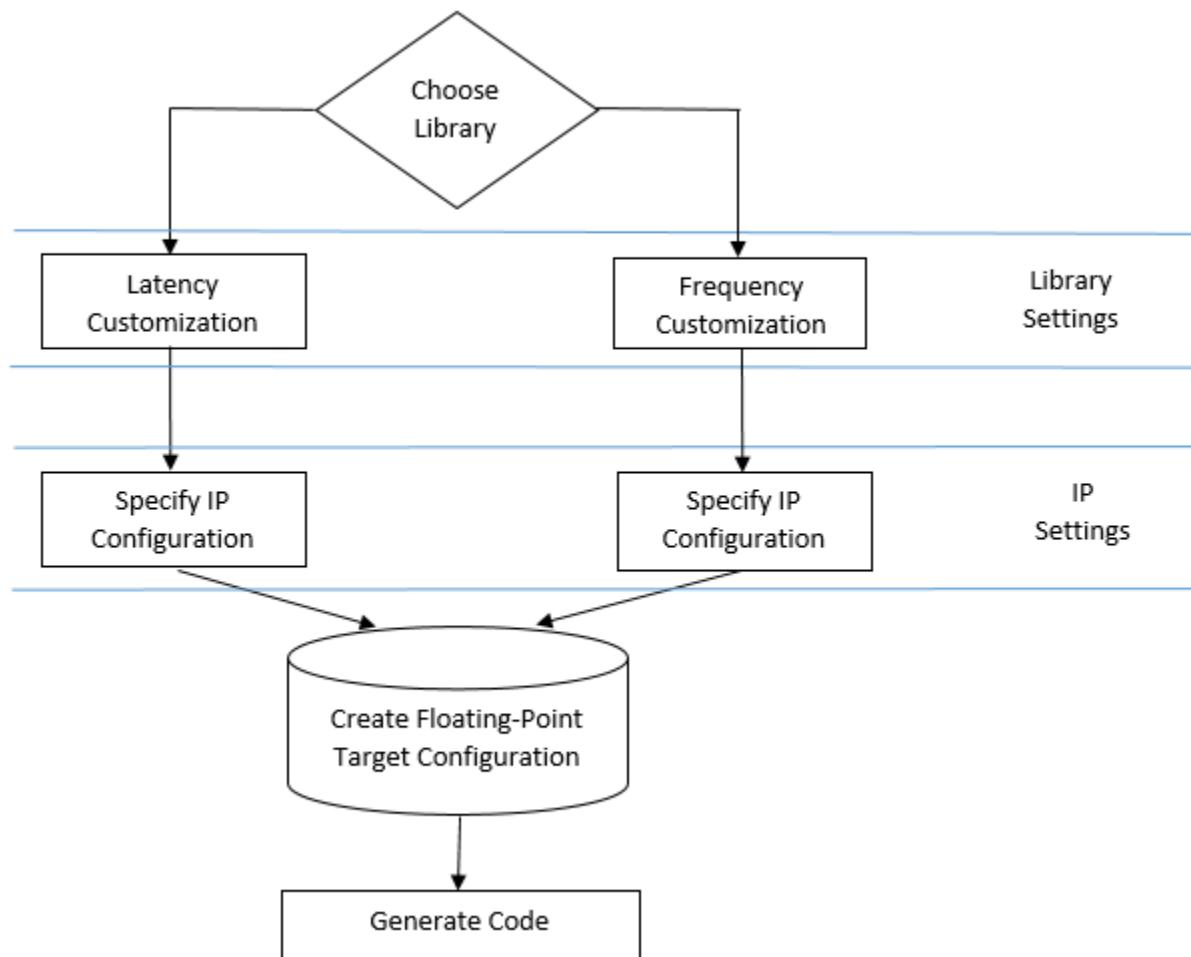
```
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE', ...
    'TargetWorkflow','Generic ASIC/FPGA');
hWC.SkipPreRouteTimingAnalysis = true;
hWC.RunTaskAnnotateModelWithSynthesisResult = false;
hWC.GenerateRTLCode = true;
hWC.validate;
hdlcoder.runWorkflow('hdlcoder_targetIP_configuration/Add_Subsystem', hWC);
```

## Summary

HDL Coder bridges the gap between high level algorithms modeled with floating-point and low level FPGA implementation details. It not only automates the process for fast prototyping, but also allows you to explore high level algorithm design choices efficiently. This example demonstrates the necessary steps to generate synthesizable floating-point HDL code. The command-line APIs used in this example helps to automate your entire code generation process and design space exploration. All the APIs have corresponding GUI settings for ease of usage. For more information about the APIs and GUI options, check “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20.

## Customize Floating-Point IP Configuration

When mapping your Simulink model to floating-point target libraries, you can create a floating-point target configuration with your own custom IP settings. To customize the IP settings, you can use an IP configuration table to choose from different combinations of IP names and data types. The table contains a list of IP types and additional columns that you can use to specify your own custom latency value and other IP settings.



The IP configuration depends on the library settings. The library settings are specific to the floating-point library that you choose. You can customize the IP latency by using the target frequency or the latency strategy setting.

### In this section...

- “Customize the IP Latency with Target Frequency” on page 31-40
- “Customize the IP Latency with Latency Strategy” on page 31-43

## Customize the IP Latency with Target Frequency

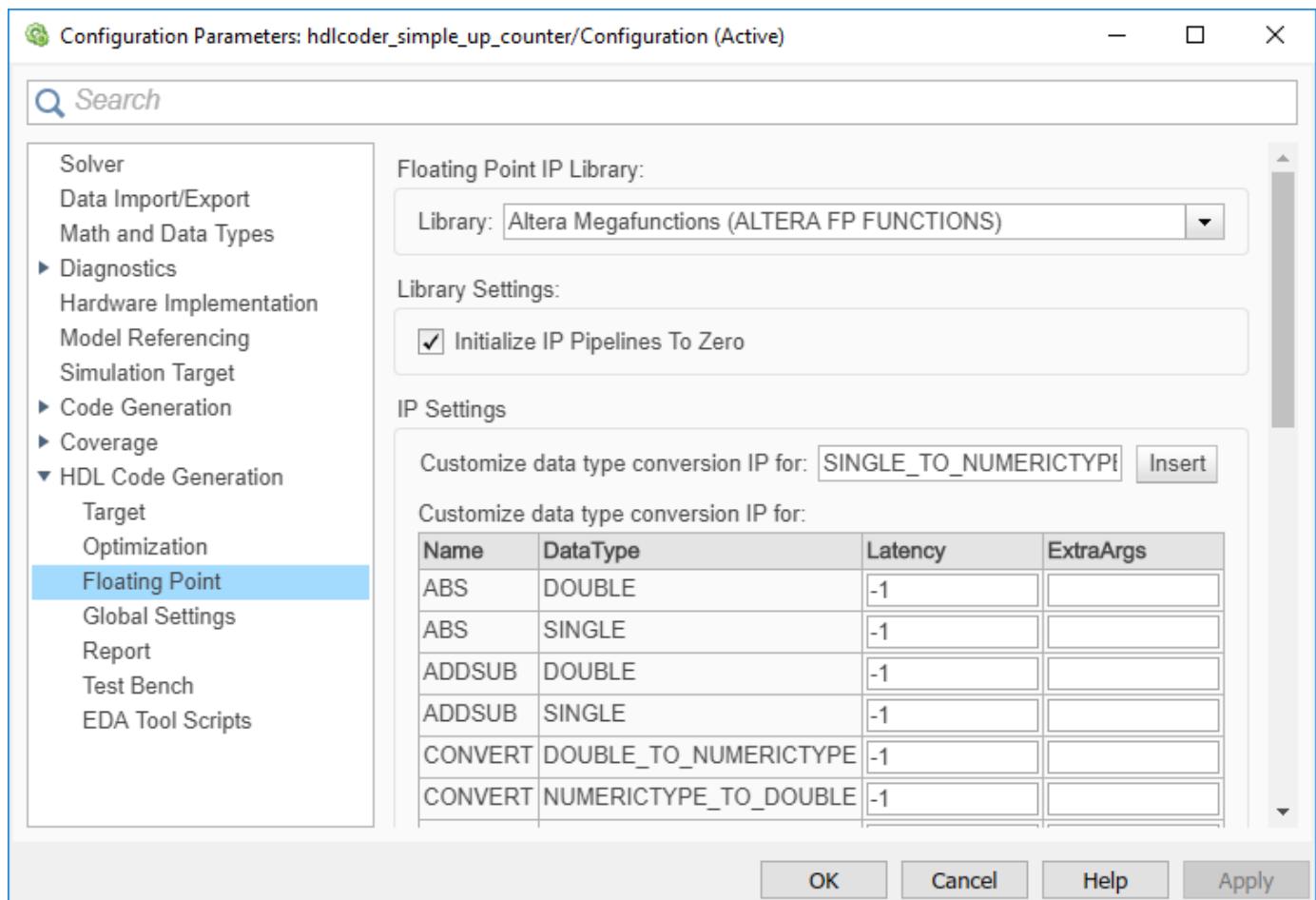
To specify the target frequency that you want the IP to achieve, use the **Altera Megafunctions (ALTERA FP Functions)** library. HDL Coder infers the latency of the IP based on the target frequency value. If you do not specify the target frequency, HDL Coder sets the target frequency to a default value of 200 MHz.

You can customize the IP latency by using the **Target Frequency** setting in the Configuration Parameters dialog box or the **TargetFrequency** property from the command line.

### From the UI

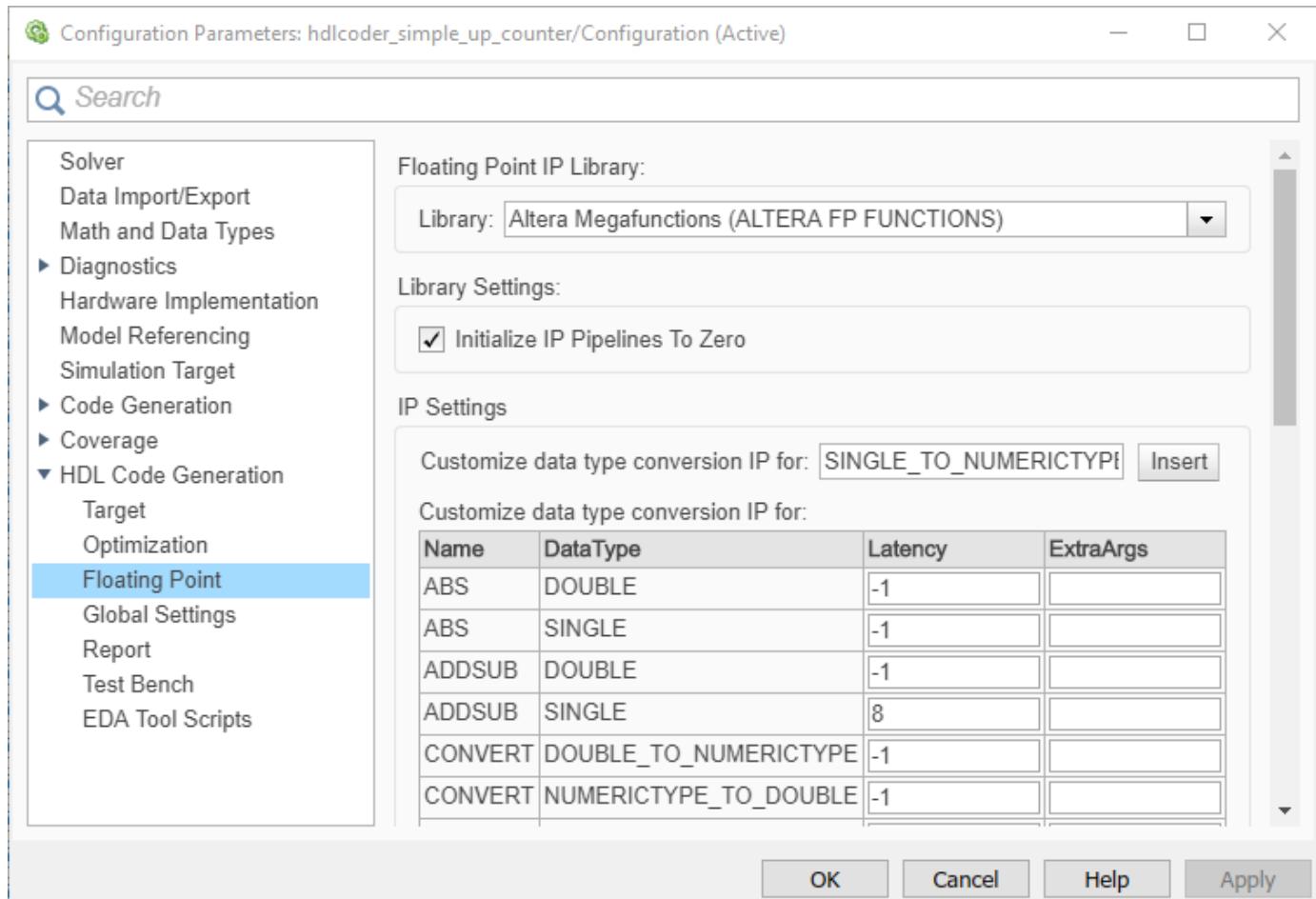
To customize the IP latency by using the target frequency setting:

- 1 Specify the library:
  - a In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Click **Settings**.
  - b On the **HDL Code Generation > Floating Point Target** pane, for **Library**, select **Altera Megafunctions (ALTERA FP Functions)**.



- 2 Specify the target frequency: In the **Target** pane, for **Target Frequency (MHz)**, enter the target frequency that you want the floating-point IP to achieve. If you do not specify a target frequency, HDL Coder sets the target frequency to a default value of 200 MHz.

- 3 Specify the library settings: By using the **Initialize IP Pipelines to Zero** option, you can specify whether to initialize pipeline registers in the IP to zero. To avoid potential numerical mismatches in the HDL simulation, it is recommended to leave the **Initialize IP Pipelines to Zero** option set to **true**.
- 4 Specify the IP settings: In the IP configuration table, you can optionally specify a custom latency and any additional settings specific to the IP.
  - In the **Latency** column of the table, the default latency value of **-1** means that the IP inherits the latency value from the target frequency. If you specify a latency value, HDL Coder tries to map your Simulink model to the IP at a target frequency corresponding to that latency value.
  - In the **ExtraArgs** column of the table, you can specify additional settings specific to the IP.



- 5 Generate code: Click **Apply**. On the Simulink Toolstrip, click **Generate HDL Code**.

## At the Command Line

To customize the IP latency from the command line:

- 1 Specify the library: Create a `hdlcoder.FloatingPointTargetConfig` object for the floating-point library by using the `hdlcoder.createFloatingPointTargetConfig` function. Then, use `hdlset_param` to save the configuration on the model.

For example, for an `sfir_single` model, to create a floating-point target configuration for the Altera Megafunctions (ALTERA FP FUNCTIONS) library with the default settings, enter:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTERA FP FUNCTIONS');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

To see the default settings for the floating-point IP, enter `fpconfig`.

```
fpconfig =
    FloatingPointTargetConfig with properties:
        Library: 'ALTERA FP FUNCTIONS'
        LibrarySettings: [1x1 fpconfig.FrequencyDrivenMode]
        IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

- 2 Specify the target frequency: If you choose ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS) as the library, you can create a floating-point configuration with a custom target frequency. To specify the target frequency for the IP to achieve, use the `TargetFrequency` property. For example:

```
hdlset_param('sfir_single', 'TargetFrequency', 300);
```

- 3 Specify the library settings: Specify whether you want to initialize the pipeline registers in the IP to zero. Use the `InitializeIPPIPelinesToZero` property of the `fpconfig.LibrarySettings` function.

For example, to set the `InitializeIPPIPelinesToZero` property to false, enter:

```
fpconfig.LibrarySettings.InitializeIPPIPelinesToZero = false;
```

To see the library settings that you have applied, enter `fpconfig.LibrarySettings`.

```
ans =
    FrequencyDrivenMode with properties:
        InitializeIPPIPelinesToZero: 0
```

To avoid potential numerical mismatches in the HDL simulation, it is recommended to leave `InitializeIPPIPelinesToZero` set to `true`.

- 4 Specify the IP settings: With the `IPConfig` method, use the `Latency` and `ExtraArgs` to customize the latency of the IP and specify any additional settings specific to the IP.

For example, when mapping to the **ADDSUB** IP with Xilinx LogiCORE libraries, to specify a custom latency of 8:

```
fpconfig.IPConfig.customize('ADDSUB', 'SINGLE', 'Latency', 8);
```

To see the IP settings that you have applied, enter `fpconfig.IPConfig`.

```
ans =
```

Name	DataType	Latency	ExtraArgs
'ABS'	'DOUBLE'	-1	''
'ABS'	'SINGLE'	-1	''
'ADDSUB'	'DOUBLE'	-1	''
'ADDSUB'	'SINGLE'	8	''
'CONVERT'	'DOUBLE_TO_NUMERICTYPE'	-1	''
'CONVERT'	'NUMERICTYPE_TO_DOUBLE'	-1	''
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	-1	''
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	-1	''

- 5 Generate HDL code: To generate code from the subsystem, use `makehdl`.

## Customize the IP Latency with Latency Strategy

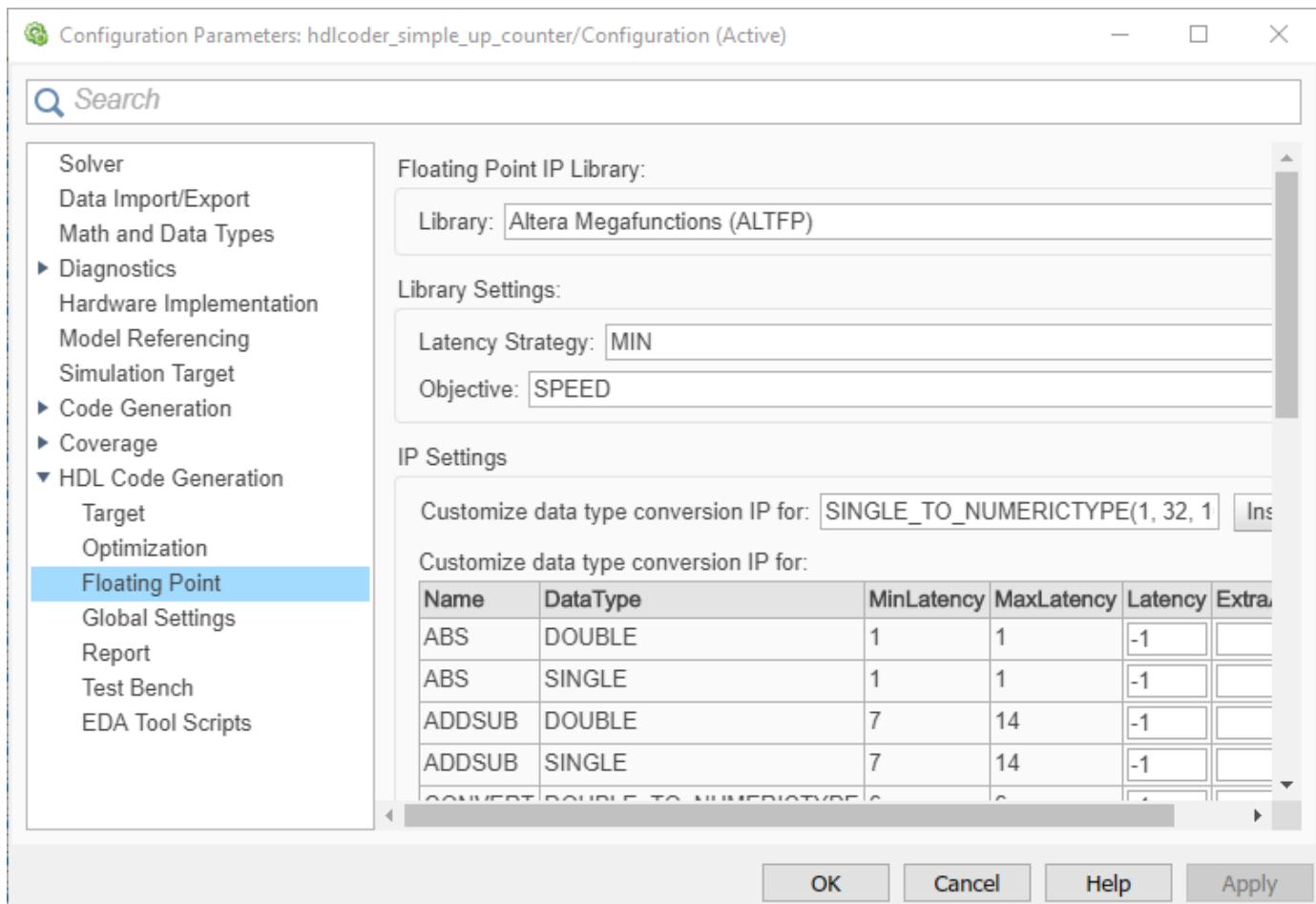
To customize the IP latency with the latency strategy setting, use the **ALTERA MEGAFUNCTION** (ALTFP) or **XILINX LOGICORE** libraries. Specify whether to map your Simulink model to maximum or minimum latency. HDL Coder infers the latency of the IP from the latency strategy setting.

You can customize the IP latency in the Configuration Parameters dialog box or from the command line.

### From the UI

To customize the IP latency with the latency strategy setting:

- 1 Specify the library: In the Configuration Parameters dialog box, on the **HDL Code Generation > Floating Point Target** pane, for **Library**, select **ALTERA MEGAFUNCTION** (ALTFP) or **XILINX LOGICORE**.



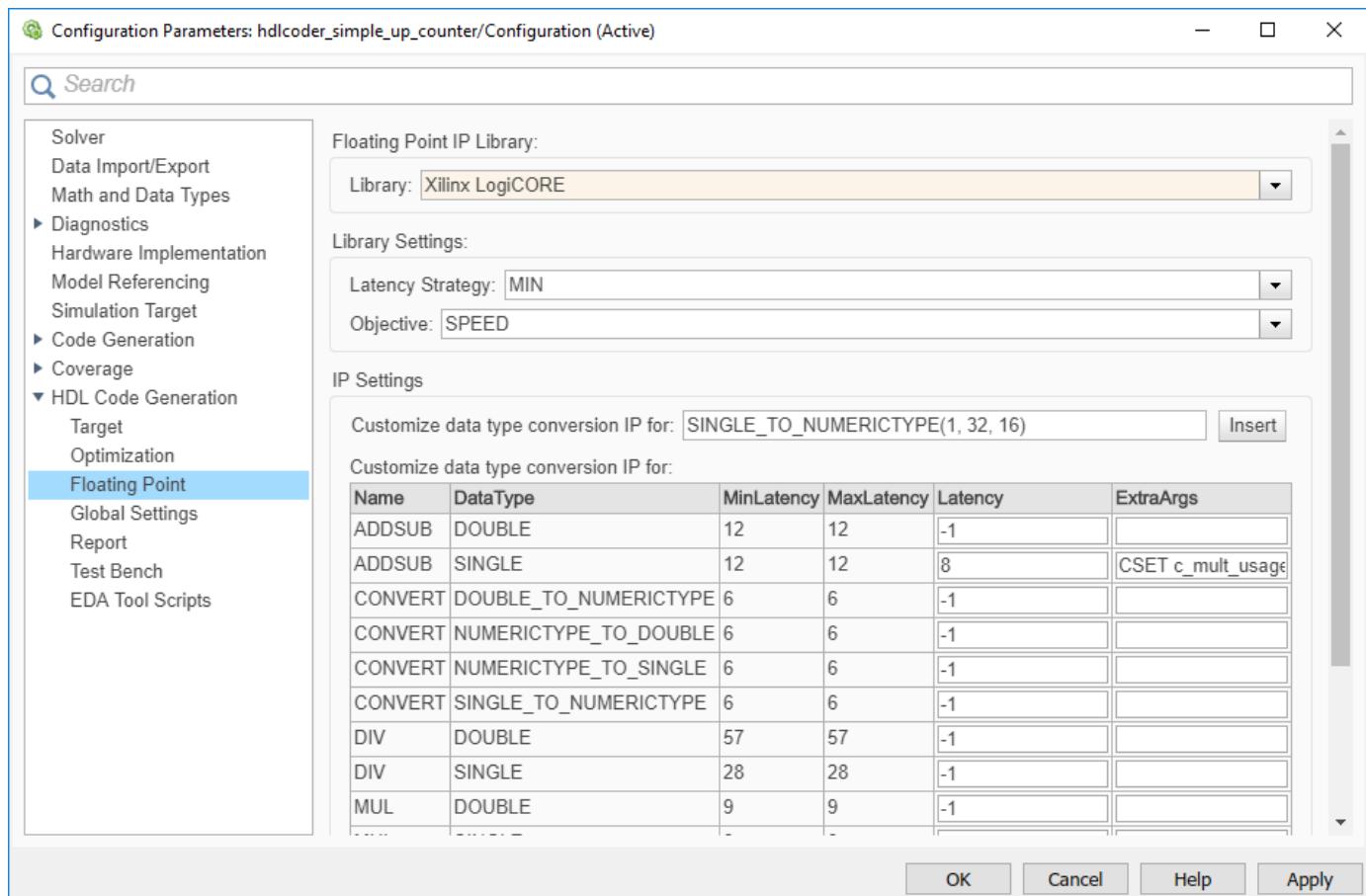
- 2 Specify the library settings: For **Latency Strategy**, specify whether to map your Simulink model to the minimum or maximum latency for the IP. For **Objective**, specify whether to optimize for speed or area.
- 3 Specify the IP settings: An IP configuration table appears that contains the IP types, and their maximum and minimum latencies. In the table, you can optionally specify a custom latency and any additional settings specific to the IP.

- In the **Latency** column of the table, the default latency value of -1 means that the IP inherits the latency value from the library settings. To customize the latency of the IP that your Simulink blocks map to, enter a value for the latency.

For example, when mapping to the **ADDSUB** IP with Xilinx LogiCORE, if you specify a latency of 8, the latency of the IP changes to 8 instead of the default value of 12.

- In the **ExtraArgs** column of the table, specify any additional settings specific to the IP.

For example, when mapping to Xilinx LogiCORE IP, for **ExtraArgs**, you can specify the parameter **c\_mult\_usage** to control the DSP resources that you want to use. To learn more about the parameter usage and syntax, see the IP library documentation.



- 4 Generate code:** Click **Apply**. On the Simulink ToolStrip, click **Generate HDL Code**.

### From the Command Line

To customize the IP latency from the command line:

- Specify the library: Create a `hdlcoder.FloatingPointTargetConfig` object for the floating-point library by using the `hdlcoder.createFloatingPointTargetConfig` function. Then, use `hdlset_param` to save the configuration on the model.

For example, for an `sfir_single` model, to create a floating-point target configuration for the ALTERA MEGAFUNCTION (ALTFP) library with the default settings, enter:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

By default, the library uses the minimum latency and speed objective for the floating-point IP.

- Specify the library settings: Customize the library settings with the `Objective` and `LatencyStrategy` of the `fpconfig.LibrarySettings` function.

For example, to customize the ALTERA MEGAFUNCTION (ALTFP) library to use the maximum latency and objective as area, enter:

```
fpconfig.LibrarySettings.Objective = 'AREA';
fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
```

To see the library settings that you have applied, enter `fpconfig.LibrarySettings`.

```
ans =
    LatencyDrivenMode with properties:
        LatencyStrategy: 'MAX'
        Objective: 'AREA'

    fpconfig is a variable of type hdlcoder.FloatingPointTargetConfig.
```

- 3 Specify the IP settings: With the `IPConfig` method, use the `Latency` and `ExtraArgs` to customize the latency of the IP and specify any additional settings specific to the IP.

For example, when mapping to the **ADDSUB** IP with Xilinx LogiCORE libraries, to use a custom latency of 8 and to specify the DSP resource usage with the `cmultusage` parameter:

```
fpconfig.IPCustomize('ADDSUB', 'SINGLE', 'Latency', 8, 'ExtraArgs', 'CSET c_mult_usage')
```

To see the IP settings that you have applied, enter `fpconfig.IPCustomize`.

Name	DataType	MinLatency	MaxLatency	Latency	
'ADDSUB'	'DOUBLE'	7	14	-1	''
'ADDSUB'	'SINGLE'	7	14	8	'CSET c_mu'
'CONVERT'	'DOUBLE_TO_NUMERICTYPE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_DOUBLE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6	-1	''

- 4 Generate HDL code: To generate code from the subsystem, use `makehdl`.

## See Also

### Related Examples

- “FPGA Floating-Point Library IP Mapping” on page 31-27

### More About

- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-47

# HDL Coder Support for FPGA Floating-Point Library Mapping

## In this section...

["Supported Blocks That Map to FPGA Floating-Point Target IP" on page 31-47](#)

["Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP" on page 31-49](#)

["Limitations for FPGA Floating-Point Library Mapping" on page 31-50](#)

In the HDL Coder block library, a subset of Simulink blocks support floating-point library mapping. The subset includes:

- Blocks that perform basic math operations such as addition, multiplication, and complex trigonometric sine and cosine functions. These blocks map to one or more floating-point IP units on the target FPGA device.
- Discrete blocks, blocks that perform signal routing, and blocks that perform math operations such as matrix concatenation. These blocks need not map to a floating-point IP unit on the target FPGA device.

## Supported Blocks That Map to FPGA Floating-Point Target IP

The following table summarizes the Simulink blocks that can map to FPGA floating-point IP cores.

When mapping to floating-point IP cores, some blocks have mode restrictions.

**Note** Some blocks do not map to a floating-point IP core in the third-party hardware. For example, the Abs block maps to an Altera target IP core but not to a Xilinx target IP core.

Block	Altera Megafunction IP ( <b>ALTFP</b> and <b>ALTERA FP</b> Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Abs	✓	—	—
Add	✓	✓	—
Bias	✓	✓	—
Compare To Constant	✓	✓	—
Compare To Zero	✓	✓	—

Block	Altera Megafunction IP (ALTFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Data Type Conversion	✓	✓	<ul style="list-style-type: none"> <li>Conversions between single and double data types are not supported.</li> <li><b>Integer rounding mode</b> attribute in the Block Parameters dialog box must be set to <b>Nearest</b>.</li> <li>If you use Altera Megafunction IP for conversion between floating-point and fixed-point data types, the input bitwidth must be between 16 and 128 bits.</li> </ul>
Decrement Real World	✓	✓	—
Discrete FIR Filter	✓	✓	—
Discrete Transfer Fcn	✓	✓	—
Discrete-Time Integrator	✓	✓	—
Divide	✓	✓	—
Dot Product	✓	✓	—
Gain	✓	✓	—
Math Function	✓		<ul style="list-style-type: none"> <li>Set the <b>Function</b> attribute in Block Parameters dialog box to either <b>reciprocal</b>, <b>log</b> or <b>exp</b>.</li> </ul>
MinMax	✓	✓	—
Multiply-Add	✓	✓	—
Product	✓	✓	<ul style="list-style-type: none"> <li>Product block with more than two inputs is not supported.</li> </ul>
Product of Elements	✓	✓	<ul style="list-style-type: none"> <li>The <b>Architecture</b> in HDL Block Properties must be set to <b>Tree</b>.</li> </ul>
Reciprocal Sqrt	✓		—
Relational Operator	✓	✓	—
Sqrt	✓	✓	—
Subtract	✓	✓	—
Sum	✓	✓	<ul style="list-style-type: none"> <li>Sum block with - ports is not supported.</li> <li>The block cannot have more than two inputs.</li> </ul>
Sum of Elements	✓	✓	<ul style="list-style-type: none"> <li>The <b>Architecture</b> in HDL Block Properties must be set to <b>Tree</b>.</li> </ul>

Block	Altera Megafunction IP (ALTFFP and ALTERA FP Functions)	Xilinx LogiCORE IP	Remarks and Limitations
Trigonometric Function	✓		<ul style="list-style-type: none"> <li>Only single data types are supported for floating-point library mapping.</li> <li>In the Block Parameters dialog box, <b>Function</b> must be set to either <b>sin</b> or <b>cos</b> and <b>Approximation method</b> must be set to <b>None</b>.</li> <li>If you are using Altera Quartus 10.1 or 11.0, turn on the <b>AlteraBackward Incompatible SinCosPipeline</b> global property using <b>hdlset_param</b>.</li> </ul>
Unary Minus	✓	✓	—

## Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP

Following are the Simulink blocks that generate HDL code but need not map to an FPGA floating-point IP core.

- Bus Assignment
- Bus Creator
- Bus Selector
- Constant
- Delay
- Demux
- Deserializer1D
- DownSample
- From
- Goto
- Index Vector
- Vector Concatenate, Matrix Concatenate
- Memory
- Model Info
- Multiport Switch
- Mux
- Rate Transition
- Reshape

- Serializer1D
- Subsystem
- Switch block with control input other than  $u2 \approx 0$ .
- Unit Delay
- Upsample
- Zero-Order Hold

## Limitations for FPGA Floating-Point Library Mapping

- If your synthesis tool is Xilinx Vivado, you cannot use FPGA floating-point library mapping.
- Complex data types are not supported.
- The streaming optimization is not supported with floating-point library mapping.
- The resource sharing optimization is not supported with Unary Minus and Abs blocks.
- For IP Core Generation, FPGA Turnkey, and Simulink Real-Time FPGA I/O workflows, your DUT ports cannot use floating-point data types.

## See Also

### Related Examples

- “FPGA Floating-Point Library IP Mapping” on page 31-27

### More About

- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20
- “Customize Floating-Point IP Configuration” on page 31-39

# Synthesis Objective to Tcl Command Mapping

In this section...
"Altera Quartus II" on page 31-51
"Xilinx Vivado 2014.4" on page 31-51
"Xilinx ISE 14.7 with PlanAhead" on page 31-52

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

When you specify a synthesis objective in the HDL Workflow Advisor **Synthesis objective** field, or in the HDL Workflow CLI workflow `hdlcoder.Objective`, the HDL Coder software generates Tcl commands that are specific to your synthesis tool.

## Altera Quartus II

Synthesis objective	Tcl Commands
Area Optimized	<code>set_global_assignment -name OPTIMIZATION_TECHNIQUE "Area"</code> <code>set_global_assignment -name FITTER EFFORT "Standard Fit"</code>
Compile Optimized	<code>set_global_assignment -name OPTIMIZATION_TECHNIQUE "Balanced"</code> <code>set_global_assignment -name FITTER EFFORT "Fast Fit"</code>
Speed Optimized	<code>set_global_assignment -name OPTIMIZATION_TECHNIQUE "Speed"</code> <code>set_global_assignment -name FITTER EFFORT "Standard Fit"</code>

## Xilinx Vivado 2014.4

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	<code>set_property strategy {Vivado Synthesis Defaults}</code> <code>[get_runs synth_1]</code> <code>set_property strategy "Area_Explore" [get_runs impl_1]</code>

Synthesis objective	Tcl Commands
Compile Optimized	<pre>set_property strategy "Flow_RuntimeOptimized" [get_runs synth1] set_property strategy "Flow_Quick" [get_runs impl_1]</pre>
Speed Optimized	<pre>set_property strategy {Vivado Synthesis Defaults} [get_runs synth_1] set_property strategy "Performance_Explore" [get_runs impl_1]</pre>

## Xilinx ISE 14.7 with PlanAhead

If your tool version is different, the Tcl commands are slightly different.

Synthesis objective	Tcl Commands
Area Optimized	<pre>set_property strategy "AreaReduction" [get_runs synth_1] set_property strategy "MapCoverArea" [get_runs impl_1]</pre>
Compile Optimized	<pre>set_property strategy "{XST Defaults}" [get_runs synth_1] set_property strategy "{ISE Defaults}" [get_runs impl_1]</pre>
Speed Optimized	<pre>set_property strategy "TimingWithIOBPacking" [get_runs synth_1] set_property strategy "MapTiming" [get_runs impl_1]</pre>

## See Also

### Related Examples

- “Getting Started with the HDL Workflow Advisor” on page 31-6
- “HDL Workflow Advisor Tasks” on page 37-2

# Run HDL Workflow with a Script

## In this section...

- “Export an HDL Workflow Script” on page 31-54
- “Specify Verbosity of Workflow Script” on page 31-54
- “Enable or Disable Tasks in HDL Workflow Script” on page 31-54
- “Run a Single Workflow Task” on page 31-54
- “Import an HDL Workflow Script” on page 31-55
- “Generic ASIC/FPGA Workflow Script Example” on page 31-55
- “FPGA-in-the-Loop Script Example” on page 31-56
- “FPGA Turnkey Workflow Script Example” on page 31-58
- “IP Core Generation Workflow Script Example” on page 31-60
- “Simulink Real-Time FPGA I/O Workflow Example” on page 31-62

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

To run the HDL workflow as a command-line script, configure and run the HDL Workflow Advisor with your Simulink design, then export a script. The script uses HDL Workflow CLI commands to perform the same tasks as the HDL Workflow Advisor, including FPGA bitstream or synthesis project generation.

You can export an HDL workflow script for these target workflows:

- Generic ASIC/FPGA
- FPGA-in-the-Loop (requires HDL Verifier license)
- FPGA Turnkey
- IP Core Generation
- Simulink Real-Time FPGA I/O (requires Simulink Real-Time)

To update an existing script, import it into the HDL Workflow Advisor, modify the tasks, and export the updated script. Alternatively, you can manually edit the script.

## Export an HDL Workflow Script

- 1 In the HDL Workflow Advisor, configure and run all the tasks.
- 2 Select **File > Export to Script**.
- 3 In the Export Workflow Configuration dialog box, enter a file name and save the script.

The script is a MATLAB file that you can run from the command line.

**Note** When you export to script, default values such as **Asynchronous** value for **Reset type** are not exported. When you import from the script, if the model is unchanged, you do not see the default settings in the script.

---

## Specify Verbosity of Workflow Script

You can use the **Verbosity** property of the `hdlcoder.runWorkflow` function to specify the level of detail for progress messages generated as code generation and deployment proceeds. To generate verbose messages while running the workflow for a `hdlcoder.WorkflowConfig` workflow configuration object, `hWC` and Simulink design, `model/DUTname`, set **Verbosity** to `on`.

```
hdlcoder.runWorkflow('model/DUTname', hWC, 'Verbosity', 'on');
```

## Enable or Disable Tasks in HDL Workflow Script

To disable all workflow tasks, update the workflow configuration object with the `clearAllTasks` method.

To reenable all workflow tasks, update the workflow configuration object with the `setAllTasks` method.

## Run a Single Workflow Task

To run a single workflow task without rerunning other workflow tasks:

- 1 Disable all tasks in the workflow configuration object by running the `clearAllTasks` method.
- 2 In the workflow configuration object, enable the task that you want to run.

For example, if you previously ran an HDL workflow script and generated a bitstream, you can program your target hardware without rerunning the other workflow tasks. To run the target device programming task for an `hdlcoder.WorkflowConfig` workflow configuration object, `hWC` and Simulink design, `model/DUTname`:

- 1 Run the `clearAllTasks` method.  
`hWC.clearAllTasks;`
- 2 Enable the target device programming task.  
`hWC.RunTaskProgramTargetDevice = true;`
- 3 Run the workflow.  
`hdlcoder.runWorkflow('model/DUTname', hWC);`

## Import an HDL Workflow Script

- 1 In the HDL Workflow Advisor, select **File > Import from Script**.
- 2 In the Import Workflow Configuration dialog box, select the script file and click **Open**.

The HDL Workflow Advisor updates the tasks with the imported script settings.

---

**Note** When you import a HDL Workflow Advisor script, make sure you use the same script that was exported from the HDL Workflow Advisor UI.

---

## Generic ASIC/FPGA Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex 7 device. It uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 14:42:37 on 29/03/2018
% This script was generated using the following parameter values:
%   Filename : 'S:\generic_workflow_example.m'
%   Overwrite : true
%   Comments : true
%   Headers : true
%   DUT      : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-----

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('sfir_fixed', 'GenerateHDLTestBench', 'off');
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7vx485t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg1761');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Generic ASIC/FPGA');

% Specify the top level project directory
```

```
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskRunSynthesis = true;
hWC.RunTaskRunImplementation = false;
hWC.RunTaskAnnotateModelWithSynthesisResult = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskRunSynthesis' Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set properties related to 'RunTaskRunImplementation' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set properties related to 'RunTaskAnnotateModelWithSynthesisResult' Task
hWC.CriticalPathSource = 'pre-route';
hWC.CriticalPathNumber = 1;
hWC.ShowAllPaths = false;
hWC.ShowDelayData = true;
hWC.ShowUniquePaths = false;
hWC.ShowEndsOnly = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `generic_workflow_example.m`, at the command line, enter:

```
generic_workflow_example.m
```

## FPGA-in-the-Loop Script Example

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an FPGA-in-the-Loop workflow script that targets a Xilinx Virtex 5 development board and uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
%-----
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 15:11:23 on 04/05/2018
% This script was generated using the following parameter values:
%     Filename : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\hdlworkflow'
%     Overwrite : true
%     Comments : true
%     Headers : true
%     DUT       : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-----

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestorereparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 25);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Xilinx Kintex-7 KC705 development board');
hdlset_param('sfir_fixed', 'Workflow', 'FPGA-in-the-Loop');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','FPGA-in-the-Loop');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = false;
hWC.RunTaskBuildFPGAInTheLoop = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskBuildFPGAInTheLoop' Task
hWC.IPAddress = '192.168.0.2';
hWC.MACAddress = '00-0A-35-02-21-8A';
```

```
hWC.SourceFiles = '';
hWC.Connection = 'Ethernet';
hWC.RunExternalBuild = true;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `FIL_workflow_example.m`, at the command line, enter:

```
fil_workflow_example.m
```

## FPGA Turnkey Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is an FPGA Turnkey workflow script that targets a Xilinx Virtex 5 development board. It uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:24:32 on 08/07/2015
% Parameter Values:
%   Filename : 'S:\turnkey_workflow_example.m'
%   Overwrite : true
%   Comments : true
%   Headers : true
%   DUT       : 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA'

%% Load the Model
load_system('hdlcoderUARTServoControllerExample');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'HDLSubsystem', 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisTool', 'Xilinx ISE');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolChipFamily', 'Virtex5');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolDeviceName', 'xc5vsx50t');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolPackageName', 'ff1136');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoderUARTServoControllerExample', ...
```

```

'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetPlatform', 'Xilinx Virtex-5 ML506 development board');
hdlset_param('hdlcoderUARTServoControllerExample', 'Workflow', 'FPGA Turnkey');

% Set Import HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterface', 'RS-232 Serial Port Rx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterfaceMapping', '[0]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterface', 'RS-232 Serial Port Tx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterfaceMapping', '[0]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterface', 'LEDs General Purpose [0:7]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterfaceMapping', '[0:3]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterfaceMapping', '[0]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterfaceMapping', '[1]');

% Set Outport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterfaceMapping', '[2]');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE', ...
    'TargetWorkflow','FPGA Turnkey');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskPerformLogicSynthesis = true;
hWC.RunTaskPerformMapping = true;
hWC.RunTaskPerformPlaceAndRoute = true;
hWC.RunTaskGenerateProgrammingFile = true;

```

```
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set Properties related to Perform Mapping Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set Properties related to Perform Place and Route Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `turnkey_workflow_example.m`, at the command line, enter:  
`turnkey_workflow_example.m`

## IP Core Generation Workflow Script Example

This example shows how to configure and run an exported HDL workflow script.

This script is an IP core generation workflow script that targets the Altera Cyclone V SoC development kit. It uses the Altera Quartus II synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:42:16 on 08/07/2015
% Parameter Values:
%   Filename : 'S:\ip_core_gen_workflow_example.m'
%   Overwrite : true
%   Comments : true
%   Headers : true
%   DUT       : 'hdlcoder_led_blinking/led_counter'

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoder_led_blinking', ...
    'HDLSubsystem', 'hdlcoder_led_blinking/led_counter');
hdlset_param('hdlcoder_led_blinking', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_led_blinking', ...
    'ReferenceDesign', 'Default system (Qsys 14.0)');
hdlset_param('hdlcoder_led_blinking', 'ResetType', 'Synchronous');
```

```

hdlset_param('hdlcoder_led_blinking', 'ResourceReport', 'on');
hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Altera QUARTUS II');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolChipFamily', 'Cyclone V');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolDeviceName', '5CSXFC6D6F31C6');
hdlset_param('hdlcoder_led_blinking', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_led_blinking', ...
    'TargetPlatform', 'Altera Cyclone V SoC development kit - Rev.D');
hdlset_param('hdlcoder_led_blinking', 'Traceability', 'on');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'IP Core Generation');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter', ...
    'ProcessorFPGASynchronization', 'Free running');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceMapping', 'x"100"');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceOptions', {'RegisterInitialValue', 5});

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterfaceMapping', 'x"104"');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterfaceOptions', {'RegisterInitialValue', 1});

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'External Port');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', 'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', ...
    'IOInterfaceMapping', 'x"108"');
hdlset_param('hdlcoder_led_blinking/led_counter/Read back', ...
    'IOInterfaceOptions', {'RegisterInitialValue', 3});

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool', 'Altera QUARTUS II', ...
    'TargetWorkflow', 'IP Core Generation');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskGenerateSoftwareInterface = false;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Generate RTL Code And IP Core Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;

```

```
% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.AreaOptimized;

% Set Properties related to Generate Software Interface Model Task
hWC.OperatingSystem = '';
hWC.AddLinuxDeviceDriver = false;

% Set Properties related to Build FPGA Bitstream Task
hWC.RunExternalBuild = true;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `ip_core_workflow_example.m`, at the command line, enter:

```
ip_core_gen_workflow_example.m
```

## Simulink Real-Time FPGA I/O Workflow Example

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a Simulink Real-Time FPGA I/O workflow script that targets the Speedgoat I0333-325K board that uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
%-----
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 18:14:33 on 08/05/2018
% This script was generated using the following parameter values:
%   Filename    : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\hdlworkflow'
%   Overwrite   : true
%   Comments    : true
%   Headers     : true
%   DUT         : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-----

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
```

```
%hdlrestorereparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 100);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Speedgoat I0333-325K');
hdlset_param('sfir_fixed', 'Workflow', 'Simulink Real-Time FPGA I/O');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink Real-Ti

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';
hWC.ReferenceDesignToolVersion = '2017.4';
hWC.IgnoreToolVersionMismatch = false;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndIPCore' Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;
hWC.GenerateIPCoreTestbench = false;
hWC.CustomIPTopHDLFile = '';
hWC.AXI4RegisterReadback = false;
hWC.IPDataCaptureBufferSize = '128';

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';
hWC.EnableIPCaching = true;

% Set properties related to 'RunTaskBuildFPGABitstream' Task
hWC.RunExternalBuild = false;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;
hWC.CustomBuildTclFile = '';
hWC.ReportTimingFailure = hdlcoder.ReportTiming.Error;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `slrt_workflow_example.m`, at the command line, enter:

```
slrt_workflow_example.m
```

## See Also

### Functions

`clearAllTasks` | `hdlcoder.runWorkflow` | `setAllTasks`

### Classes

`hdlcoder.WorkflowConfig`

## Related Examples

- “Getting Started with the HDL Workflow Advisor” on page 31-6
- “HDL Workflow Advisor Tasks” on page 37-2

# Getting Started with the HDL Workflow Command-Line Interface

This example shows how to use the HDL Workflow Advisor to run HDL workflows from the command-line and the 'Export to script' functionality.

## Introduction

This example is a step-by-step guide that helps introduce you to the HDL Workflow Command Line Interface.

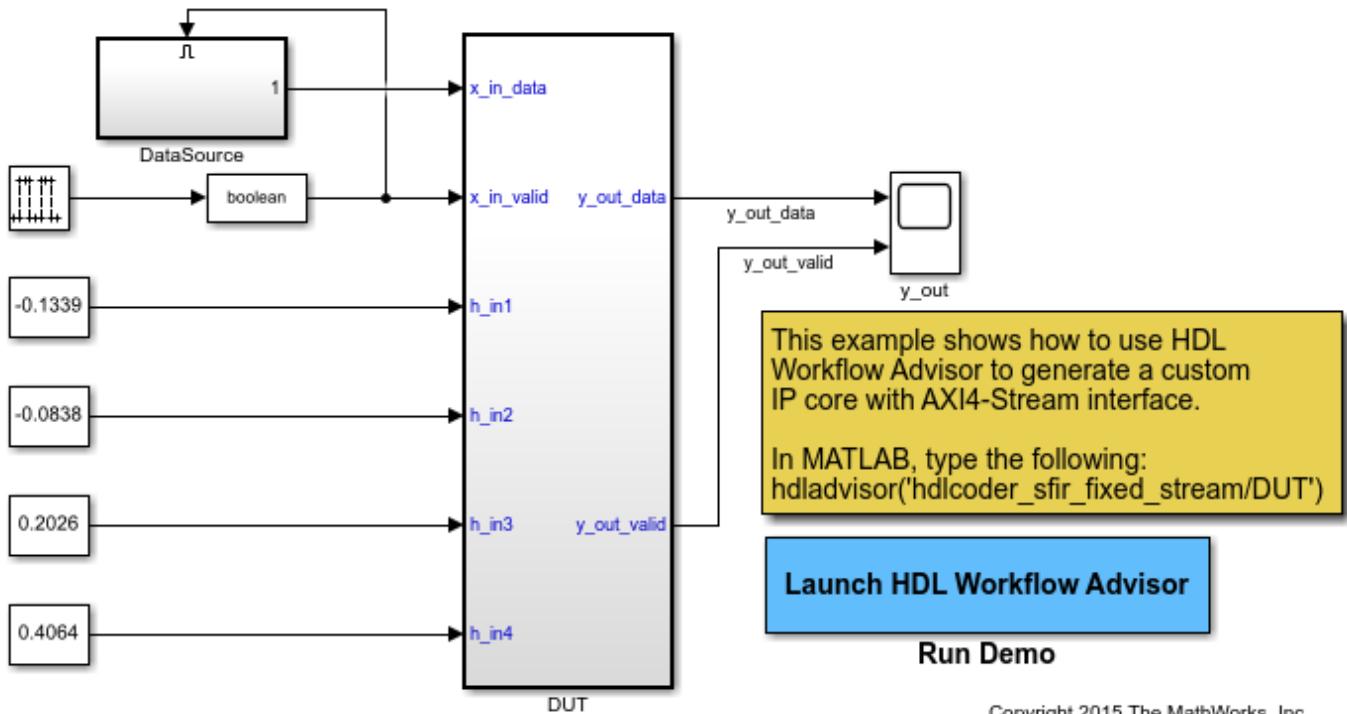
Using the HDL Workflow Command Line Interface, you can run the same sequence of steps and control the same configuration settings that are available in the HDL Workflow Advisor for the following workflows:

- 1 Generic ASIC/FPGA
- 2 FPGA Turnkey
- 3 IP Core Generation
- 4 Simulink Real-Time FPGA I/O

## Open the Model

In this example we will use the **hdlcoder\_sfir\_fixed\_stream** model, but the HDL Workflow Command Line Interface can be used with any model that works with the workflows listed above.

```
open_system('hdlcoder_sfir_fixed_stream')
```

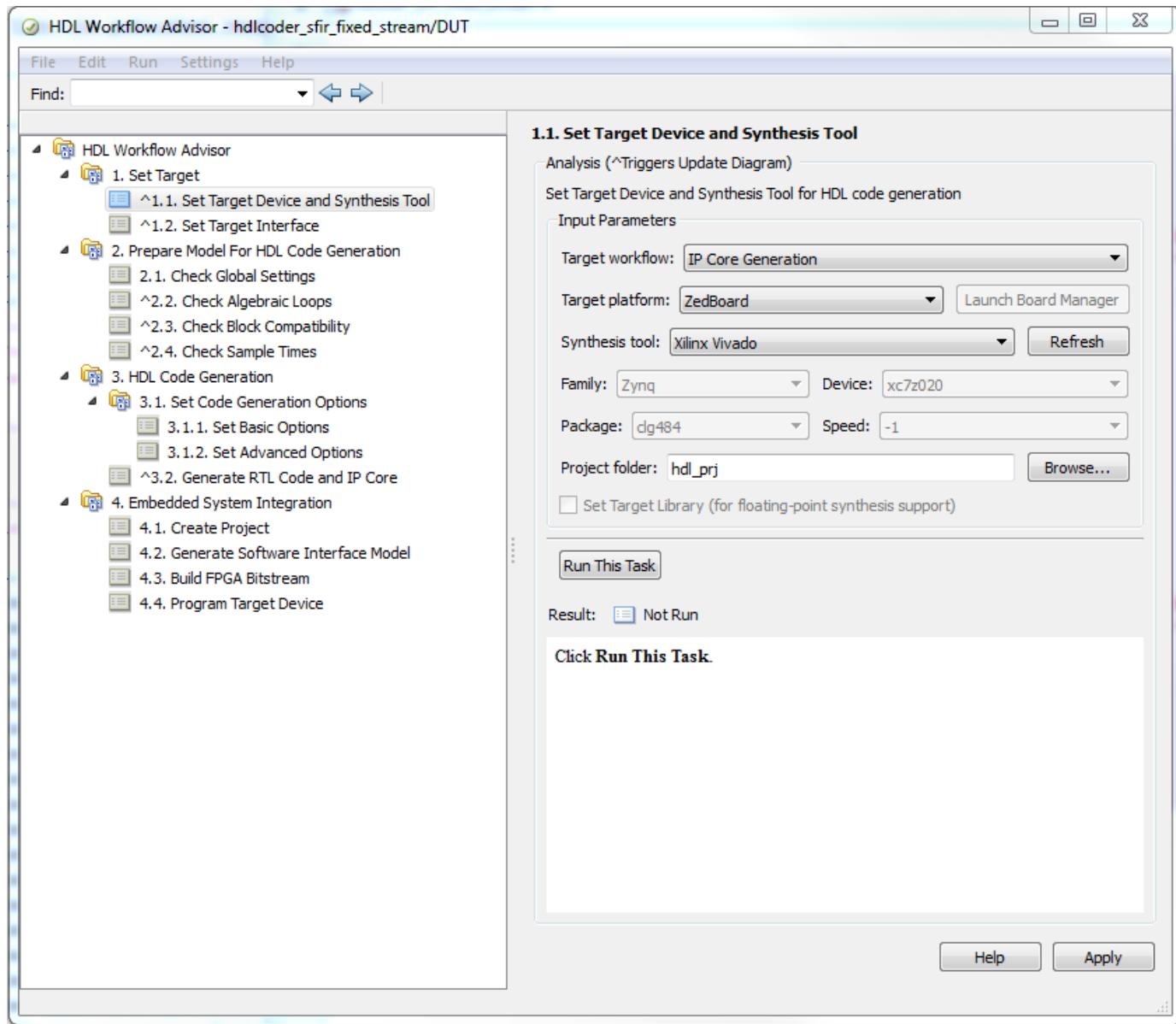


Copyright 2015 The MathWorks, Inc.

Make sure that the desired third party tools are included on the path. For example, to include Vivado installed locally in its default windows location, use the following command:

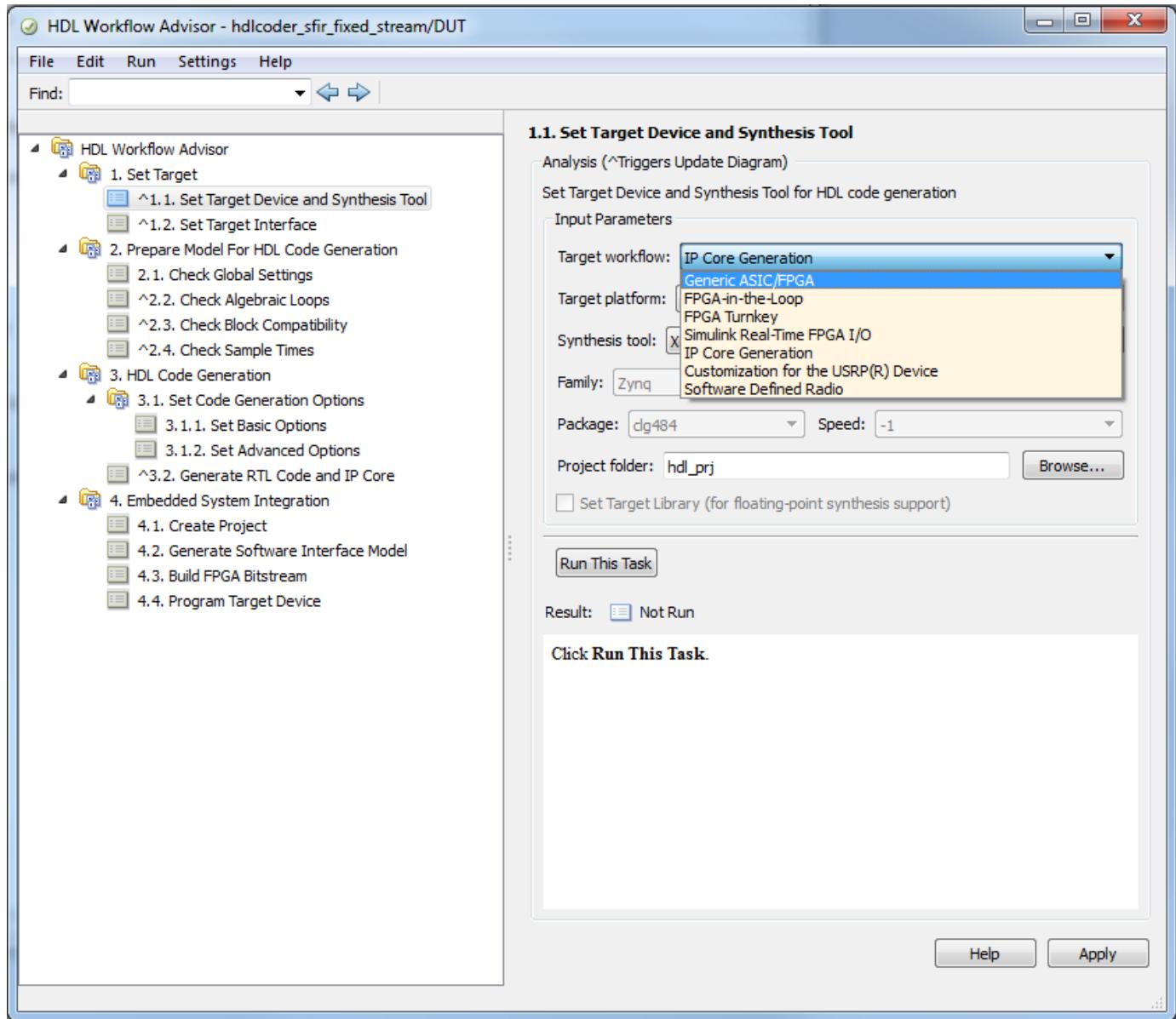
```
>> hdlsetupoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2017.4\bin\vivado.hip')
```

Next, launch the workflow advisor and select the appropriate subsystem as the DUT.

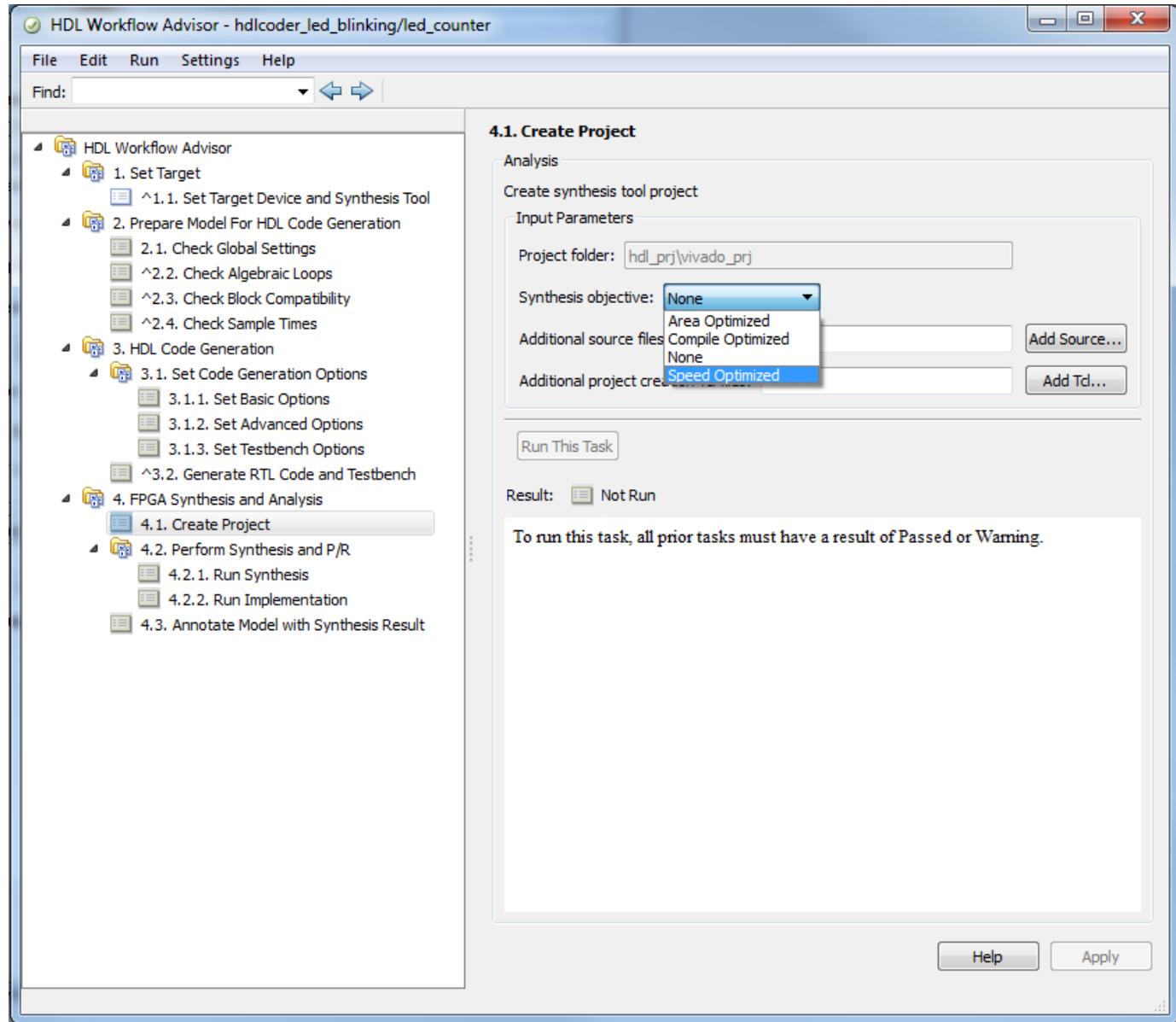


### Setup the Workflow

Use the HDL Workflow Advisor to setup your project with your desired settings, such as a Synthesis Tool and Device. Start by changing the workflow to "Generic ASIC/FPGA" so that we can Annotate the model with synthesis results.



You can also specify high level objectives for the synthesis tool. For example, try setting the tool to "Speed Optimized".



### 4.1. Create Project

#### Analysis

Create synthesis tool project

#### Input Parameters

Project folder: `hdl_prj\vivado_prj`

Synthesis objective: `None`

Additional source files `Compile Optimized`

`None`

`Speed Optimized`

Additional project creation options

`Add Source...`

`Add Tcl...`

`Run This Task`

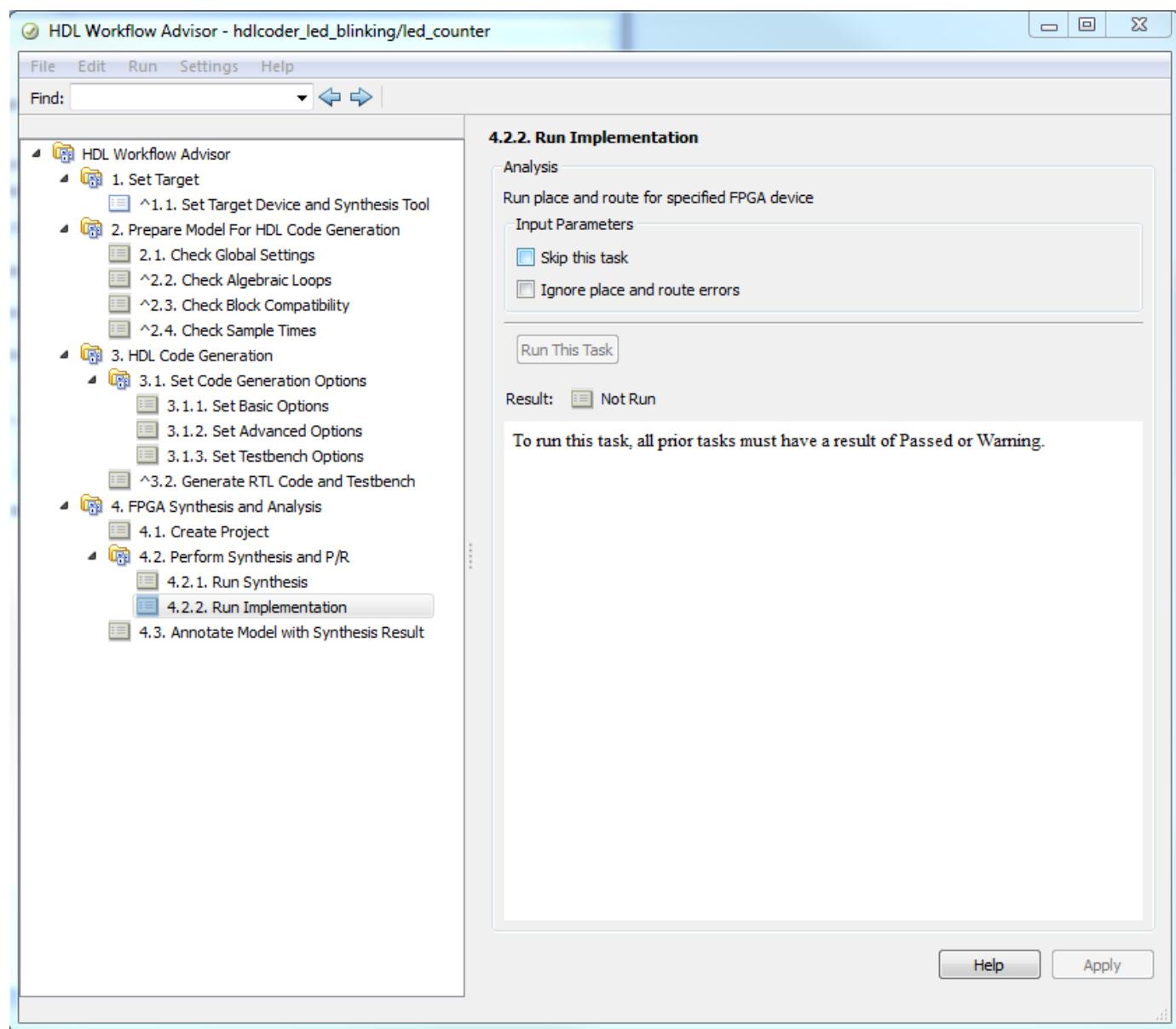
Result: `Not Run`

To run this task, all prior tasks must have a result of Passed or Warning.

`Help`

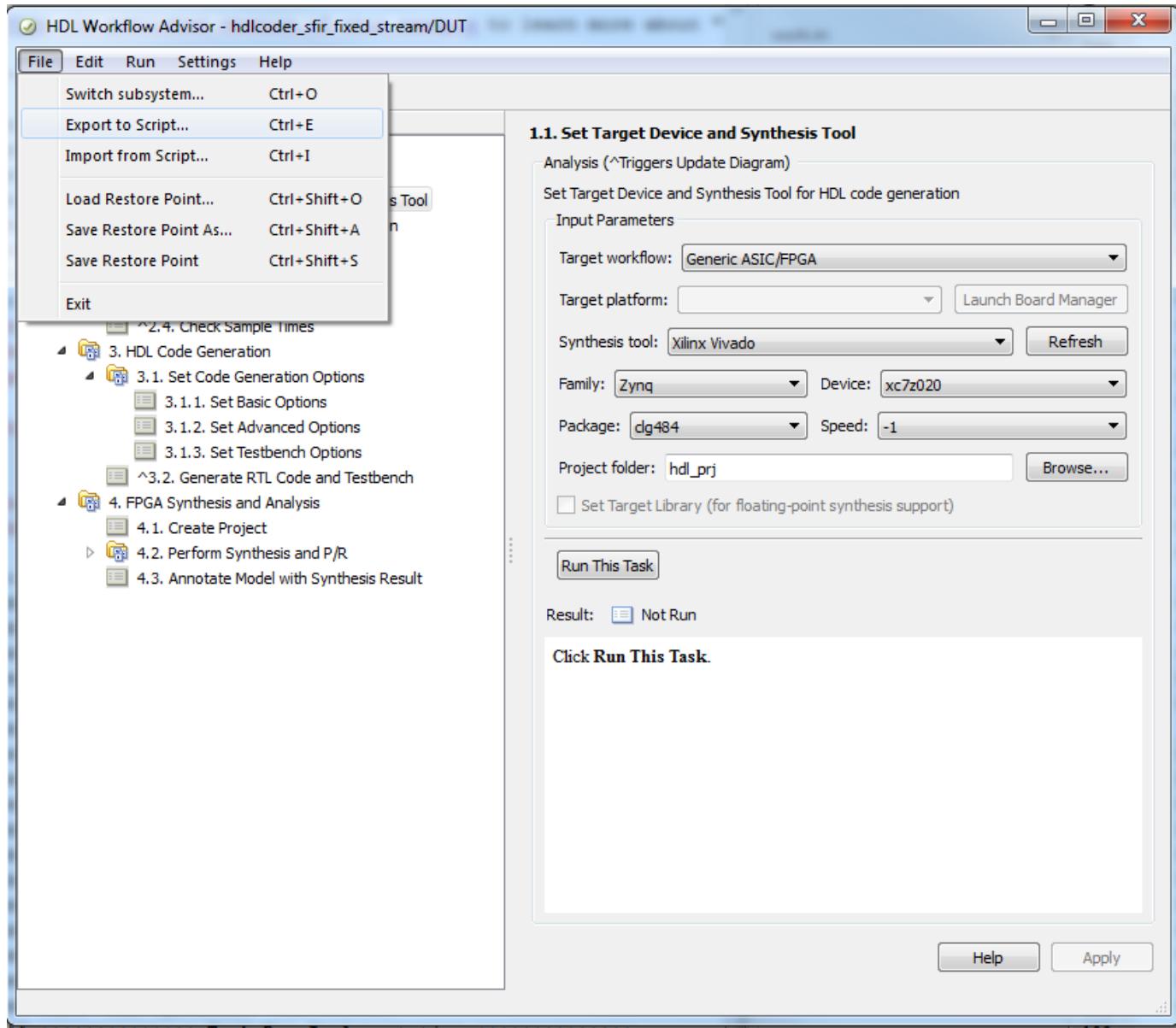
`Apply`

Also, change the "Skip this task" checkbox for Run Implementation so that the exported script runs this step as well



## Export to Script

After all the initial settings have been entered, export your workflow to a script which can be run directly from the command-line for faster design iterations.



Save the file as any name you like. The default will be "hdlworkflow.m". The exported script is shown below:

```
%-----
% HDL Workflow Script
% Generated with MATLAB 9.0 (R2016b Prerelease) at 10:40:45 on 31/12/2015
% This script was generated using the following parameter values:
%     Filename : '/mathworks/devel/sandbox/cberry/work/demo/hdlworkflow.m'
%     Overwrite : true
%     Comments : true
%     Headers   : true
%     DUT       : 'hdlcoder_sfir_fixed_stream/DUT'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','hdlcoder_sfir_fixed_stream/DUT');
```

```
%-----%
% Copyright 2018 The MathWorks, Inc.

%% Load the Model
load_system('hdlcoder_sfir_fixed_stream');

%% Restore the Model to default HDL parameters
%hdlrestorereparams('hdlcoder_sfir_fixed_stream/DUT');

%% Model HDL Parameters
%% Set Model 'hdlcoder_sfir_fixed_stream' HDL parameters
hdlset_param('hdlcoder_sfir_fixed_stream', 'HDLSubsystem', 'hdlcoder_sfir_fixed_stream/DUT');
hdlset_param('hdlcoder_sfir_fixed_stream', 'ReferenceDesign', 'Default system with AXI4-Stream interface');
hdlset_param('hdlcoder_sfir_fixed_stream', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolDeviceName', 'xc7z020');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolPackageName', 'clg484');
hdlset_param('hdlcoder_sfir_fixed_stream', 'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoder_sfir_fixed_stream', 'TargetDirectory', 'hdl_prj/hdlsrc');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Generic ASIC/FPGA');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskRunSynthesis = true;
hWC.RunTaskRunImplementation = false;
hWC.RunTaskAnnotateModelWithSynthesisResult = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateRTLTBench = false;
hWC.GenerateCosimulationModel = false;
hWC.CosimulationModelForUseWith ='Mentor Graphics ModelSim';
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskRunSynthesis' Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set properties related to 'RunTaskRunImplementation' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set properties related to 'RunTaskAnnotateModelWithSynthesisResult' Task
hWC.CriticalPathSource ='pre-route';
hWC.CriticalPathNumber = 1;
hWC.ShowAllPaths = false;
```

```
hWC>ShowDelayData = true;
hWC>ShowUniquePaths = false;
hWC>ShowEndsOnly = false;

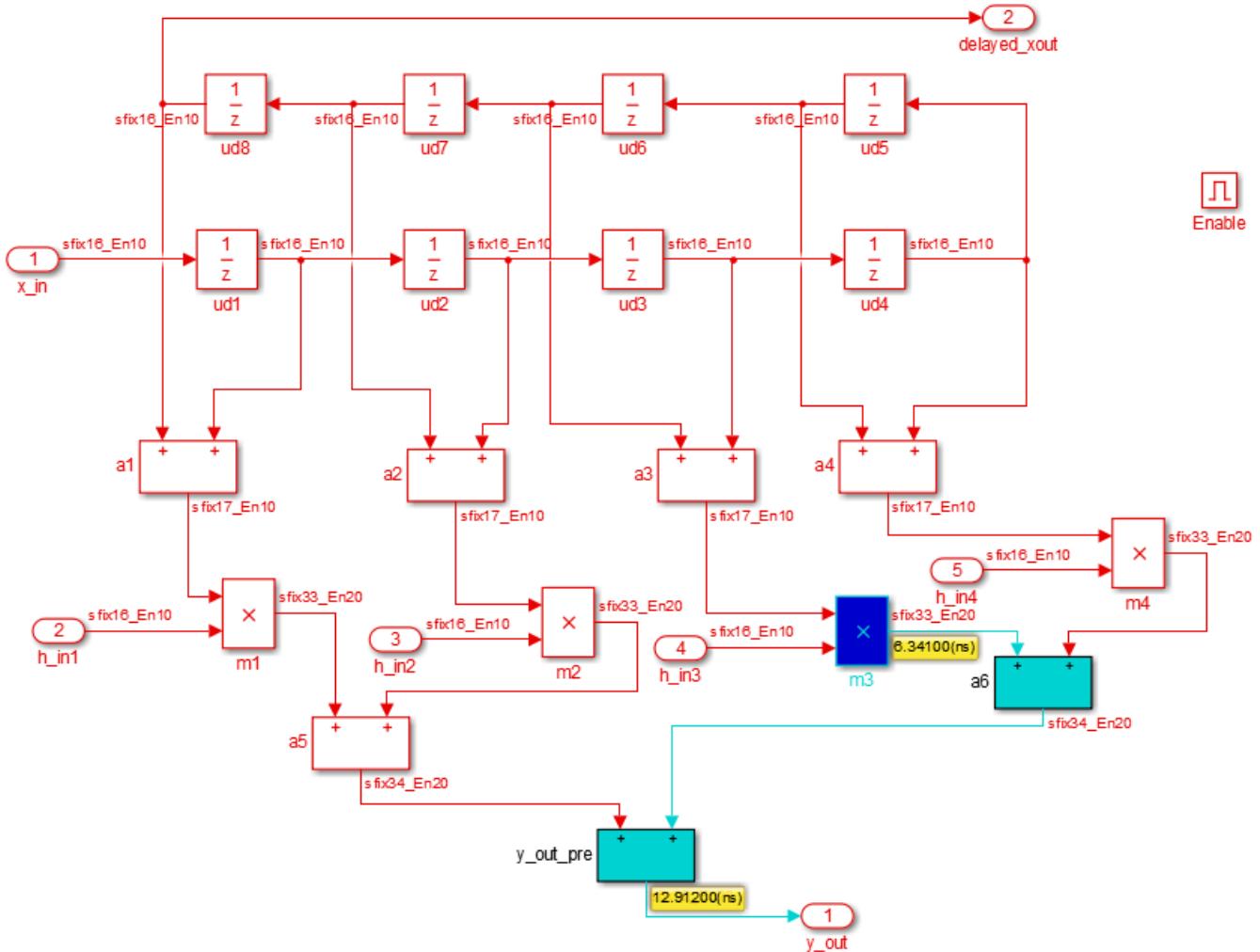
% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_sfir_fixed_stream/DUT', hWC);
```

### Run workflow from Script

Running the script directly will execute your workflow and output a compact set of runtime messages to the cmd window. If you would like to see detailed synthesis tool output information, click on the relevant "Synthesis tool log: " hyperlink under the desired task header to open this file in the MATLAB editor.

```
>> hdlworkflow
### Workflow begin.
### ++++++ Task Generate RTL Code and Testbench ++++++
### Generating HDL for 'hdlcoder_sfir_fixed_stream/DUT'.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_sfir_fixed_stream'.
### Working on hdlcoder_sfir_fixed_stream/DUT/symmetric_fir as hdl_prj\hdlsrc\hdlcoder_sfir_fixed_
### Working on hdlcoder_sfir_fixed_stream/DUT as hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\DUt.vl
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'hdlcoder_sfir_fixed_stream' complete with 0 errors, 0 warnings, and 0 messages
### HDL code generation complete.
### ++++++ Task Create Project ++++++
### Generating Xilinx Vivado 2014.4 project: hdl_prj\vivado_prj\DUt_vivado.xpr
### Synthesis tool log: hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\workflow_task_CreateProject.log
### Task "Create Project" successful.
### ++++++ Task Run Synthesis ++++++
### Synthesis tool log: hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\workflow_task_RunSynthesis.log
### Task "Run Synthesis" successful.
### ++++++ Task Run Implementation ++++++
### Synthesis tool log: hdl_prj\hdlsrc\hdlcoder_sfir_fixed_stream\workflow_task_RunImplementation.log
### Task "Run Implementation" successful.
### ++++++ Task Annotate Model with Synthesis Result ++++++
### Parsing the timing file...
### Matched Source = 'hdlcoder_sfir_fixed_stream/DUT/symmetric_fir/m3_out1'
### Matched Destination = 'hdlcoder_sfir_fixed_stream/DUT/y_out_data'
### Highlighting CP 1 from 'hdlcoder_sfir_fixed_stream/DUT/symmetric_fir/m3_out1' to 'hdlcoder_sfir_
### Click here to reset highlighting.
### Workflow complete.
```



## Run workflow interactively

The HDL Workflow Command-Line interface can also be used interactively. For example, after either running the entire script or just the section "Workflow Configuration Settings", the `WorkflowConfig` object, `hWC`, will be populated in the workspace:

```
>> hWC =
GenericTurnkeyConfig with properties:

    SynthesisTool: 'Xilinx Vivado'
    TargetWorkflow: 'Generic ASIC/FPGA'
    ProjectFolder: 'hdl_prj'

    RunTaskGenerateRTLCodeAndTestbench: true
    RunTaskCreateProject: true
    RunTaskRunSynthesis: true
    RunTaskRunImplementation: true
    RunTaskAnnotateModelWithSynthesisResult: true
```

```
TaskGenerateRTLCodeAndTestbench
    GenerateRTLCode: true
    GenerateRTLTesbench: false
    GenerateCosimulationModel: false
    CosimulationModelForUseWith: 'Mentor Graphics ModelSim'
    GenerateValidationModel: false

    TaskCreateProject
        Objective: hdlcoder.Objective.SpeedOptimized
    AdditionalProjectCreationTclFiles: ''

    TaskRunSynthesis
    SkipPreRouteTimingAnalysis: false

    TaskRunImplementation
    IgnorePlaceAndRouteErrors: false

TaskAnnotateModelWithSynthesisResult
    CriticalPathSource: 'pre-route'
    CriticalPathNumber: 1
    ShowAllPaths: false
    ShowDelayData: true
    ShowUniquePaths: false
    ShowEndsOnly: false
```

You can edit this configuration object and then run the workflow with the modified settings. For example, since the task "Run Implementation" was enabled in the previous run, we can change the critical path source to "post-route" with and rerun just the Annotate model task:

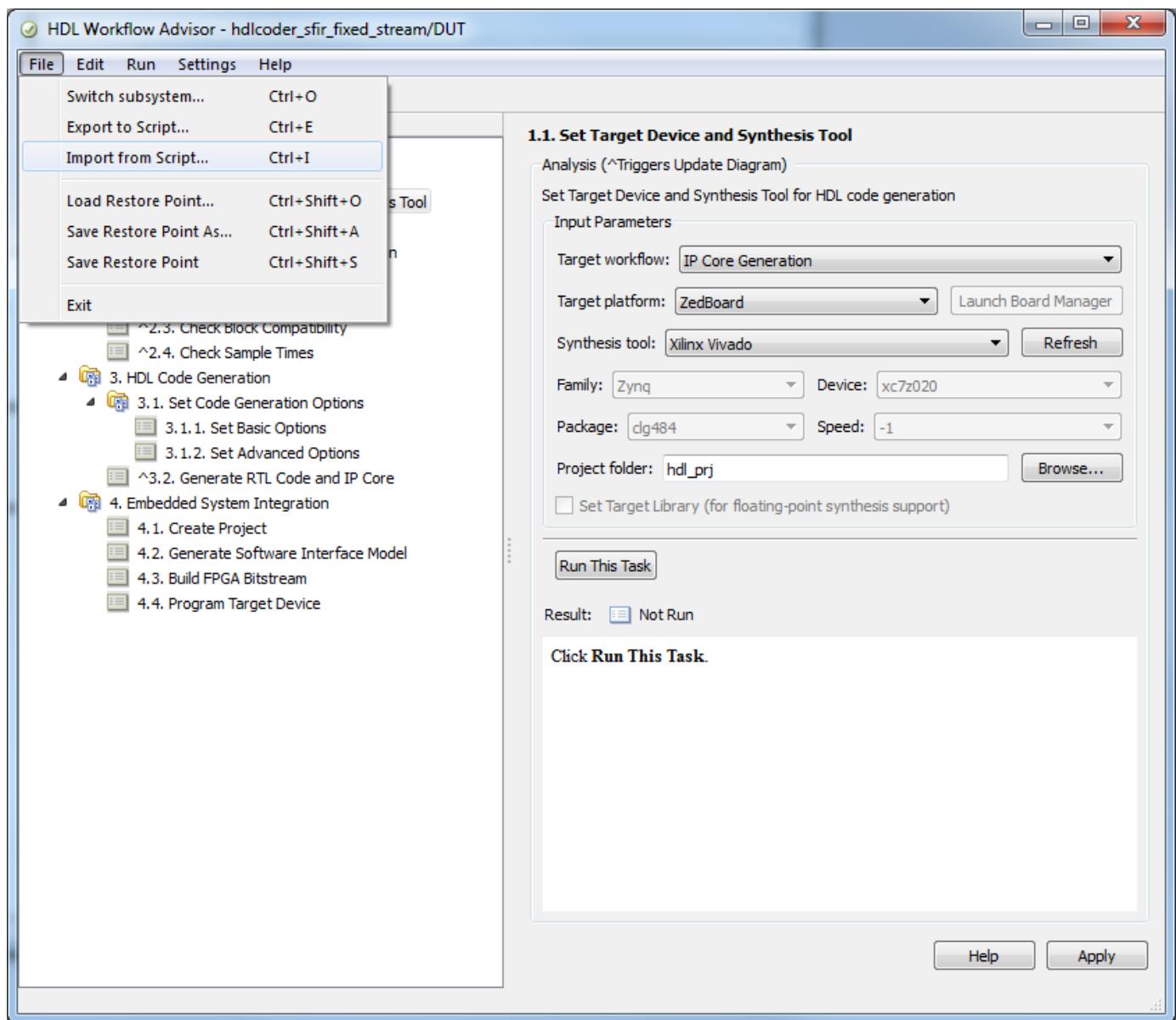
```
>> hWC.clearAllTasks;
>> hWC.RunTaskAnnotateModelWithSynthesisResult = true;
>> hWC.CriticalPathSource = 'post-route';
```

Then run the modified workflow configurations directly using the `hdlcoder.runWorkflow` command:

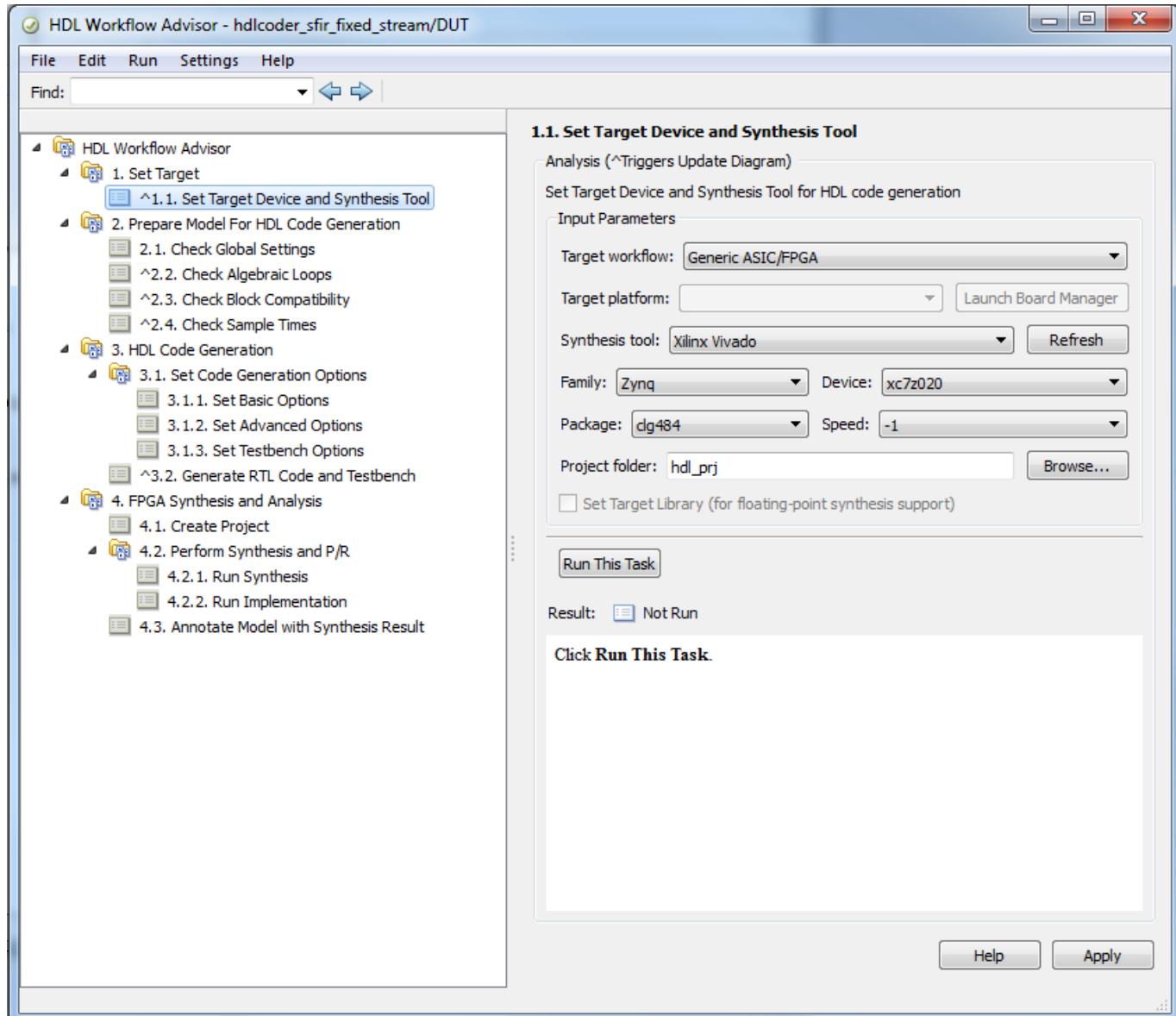
```
>> hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC)
```

### **Import script into HDL Workflow Advisor**

Any changes you make to the exported script can also be imported back into the HDL Workflow Advisor at any time. To do this, make sure the model loaded is the same as the model used in the script, and select "Import from script" from the File menu.



After importing, all of the script settings will be populated in the HDL Workflow Advisor.



### Save the HDL Workflow Command-Line Interface programming script for later use

In certain cases, you can avoid re-running all the steps in the HDL Workflow Advisor just to perform specific tasks in the Advisor. For example, when running the IP Core Generation Workflow, after you generate the FPGA bitstream in the Build FPGA Bitstream Task, the Advisor provides a link that generates a script to run the Program Target Device Task.

Click on the link for `hdlworkflow_ProgramTargetDevice.m` in Build FPGA Bitstream Task to generate an HDL Workflow Command-Line Interface script which will execute only the Program Target Device Task from an existing `hdl_prj` directory. Therefore, you can avoid running all the steps in the workflow just to re-program the target device.

The screenshot shows the 'Build FPGA Bitstream' task results. It includes sections for Analysis, Input Parameters (with a checked checkbox for 'Run build process externally'), and a 'Run This Task' button. The result is listed as 'Passed'. Below the result, there is a 'Synthesis Tool Log' containing command-line output related to the build process.

**Analysis**  
Synthesis and generate bitstream for embedded system on FPGA

**Input Parameters**

Run build process externally

Tcl file for synthesis build: Default

**Run This Task**

**Result:** Passed

**Passed Build Embedded System.**

**Synthesis Tool Log:**

```
Task "Build FPGA Bitstream" successful.  
Generated logfile: hdl\_prj\hdlsrc\hdlcoder\_led\_vector\workflow\_task\_BuildFPGABitstream.log  
Running embedded system build outside MATLAB.  
Please check external shell for system build progress.  
The generated bitstream file is located at: hdl\_prj\vivado\_ip\_prj\vivado\_prj.runs\impl\_1\system\_top\_wrapper.bit  
Generate an HDL Workflow Command-Line Interface script to program the target device: hdlWorkflow\_ProgramTargetDevice.m.
```

The generated script is a standard HDL Workflow CLI script. When running HDL Workflow in command-line interface, the **Program Target Device** Task can be run independent of previous tasks, as long as the FPGA bitstream is already generated.

As a related note, if you are using the **Download** programming method in the **Program Target Device** Task, the HDL Workflow Advisor copies the generated bitstream file onto the SD card on the Zynq or Intel SoC board, so you do not need to re-run the **Program Target Device** Task to download the bitstream. The FPGA bitstream will be reloaded from the SD card automatically during the Linux boot up process.

## Summary

The HDL Workflow Command-Line Interface provides an easily scripted alternative to the graphical HDL Workflow Advisor. Workflows can be setup initially using the HDL Workflow Advisor and then exported to script for iterative or automated use.

## Getting Started with FPGA Turnkey Workflow

This example shows how to program a standalone FPGA with your MATLAB design, using the FPGA Turnkey workflow.

The target device in this example is a Xilinx ® Virtex-5 ML506 development board.

### Introduction

In this example, the function '**mlhdlc\_ip\_core\_led\_blinking**' models a counter that blinks the LEDs on an FPGA board.

Two input ports, **Blink\_frequency** and **Blink\_direction**, are control ports that determine the LED blink frequency and direction.

You can adjust the input values of the hardware via push-buttons on Xilinx ® Virtex-5 ML506 development board. The output port of the design function, 'LED', connects to the LED hardware.

```
design_name = 'mlhdlc_turnkey_led_blinking';
testbench_name = 'mlhdlc_turnkey_led_blinking_tb';
```

Let us take a look at the MATLAB design

```
type(design_name);

function [LED, Read_back] = mlhdlc_turnkey_led_blinking(Blink_frequency, Blink_direction)
%
% Copyright 2013-2015 The MathWorks, Inc.

persistent freqCounter LEDCounter

if isempty(freqCounter)
    freqCounter = 0;
    LEDCounter = 255;
end

if Blink_frequency <= 0
    Blink_frequency = 0;
elseif Blink_frequency >= 15
    Blink_frequency = 15;
end

blinkFrequencyOut = LookupTable(Blink_frequency);
if blinkFrequencyOut == freqCounter
    freqMatch = 1;
else
    freqMatch = 0;
end

freqCounter = freqCounter + 1;

if freqMatch
    freqCounter = 0;
end
```

```

if Blink_direction
    LED = 255 - LEDCounter;
else
    LED = LEDCounter;
end

if LEDCounter == 255
    LEDCounter = 0;
elseif freqMatch
    LEDCounter = LEDCounter + 1;
end

Read_back = LED;
end

function y = LookupTable(idx)
s = 2.^{26:-1:11}';

y = s(idx+1);
end

type(testbench_name);

%
% Copyright 2013-2015 The MathWorks, Inc.

for i=1:16
    [yout, ~] = mlhdlc_turnkey_led_blinking(i-1, 0);
    [yout2, ~] = mlhdlc_turnkey_led_blinking(i-1, 1);
end

```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_turnkey_led_blinking'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);

```

### Create a New HDL Coder™ Project

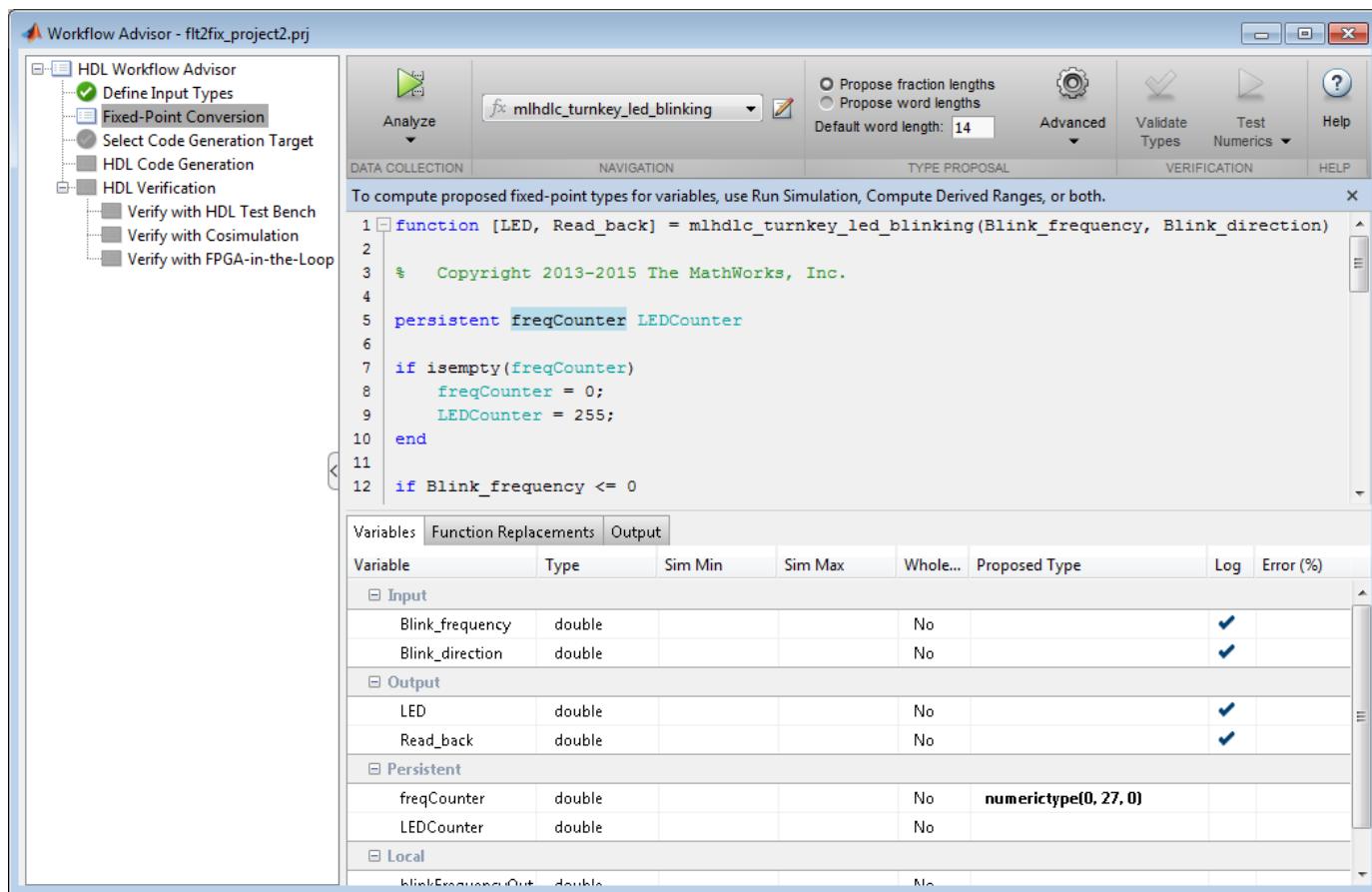
```
coder -hdlcoder -new mlhdlc_turnkey_led_blinking_prj
```

Next, add the file 'mlhdlc\_turnkey\_led\_blinking.m' to the project as the MATLAB Function and 'mlhdlc\_turnkey\_led\_blinking\_tb.m' as the MATLAB Test Bench.

See “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Convert Design To Fixed-Point

1. Right-click the **Define Input Types** task and select **Run This Task**.
2. In the Fixed-Point Conversion task, click **Advanced** and set the **Safety margin for sim min/max (%)** to 0.
3. Set the proposed type of the **freqCounter** variable to unsigned 27-bit integer by entering **numerictype(0, 27, 0)** in its 'Proposed Type' column.



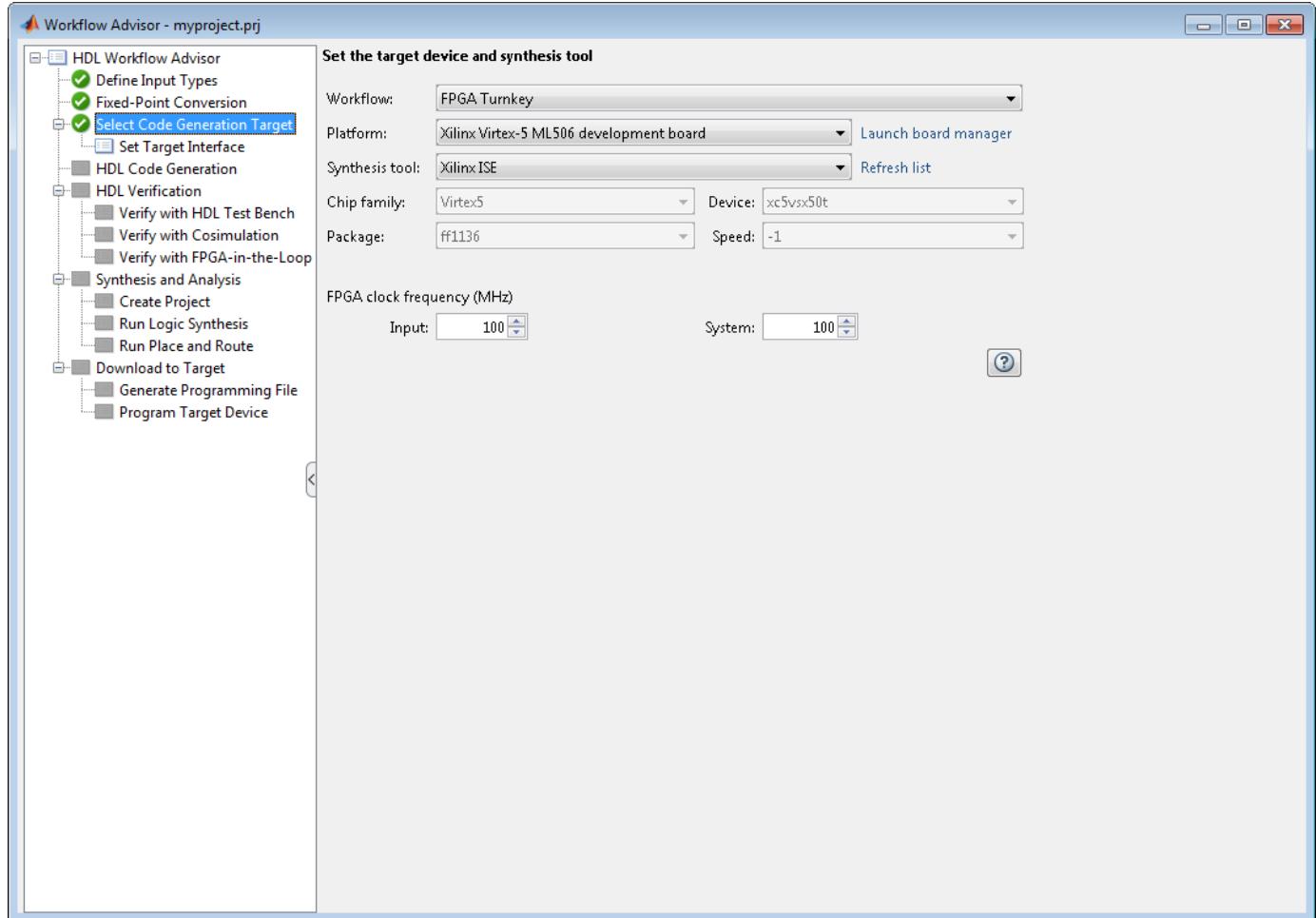
4. On the left, right-click the **Fixed-Point Conversion** task and select **Run This Task**.

### Map Design Ports to Target Interface

In the **Select Code Generation Target** task, select the **FPGA Turnkey** workflow and **Xilinx Virtex-5 ML506 development board** as follows:

1. For **Workflow**, select **FPGA Turnkey**.
2. For **Platform**, select **Xilinx Virtex-5 ML506 development board**. If your target device is not in the list, select **Get more** to download the support package. The coder automatically sets **Chip family**, **Device**, **Package**, and **Speed** according to your platform selection.

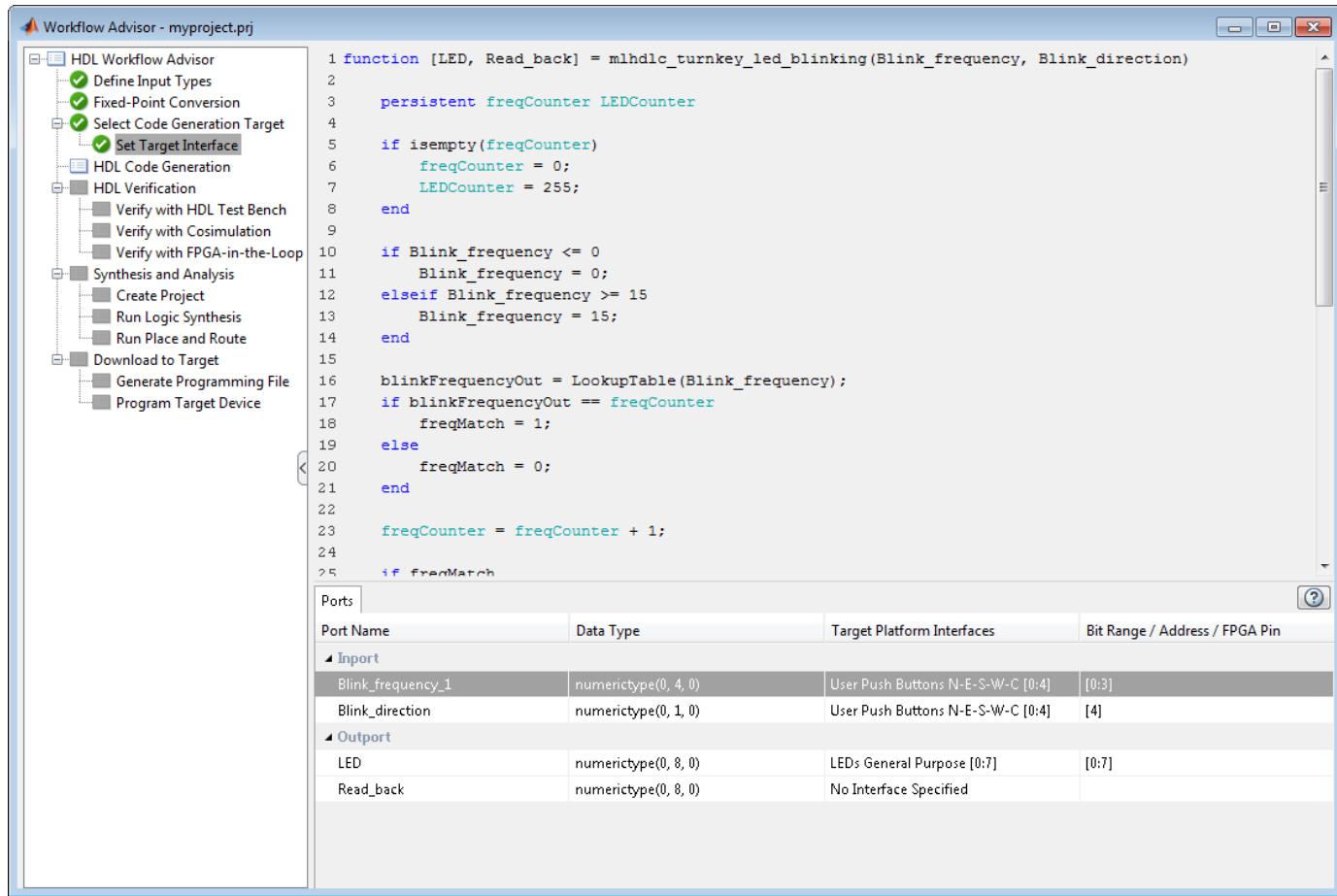
3. For FPGA clock frequency, for both **Input** and **System**, enter 100.



4. In the **Set Target Interface** task, map the design input and output ports to interfaces on the target device by setting the fields in the **Target Platform Interfaces** column as follows:

- 1 Blink\_frequency\_1 to **User Push Buttons N-E-S-W-C [0:4]**
- 1 Blink\_direction to User Push Buttons **N-E-S-W-C [0:4]**
- 1 LED to **LEDs General Purpose [0:7]**

You can leave the 'Read\_back' port unmapped.



### Generate Programming File and Download To Hardware

You can generate code, perform synthesis and analysis, and download the design to the target hardware using the default settings:

1. For the **Synthesis and Analysis** task group, uncheck the **Skip this Step** option.
2. For the **Download to Target** task group, uncheck the **Skip this Step** option.
3. Right-click **Download to Target > Generate Programming File** and select **Run to Selected Task**.
4. If your target hardware is connected and ready to program, select the **Program Target Device** subtask and click **Run**.

### Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde');
mlhdlc_temp_dir = [tempdir 'mlhdlc_turnkey_led_blinking'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Simscape to HDL Workflow

---

- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 32-2
- “Modeling Guidelines for Simscape Subsystem Replacement” on page 32-5
- “Generate HDL Code for Simscape Models” on page 32-9
- “Generate Optimized HDL Implementation Model from Simscape” on page 32-17
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25
- “Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script” on page 32-33
- “Partition Simscape Models Containing a Large Network into Multiple Smaller Networks” on page 32-47
- “Generate HDL Code for Simscape Models with Multiple Networks” on page 32-54
- “Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model” on page 32-63
- “Troubleshoot Conversion of Simscape Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model” on page 32-70
- “Replacing Variable Resistors” on page 32-86
- “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 32-90
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97
- “Improve Sampling Rate of HDL Implementation Model Generated from Simscape Algorithm” on page 32-104

## Get Started with Simscape Hardware-in-the-Loop Workflow

To perform hardware-in-the-loop (HIL) simulation with smaller timesteps and increased accuracy, deploy the Simscape plant models to the FPGAs on board the Speedgoat I/O modules.

- Generate an HDL implementation model by using the Simscape HDL Workflow Advisor. The implementation model is a Simulink model that replaces the Simscape components with HDL-compatible Simulink blocks.
- Generate HDL code for the implementation model, and then deploy the generated code to generic FPGAs, SoCs, or FPGAs on board Speedgoat FPGA I/O modules by using the HDL Workflow Advisor.

By using this capability, you can model and deploy complex physical systems in Simscape that previously took long time to model by using Simulink blocks.

### Simscape Example Models for HDL Code Generation

For HDL code generation, you can design your own Simscape algorithm or choose from a list of example models that are created in Simscape. The example models include:

- Boost converter
- Bridge rectifier
- Buck converter
- Half-wave rectifier
- Three phase rectifier
- Two level converter ideal
- Two level converter IGBT
- Solar power inverter model
- Swiss rectifier
- Vienna rectifier

All examples are prefixed with `sschdlex` and postfixed with `Example`. For example, to open the boost converter model, at the MATLAB command prompt, enter:

```
load_system('sschdlexBoostConverterExample')
open_system('sschdlexBoostConverterExample/Simscape_system')
```

### Guidelines for Modeling Simscape for HDL Compatibility

Follow these guidelines when designing your Simscape algorithm for compatibility with Simscape HDL Workflow Advisor. To replace the subsystem that uses Simscape blocks with the corresponding state-space algorithm, follow these additional guidelines as described in “Modeling Guidelines for Simscape Subsystem Replacement” on page 32-5.

#### Use Linear and Switched Linear Blocks

Create a Simscape model by using linear and switched linear blocks. Linear blocks are blocks that are defined by a linear relationship such as resistors. Switched linear blocks are blocks such as diodes and switches. These blocks are also defined by a linear relationship such as  $V = IR$  where  $R$  can switch between two or more values depending on the state of the diodes or switches.

Nonlinear blocks are not supported. To verify that the Simscape model does not contain nonlinear blocks, use the `simscape.findNonlinearBlocks` function. Provide the path to your Simscape model as an argument to this function.

```
simscape.findNonlinearBlocks('current_model')
```

Alternatively, to verify that the model does not contain nonlinear blocks, run the “Check switched linear task” on page 33-3 of the Simscape HDL Workflow Advisor.

### Specify Backward Euler Solver with Discrete Sample Time

Configure the solver options for HDL code generation by using a Solver Configuration block.

In the Block Parameters dialog box of this block:

- Select **Use local solver**.
- Use Backward Euler as the **Solver type**.
- Specify a discrete sample time, Ts.

To verify that the solver settings are specified correctly, run the “Check solver configuration task” on page 33-2 of the Simscape HDL Workflow Advisor.

### Run `hdlsetup` function

After creating the model, configure the model for HDL code generation by running the `hdlsetup` function. `hdlsetup` configures the solver settings such as using a fixed-step solver, specifies the simulation start and stop times, and so on. To run the command for your `current_model`:

```
hdlsetup('current_model')
```

## Restrictions for HDL Code Generation from Simscape Models

HDL Coder does not support code generation from Simscape networks that contain:

- Events
- Mode charts
- Delays
- Runtime parameters
- Periodic sources
- Simscape Multibody™ blocks
- Simscape Electrical™ Specialized Power Systems blocks
- Nonlinear and time-varying Simscape blocks. Time-varying blocks include blocks such as Variable Inductor and Variable Capacitor.
- Nonscalar states or inputs to the network. Split nonscalar inputs into scalar inputs and reduce the second operand of the colon operator by one for error caused by nonscalar states. For example:

```
% Suppose this code generates an error
tmp1 = u(1:4);

% Fix the error by reducing second operand by 1
tmp1 = u(1:3);
```

**See Also**

`makehdl` | `sschdladvisor`

**More About**

- “Generate HDL Code for Simscape Models” on page 32-9
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25

# Modeling Guidelines for Simscape Subsystem Replacement

To generate HDL code for Simscape algorithms, you generate an HDL implementation model by using the Simscape HDL Workflow Advisor. If you follow certain guidelines when modeling the Simscape algorithm, the Simscape HDL Workflow Advisor replaces the Simscape subsystem with a corresponding **HDL Subsystem** block in the HDL implementation model. The **HDL Subsystem** block contains the state-space algorithm that uses HDL-compatible Simulink blocks instead of Simscape blocks. You can generate HDL code for the **HDL Subsystem** block and deploy the code onto FPGA target devices and FPGAs on board Speedgoat FPGA I/O modules. In this case, when you select the **Generate validation logic for the implementation model** check box in the **Generate implementation model** task of the Simscape HDL Workflow Advisor, the Advisor generates a separate state-space validation model. This model compares the outputs from the **HDL Subsystem** and the original Simscape subsystem to verify that they are functionally equivalent.

If you do not follow the guidelines, the Simscape HDL Workflow Advisor might not be able to perform this replacement. In that case, the HDL implementation model contains the state-space algorithm with the original Simscape subsystem beside it. Before generating code, you modify the implementation model and rearrange the blocks such that it replaces the Simscape subsystem with the state-space algorithm. In this case, when you select the **Generate validation logic for the implementation model** check box, the Advisor places a validation logic subsystem inside the implementation model to verify functional equivalence.

In addition to these guidelines, make sure that the Simscape model is configured for compatibility with Simscape HDL Workflow Advisor. See “Guidelines for Modeling Simscape for HDL Compatibility” on page 32-2.

## Enclose Simscape Blocks Inside a Subsystem

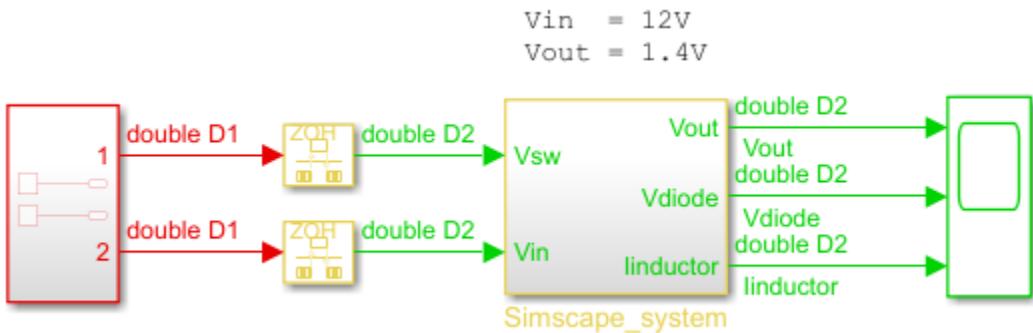
- Enclose the Simscape blocks for which you are generating an HDL implementation model inside a Subsystem block and provide the test inputs. Inside the Subsystem block, your model can have multiple hierarchies that use Simscape blocks.
- Do not use masked subsystems. The Simscape HDL Workflow Advisor cannot replace masked subsystems in the HDL implementation model.
- Inside the Subsystem block that contains Simscape blocks, at the input ports, add Simulink-PS Converter blocks. At the output ports of this subsystem, add PS-Simulink Converter blocks.
  - Use a meaningful name for the Simulink-PS Converter and PS-Simulink Converter blocks.

The Simscape HDL Workflow Advisor uses the names of the Simulink-PS Converter and PS-Simulink Converter blocks for the input and output ports of the **HDL Subsystem** block. Using a meaningful name makes it easier to identify what the input and output ports in the HDL implementation model correspond to.

- In the Block Parameters dialog box of the Simulink-PS Converter and PS-Simulink Converter blocks, on the **Input Handling** tab, leave **Filtering and derivatives** set to **Provide signals** and **Provided signals** set to **Input only**.

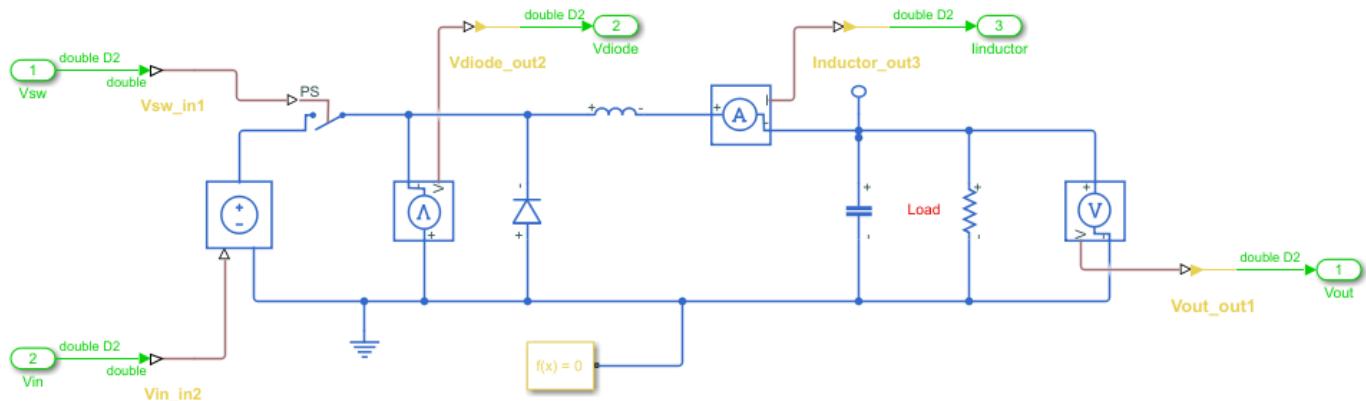
For example, open the buck converter model. The Simscape\_system block contains Simscape blocks. Blocks outside this subsystem form the test environment.

```
open_system('sschdlexBuckConverterExample')
sim('sschdlexBuckConverterExample')
```



Inside the `Simscape_system` subsystem, the model uses Simscape blocks and physical signals. The model has Simulink-PS Converter and PS-Simulink Converter blocks at the interfaces. Provide unique names for these blocks such that they match the corresponding port names.

```
open_system('sschdlexBuckConverterExample/Simscape_system')
```

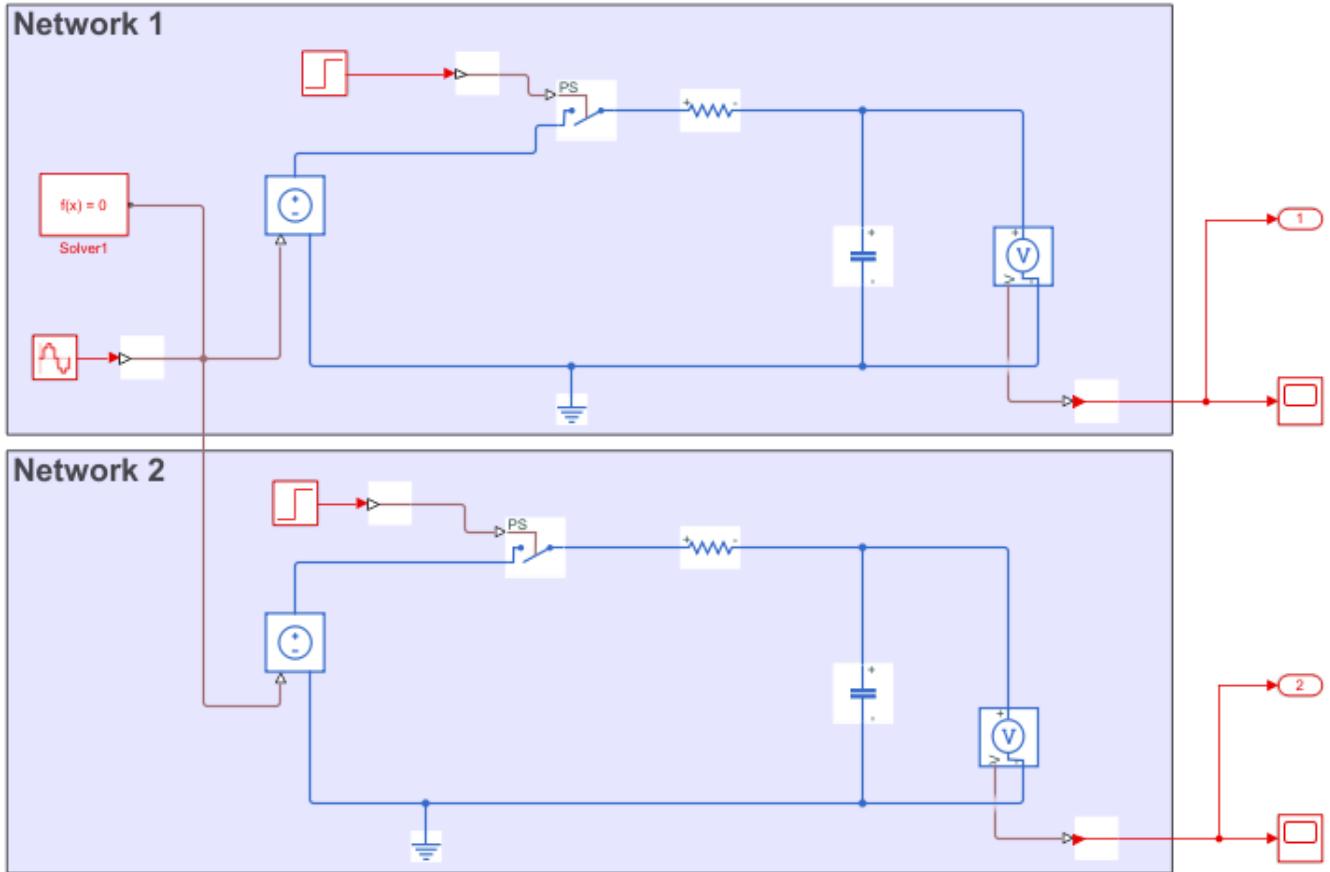


## Multiple Simscape Network Considerations

If your Simscape model contains multiple networks:

- Enclose each network inside a subsystem. Add Simulink-PS Converter and PS-Simulink Converter blocks at the subsystem interface.
- Use a Solver Configuration block for each network. Use the same sample time across Solver Configuration blocks inside the different networks.

For example, this model contains more Simscape networks than Solver Configuration blocks, the Simscape network is not replaced with the HDL subsystem.



The Simscape HDL Workflow Advisor then replaces each Simscape subsystem with the corresponding **HDL Subsystem**.

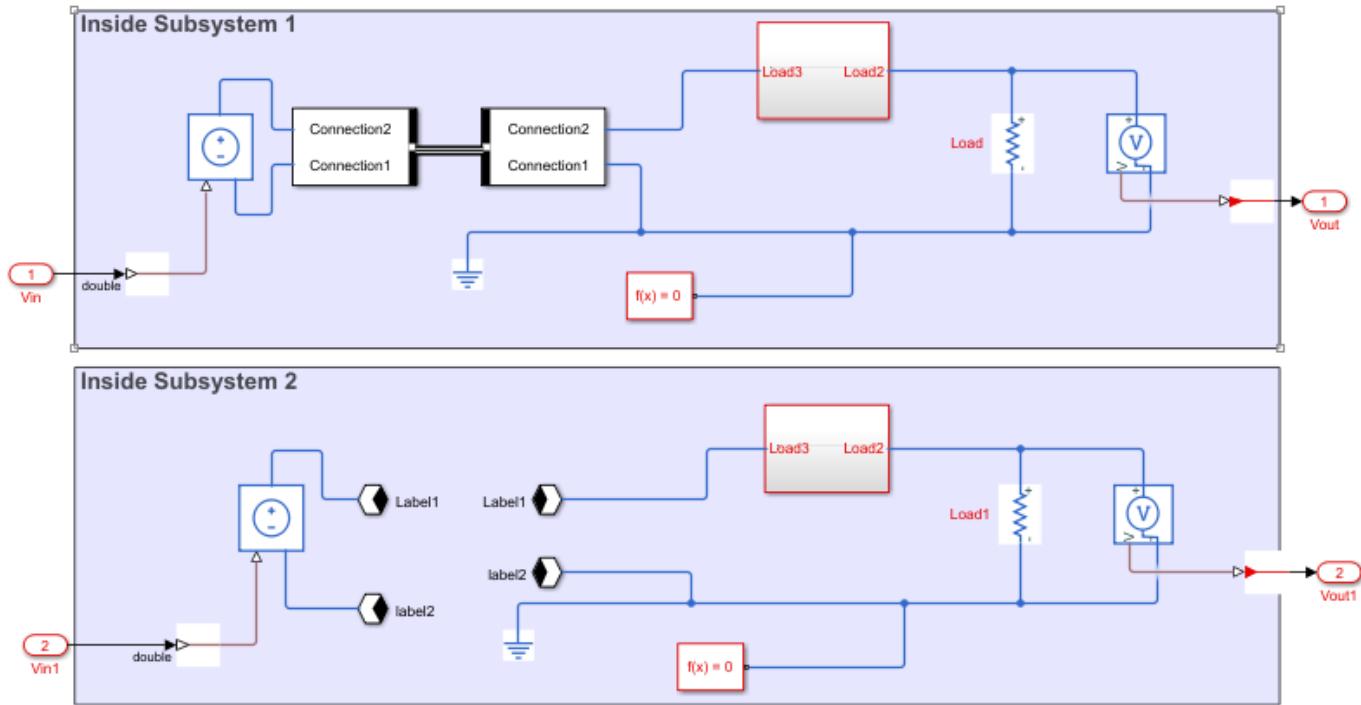
For an example that shows how to generate HDL code for a model that has multiple networks, see “Generate HDL Code for Simscape Models with Multiple Networks” on page 32-54.

## Avoid Using Certain Blocks in Simscape Utilities Library

To generate an implementation model that replaces the Simscape subsystem with the state-space algorithm, in your original Simscape model, do not use these blocks from the **Simscape > Utilities** Library:

- Simscape Bus
- Connection Port
- Connection Label

For example, this model contains Connection Label and Simscape Bus blocks inside two different subsystems. The Simscape HDL Workflow Advisor cannot replace these subsystems with the state-space algorithm.



## See Also

[makehdl](#) | [sschdladvisor](#)

## More About

- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 32-2
- “Generate HDL Code for Simscape Models” on page 32-9
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25

# Generate HDL Code for Simscape Models

This example shows how to generate HDL code for a halfwave rectifier model that uses Simscape™ blocks. Use the Simscape HDL Workflow Advisor to generate an HDL implementation model. You can then generate HDL code for the implementation model. See “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 32-2.

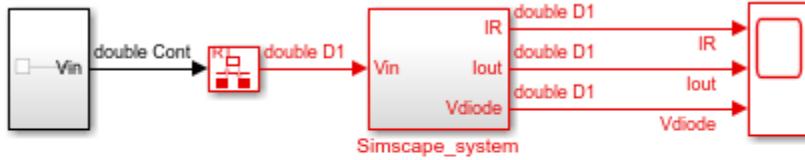
## The Halfwave Rectifier Model

To open the half-wave rectifier model, at the MATLAB command prompt, enter:

```
open_system('sschdlexHalfWaveRectifierExample')
```

Save this model locally as `HalfWaveRectifier_HDL` to run the workflow.

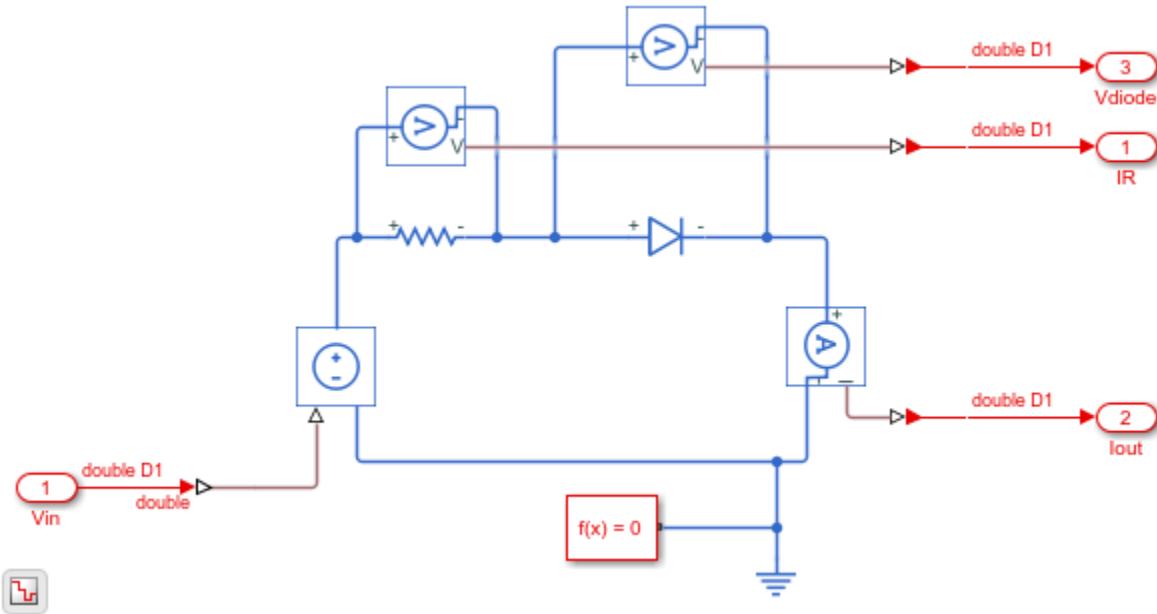
```
open_system('HalfWaveRectifier_HDL')
set_param('HalfWaveRectifier_HDL', 'SimulationCommand', 'Update')
```



Copyright 2020 The MathWorks, Inc.

At the top level of the model, a `Simscape_system` block models the half-wave rectifier algorithm. The model accepts a Sine Wave input, uses a Rate Transition block to discretize the continuous time input, and has a Scope block that calculates the output. To see the half-wave rectifier algorithm, double-click the `Simscape_system` subsystem.

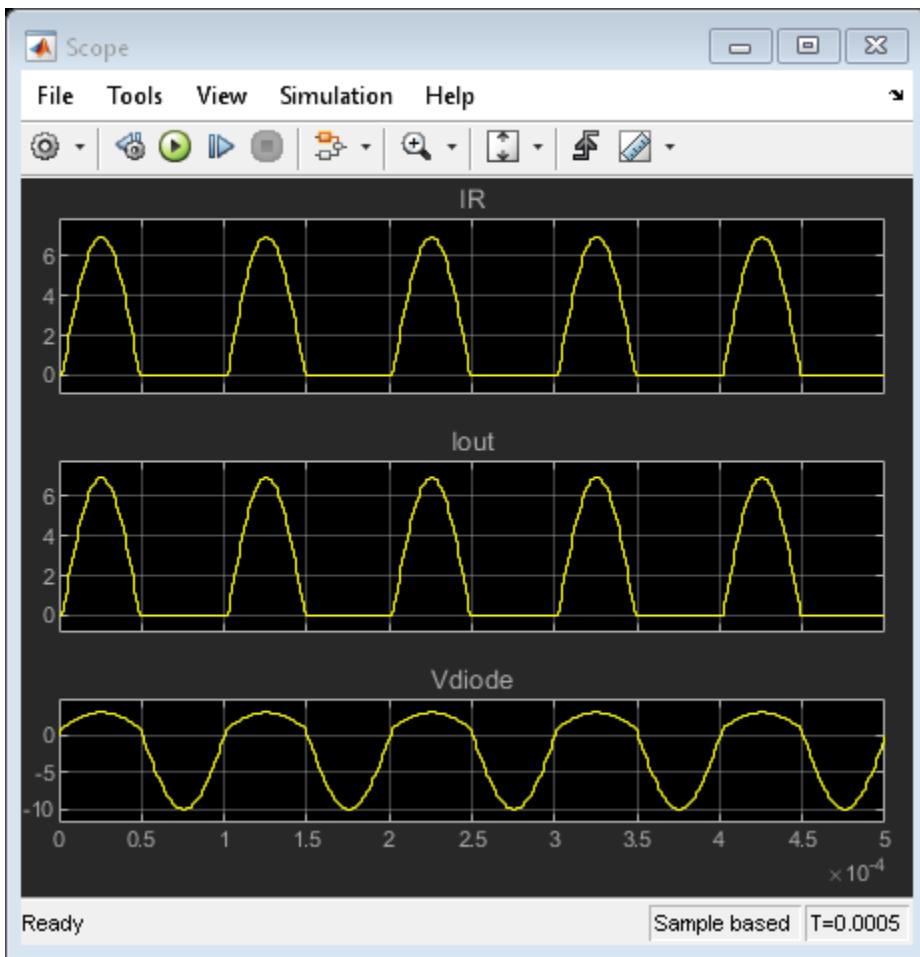
```
open_system('HalfWaveRectifier_HDL/Simscape_system')
```



The half-wave rectifier consists of a resistor, which is a linear block, and a diode, which is a switched linear block. The Simscape model is preconfigured for HDL compatibility. At the input and output port interfaces, the model has Simulink-PS Converter and PS-Simulink Converter blocks. The solver settings are configured for compatibility with Simscape HDL Workflow Advisor. If you open the Block Parameters dialog box for the Solver Configuration block, **Use local solver** is selected and Backward Euler is specified as the **Solver type**. See “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 32-2.

To see the functionality, simulate the model and then open the Scope block.

```
sim('HalfWaveRectifier_HDL')
open_system('HalfWaveRectifier_HDL/Scope')
```



### Run Simscape HDL Workflow Advisor

To generate an HDL implementation model from which you generate code, use the Simscape HDL Workflow Advisor. To open the Advisor, run this command:

```
sschdladvisor('HalfWaveRectifier_HDL')
```

```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(HalfWaveRectifier_HDL)">HalfWaveRe
```

This command updates the model advisor cache and opens the Simscape HDL Workflow Advisor. To learn more about the Simscape HDL Workflow Advisor and the various tasks, right-click that folder or task, and select **What's This?**. See also “Simscape HDL Workflow Advisor Tasks” on page 33-2.

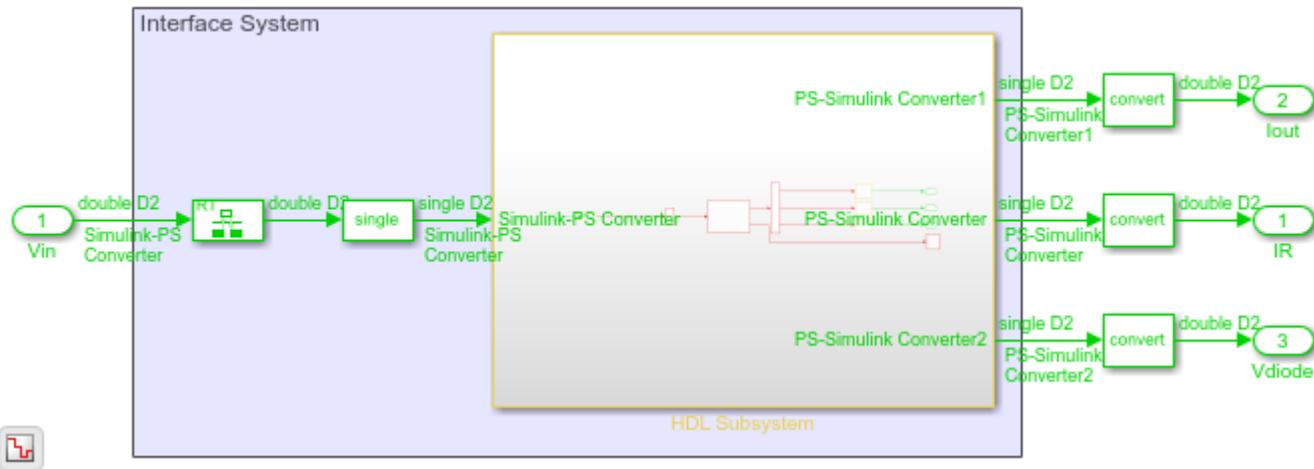
To run the workflow and compare functionality of the HDL implementation model with the original Simscape algorithm, select the **Generate implementation model** step, and then select the **Generate validation logic for the implementation model** check box. Use a **Validation logic tolerance** of 0.001. Right-click the **Generate implementation model** step and select **Run to Selected Task**.

The Advisor generates an HDL implementation model and a state-space validation model. The implementation model has the same name as the original Simscape model and uses the prefix `gmStateSpaceHDL_`. The state-space validation model has the same name as the implementation model and uses the postfix `_vnl`.

## Open and Examine HDL Implementation Model

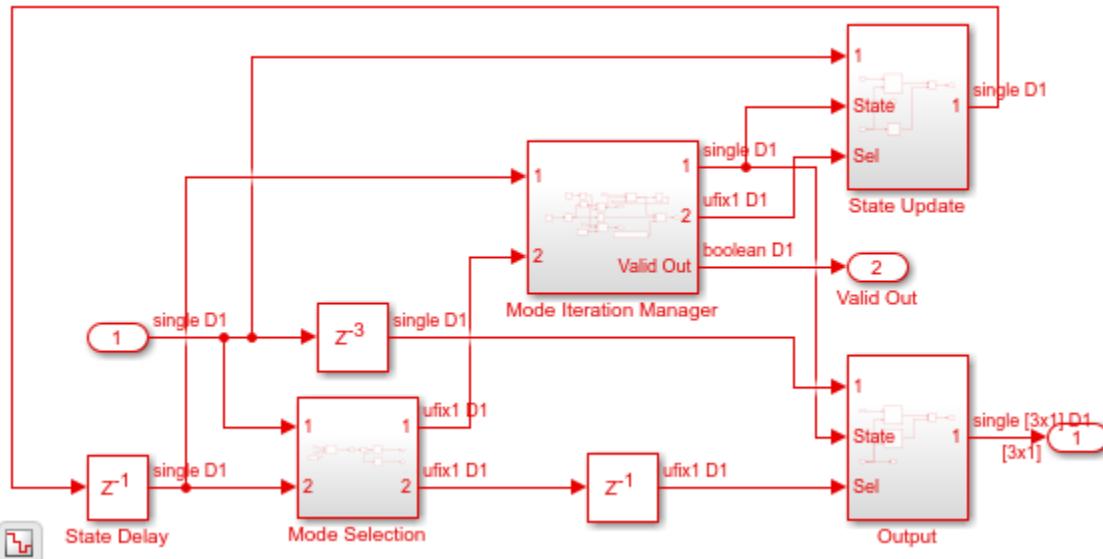
In the **Generate implementation model task**, click the link to open the implementation model. The model contains a `Simscape_system` subsystem that contains a `HDL Subsystem` block. The `HDL Subsystem` models the state-space representation that you generated from the Simscape model.

```
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL')
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system')
set_param('gmStateSpaceHDL_HalfWaveRectifier_HDL','SimulationCommand','Update')
```



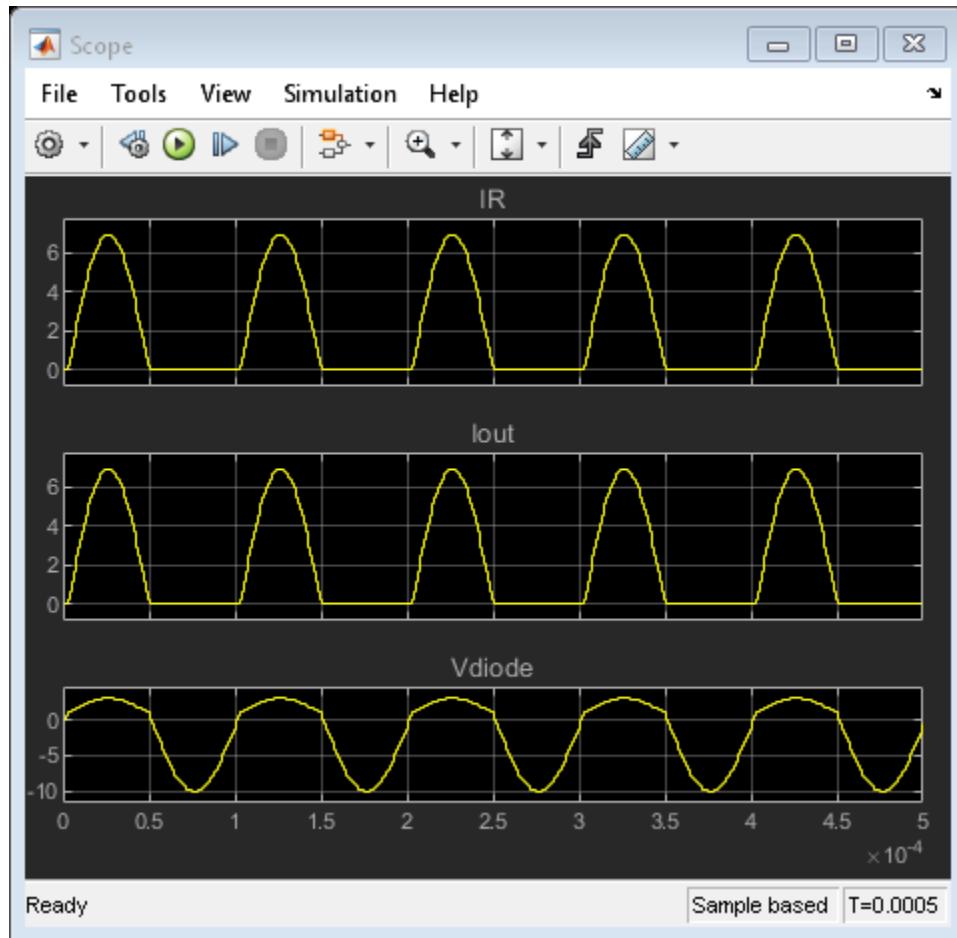
The ports of this subsystem use the same name as the Simulink-PS Converter and PS-Simulink Converter blocks in your original Simscape model. If you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations.

```
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system/HDL_Subsystem/HDL_Algorithm')
```



To simulate the HDL Implementation model, enter these commands. Open the Scope block to view results.

```
sim('gmStateSpaceHDL_HalfWaveRectifier_HDL')
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL/Scope')
```

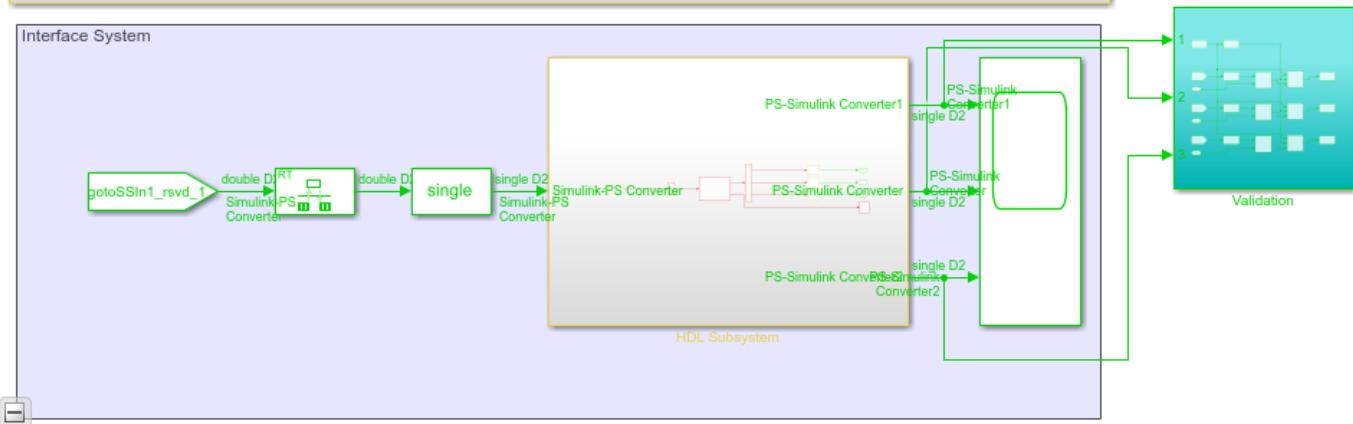
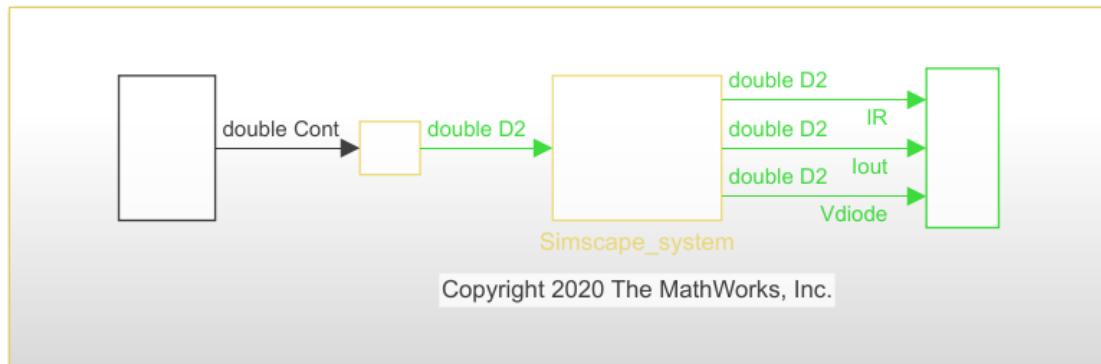


HDL code is generated for the HDL Subsystem block inside this model.

### Validate HDL Algorithm

To compare functionality of the HDL implementation model with the original Simscape algorithm, open and simulate the state-space validation model.

```
open_system('gmStateSpaceHDL_HalfWaveRectifier_HDL_vnl')
sim('gmStateSpaceHDL_HalfWaveRectifier_HDL_vnl')
```



The output of this model matches the original Simscape model. The simulation does not generate assertions, which indicates that the outputs match. For a more systemic verification, see “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97.

In some cases, your Simscape algorithm might not be compatible for generating an implementation model by using the Simscape HDL Workflow Advisor. In such cases, running certain tasks in the Advisor can result in the task to fail. To learn how you can make it HDL compatible, see:

- “Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model” on page 32-63
- “Troubleshoot Conversion of Simscape Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model” on page 32-70

### Generate HDL Code and Validation Model

The HDL model and subsystem parameter settings are saved using this command:

```
hdlsaveparams('gmStateSpaceHDL_HalfWaveRectifier_HDL');

%% Set Model 'gmStateSpaceHDL_HalfWaveRectifier_HDL' HDL parameters
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'FloatingPointTargetConfiguration', hdlcode
, 'LatencyStrategy', 'MIN') ...
);
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'HDLSubsystem', 'gmStateSpaceHDL_HalfWaveRe
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'MaskParameterAsGeneric', 'on');
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL', 'Oversampling', 60);

% Set SubSystem HDL parameters
```

```
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system/HDL Subsystem', 'FlattenHierarchicalModel', 'on')
hdlset_param('gmStateSpaceHDL_HalfWaveRectifier_HDL/Simscape_system/HDL Subsystem/HDL Algorithm/Mux', 'OversamplingFactor', 60)
```

The model uses single data types and generates HDL code in **Native Floating Point** mode. Floating-point operators can introduce delays. Because the design contains feedback loops, to allocate sufficient delays for the operators inside the feedback loops, the model uses clock-rate pipelining in conjunction with a large value for the **Oversampling factor**. An **Oversampling factor** of **60** and the clock-rate pipelining optimization is saved on this model.

For more information, see:

- “Clock-Rate Pipelining” on page 24-114
- “Oversampling factor” on page 17-15
- “Allocate Sufficient Delays for Floating-Point Operations” on page 10-67

Before you generate HDL code, enable generation of the validation model. The validation model compares the output of the generated model after code generation and the original model. To learn more, see “Generated Model and Validation Model” on page 24-10.

Run these commands to save validation model generation settings on your Simulink model:

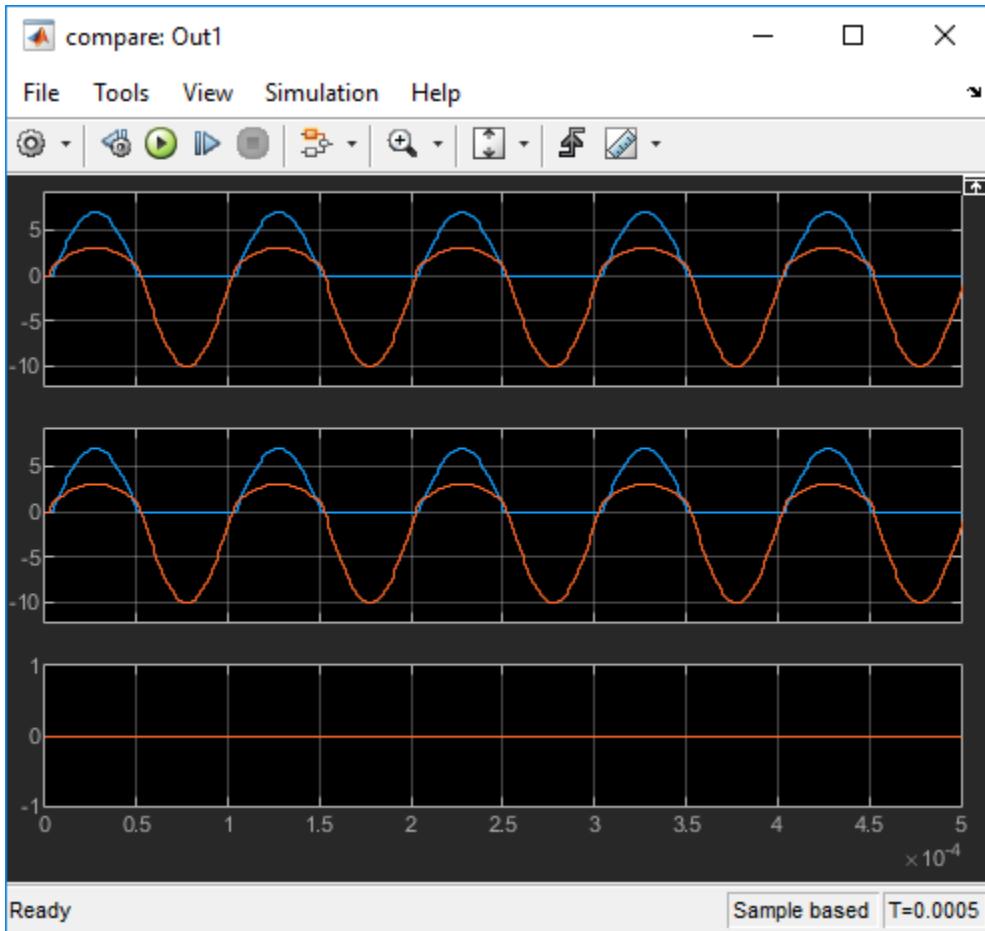
```
HDLmodelName = 'gmStateSpaceHDL_HalfWaveRectifier_HDL';
hdlset_param(HDLmodelName, 'TargetDirectory', 'C:/Temp/hdlsrc');
hdlset_param(HDLmodelName, 'GenerateValidationModel', 'on');
```

To generate HDL code, run this command:

```
makehdl('gmStateSpaceHDL_HalfWaveRectifier_HDL/HDL Subsystem');
```

The generated HDL code and validation model are saved in **C:/Temp/hdlsrc** directory. The generated code is saved as **HDL\_Subsystem\_tc.vhd**. To open the validation model, click the link to **gm\_gmStateSpaceHDL\_HalfWaveRectifier\_HDL\_vnl.slx** in the code generation logs in the Command Window.

Open the **Compare** block at the output of **HDL Subsystem\_vnl** subsystem of the validation model. Then, open the **Assert\_Out1** block. To see the simulation results after HDL code generation, open the **Compare: Out1** Scope block. The top graph represents the output of the generated model, and the middle graph represents the output of the implementation model. The bottom graph calculates the difference between outputs of both models. As the outputs match, the error is zero.



## See Also

### Functions

[checkhdl](#) | [makehdl](#)

## More About

- “Get Started with Simscape Electrical” (Simscape Electrical)
- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 32-2
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97

# Generate Optimized HDL Implementation Model from Simscape

This example shows how you can generate an optimized HDL implementation model for a Simscape™ vienna rectifier model by using optimizations such as resource sharing and RAM mapping.

## Why Optimize the HDL Implementation Model

For Simscape models that have many switching elements, the state-space representation contains a large number of configurations. The Simscape HDL Workflow Advisor simulates the Simscape model to calculate the number of relevant configurations. Certain Simscape models can have a large number of configurations that are relevant. The generated HDL implementation model for such a large design can consume a significantly large number of resources. Synthesizing the generated code can cause the design to occupy a large amount of resources on the FPGA device. In some cases, the design might not fit on the target FPGA device. To save resources and make the design fit on the FPGA, the Simscape HDL Workflow Advisor uses HDL Coder™ optimizations such as clock-rate pipelining, resource sharing, and RAM mapping.

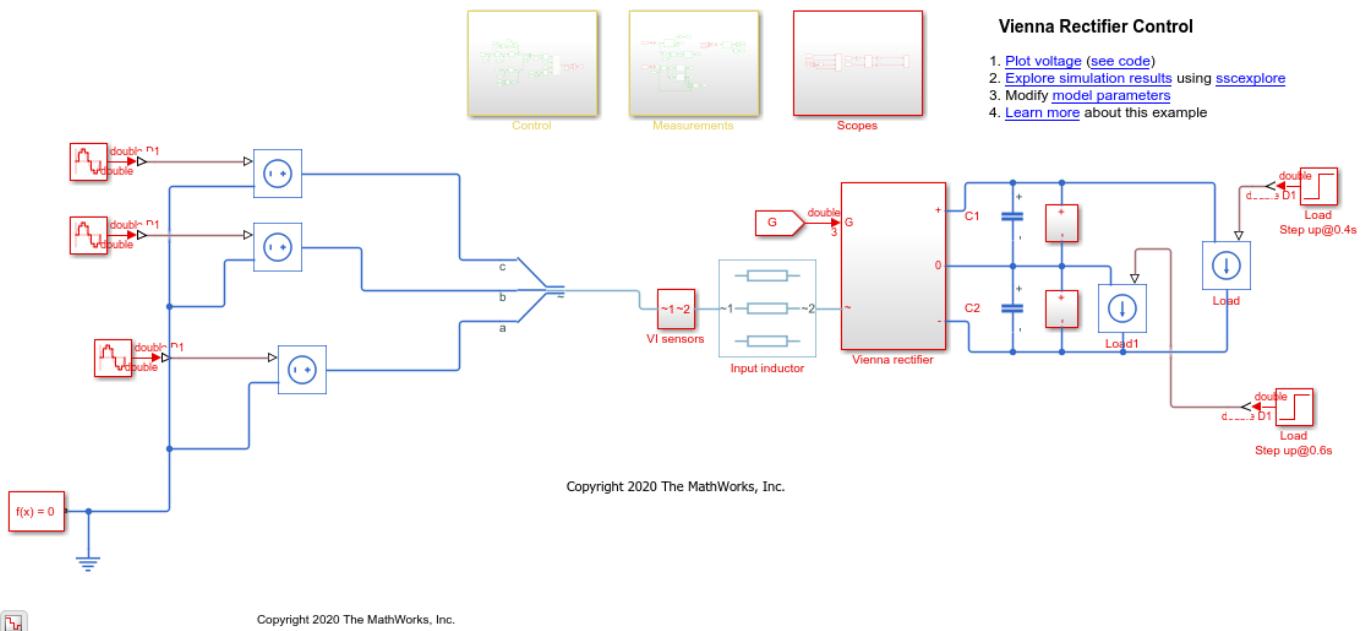
## Vienna Rectifier Model

To open the model, at the MATLAB® command prompt, enter:

```
open_system('sschdlexViennaRectifierExample')
```

Save this model as ViennaRectifier\_HDL to run the workflow.

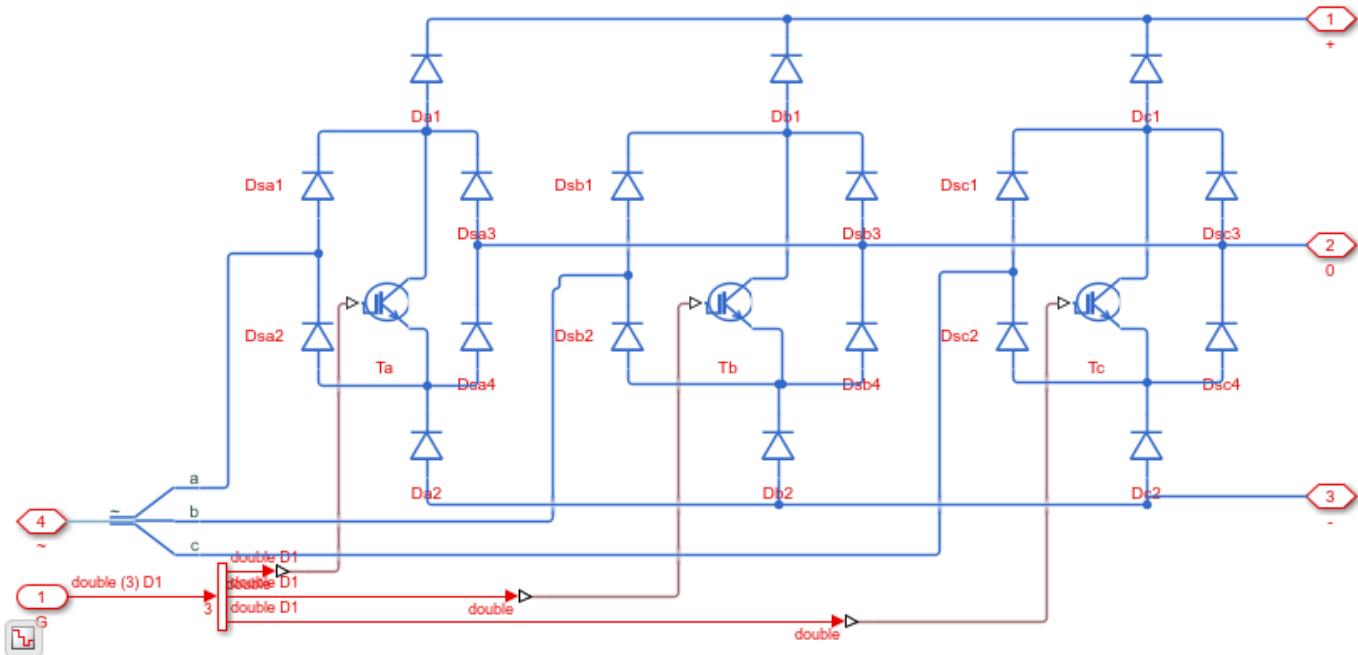
```
open_system('ViennaRectifier_HDL')
set_param('ViennaRectifier_HDL', 'SimulationCommand', 'Update')
```



The Control subsystem implements a closed-loop control strategy for the Vienna rectifier subsystem by using space-vector modulation. At simulation time  $0.1\text{s}$ , the vienna rectifier is engaged. At times  $0.4\text{s}$  and  $0.6\text{s}$ , the load steps up on the DC side.

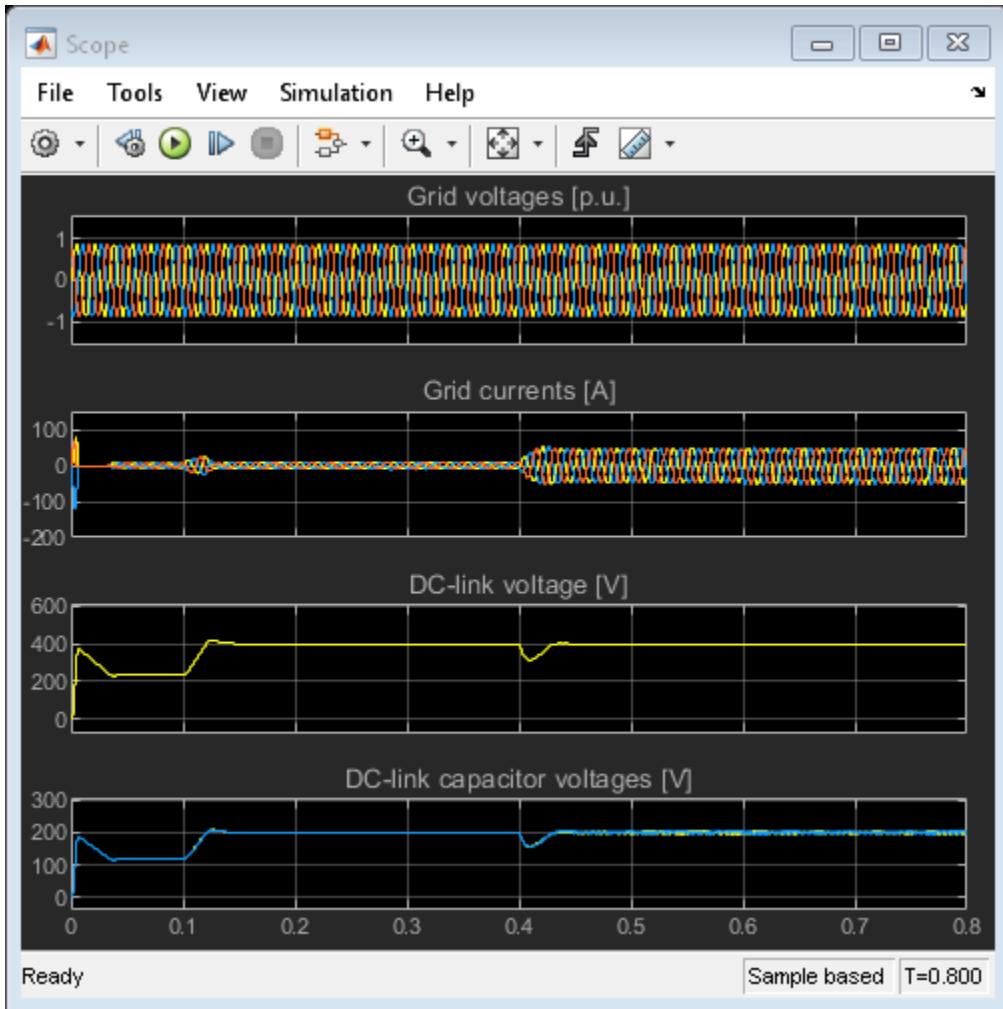
The Vienna rectifier subsystem consists of three-phase legs. Each leg has one power switch and six power diodes. See “Vienna Rectifier Control” (Simscape Electrical).

```
open_system('ViennaRectifier_HDL/Vienna_rectifier')
```



Simulate the model. View the simulation results by double-clicking the Scope blocks inside the Scopes subsystem.

```
sim('ViennaRectifier_HDL')
open_system('ViennaRectifier_HDL/Scopes/Scope')
```



### Generate HDL Implementation Model and Validate HDL Algorithm

To generate an HDL implementation model, use the Simscape HDL Workflow Advisor. You can generate HDL code for the implementation model. To open the Advisor, run this command:

```
sschdladvisor('ViennaRectifier_HDL')
```

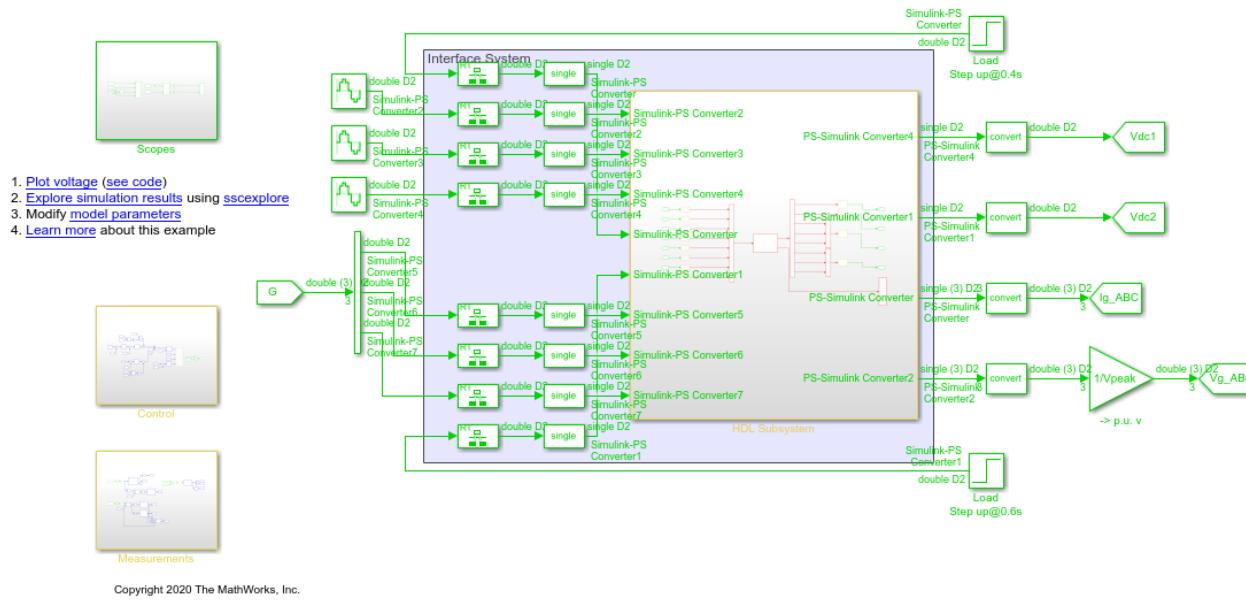
```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(ViennaRectifier_HDL)">ViennaRectifier_HDL</a>
```

To run the workflow, right-click the **Generate implementation model** task and select **Run to Selected Task**. After the task passes, you see a link to the HDL implementation model. To see the number of configurations, select the **Extract Equations** task. On the task, you see that simulating the model reaches 558 modes. Such a large number of modes can increase resource consumption of the design on the FPGA.

To open the HDL implementation model, enter these commands:

```
open_system('gmStateSpaceHDL_ViennaRectifier_HDL')
set_param('gmStateSpaceHDL_ViennaRectifier_HDL', 'SimulationCommand', 'Update')
```

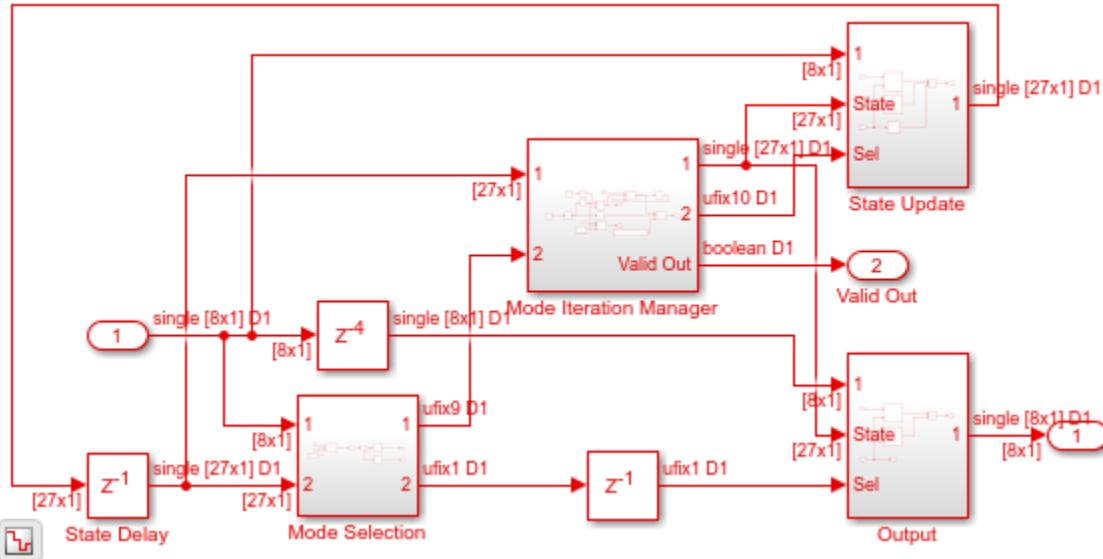
### Vienna Rectifier Control



Copyright 2020 The MathWorks, Inc.

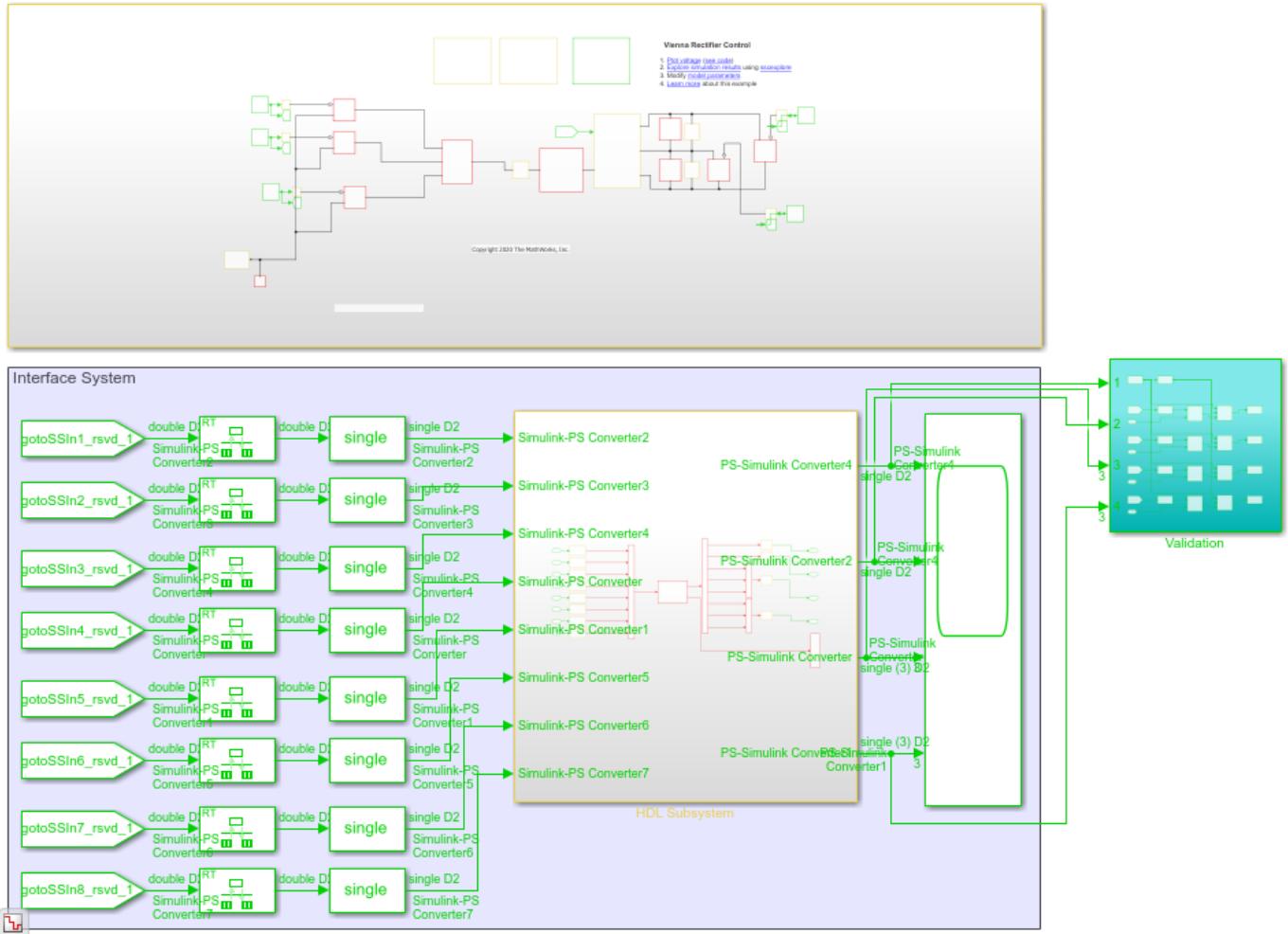
The ports of this subsystem use the same name as the Simulink-PS Converter and PS-Simulink Converter blocks in your original Simscape model. If you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations.

```
open_system('gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem/HDL Algorithm')
```



To validate the HDL algorithm, in the **Generate implementation model** task, select the **Generate validation logic for the implementation model** check box, set the **Validation logic tolerance** to **0.001**, and rerun this task. The task generates a state-space validation model that compares the implementation model and the original Simscape model.

```
open_system('gmStateSpaceHDL_ViennaRectifier_HDL_vnl')
set_param('gmStateSpaceHDL_ViennaRectifier_HDL_vnl', 'SimulationCommand', 'Update')
```



Simulating the model does not display assertions, which indicates that the HDL algorithm matches the original model.

```
sim('gmStateSpaceHDL_ViennaRectifier_HDL_vnl')
```

### Map State-Space Parameters in Implementation Model to RAM

The HDL implementation model uses `single` data types and contains large Delay blocks that are inside a feedback loop in the `HDL Algorithm` subsystem. To accommodate the large delays and make the design run at a faster clock rate on the target FPGA, the model uses clock-rate pipelining in conjunction with a large value of **Oversampling factor**.

```
hdlsaveparams('gmStateSpaceHDL_ViennaRectifier_HDL')
```

For more information, see:

- “Clock-Rate Pipelining” on page 24-114
- “Oversampling factor” on page 17-15

- “Allocate Sufficient Delays for Floating-Point Operations” on page 10-67

In the **Generate implementation model** task, the **Map state space parameters to RAMs** setting uses the default value of Auto. This setting maps large state-space parameters in the HDL implementation model to RAMs when the number of modes exceed a threshold value of 200. As the vienna rectifier model uses a large number of modes, the state-space parameters are mapped to RAMs. By mapping to RAMs, you save lookup table resources on the FPGA. To enable the RAM mapping, the “UseRAM” on page 22-25 parameter is enabled on the masked subsystem blocks that perform the state update and compute the output.

To map the parameters to RAMs irrespective of the threshold, set **Map state space parameters to RAMs** to on.

To see the effect of RAM mapping on the vienna rectifier model:

1. Verify the **UseRAM** parameter setting by running the `hdlget_param` function on the **Multiply Input** and **Multiply State** blocks.

```
MultiplySubsys1 = 'gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem/HDL Algorithm/State Update';
MultiplySubsys2 = 'gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem/HDL Algorithm/Output';
UseRAM1 = hdlget_param([MultiplySubsys1 '/Multiply Input'], 'UseRAM')
UseRAM2 = hdlget_param([MultiplySubsys1 '/Multiply State'], 'UseRAM')
```

```
UseRAM1 =
```

```
'on'
```

```
UseRAM2 =
```

```
'on'
```

2. Enable generation of the resource utilization report.

```
hdlset_param('gmStateSpaceHDL_ViennaRectifier_HDL', 'ResourceReport', 'on')
```

3. Generate HDL code for the implementation model.

```
makehdl('gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem');
```

When you generate code, HDL Coder opens a Code Generation report. The **High-level Resource Report** shows 136 RAMs utilized.

## Summary

Multipliers	294
Adders/Subtractors	4960
Registers	29079
Total 1-Bit Registers	278646
RAMs	136
Multiplexers	45795
I/O Bits	516
Static Shift operators	0
Dynamic Shift operators	644

### Resource Sharing of State Update and Output Computation Blocks

Before you generate HDL code for the HDL Subsystem, you can optimize the algorithm by using the resource sharing optimization in HDL Coder. Resource sharing is an area optimization that identifies multiple functionally equivalent resources and replaces them with a single, equivalent resource. The data is time-multiplexed over the shared resource to perform the same operations. See “Resource Sharing” on page 24-32.

In the HDL implementation model, you can share the masked subsystem blocks that perform state updates and compute the output.

To share these subsystems for the vienna rectifier and generate HDL code:

1. Specify a **SharingFactor** of 2 on the Multiply Input and Multiply State subsystems.

```
hdlset_param([Multiplysubsys1 '/Multiply Input'], 'SharingFactor', 2)
hdlset_param([Multiplysubsys1 '/Multiply State'], 'SharingFactor', 2)
hdlset_param([Multiplysubsys2 '/Multiply Input'], 'SharingFactor', 2)
hdlset_param([Multiplysubsys2 '/Multiply State'], 'SharingFactor', 2)
```

2. Enable generation of the optimization report

```
hdlset_param('gmStateSpaceHDL_ViennaRectifier_HDL', 'OptimizationReport', 'on')
```

3. Generate HDL code for the HDL Subsystem block in the implementation model.

```
makehdl('gmStateSpaceHDL_ViennaRectifier_HDL/HDL Subsystem');
```

When you generate code, HDL Coder opens a Code Generation report. To see the status of the resource sharing optimization, click the **Streaming and Sharing** section of the report. This sharing group shows the dot products that the optimization shared. When you click the **High-level Resource Report**, you see that the consumption of adders, multipliers, and registers have decreased.

**Subsystem: Multiply Input**

SharingFactor: 2

[Highlight shared resources and diagnostics](#)

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1			2	<a href="#">dot_product_5</a>	
2			2	<a href="#">dot_product_5</a>	
3			2	<a href="#">dot_product_5</a>	
4			2	<a href="#">dot_product_5</a>	
5			2	<a href="#">dot_product_5</a>	
6			2	<a href="#">dot_product_5</a>	
7			2	<a href="#">dot_product_5</a>	
8			2	<a href="#">dot_product_5</a>	
9			2	<a href="#">dot_product_5</a>	
10			2	<a href="#">dot_product_5</a>	
11			2	<a href="#">dot_product_5</a>	
12			2	<a href="#">dot_product_5</a>	
13			2	<a href="#">dot_product_5</a>	

**See Also****Functions**

checkhdl | makehdl

**More About**

- “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 32-2
- “Speed and Area Optimizations in HDL Coder” on page 24-2
- “Generate HDL Code for Simscape Models” on page 32-9

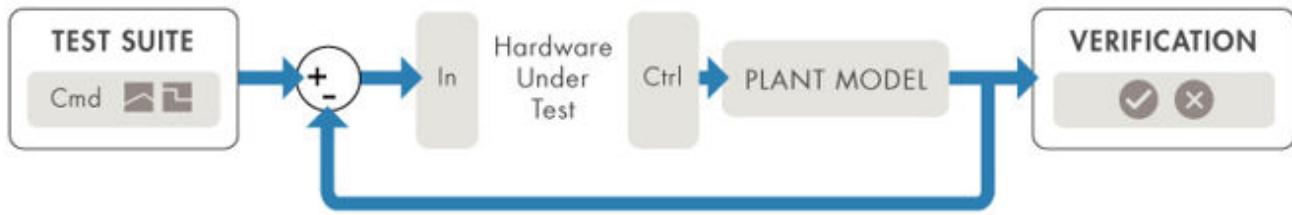
# Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model

This example shows how to generate a Simulink® Real-Time Interface subsystem for a Simscape™ two-level converter plant model. You can then deploy the interface model on the Speedgoat FPGA IO module. This example uses the Speedgoat IO334-325k module.

## Real-Time Simulation

Simulating the plant model on the FPGA provides:

- **Real-time Simulation:** Hardware-in-the-loop provides real-time simulation of your Simscape plant model on the target hardware.



- **Hardware Acceleration:** Accelerated simulation of complex physical systems on hardware while reconfigurable FPGAs provide rapid prototyping.

To use the workflow:

- 1 Develop the Simscape model and convert it into an implementation model by using the Simscape HDL Workflow Advisor.
- 2 Generate HDL code and deploy the code to the Speedgoat I/O module by using the HDL Workflow Advisor.

## Setup and Configuration

Before deploying your algorithm on the Speedgoat IO module:

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).

3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

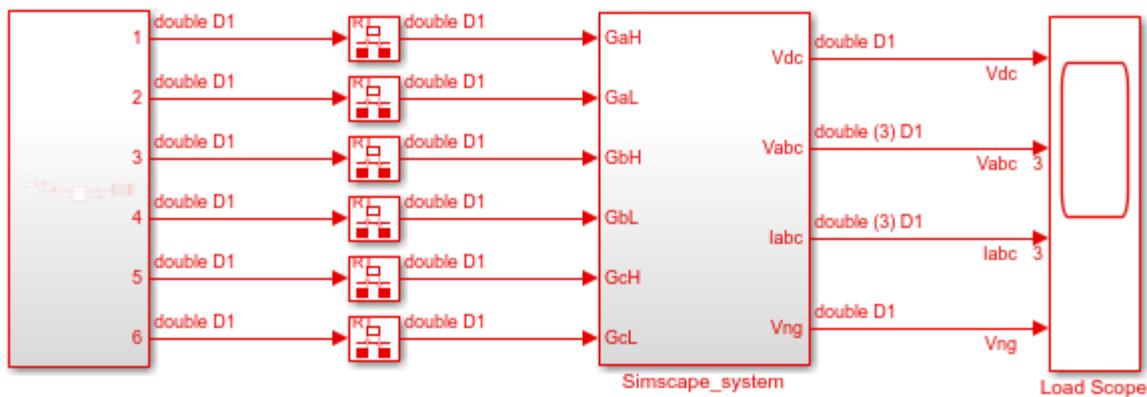
## Two-Level Converter Ideal Model

To open this model, enter:

```
open_system('sschdlexTwoLevelConverterIdealExample')
```

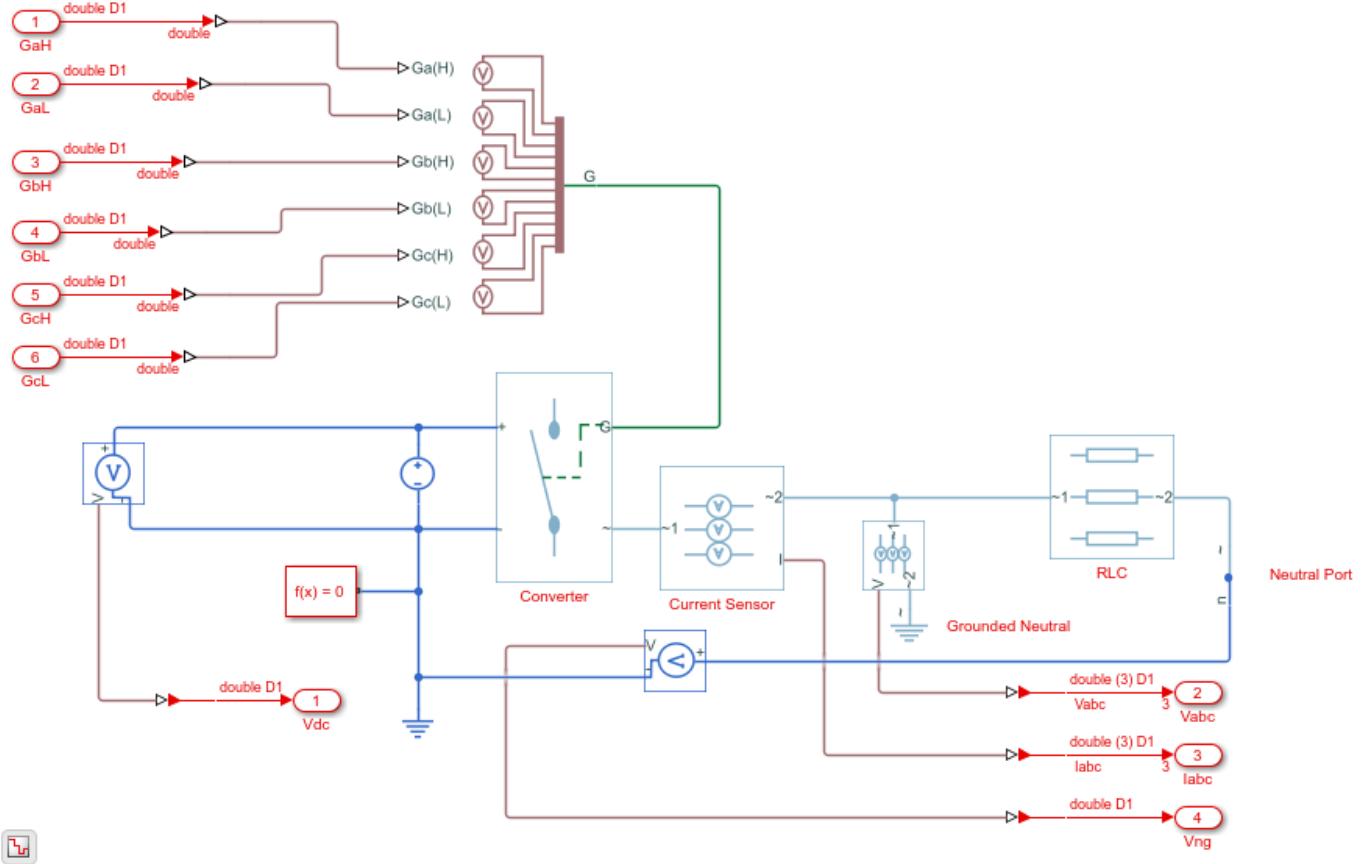
Save this model locally as TwoLevelConverter\_HDL.slx to run this workflow.

```
open_system('TwoLevelConverter_HDL')
set_param('TwoLevelConverter_HDL','SimulationCommand','update')
```



Copyright 2020 The MathWorks, Inc.

```
open_system('TwoLevelConverter_HDL/Simscape_system')
```

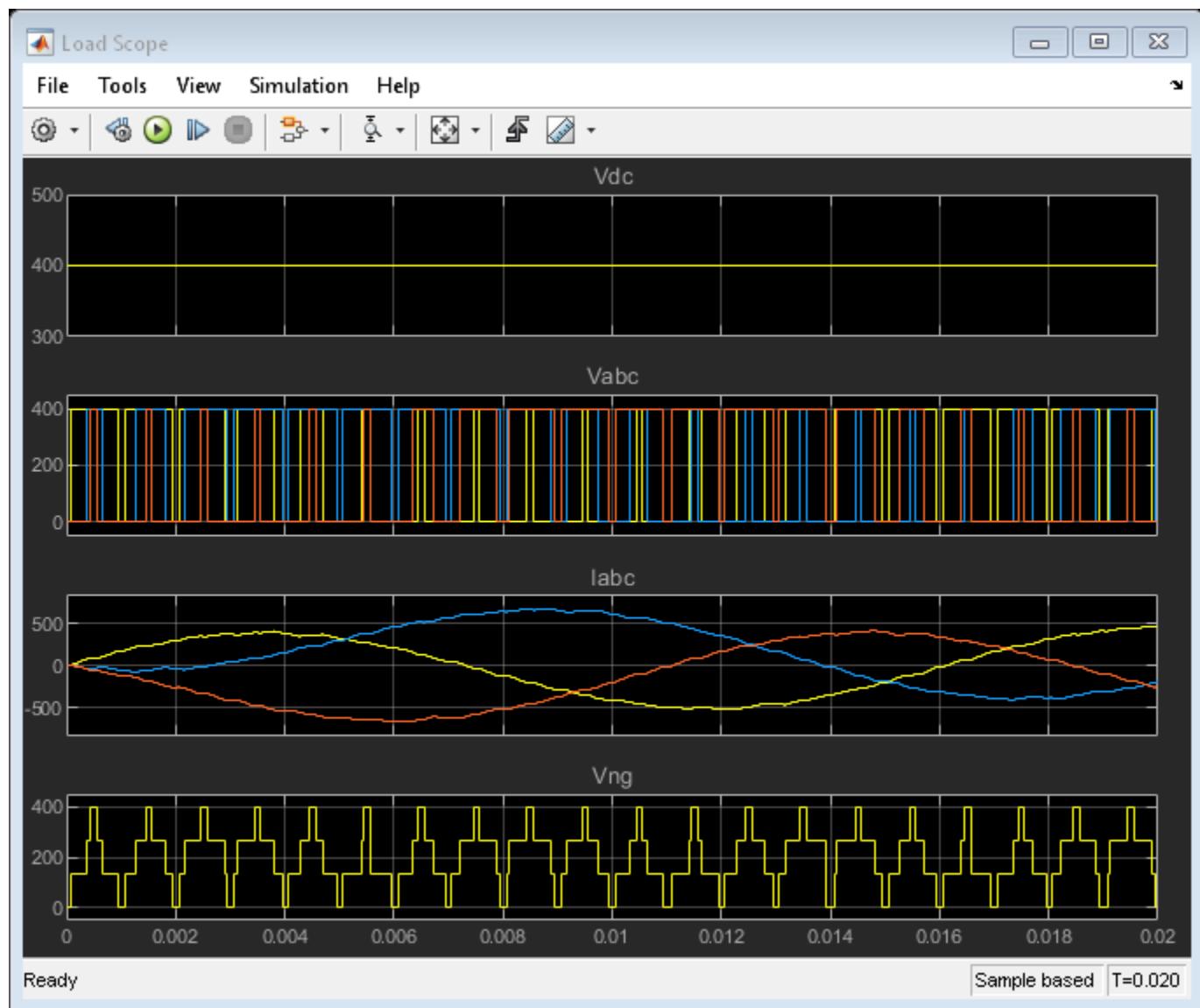


The Simscape subsystem receives six-switch controlling pulses as input. The Simscape subsystem acts as a generator that uses a two-level, carrier-based PWM method to:

- 1** Sample a reference wave.
- 2** Compare the sample to a triangular carrier wave.
- 3** Generate a switch-on pulse if a sample is higher than the carrier signal or a switch-off pulse if a sample is lower than the carrier wave.

Simulate the model.

```
sim('TwoLevelConverter_HDL')
open_system('TwoLevelConverter_HDL/Load Scope')
```



### Generate HDL Implementation Model

To generate an implementation model, use the Simscape HDL Workflow Advisor. Run the `sschdladvisor` function for your model:

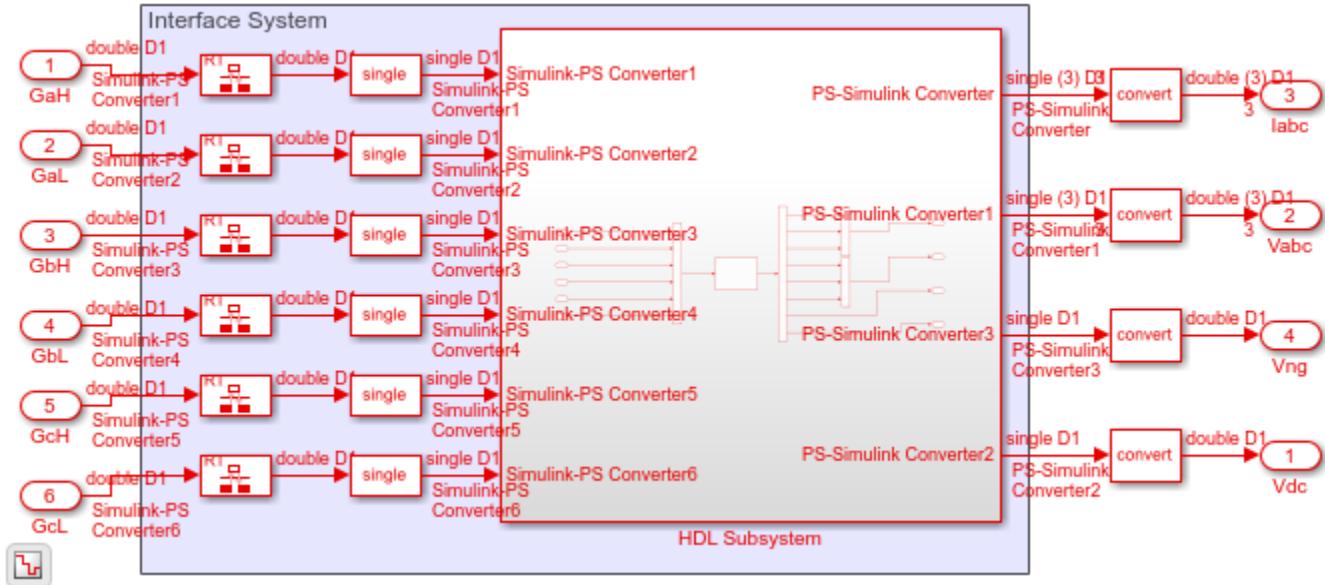
```
sschdladvisor('TwoLevelConverter_HDL')
```

```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(TwoLevelConverter_HDL)">TwoLevelC
```

To generate the implementation model, in the Simscape HDL Workflow Advisor, keep the default settings for the tasks, and then run the tasks. You see a link to the model in the **Generate implementation model** task. This model has the same name as the original model prefixed with `gmStateSpaceHDL`.

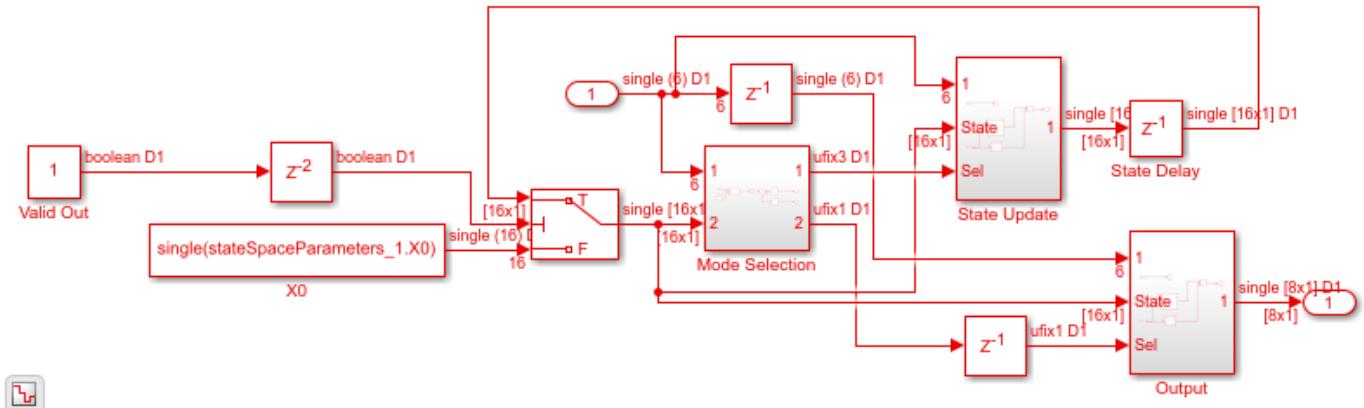
To open the implementation model, enter:

```
load_system('gmStateSpaceHDL_TwoLevelConverter_HDL')
open_system('gmStateSpaceHDL_TwoLevelConverter_HDL/Simscape_system')
set_param('gmStateSpaceHDL_TwoLevelConverter_HDL','SimulationCommand','update')
```



The implementation model replaces the Simscape subsystem with the HDL algorithm that performs the state-space computations. When you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations. From and Goto blocks inside this subsystem provide the same input as that of the original model to the HDL Subsystem.

```
open_system('gmStateSpaceHDL_TwoLevelConverter_HDL/Simscape_system/HDL_Subsystem/HDL_Algorithm')
```



## HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.

- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

To open the HDL Workflow Advisor for a subsystem inside the model, use the `hdladvisor` function.

```
load_system('sschdlexTwoLevelConverterIgbtExample')
hdladvisor('sschdlexTwoLevelConverterIgbtExample/Simscape_system')
```

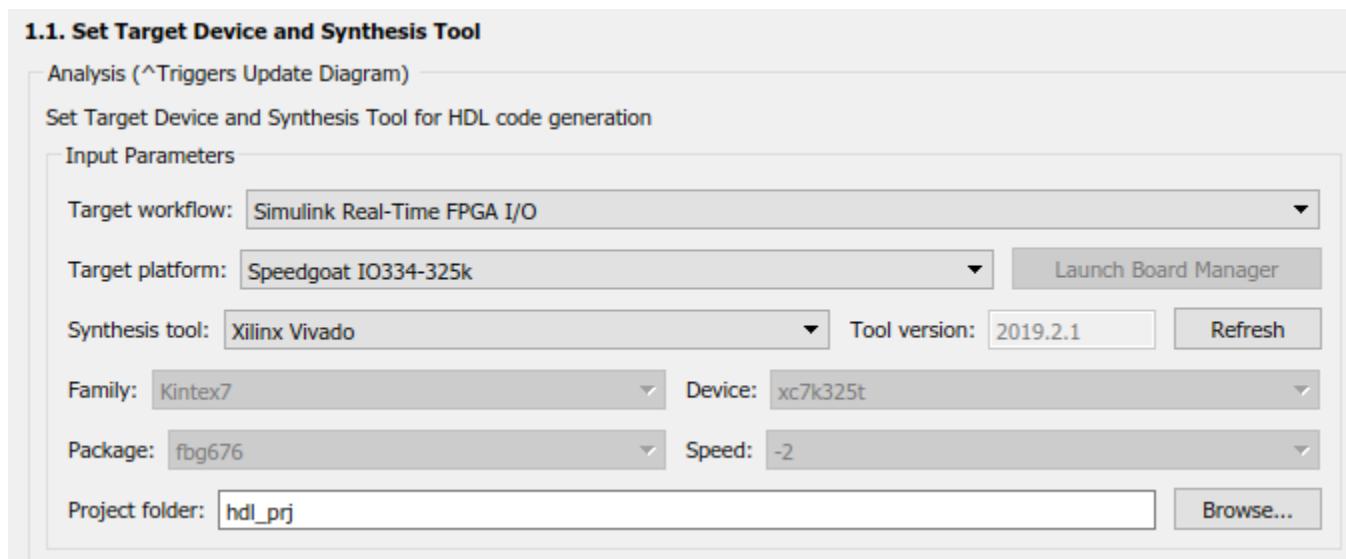
The left pane contains folders that represent a group of related tasks. Expanding the folders and selecting a task displays information about that task in the right pane. The right pane contains simple controls for running the task to advanced parameters and option settings that control code and test bench generation. To learn more about each task, right-click that task, and select **What's This?**. See “Getting Started with the HDL Workflow Advisor” on page 31-6.

### Deploy Two Level Ideal Converter Model to Speedgoat IO334-325K Module

1. Open the HDL Workflow Advisor for the implementation model.

```
hdladvisor('gmStateSpaceHDL_TwoLevelConverter_HDL/Simscape_system/HDL_Subsystem')
```

2. In **Set Target Device and Synthesis Tool** task, specify **Target workflow** as Simulink Real-Time FPGA I/O and **Target platform** as Speedgoat IO334-325K



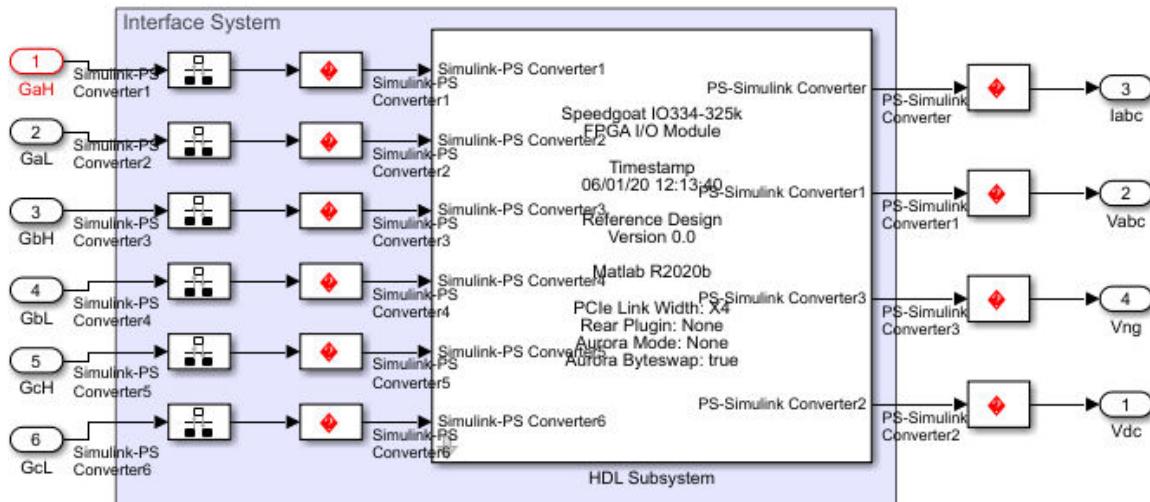
3. Run the **Set Target Reference Design** task, select a value of x4 for the parameter **PCIe lanes**, and select **Run This Task**.
4. In **Set Target Interface** task, map the input and output single data type ports to **PCIe Interface** and select **Run This Task**.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
Simulink-PS Converte...	Import	single	PCIe Interface	x"100"	Options...
Simulink-PS Converte...	Import	single	PCIe Interface	x"104"	Options...
Simulink-PS Converte...	Import	single	PCIe Interface	x"108"	Options...
Simulink-PS Converte...	Import	single	PCIe Interface	x"10C"	Options...
Simulink-PS Converte...	Import	single	PCIe Interface	x"110"	Options...
Simulink-PS Converte...	Import	single	PCIe Interface	x"114"	Options...
PS-Simulink Converter	Outport	single (3)	PCIe Interface	x"120"	
PS-Simulink Converte...	Outport	single (3)	PCIe Interface	x"140"	
PS-Simulink Converte...	Outport	single	PCIe Interface	x"118"	
PS-Simulink Converte...	Outport	single	PCIe Interface	x"11C"	

5. Right-click **Generate RTL Code and IP Core** task and select **Run to Selected Task**. As the model uses vector data types, the **Generate RTL Code and IP Core** fails because the **ScalarizePorts** property must be set to **dutlevel**. Click the link to change this setting and rerun the task.

6. Run the workflow to the **Generate Simulink Real-Time interface** task. In **Create Project** task, you can open the Vivado project and see the implemented design. After the **Generate Simulink Real-Time interface** task passes, click the link to open the Simulink Real-Time Interface Model.



## Export HDL Workflow to Script

For rapid prototyping, you can export the HDL Workflow Advisor settings to a script. The script is a MATLAB® file that you can run from the command line. You can then modify and run the script, or import the settings into the HDL Workflow Advisor User Interface.

To export an HDL Workflow script, after you run the tasks in the Advisor, select **File > Export to Script**. For this example, when you export to script, this file shows the settings you saved.

```
edit('hdlworkflow_slrt.m')
```

To import an HDL Workflow script, in the HDL Workflow Advisor, select **File > Import from Script**. Select the script file and click **Open**. The HDL Workflow Advisor updates the tasks with the imported script settings.

For an example that shows how to run the real-time application by deploying the FPGA bitstream, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 32-90.

## See Also

### Functions

`checkhdl | makehdl`

## More About

- “Run HDL Workflow with a Script” on page 31-53
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 41-93
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- Speedgoat I/O Examples

# Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script

This example shows how to deploy a Simscape™ buck converter model to a Speedgoat IO334 Simulink®-programmable I/O module and then run the model in real-time at a sample step size as small as 1 microsecond. The example uses a DCDC converter topology to show how to prepare your power electronic converter model for hardware in the loop (HIL) simulation on a Speedgoat real-time target machine.

To use this workflow:

- 1** Convert your model into an HDL-compatible implementation model by using the Simscape HDL Workflow Advisor
- 2** Generate HDL code and FPGA bitstream for the IO334 module by using the HDL Workflow Advisor.
- 3** Deploy the real-time model to the Speedgoat real-time target machine by using Simulink Real-Time.

The model runs at a sample time of 1us till HDL code generation and then runs at 50us on the CPU in real time. To generate HDL code and FPGA bitstream, the example shows how to run the HDL workflow script from the command line. For an example that shows how you can use the Workflow Advisor User Interface to run this workflow, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 32-90.

## Setup and Configuration

Before deploying your algorithm on the Speedgoat IO module:

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).

3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See Install Speedgoat HDL Coder Integration Packages.

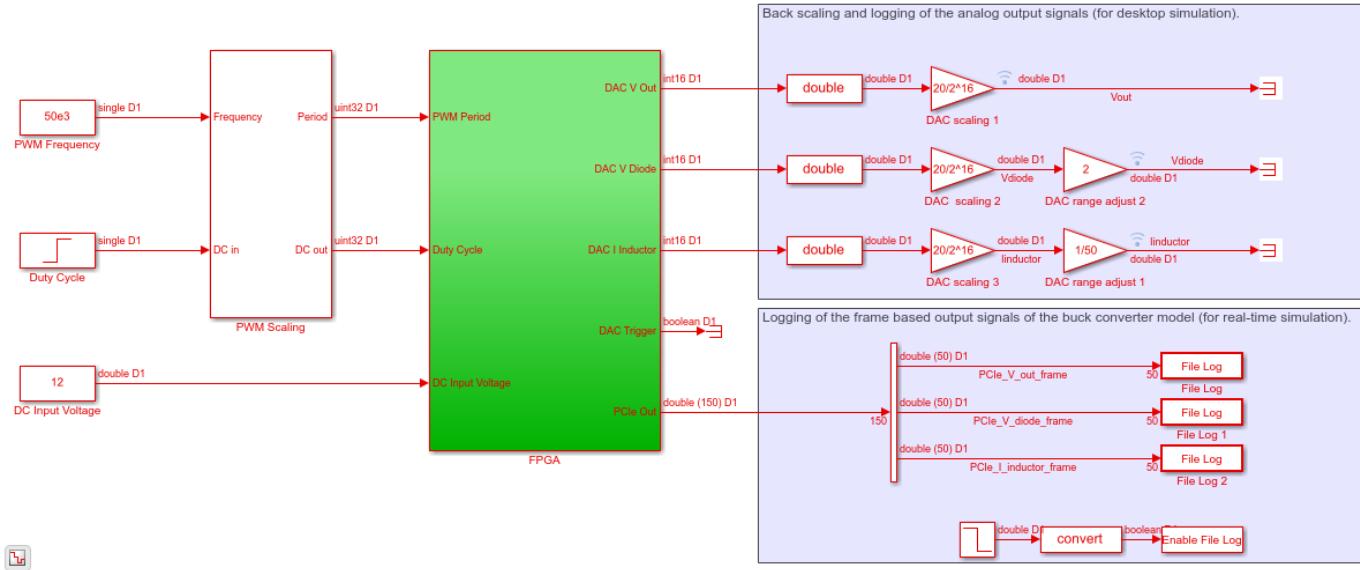
## Buck Converter Model

To see the buck converter model, run this command:

```
open_system('sschdlexBuckConverterExample')
```

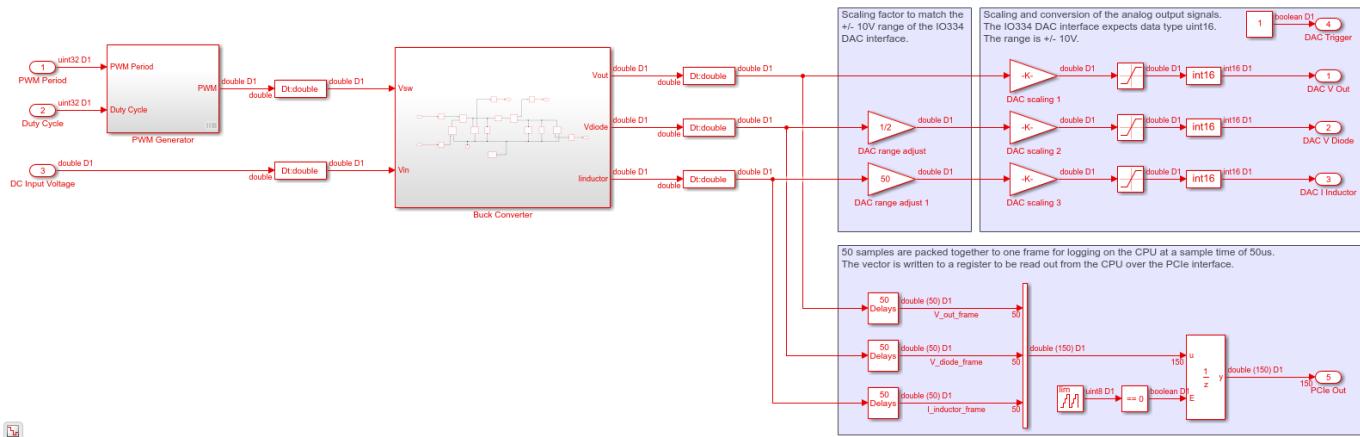
This model is modified for real-time deployment and saved as `sschdlex_I0334_BuckConverter`. The model has been partitioned into parts that run on the FPGA and parts that run on CPU. Parts inside the green FPGA subsystem run on the FPGA. Parts outside this subsystem run on the CPU in real time.

```
open_system('sschdlex_I0334_BuckConverter')
set_param('sschdlex_I0334_BuckConverter', 'SimulationCommand', 'Update')
```



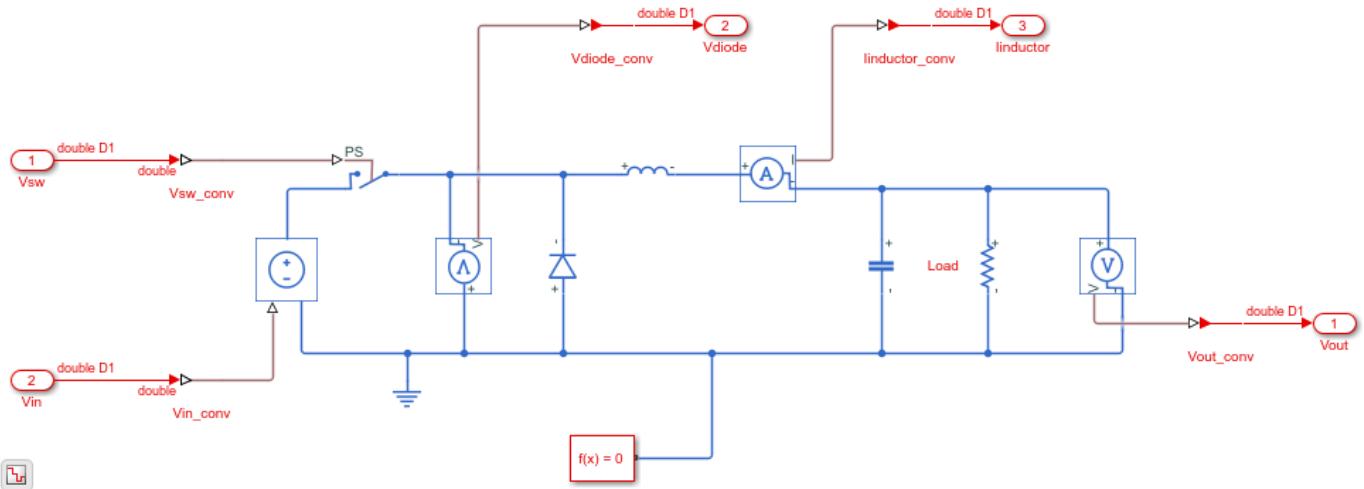
You generate VHDL code for blocks that are inside the green FPGA subsystem that contains the PWM generator and buck converter. The code is then deployed to the FPGA on board the IO334 module. The outputs of the subsystem are mapped to DAC interfaces. The output signals from the Buck Converter subsystem are scaled within a 10V range and converted to use uint16 data types. 50 samples are packed together to one frame to log the output signals on the CPU.

```
open_system('sschdlex_I0334_BuckConverter/FPGA')
```



To see the buck converter model, double-click the Buck Converter subsystem. The buck converter is a power converter model that steps down the input voltage at the output. The voltage at the output is stepped down by the duty cycle, D. The output voltage,  $V_{out}$ , is calculated as  $V_{in}/D$

```
open_system('sschdlex_I0334_BuckConverter/FPGA/Buck_Converter')
```



### Run Desktop Simulation of Simscape model

The Simulation input is a duty cycle step wave from  $0.2$  to  $0.8$ . The input signals that include the DC input voltage, PWM frequency, and duty cycle are generated on the top level of the model. The sample time for the Simscape model is set to  $1\mu s$ . Signal logging is enabled on the top level of the model.

```

sim('sschdlex_I0334_BuckConverter')

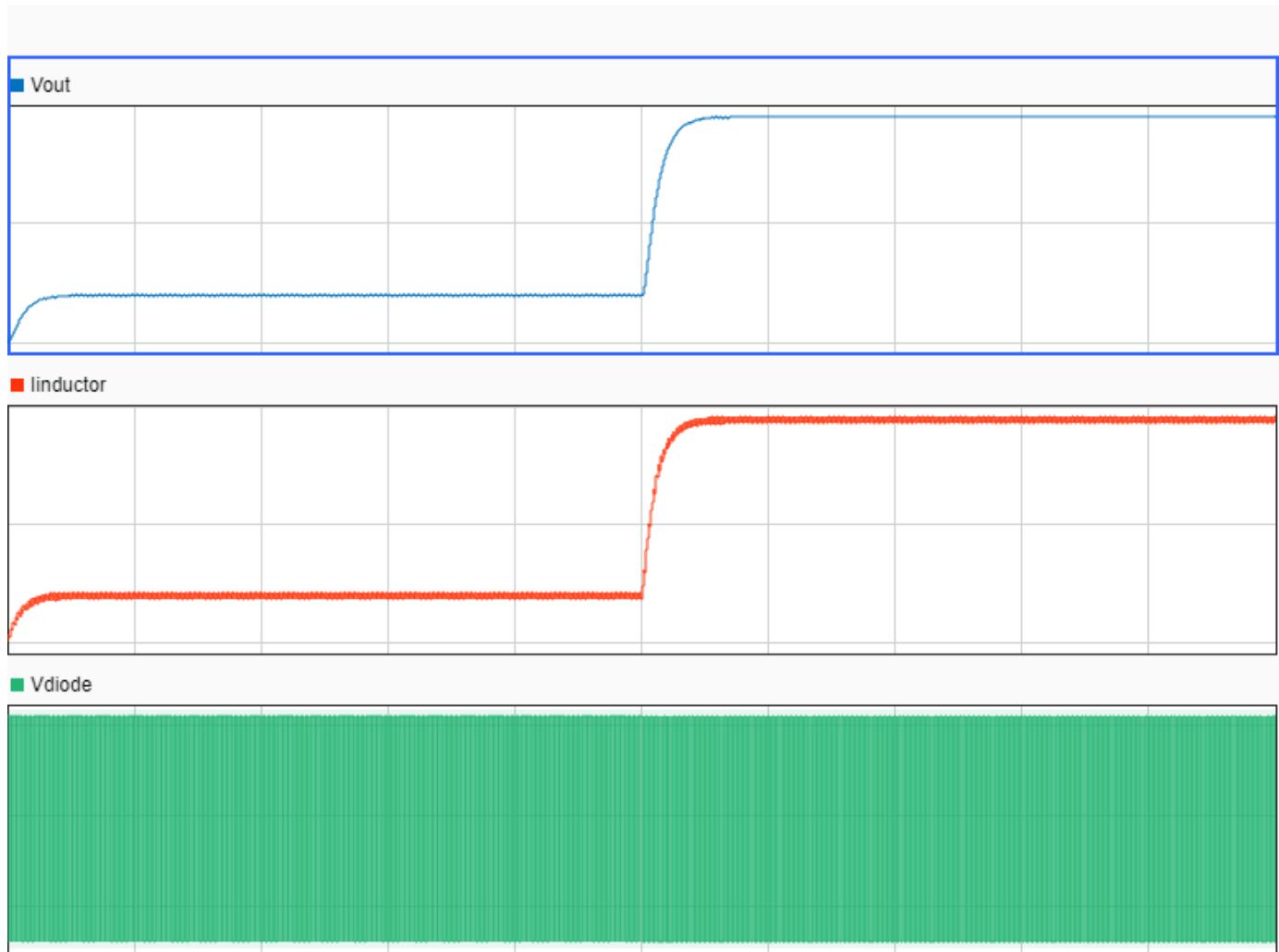
% Display the buck converter output signals in SDI
Simulink.sdi.clearAllSubPlots
Simulink.sdi.setSubPlotLayout(3,1);

allIDs2 = Simulink.sdi.getAllRunIDs;
runID2 = allIDs2(end);
run2 = Simulink.sdi.getRun(runID2);
run2.name = 'Simscape Desktop Simulation';

run2.getAllSignals;
plotOnSubPlot(run2.getSignalsByName('Vout'),1,1,true);
plotOnSubPlot(run2.getSignalsByName('Iinductor'),2,1,true);
plotOnSubPlot(run2.getSignalsByName('Vdiode'),3,1,true);

Simulink.sdi.view;

```



### Generate HDL Implementation Model

For HDL code generation compatibility, you run the Simscape HDL Workflow Advisor to generate an HDL implementation model.

The Simscape solver is set to run for two iterations at each sample step. The Simscape HDL Workflow Advisor uses the solver settings in the next step for deterministic real-time behaviour.

```
set_param('sschdlex_I0334_BuckConverter/FPGA/Buck Converter/Solver Configuration','DoFixedCost',  
set_param('sschdlex_I0334_BuckConverter/FPGA/Buck Converter/Solver Configuration','MaxNonlinIter',
```

To open the Advisor, run the `sschdladvisor` function for your model:

```
sschdladvisor('sschdlex_I0334_BuckConverter')
```

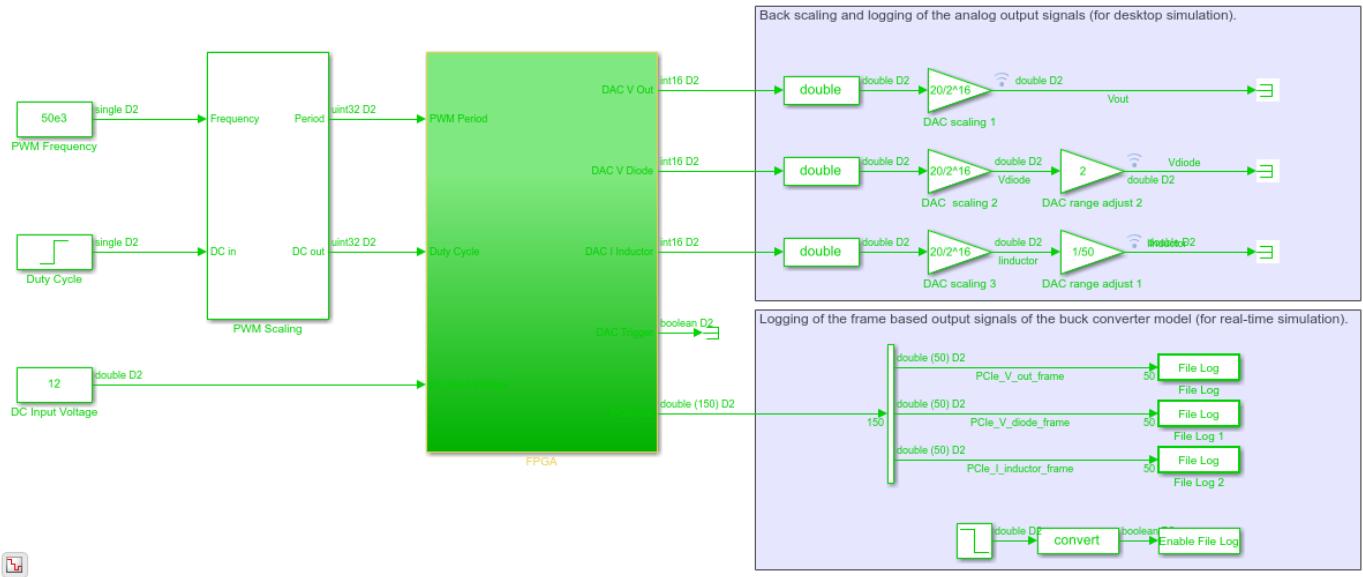
```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(sschdlex_I0334_BuckConverter)">sschdlex_I0334_BuckConverter</a>
```

To generate the implementation model, in the Simscape HDL Workflow Advisor, keep the default settings for the tasks, and then run the tasks. Run the tasks in the Advisor by clicking the **Run all** button. You see a link to the model in the **Generate implementation model** task. This model has the same name as your original model with the prefix `gmStateSpaceHDL_`.

## Prepare Implementation Model for HDL Code Generation

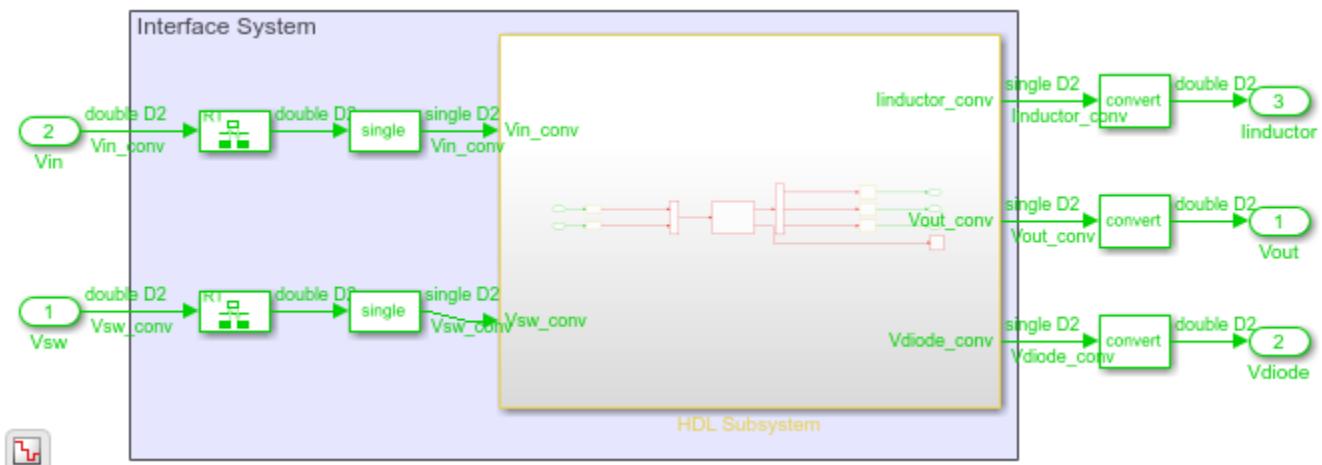
To open the implementation model, click the link in the **Generate implementation model** task.

```
open_system('gmStateSpaceHDL_sschedlex_I0334_BuckConverte');
set_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'SimulationCommand', 'Update')
```



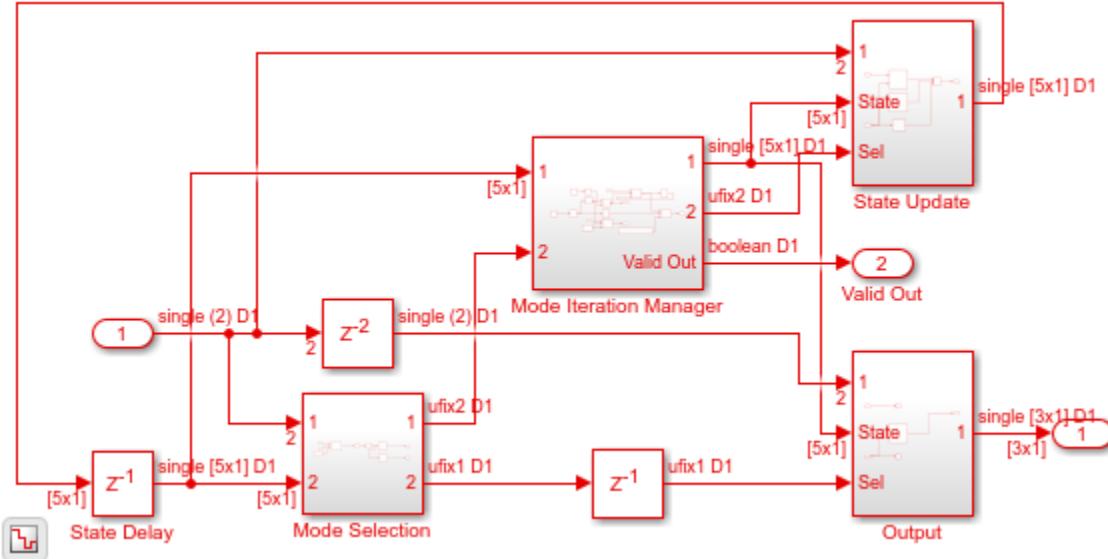
The model contains a switched linear Simulink replacement of the original buck converter model. You see that the Simscape model was replaced.

```
open_system('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Buck_Converter');
```



The implementation model replaces the Simscape subsystem with the HDL-compatible algorithm that performs the state-space computations. When you navigate inside this subsystem, you see several delays, adders, and Matrix Multiply blocks that model the state-space equations. From and Goto blocks inside this subsystem provide the same input as that of the original model to the **HDL Subsystem**.

```
open_system('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Buck Converter/HDL Subsystem/HDL A')
```



The data type of the buck converter output signals is set to single precision floating point for HDL code generation.

```
set_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Signal Specification','OutDataTypeStr');
set_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Signal Specification1','OutDataTypeStr');
set_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Signal Specification2','OutDataTypeStr');
set_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Signal Specification3','OutDataTypeStr');
set_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Signal Specification4','OutDataTypeStr');
```

### Run Desktop Simulation of HDL Implementation Model and Validate HDL Algorithm

You can simulate the switched linear state-space model of the buck converter in Simulink and display the signals in Simulation Data Inspector. The comparison of the runs show that the numeric results match.

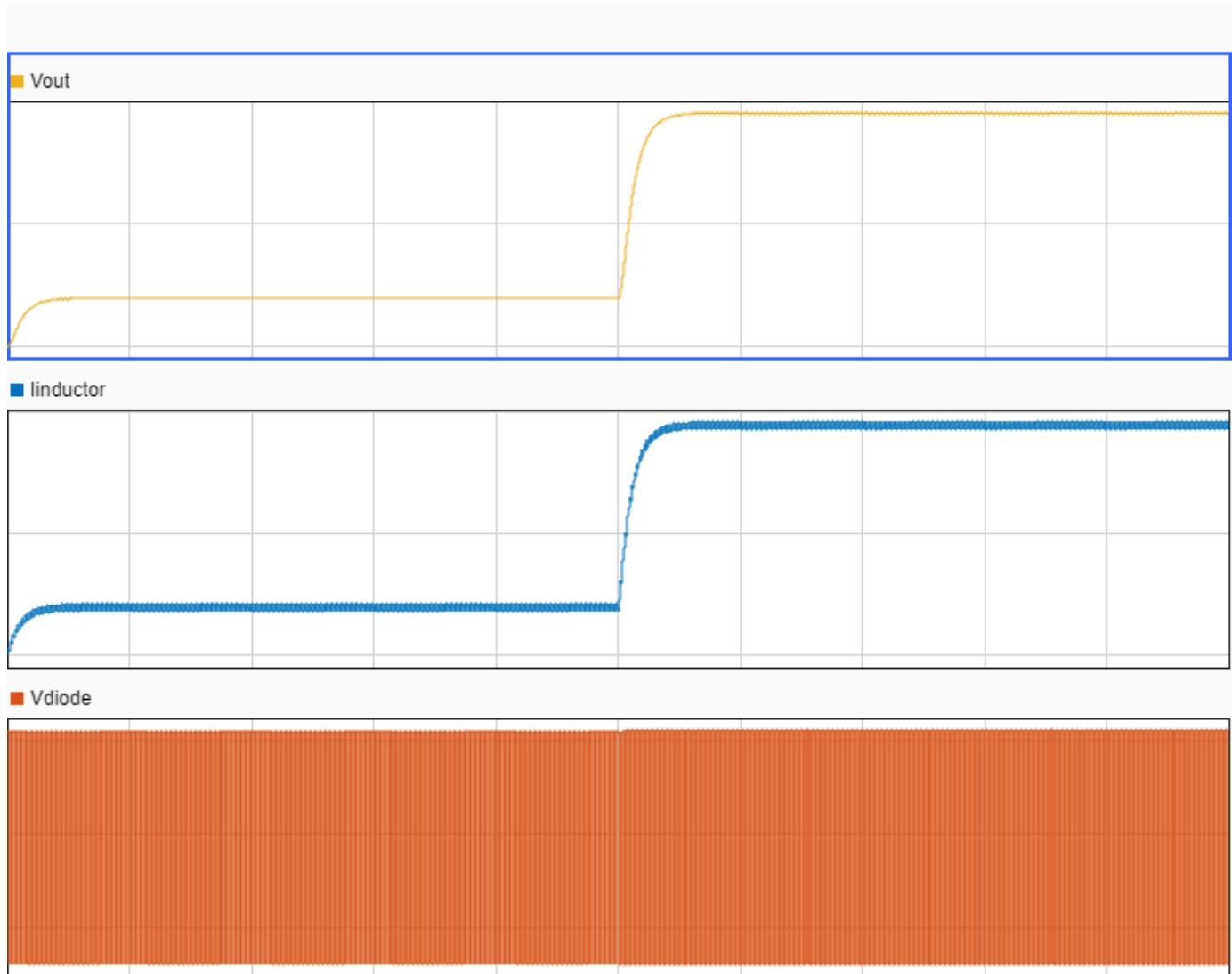
Simulate the HDL implementation model.

```
sim('gmStateSpaceHDL_sschedlex_I0334_BuckConverte')
%
% Display output signals of buck converter Simscape model in Simulation
% Data Inspector.
Simulink.sdi.clearAllSubPlots
Simulink.sdi.setSubPlotLayout(3,1);

allIDs2 = Simulink.sdi.getAllRunIDs;
runID2 = allIDs2(end);
run2 = Simulink.sdi.getRun(runID2);
run2.name = 'HDL Desktop Simulation';

run2.getAllSignals;
plotOnSubPlot(run2.getSignalsByName('Vout'),1,1,true);
plotOnSubPlot(run2.getSignalsByName('Iinductor'),2,1,true);
plotOnSubPlot(run2.getSignalsByName('Vdiode'),3,1,true);

Simulink.sdi.view;
```



To verify that the HDL implementation model matches the original Simscape model, generate a state-space validation model. In the **Generate implementation model** task, select the **Generate validation logic for the implementation model** check box and then run this task. Simulating the model does not display assertions, which indicates that the numeric results match. See “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97.

### HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis, timing analysis, and deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

You run the Advisor for the FPGA subsystem in your model. To open the HDL Workflow Advisor for the subsystem inside the model, use the `hdladvisor` function. For example:

```
hdladvisor('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA')
```

To learn about the tasks in the Advisor, right-click that task, and select **What's This?**. See “Getting Started with the HDL Workflow Advisor” on page 31-6.

### Run Workflow Script to Generate Simulink Real-Time Interface Model

For rapid prototyping, export the HDL Workflow Advisor settings to a script. The script is a MATLAB® file that you run from the command line. You can modify and run the script, or import the settings into the HDL Workflow Advisor User Interface. See “Run HDL Workflow with a Script” on page 31-53.

This example shows how to run the HDL Workflow script. To generate a Simulink Real-Time Interface model, open and run this MATLAB script.

```
edit('hdlworkflow_buck_I0334')

%% -----
% This script contains the model, target settings, interface mapping, and
% the Workflow Configuration settings for generating HDL code for the HDL
% implementation model generated for the buck converter model, and for
% deploying the code to the FPGA on board the Speedgoat I0334-325K module.
%% -----


%% Set Parameters for HDL Code Generation

% Model HDL parameters
%
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'FloatingPointTargetConfiguration', 1);
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'HDLSubsystem', 'gmStateSpaceHDL_sschedlex_I0334_BuckConverte');
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'Oversampling', 100);
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'ScalarizePorts', 'DUTLevel');
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'TargetFrequency', 200);
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'Workflow', 'Simulink Real-Time FPGA');
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'TargetPlatform', 'Speedgoat I0334-325K');
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte', 'AdaptivePipelining', 'off');

%% Map DUT Ports to Target Interfaces

% Input port mapping
%
% All input signals to the "FPGA" subsystem are mapped to the PCIe
% interface. I.e. these signals will be transferred from the CPU of the
% real-time target machine to the I0334 FPGA over the PCIe bus.

hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/PWM Period', 'I0Interface', 'PCIe');
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/Duty Cycle', 'I0Interface', 'PCIe');
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DC Input Voltage', 'I0Interface', 'PCIe');

% Output port mapping
%
% The scaled output signals of the converter are mapped to the analog
% output interface of the I0334.
```

```

hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC V Out', 'IOInterface', 'I0334'
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC V Out', 'IOInterfaceMapping', 'I0334'

hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC V Diode', 'IOInterface', 'I0334'
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC V Diode', 'IOInterfaceMapping', 'I0334'

hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC I Inductor', 'IOInterface', 'I0334'
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC I Inductor', 'IOInterfaceMapping', 'I0334'

hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC Trigger', 'IOInterface', 'I0334'
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/DAC Trigger', 'IOInterfaceMapping', 'I0334

% The signal frames are mapped to PCIe registers. They are read by the CPU for data logging.
hdlset_param('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA/PCIe Out', 'IOInterface', 'PCIe Interface')

%% Workflow Configuration Settings

% HDL Workflow Advisor is opened with the following settings.
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink Real-Time')

% Specify the top level project directory.
hWC.ProjectFolder = 'hdl_prj';
hWC.ReferenceDesignToolVersion = '2019.2';

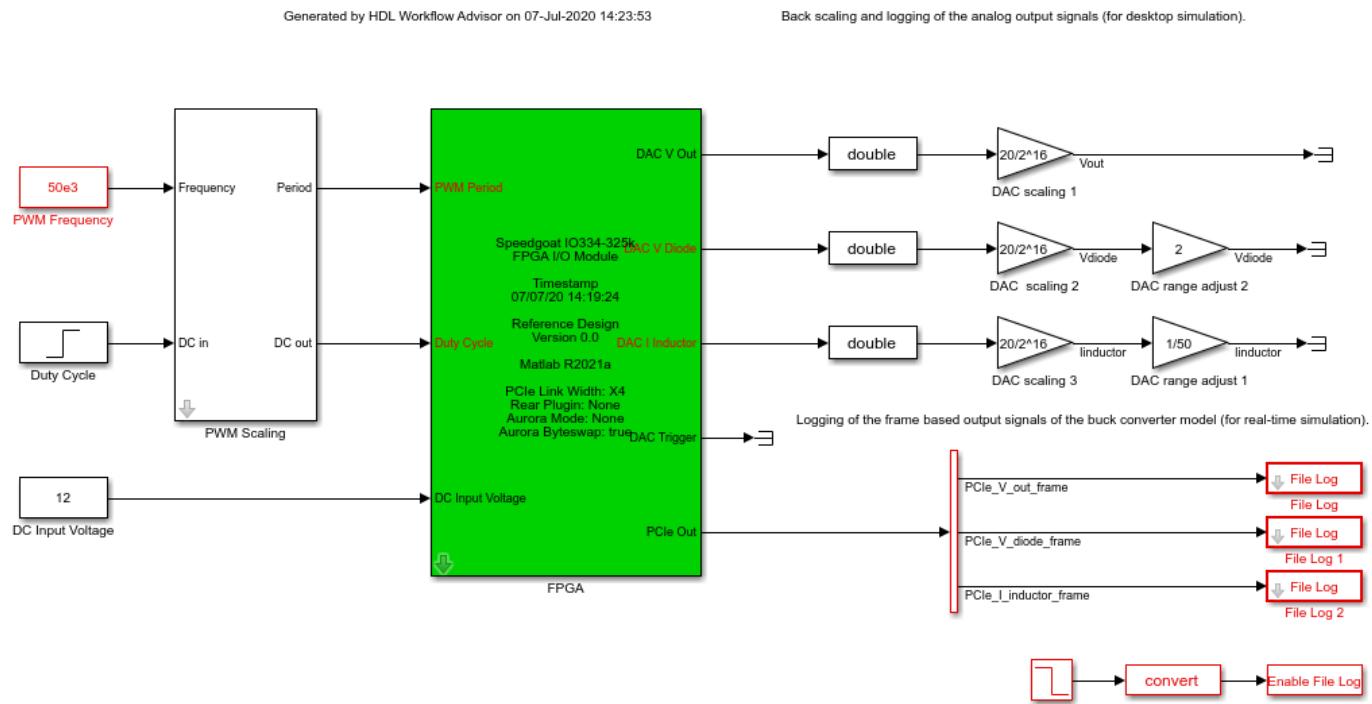
% Set Workflow tasks to run.
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

%% Run the Workflow
hdlcoder.runWorkflow('gmStateSpaceHDL_sschedlex_I0334_BuckConverte/FPGA', hWC);

```

### Prepare Simulink Real-Time Interface Model for Real-Time Simulation

Running the workflow script generates RTL code and IP core, creates a Vivado project, builds the FPGA bitstream, and then generates the Simulink Real-Time Interface model.



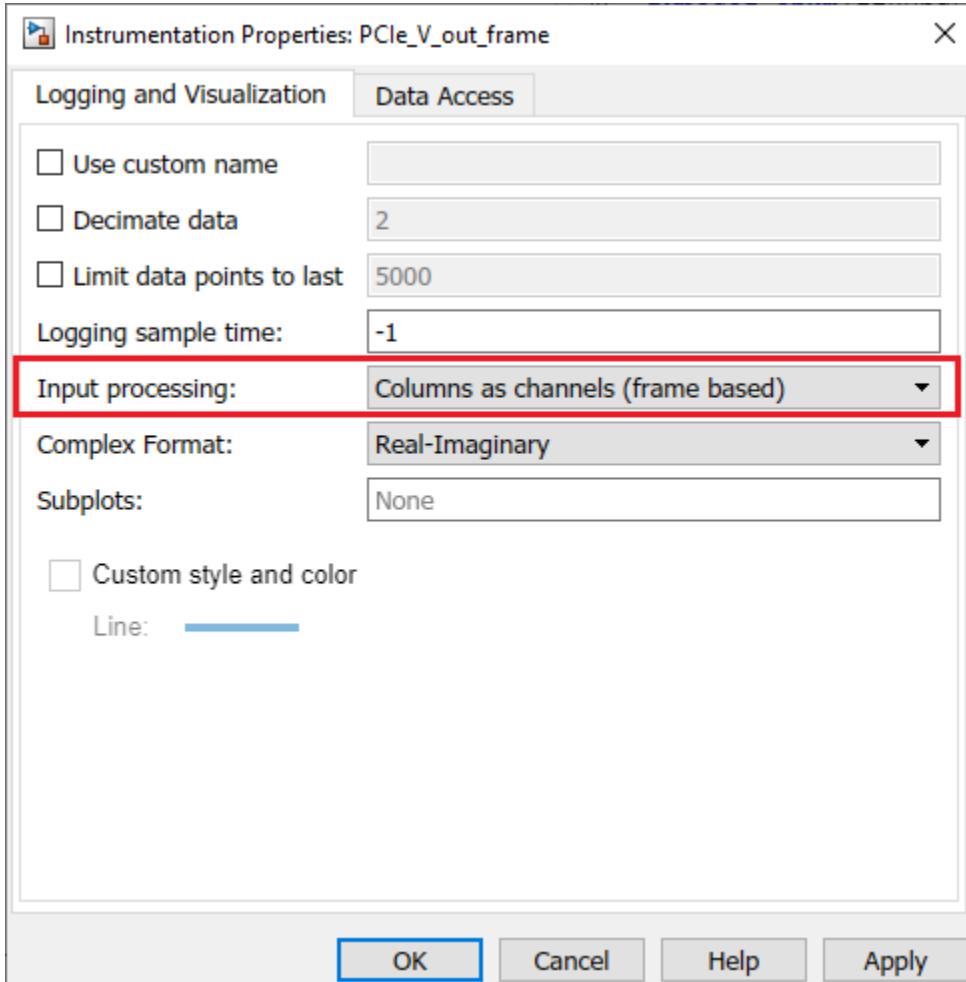
Before deploying the model to the Speedgoat real-time target machine:

1. Set the sample time for all blocks running on the CPU of the Speedgoat real-time target machine to 50us (including driver blocks for the FPGA).

```
generated_model = gcs;
Ts = 50e-6;
set_param([generated_model,'/FPGA'],'ts','Ts');
```

2. Set the Simulation Data Inspector setting **Input processing** to **Columns as channels (frame based)** for the signals PCIe\_V\_out\_frame, PCIe\_V\_diode\_frame and PCIe\_I\_inductor\_frame. Inside the mask of the File Log blocks, right-click the logging symbol and navigate to the Instrumentation Properties dialog box. To make the logging signal appear, you might have to update the model.

```
set_param(generated_model, 'SimulationCommand', 'update');
```



Alternatively, you can set signal logging to frame based mode by using these commands.

```
Simulink.sdi.setSignalInputProcessingMode([generated_model,'/File Log/Demux'], 1, 'frame');
Simulink.sdi.setSignalInputProcessingMode([generated_model,'/File Log 1/Demux'], 1, 'frame');
Simulink.sdi.setSignalInputProcessingMode([generated_model,'/File Log 2/Demux'], 1, 'frame');
```

### Connect to Target Machine and Run Real-Time Simulation

The model can now be deployed to the Speedgoat real-time target machine. The buck converter model is automatically loaded to the FPGA on the IO334.

Connect to the Speedgoat real-time target machine.

```
tg = slrealtime;
tg.connect;
```

Build and download the model to the target machine.

```
rtwbuild(generated_model);
tg.load(generated_model);
```

Start the model execution.

```
tg.start;
pause(10);
```

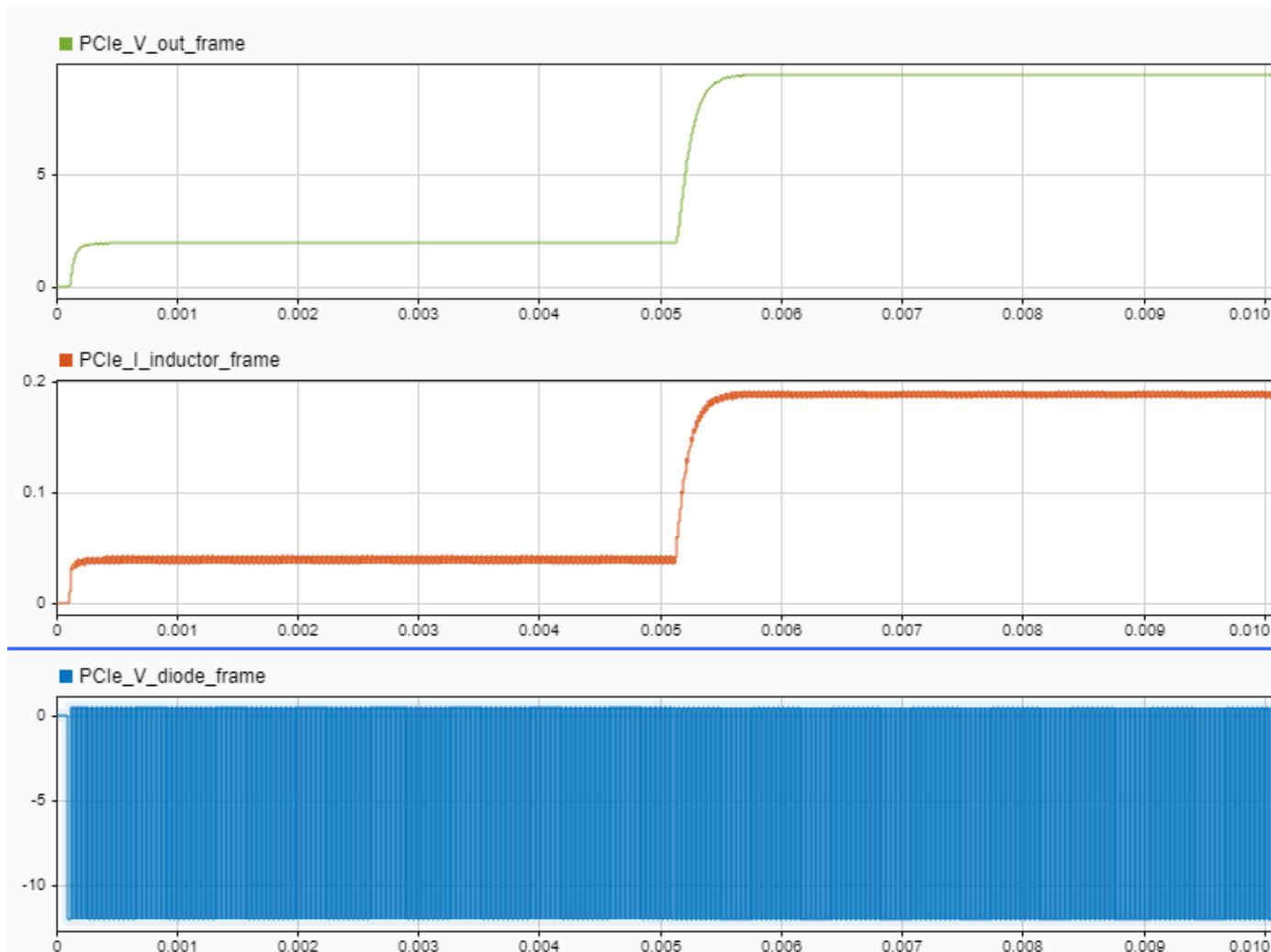
The file logging blocks store the signals on the SSD of the target-machine. The data is automatically uploaded to the host computer once the model is stopped. The data is visualized in Simulation Data Inspector. You can verify that the results of the real-time simulation matches the original Simscape model.

```
Simulink.sdi.setSubPlotLayout(3,1);

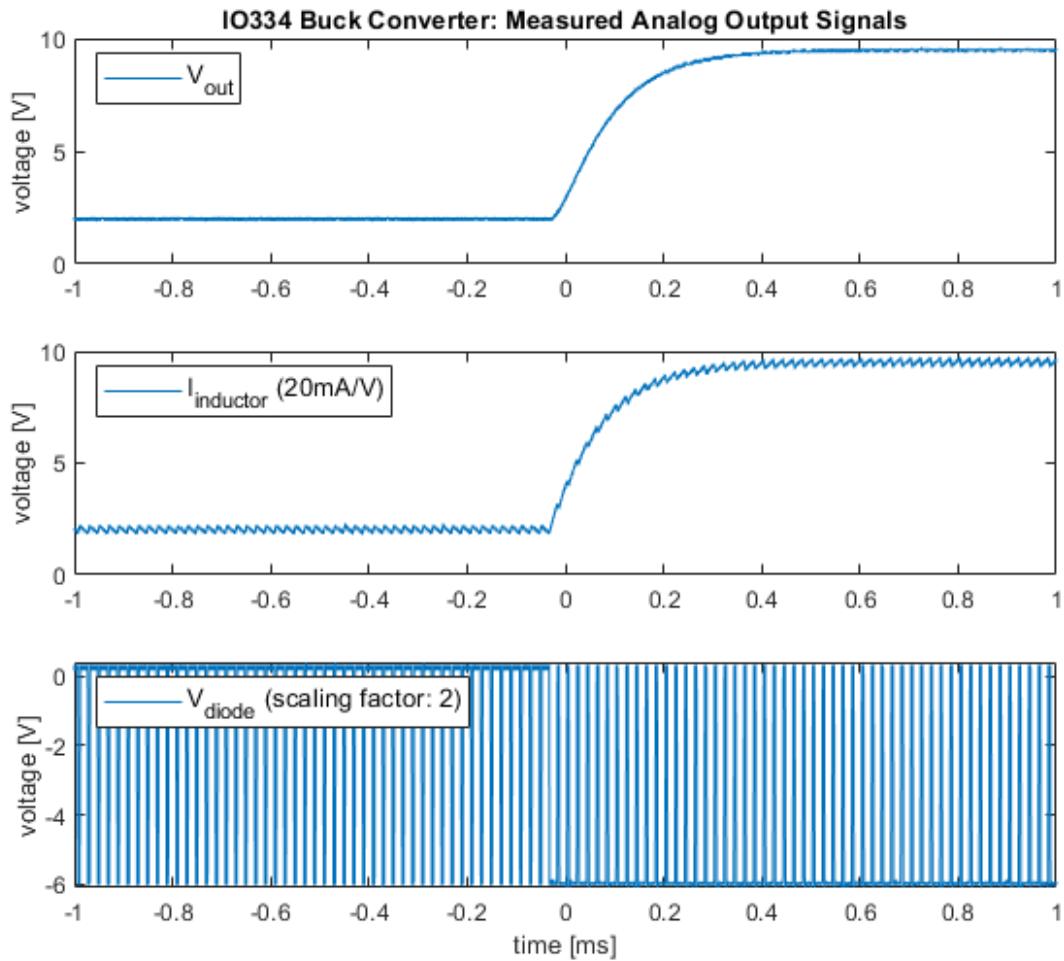
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);
run = Simulink.sdi.getRun(runID);
run.name = 'Real Time Simulation on I0334';

run.getAllSignals
plotOnSubPlot(run.getSignalsByName('PCIe_V_out_frame'),1,1,true);
plotOnSubPlot(run.getSignalsByName('PCIe_I_inductor_frame'),2,1,true);
plotOnSubPlot(run.getSignalsByName('PCIe_V_diode_frame'),3,1,true);

Simulink.sdi.view
```



Alternatively, you can measure the signals at the analog output of the IO334. This figure shows a plot of the signals in MATLAB.



## See Also

### Functions

`checkhdl` | `makehdl`

## More About

- “Run HDL Workflow with a Script” on page 31-53
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 41-93
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63

- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- Speedgoat I/O Examples

# Partition Simscape Models Containing a Large Network into Multiple Smaller Networks

This example shows how to partition a solar power inverter model that contains a single, large Simscape™ network into multiple networks. After you partition the network, you can run the Simscape HDL Workflow Advisor to generate the HDL implementation model. To learn how you run the Advisor for the model, see “Generate HDL Code for Simscape Models with Multiple Networks” on page 32-54.

## Why Partition a Simscape Network

When your Simscape model contains many switching elements, the state-space representation can contain a large number of modes. The Simscape HDL Workflow Advisor simulates the Simscape model to calculate the number of modes that are relevant. Certain Simscape models can have a large number of modes that are relevant. The generated HDL implementation model for such a large design can consume a significantly large number of resources, and the generated HDL implementation may even fail to synthesize on the target FPGA device. To reduce the number of modes, you can partition the Simscape network in your model into multiple networks, and then run the Simscape HDL Workflow Advisor.

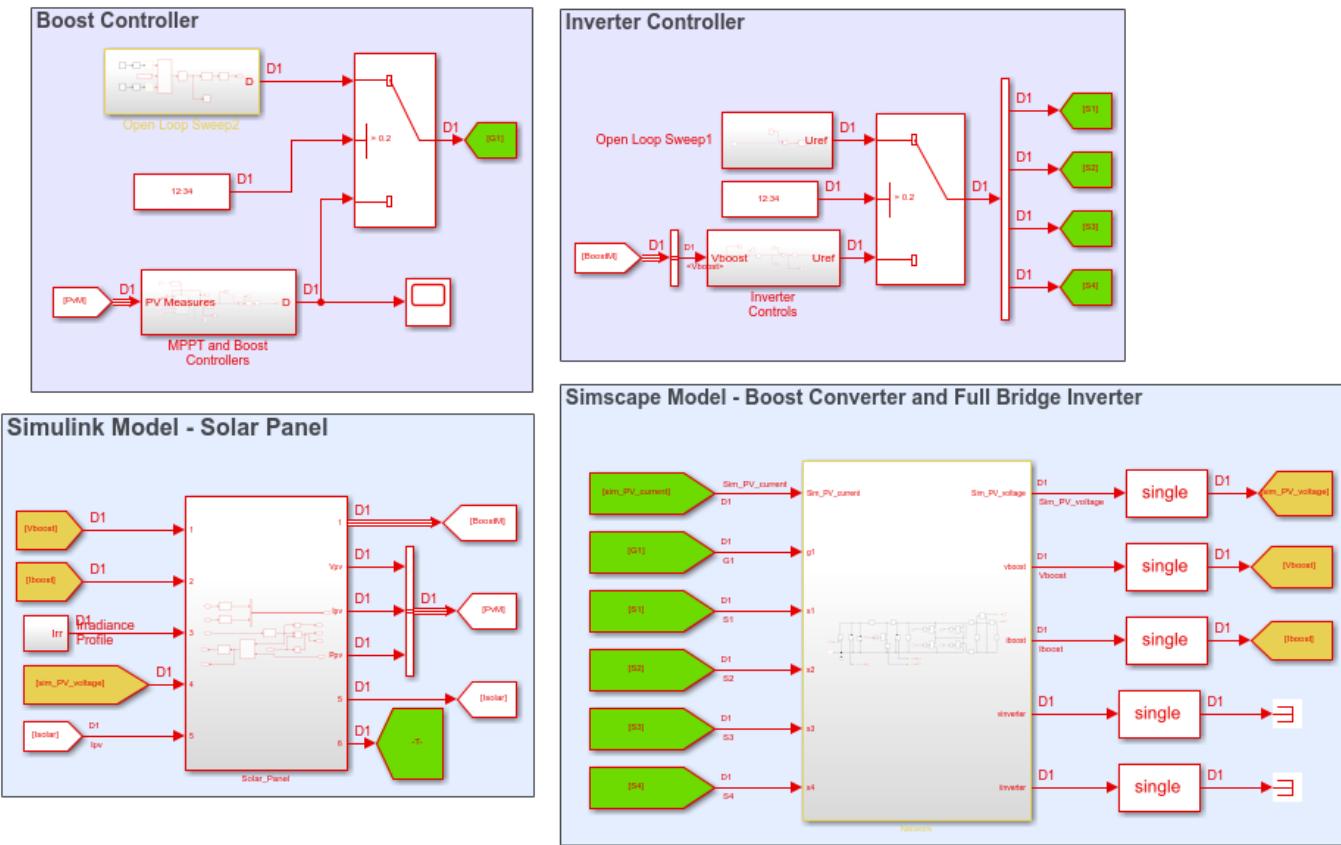
## Solar Power Inverter Model with Single Network

To open the solar power inverter example model, run:

```
open_system('sschdlexSolarInverterSingleNetworkExample')
```

For this example, the model is saved as Solar\_Power\_Inverter\_Single\_Network\_HDL. This model is the same as sschdlexSolarInverterSingleNetworkExample but has the subsystems rearranged and the logic for the solar panel placed inside a Solar\_Panel subsystem.

```
open_system('Solar_Power_Inverter_Single_Network_HDL')
set_param('Solar_Power_Inverter_Single_Network_HDL', 'SimulationCommand', 'Update')
```

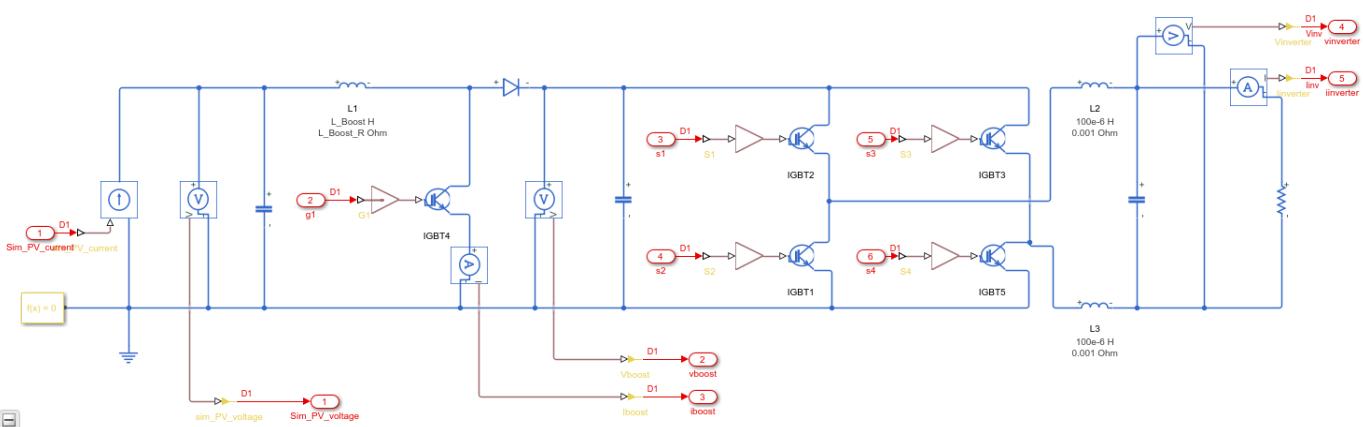


Copyright 2019 The MathWorks, Inc

The model consists of four parts: solar panel, boost controller, inverter controller, and a boost converter and full bridge inverter. The solar panel is modeled in Simulink® by using lookup tables. The boost controller and inverter controller provide the control signals for the boost converter and the full bridge inverter which is an H-bridge.

To see the boost converter and inverter, open the Network subsystem.

```
open_system('Solar_Power_Inverter_Single_Network_HDL/Network')
```



## Run Simscape HDL Workflow Advisor

1. To open the Simscape HDL Workflow Advisor for the model, enter:

```
sschdladvisor('Solar_Power_Inverter_Single_Network_HDL')
```

```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(Solar_Power_Inverter_Single_Netwo
```

2. Run the workflow to the **Discretize Equations** task. You see that the state-space representation uses around 173 modes, which is a large number of modes.

Summary of the state-space representation:

Details related to the Simscape network [Solar\\_Power\\_Inverter\\_Single\\_Network\\_HDL/Network/Solver Configuration](#)

Parameter	Parameter size
A	18 x 18 x 173
B	18 x 6 x 173
F0	18 x 1 x 173
C	5 x 18 x 173
D	5 x 6 x 173
Y0	5 x 1 x 173

Such a large number of modes can consume a significantly large number of hardware resources, and may even cause the DUT subsystem in the HDL implementation model to fail to synthesize on the target FPGA device.

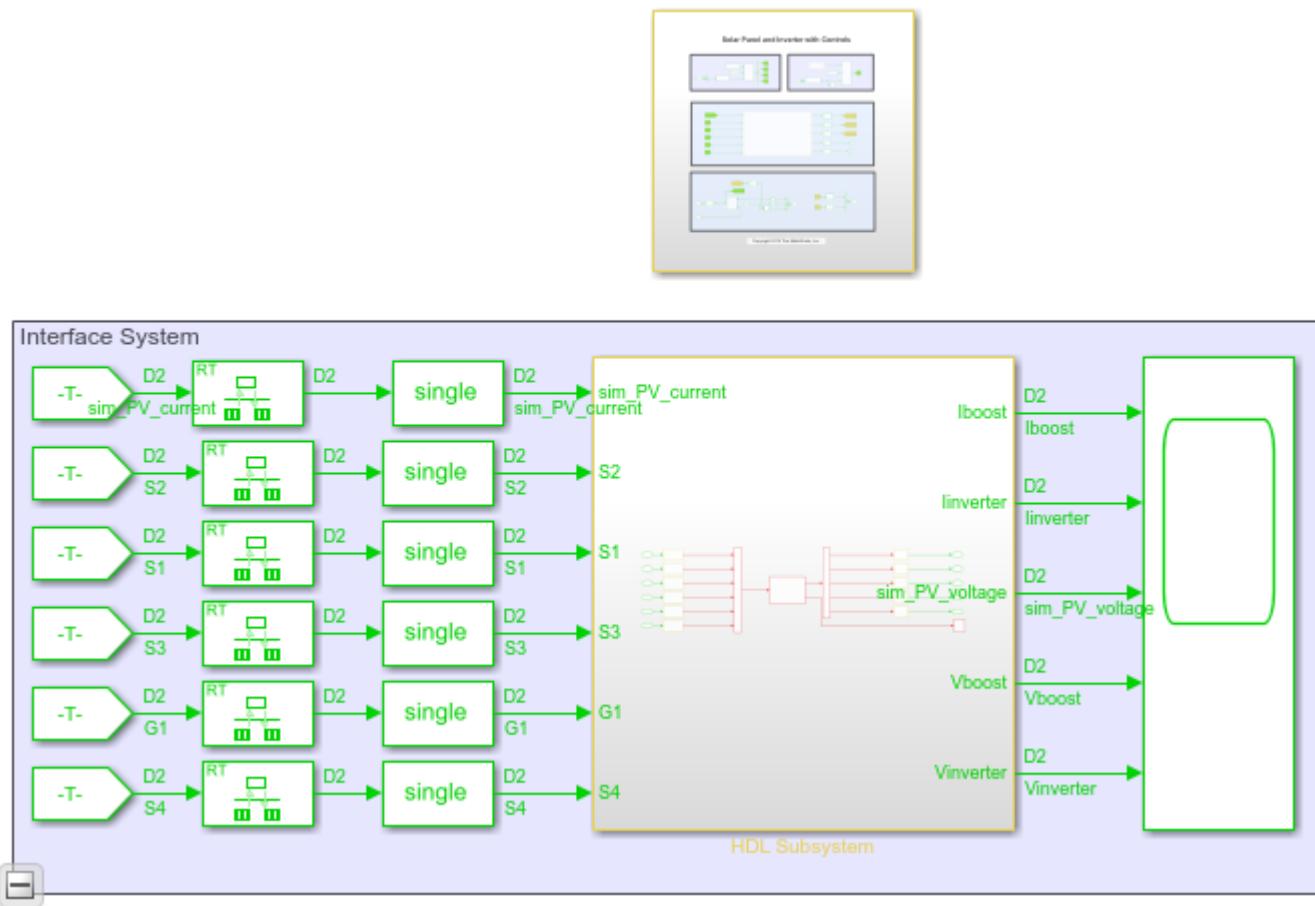
## Generate HDL Implementation Model and View Resource Consumption

To see the resource consumption:

1. Run the **Generate implementation model** task. Click the link to open the HDL implementation model.

The model contains a HDL Subsystem block that models the state-space equations for the Simscape network. Save the model as Solar\_Power\_Inverter\_Single\_Network\_StateSpace.slx.

```
open_system('Solar_Power_Inverter_Single_Network_StateSpace')
set_param('Solar_Power_Inverter_Single_Network_StateSpace', 'SimulationCommand', 'Update')
```



2. Enable generation of the resource utilization report.

```
hdlset_param('Solar_Power_Inverter_Single_Network_StateSpace', 'ResourceReport', 'on')
```

3. Run the `makehdl` function to generate code for the HDL Subsystem block.

```
makehdl('Solar_Power_Inverter_Single_Network_StateSpace/HDL_Subsystem')
```

If HDL Coder™ generates an error that it is unable to allocate delays, increase the **Oversampling factor**. Start by increasing the **Oversampling factor** to 100, and then generate HDL code. If HDL Coder is still unable to allocate delays, then further increase the **Oversampling factor**.

```
hdlset_param('Solar_Power_Inverter_Single_Network_StateSpace', 'Oversampling', 100)
```

4. As you generate HDL code, open the Code Generation Report. The resource utilization report indicates a large amount of multipliers, adders, and registers that might be consumed on the target FPGA device.

## Summary

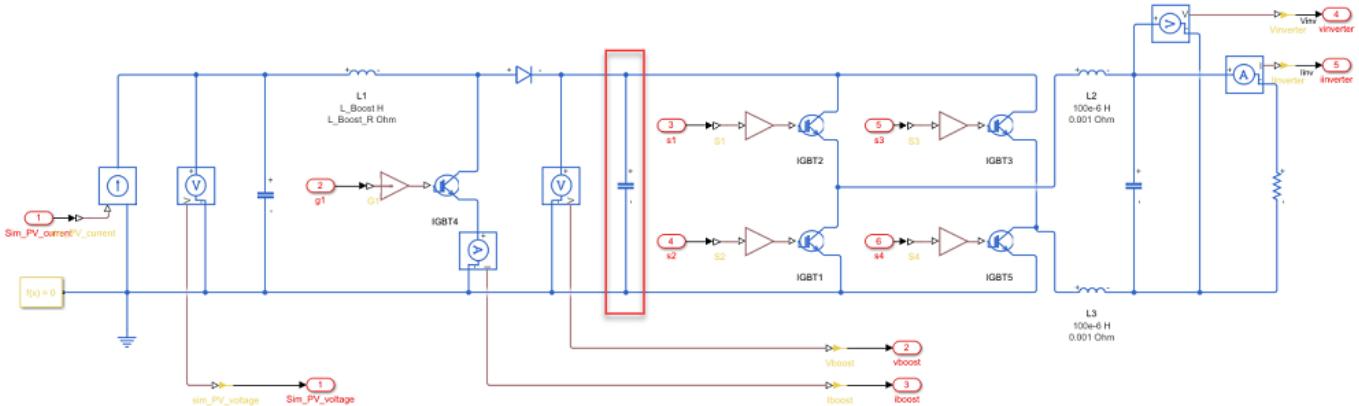
Multipliers	173
Adders/Subtractors	2959
Registers	16218
Total 1-Bit Registers	161776
RAMs	0
Multiplexers	25511
I/O Bits	356
Static Shift operators	0
Dynamic Shift operators	352

## Partition Solar Inverter Network into Multiple Simscape Networks

To reduce the number of modes, you can partition the Simscape network inside the Network subsystem into two Simscape networks. To partition the network into multiple networks:

1. Identify the boundary for partitioning the network into multiple networks. An energy storage element such as a capacitor or an inductor makes a good candidate for partitioning the network. To produce a Simscape model that contains multiple networks and effectively reduces the number of modes in the state-space representation, choose a boundary that produces identical or near identical partitions. That is, the number of switching elements on either side of the boundary are identical or nearly identical.

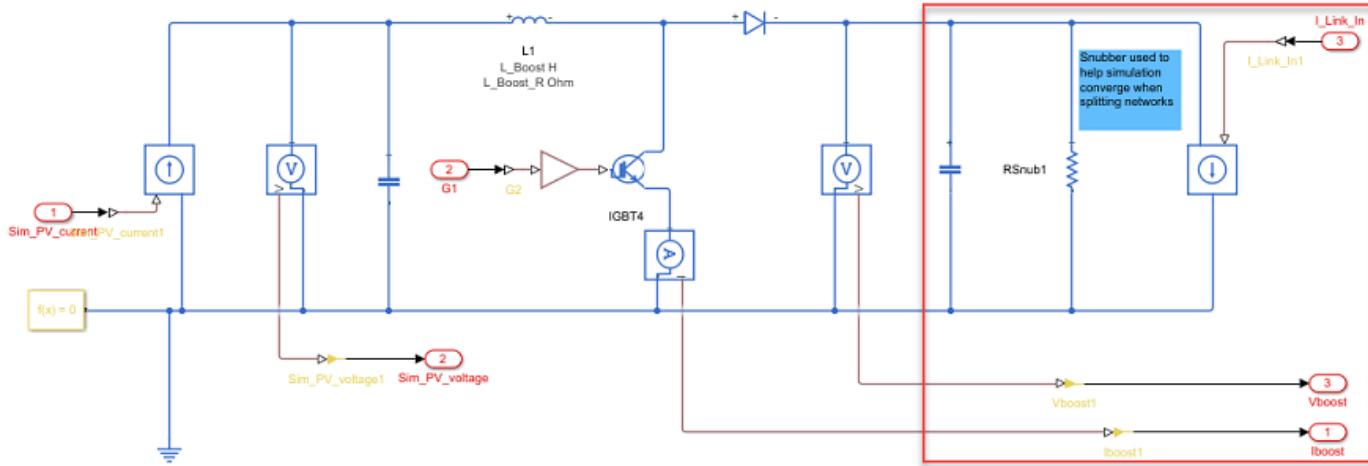
For the solar power inverter, you can choose the DC link capacitor between the H-bridge inverter and the boost converter as the boundary for partitioning the network.



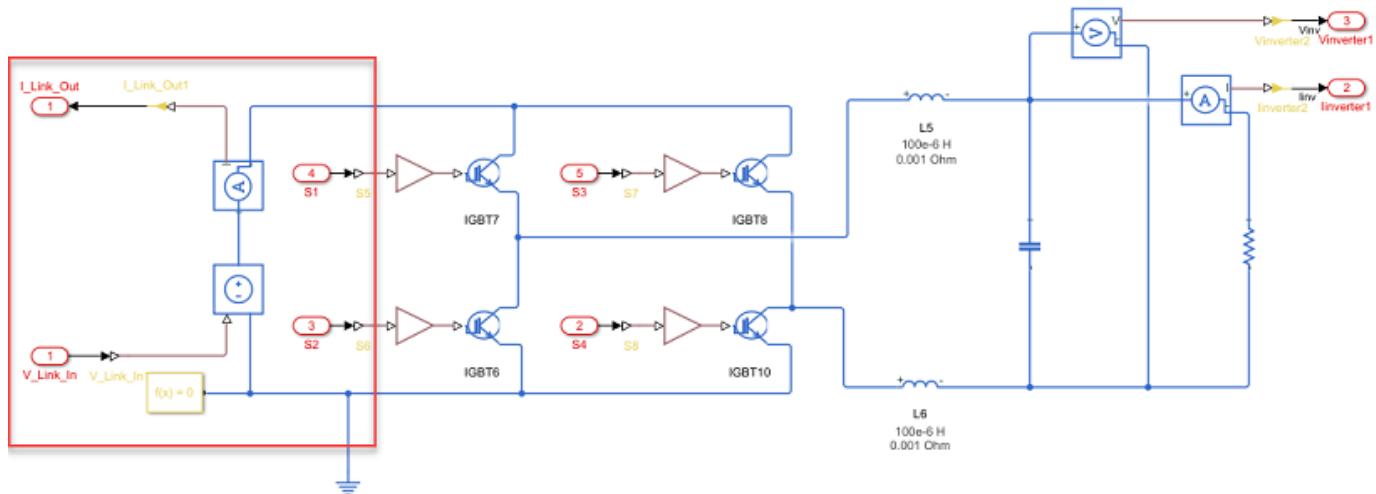
2. After you partition the network, prepare the modified Simscape model for compatibility with the Simscape HDL Workflow Advisor. Place each partitioned network inside a subsystem and use a Solver Configuration block for each network.

The Simscape HDL Workflow Advisor uses the Solver Configuration block to identify each unique network in your Simscape model.

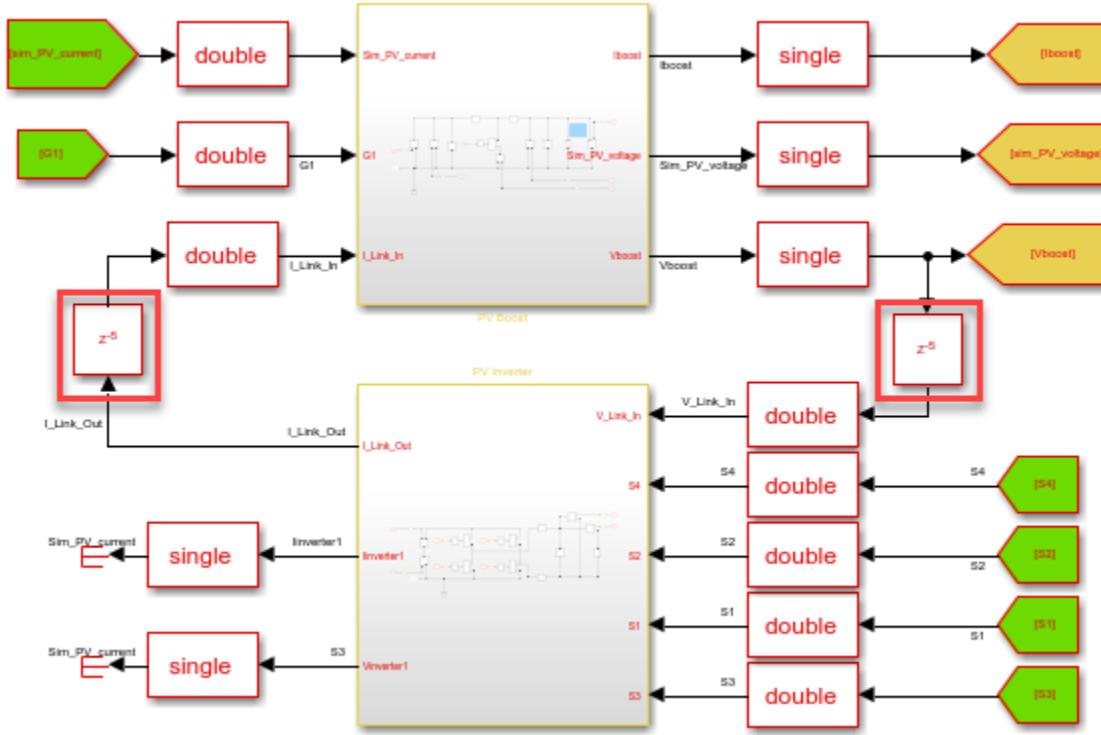
3. For the simulation to converge when using multiple networks, in the network containing the boost converter, add a snubber resistance and a controlled current source in parallel to the capacitor for the current output to the inverter network.



4. In the inverter network, add a controlled voltage source to the voltage input to the network.



5. To break the algebraic loops in the system, add Delay blocks between the signal lines that connect the output of one subsystem to the input of the other subsystem. For higher accuracy, add Data Type Conversion blocks to provide double data types as inputs to the networks.



### Solar Power Inverter Model with Multiple Networks

The single network model is now partitioned into multiple networks. To open the model containing multiple networks, enter:

```
open_system('sschdlexSolarInverterPartitionedNetworkExample')
```

To learn how you run the Simscape HDL Workflow Advisor and generate HDL code for this model, see “Generate HDL Code for Simscape Models with Multiple Networks” on page 32-54.

## See Also

### Functions

`checkhdl` | `makehdl` | `sschdladvisor`

## More About

- “Generate HDL Code for Simscape Models” on page 32-9
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-6
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97

## Generate HDL Code for Simscape Models with Multiple Networks

This example shows how you can run the Simscape HDL Workflow Advisor to generate the HDL implementation model for a Simscape™ model that contains multiple networks. You can also generate a validation logic that numerically compares each Simscape network with the corresponding state-space implementation in the HDL implementation model. The Simscape model in this example is a solar power inverter partitioned into two networks. To learn how this network is partitioned, see “Partition Simscape Models Containing a Large Network into Multiple Smaller Networks” on page 32-47.

### Why Use a Simscape Model with Multiple Networks

When your Simscape model contains many switching elements, the state-space representation can contain a large number of modes. The generated HDL implementation model for such a large design can consume a significantly large number of resources, and may even fail to synthesize on the target FPGA device. To reduce the number of modes, you can partition the Simscape network in your model into multiple networks, and then run the Simscape HDL Workflow Advisor.

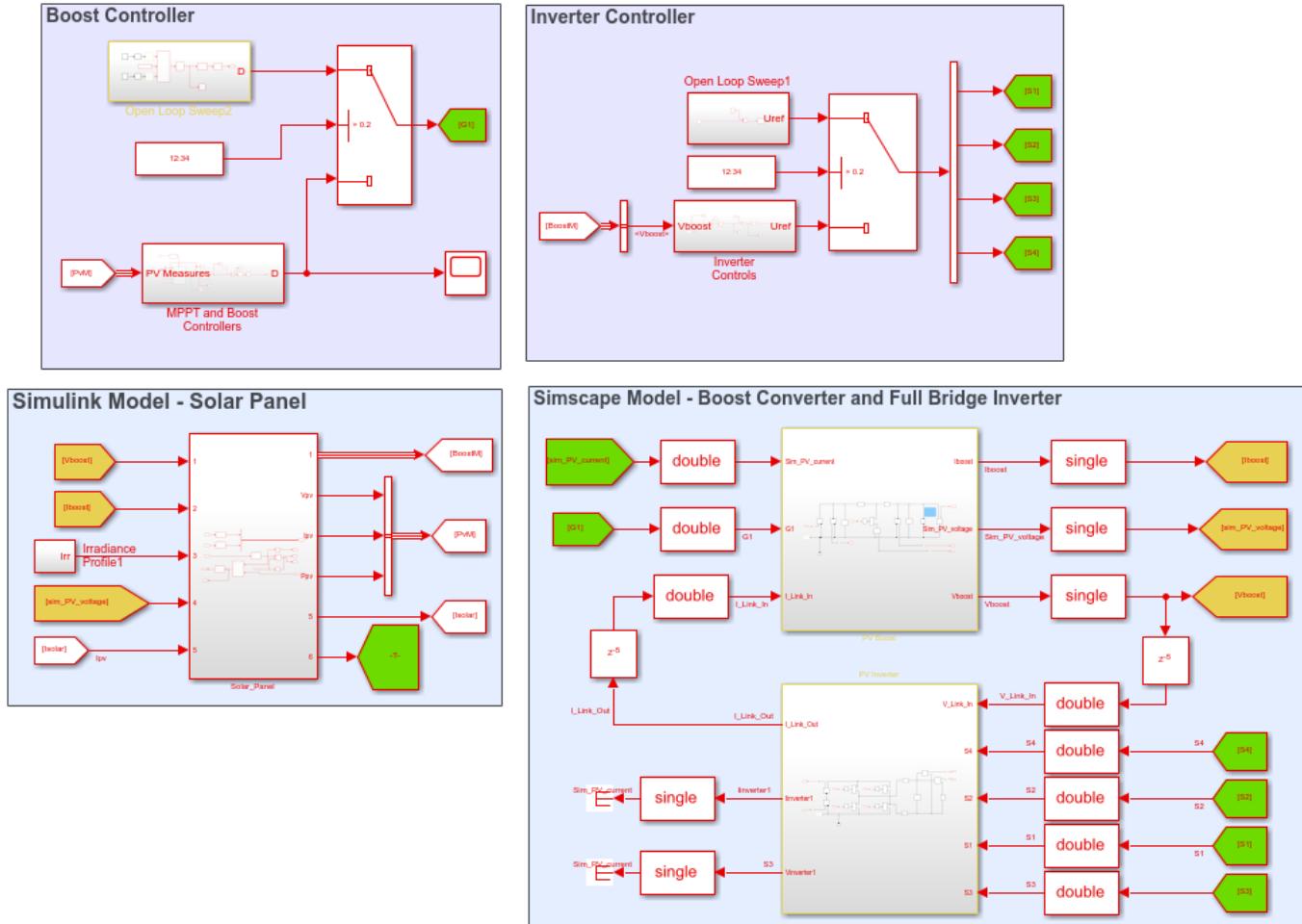
### Solar Power Inverter Model with Multiple Networks

To open the model that contains multiple networks, run:

```
open_system('sschdlexSolarInverterPartitionedNetworkExample')
```

For this example, the model is saved as Solar\_Power\_Inverter\_Multiple\_Network\_HDL. This model is the same as sschdlexSolarInverterPartitionedNetworkExample but has the subsystems rearranged and the logic for the solar panel placed inside a Solar\_Panel subsystem.

```
open_system('Solar_Power_Inverter_Multiple_Network_HDL')
set_param('Solar_Power_Inverter_Multiple_Network_HDL', 'SimulationCommand', 'Update')
```



Copyright 2019 The MathWorks, Inc

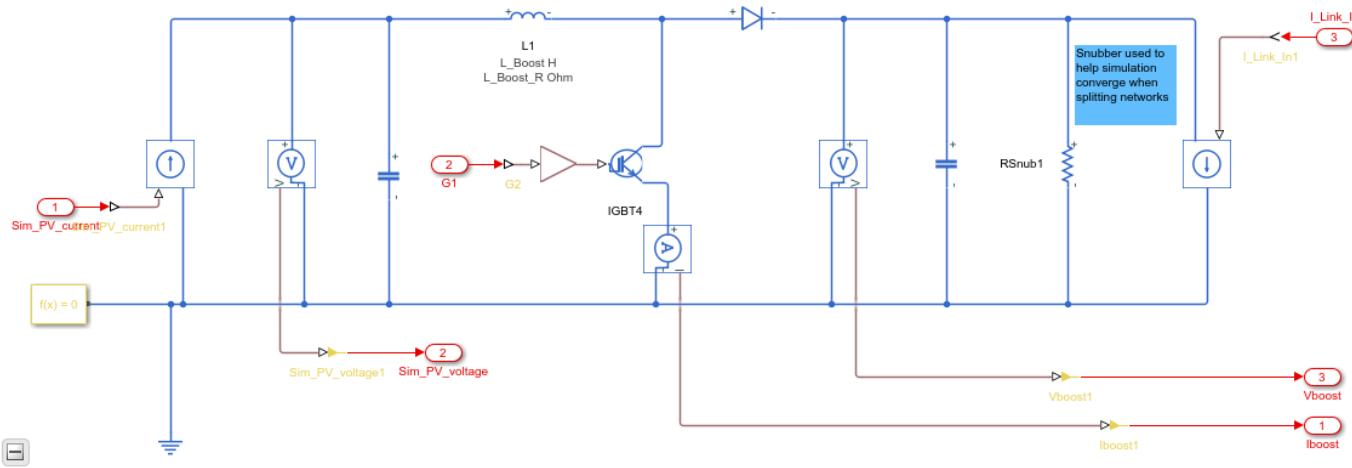
The model consists of four parts: solar panel, boost controller, inverter controller, and a boost converter and full bridge inverter. The solar panel is modeled in Simulink® by using lookup tables. The boost controller and inverter controller provide the control signals for the boost converter and the full bridge inverter which is an H-bridge.

The original model contains the boost converter and full bridge inverter as a single network inside one subsystem. To see this model, enter:

```
open_system('sschdlexSolarInverterSingleNetworkExample')
```

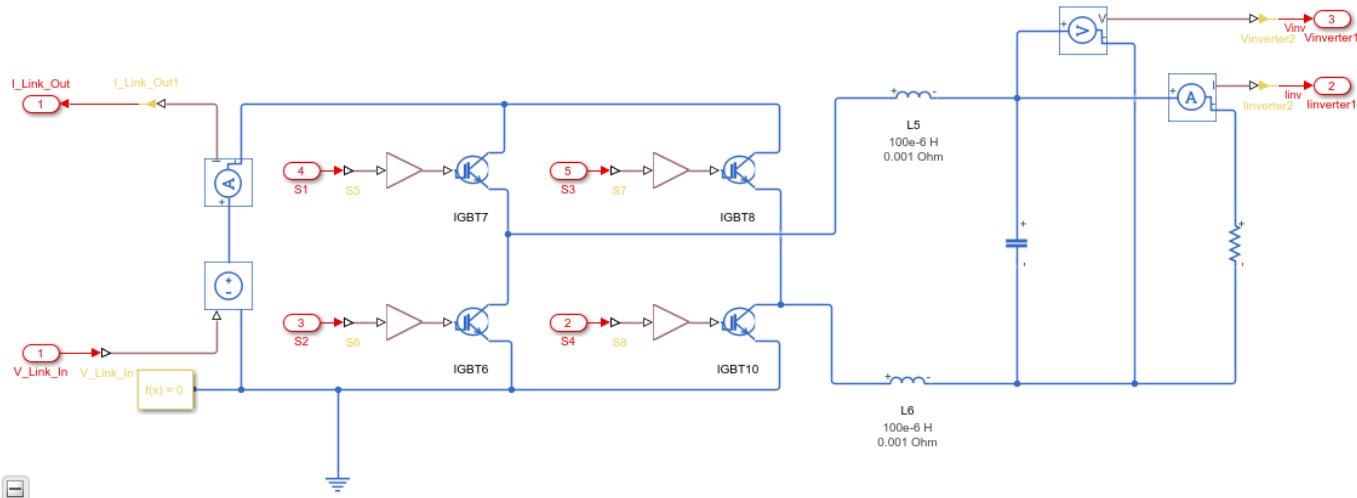
The partitioned model contains the two networks inside separate subsystems. To see the boost converter, open the PV Boost subsystem.

```
open_system('Solar_Power_Inverter_Multiple_Network_HDL/PV_Boost')
```



To see the full bridge inverter, open the PV Inverter subsystem.

```
open_system('Solar_Power_Inverter_Multiple_Network_HDL/PV Inverter')
```



### Run Simscape HDL Workflow Advisor for Model with Multiple Networks

1. To open the Simscape HDL Workflow Advisor for the model, enter:

```
sschdladvisor('Solar_Power_Inverter_Multiple_Network_HDL')
```

```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(Solar_Power_Inverter_Multiple_Netw
```

2. Run the workflow to the **Check switched linear** task.

The Simscape HDL Workflow Advisor lists the number of networks present in the model and the number of algebraic and differential variables for each network. The Advisor uses the Solver Configuration block to identify each unique network in your model.

Number of Simscape networks present in the model: 2

**Details related to the Simscape network [Solar\\_Power\\_Inverter\\_Multiple\\_Network\\_HDL/PV Inverter/Solver Configuration](#)**

**Details**

Number of Discrete Variables: 12

Number of Differential Variables: 3

Source	Value
<a href="#">PV_Inverter.Capacitor4.vc</a>	Capacitor voltage
<a href="#">PV_Inverter.L5.i_L</a>	Inductor current
<a href="#">PV_Inverter.L6.i_L</a>	Inductor current

Number of Algebraic Variables: 9

Source	Value
<a href="#">PV_Inverter.IGBT10.C.v</a>	Voltage
<a href="#">PV_Inverter.IGBT10.ideal_switch.i</a>	i
<a href="#">PV_Inverter.IGBT6.C.v</a>	Voltage
<a href="#">PV_Inverter.IGBT6.ideal_switch.i</a>	i
<a href="#">PV_Inverter.IGBT7.diode.i</a>	Current
<a href="#">PV_Inverter.IGBT7.ideal_switch.i</a>	i
<a href="#">PV_Inverter.IGBT8.diode.i</a>	Current
<a href="#">PV_Inverter.IGBT8.ideal_switch.i</a>	i
<a href="#">PV_Inverter.L6.v</a>	Voltage

**Details related to the Simscape network [Solar\\_Power\\_Inverter\\_Multiple\\_Network\\_HDL/PV Boost/Solver Configuration](#)**

**Details**

Number of Discrete Variables: 6

Number of Differential Variables: 3

Source	Value
<a href="#">PV_Boost.Capacitor.vc</a>	Capacitor voltage
<a href="#">PV_Boost.Capacitor2.vc</a>	Capacitor voltage
<a href="#">PV_Boost.L1.i_L</a>	Inductor current

3. Run the **Extract equations** task.

The task displays the number of modes, states, inputs, outputs, and differential variables for each Simscape network.

**Passed**

Details related to the Simscape network [Solar\\_Power\\_Inverter\\_Multiple\\_Network\\_HDL/PV Inverter/Solver Configuration1](#)

- Number of states: 12
- Number of inputs: 5
- Number of outputs: 3
- Number of modes: 58
- Number of differential variables: 3

Details related to the Simscape network [Solar\\_Power\\_Inverter\\_Multiple\\_Network\\_HDL/PV Boost/Solver Configuration](#)

- Number of states: 6
- Number of inputs: 3
- Number of outputs: 3
- Number of modes: 9
- Number of differential variables: 3

4. Run the **Discretize equations** task.

The state-space representation now uses fewer modes. The number of modes is 58 for the boost converter and 9 for the full bridge inverter, which results in a total number of 67 modes. The reduction in the number of modes saves area of the HDL implementation model on the target device.

Details related to the Simscape network [Solar\\_Power\\_Inverter\\_Multiple\\_Network\\_HDL/PV Boost/Solver Configuration](#)

Parameter	Parameter size
A	12 x 12 x 58
B	12 x 5 x 58
F0	12 x 1 x 58
C	3 x 12 x 58
D	3 x 5 x 58
Y0	3 x 1 x 58

Details related to the Simscape network [Solar\\_Power\\_Inverter\\_Multiple\\_Network\\_HDL/PV Inverter/Solver Configuration1](#)

Parameter	Parameter size
A	6 x 6 x 9
B	6 x 3 x 9
F0	6 x 1 x 9
C	3 x 6 x 9
D	3 x 3 x 9
Y0	3 x 1 x 9

5. Change the **Validation logic tolerance** to 1e-4 and select the **Generate validation logic for the implementation model** check box. Run the **Generate implementation model** task.

**Generate implementation model**

Solver Settings

Solver method: Iterative Number of solver iterations 5 [How to Change This?](#)

Implementation Model Settings

Floating-point precision

Single  Double  Single coefficient, double computation [What's This?](#)

Map state space parameters to RAMs Auto

Verification Settings

Generate validation logic for the implementation model Validation logic tolerance 1e-4

**Run This Task**

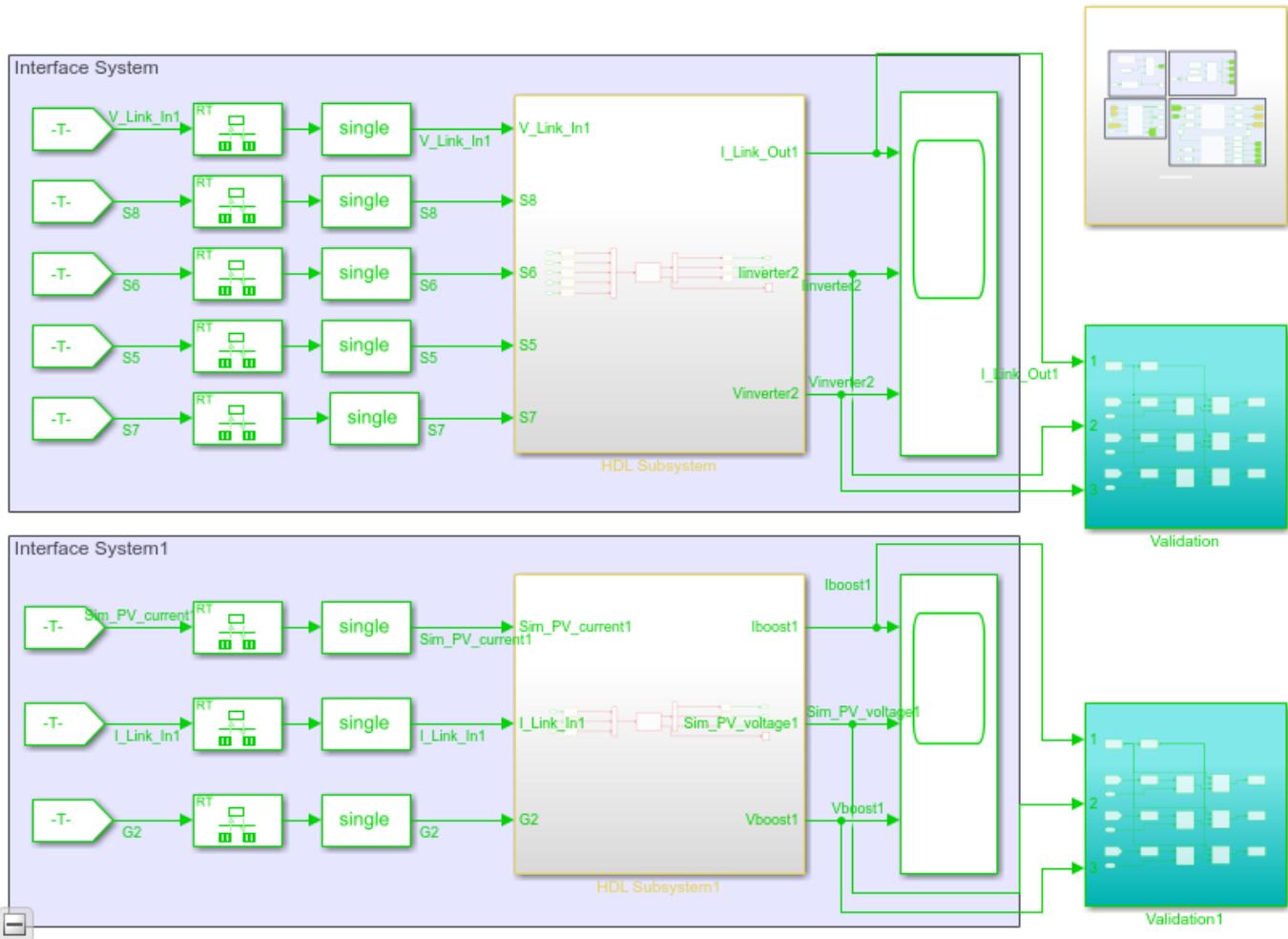
Result:  Passed

**Passed**  
Generated implementation model '[gmStateSpaceHDL\\_Solar\\_Power\\_Inverter\\_Multip](#)'.

### Open HDL Implementation Model and Validate HDL Algorithm

To open the implementation model, click the link in the **Generate implementation model** task log. Rename the model as Solar\_Power\_Inverter\_Multiple\_Network\_StateSpace.

```
open_system('Solar_Power_Inverter_Multiple_Network_StateSpace')
set_param('Solar_Power_Inverter_Multiple_Network_StateSpace', 'SimulationCommand', 'Update')
```



The model contains two HDL Subsystems. The HDL Subsystem block models the state-space equations for the boost converter. The HDL Subsystem1 block models the state-space equations for the full bridge inverter. The Validation and Validation1 subsystems compare functional equivalence of the state space representation of the boost converter and full bridge inverter with the corresponding Simscape network in the original model.

The state-space parameters are saved in a MAT file

`Solar_multiple_network_stateSpaceParameters.mat`. The file contains a cell array of two structures. One structure contains the parameters for the boost converter. The other structure contains the parameters for the full bridge inverter.

To compare the functional equivalence, simulate the model. If simulating the model produces assertions, you can resolve the validation mismatch by modifying a combination of various settings in the **Generate implementation model** task until the HDL implementation model matches the Simscape algorithm. The settings include increasing the validation logic tolerance, increasing the number of solver iterations, and changing the floating-point precision. For more information, see “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97.

### Generate HDL Code and Validation Model

1. Enable generation of the resource utilization report.

```
hdlset_param('Solar_Power_Inverter_Multiple_Network_StateSpace', 'ResourceReport', 'on')
```

2. Before you generate HDL code, it is recommended that you enable generation of the validation model. The validation model compares the output of the generated model after code generation to the output of the original model. To learn more, see “Generated Model and Validation Model” on page 24-10.

```
HDLmodelName = 'Solar_Power_Inverter_Multiple_Network_StateSpace';
hdlset_param(HDLmodelName, 'TargetDirectory', 'C:/Temp/hdlsrc');
hdlset_param(HDLmodelName, 'GenerateValidationModel', 'on');
```

3. Run the `makehdl` function to generate code. To generate HDL code for both HDL Subsystem blocks, you can place the blocks inside another top level subsystem and then generate HDL code. Name this subsystem as `HDL_DUT`.

```
makehdl('Solar_Power_Inverter_Single_Network_StateSpace/HDL_DUT')
```

The generated HDL code and validation model are saved in `C:/Temp/hdlsrc` directory. The generated code is saved as `HDL_DUT_tc.vhd`. To open the validation model, click the link to `gm_Solar_Power_Inverter_Multiple_Network_StateSpace_vnl.slx` in the code generation logs in the Command Window.

4. As you generate HDL code, open the Code Generation Report. The resource utilization report indicates a large amount of adders, multipliers, and registers that might be consumed on the target FPGA device.

## Summary

Multipliers	103
Adders/Subtractors	1751
Registers	9957
Total 1-Bit Registers	100465
RAMs	0
Multiplexers	15562
I/O Bits	452
Static Shift operators	0
Dynamic Shift operators	214

The overall resource consumption of the two networks is significantly less than the resource consumption of a single, large network. To learn about the resource consumption of the single solar power inverter network, see “Partition Simscape Models Containing a Large Network into Multiple Smaller Networks” on page 32-47.

## See Also

### Functions

`checkhdl` | `makehdl` | `sschdladvisor`

## More About

- “Generate HDL Code for Simscape Models” on page 32-9
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-6
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97

# Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model

This example shows how to modify a Simscape™ plant model to generate an HDL-compatible Simulink® model with HDL Coder™. HDL code is then generated from this Simulink model.

## Introduction

The Simscape plant model is converted to an HDL-compatible Simulink model by using the Simscape HDL Workflow Advisor. To run the Advisor, you run the `sschdla` function for the model.

The Simscape HDL Workflow Advisor generates an HDL implementation model from which you generate HDL code. Before you generate the implementation model, configure the Simscape plant model for generation of the implementation model using the Simscape HDL Workflow Advisor. For more information, see “Generate HDL Code for Simscape Models” on page 32-9.

In some cases, the Simscape plant model may not be compatible for generation of the implementation model using the Simscape HDL Workflow Advisor. For HDL compatibility, you modify the Simscape plant model and then run the Simscape HDL Workflow Advisor. This example illustrates the DC Motor Control plant model. The model contains a nonlinear Friction block. You can use the approach in this example to convert Simscape models with few nonlinear blocks to a HDL-compatible Simulink model.

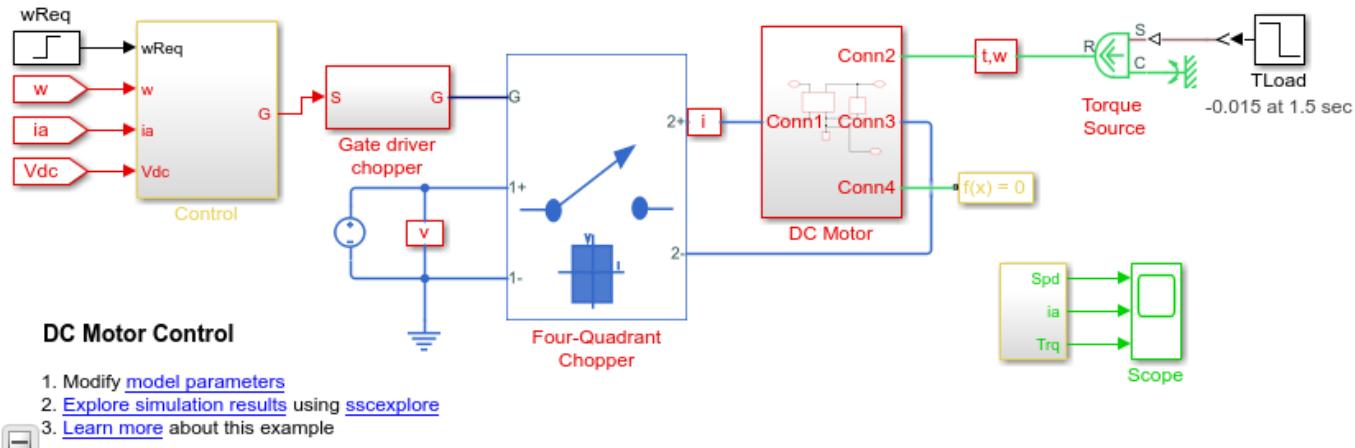
## DC Motor Control Model

The DC Motor Control model is a physical model developed in Simscape. The model contains nonlinear elements and must be modified for implementation model generation.

```
open_system('ee_dc_motor_control')
```

Enclose the DC Motor and Friction block inside a Subsystem and save the model as `ee_dc_motor_control_original`.

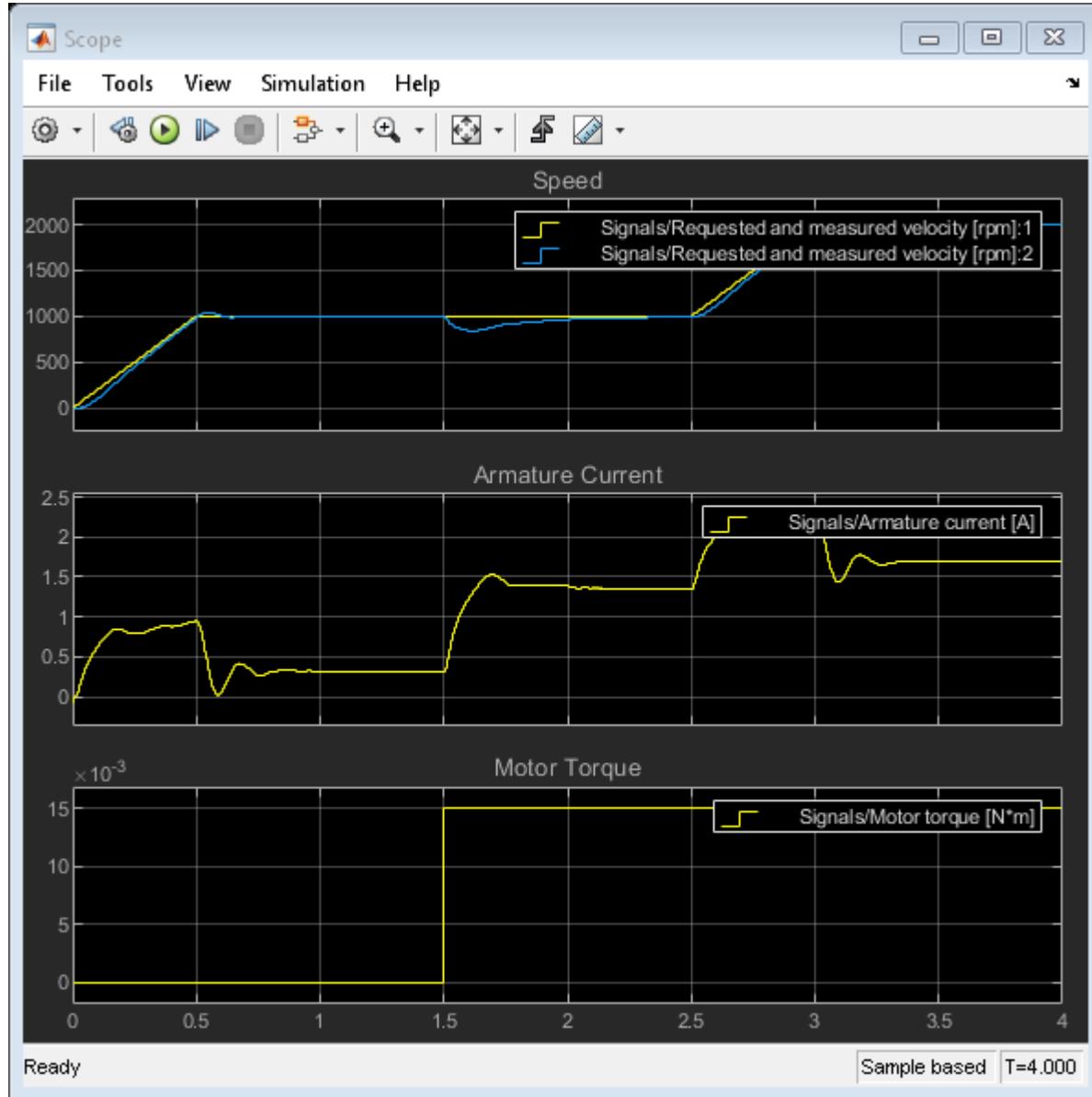
```
open_system('ee_dc_motor_control_original')
set_param('ee_dc_motor_control_original','SimulationCommand','Update')
```



DC motor control is used as a speed control structure. A PWM controlled four-quadrant Chopper is used to feed the DC motor. The DC motor consists of Rotational Electromechanical Converter,

Resistor, Inductance, Friction block and an Inertia block. The control subsystem includes the outer speed-control loop, the inner current-control loop and the PWM generation.

```
sim('ee_dc_motor_control_original')
open_system('ee_dc_motor_control_original/Scope')
```



### Make DC Motor Model HDL-Compatible

To convert the model to a model that is compatible for conversion with Simscape HDL Workflow Advisor:

1. Detect presence of nonlinear components or blocks in the model. To verify the presence of nonlinear blocks in Simscape plant model, enter:

```
simscape.findNonlinearBlocks('ee_dc_motor_control_original')
```

```
Found network that contains nonlinear equations in the following blocks:
{'ee_dc_motor_control_original/DC Motor/Friction'}
```

The number of linear or switched linear networks in the model is 0.  
The number of nonlinear networks in the model is 1.

ans =

```
1x1 cell array
{'ee_dc_motor_control_original/DC Motor/Friction'}
```

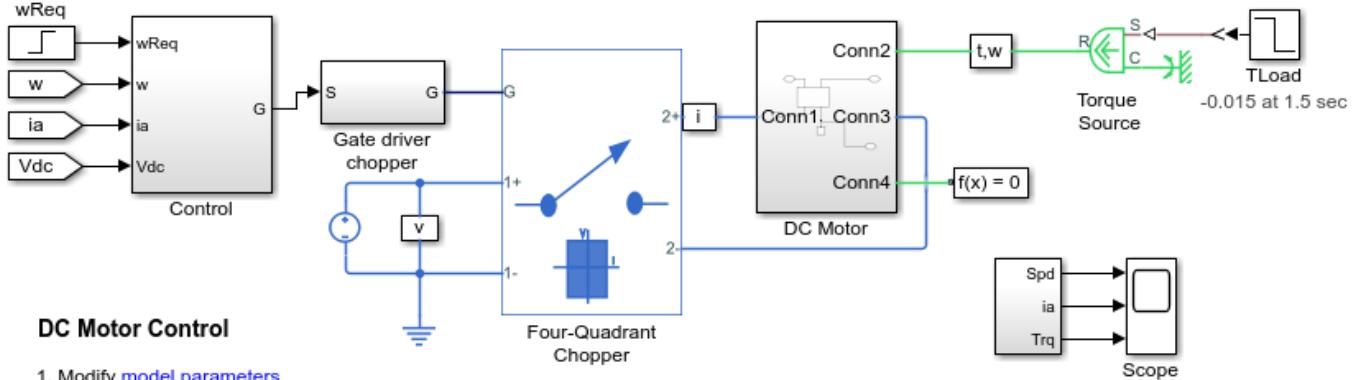
The Simscape plant model has a nonlinear block, which is the Friction block.

2. For HDL compatibility, the model must not contain nonlinear elements. Remove the Friction block from the model.

3. To simulate the model faster and to reduce the time that the Simscape HDL Workflow Advisor takes to extract the state-space equations, reduce the stop time of this model. In the Simulink Toolstrip, on the Simulation tab, change **Stop Time** to 1.

Save the changes into a new model as ee\_dc\_motor\_control\_modified.

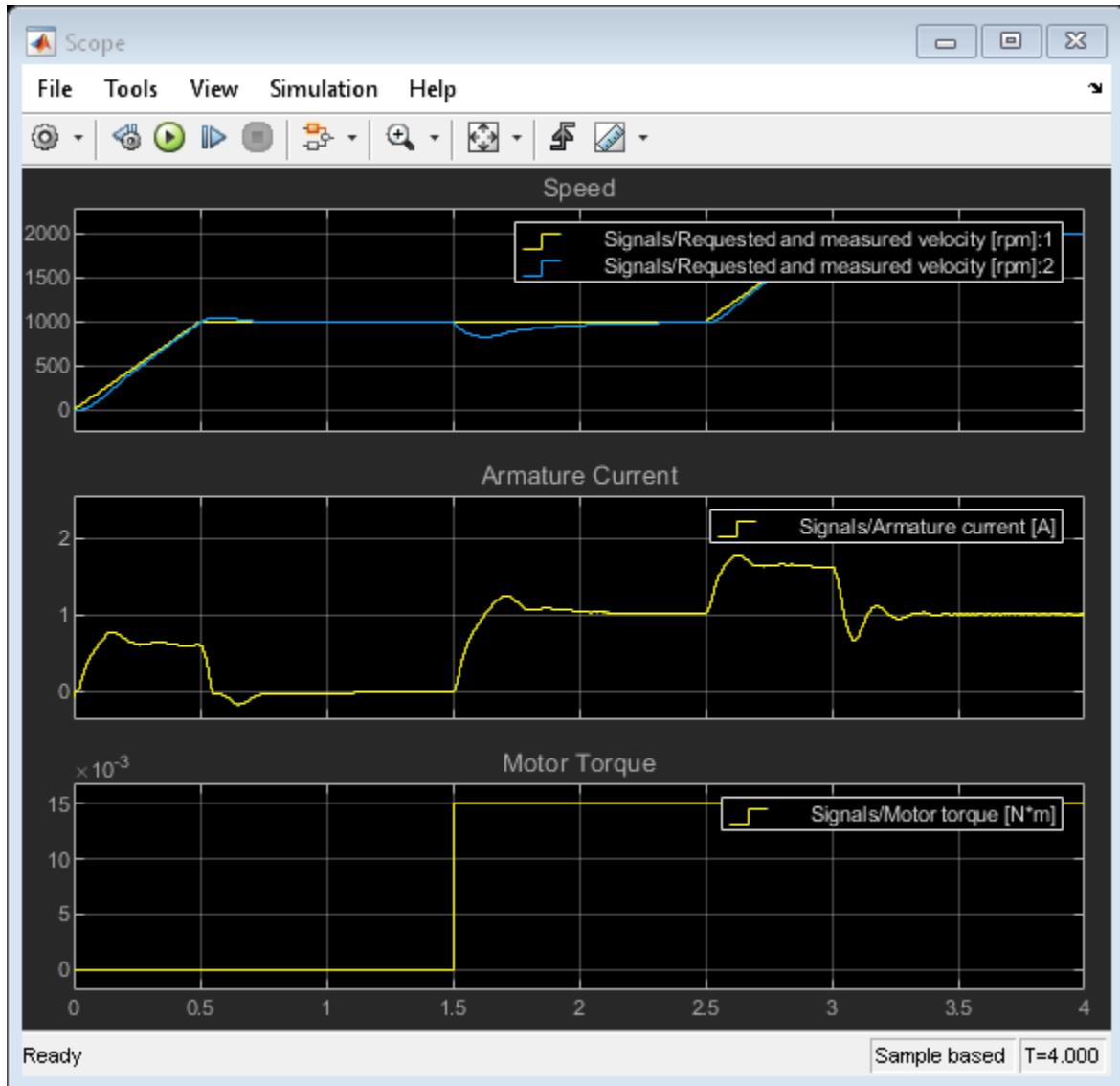
```
open_system('ee_dc_motor_control_modified')
set_param('ee_dc_motor_control_original','SimulationCommand','Update')
```



1. Modify [model parameters](#)
2. Explore simulation results using [sscexplore](#)
3. [Learn more](#) about this example

To see the simulation results of the modified model, run these commands:

```
sim('ee_dc_motor_control_modified')
open_system('ee_dc_motor_control_modified/Scope')
```



### Run Simscape HDL Workflow Advisor and Verify Simulation Results

To open the Simscape HDL Workflow Advisor, run the `sschdladvisor` for your model.

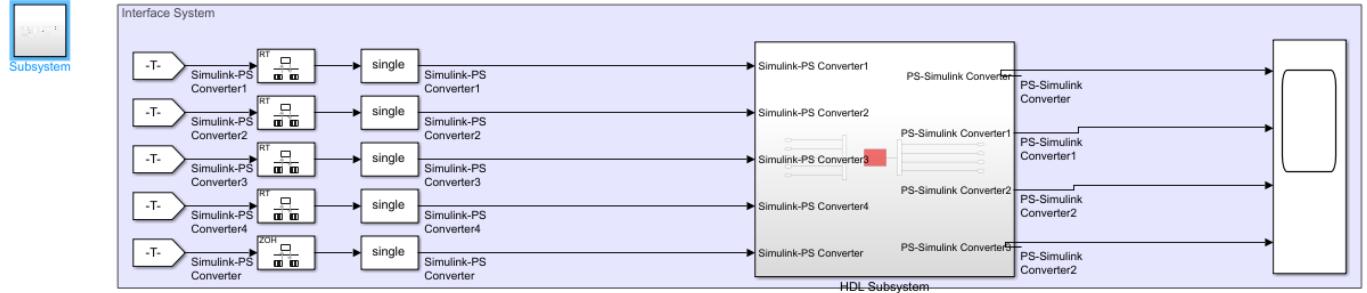
```
sschdladvisor('ee_dc_motor_control_modified')
```

```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(ee_dc_motor_control_modified)">ee_
```

Updating Model Advisor cache...

Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.Manager

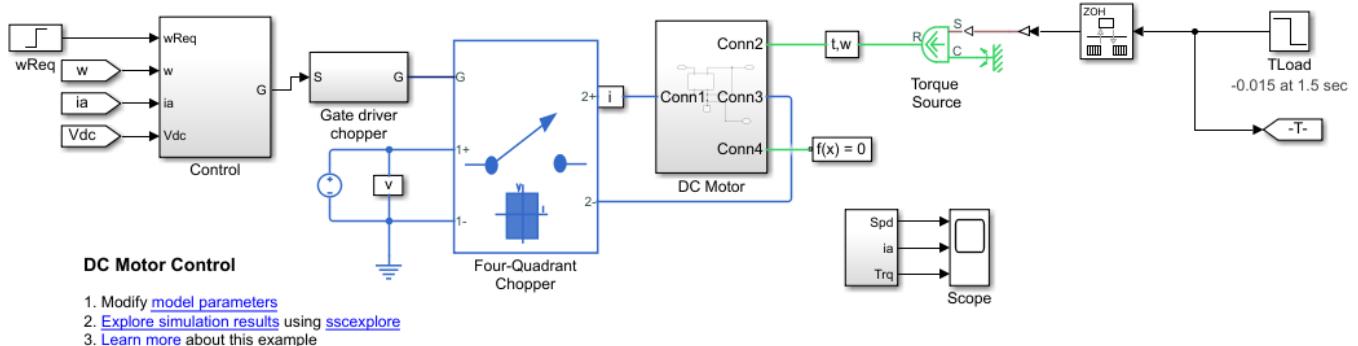
To generate the implementation model, in the Simscape HDL Workflow Advisor, leave all tasks to the default settings and then run the tasks. Click the link in the **Generate implementation model** task to open the model.



### Simulate Implementation model and Generate HDL code

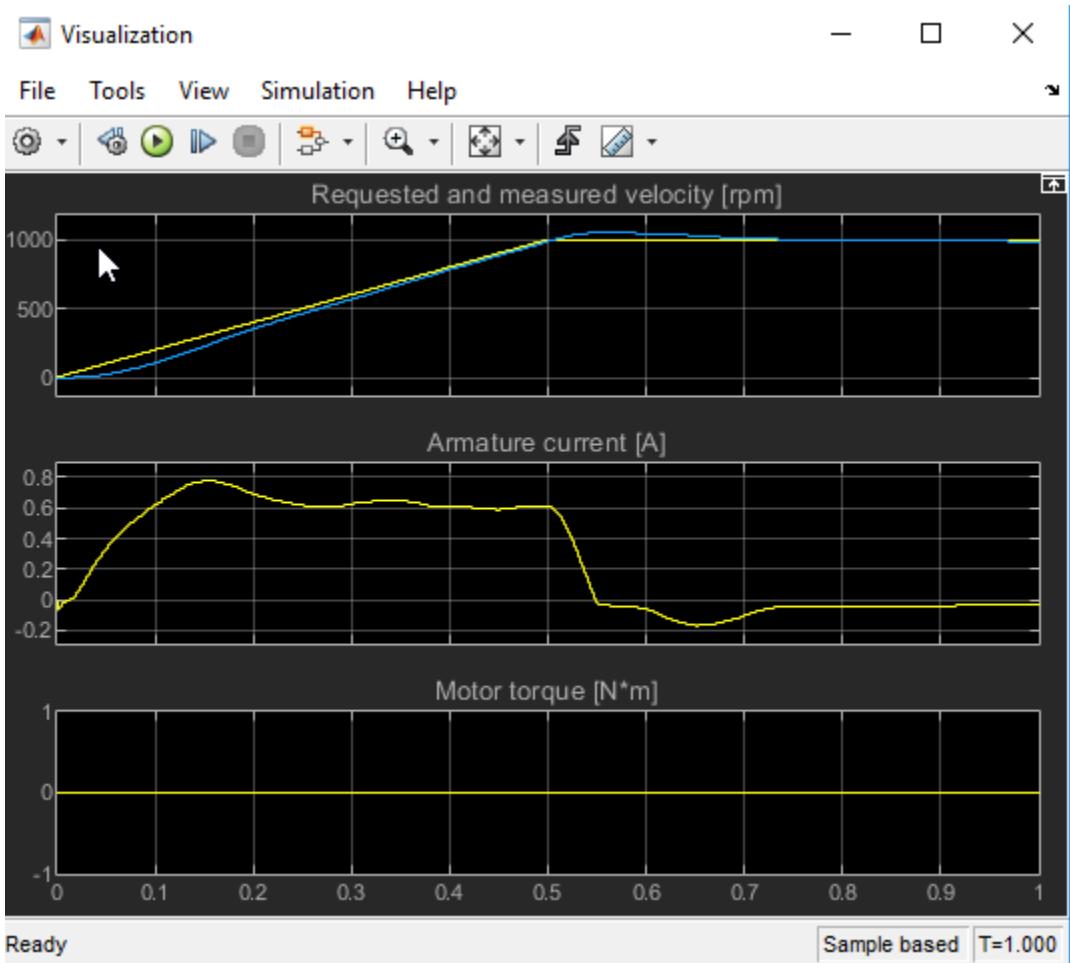
Before you can generate HDL code from the model, you must change the sample time and specify certain settings that make the model compatible for HDL code generation. The sample time of modified plant model is  $T_s$  and the number of solver iterations to compute the modes is 3. Therefore, you must change the sample time of the model. To specify the HDL-compatible settings:

- 1 In the Configuration Parameters dialog box:
  - On the **Solver** pane, set **Fixed-step size (fundamental sample time)** to  $T_s/3$  and select **Treat each discrete rate as a separate task**.
  - On the **Diagnostics > Sample Time** pane, set **Multitask rate transition** and **Single task rate transition** to error.
- 1 Add a Rate Transition block in your Simscape model that is placed inside the Subsystem block in your implementation model as illustrated in figure below.



To simulate the model, run this command and then open the Scope block to see the results:

```
sim('gmStateSpaceHDL_ee_dc_motor_control_modifie')
```



You see that the output generated by the modified Simscape plant model matches the output generated by the implementation model.

### Generate HDL Code and Validation Model

Before you generate HDL code, it is recommended that you enable generation of the validation model. The validation model compares the output of the generated model after code generation and the modified Simscape plant model. To learn more, see “Generated Model and Validation Model” on page 24-10.

To save validation model generation settings on your Simulink model, run this command:

```
hdlset_param('gmStateSpaceHDL_ee_dc_motor_control_modified', 'GenerateValidationModel', 'on');
```

To generate HDL code, run this command:

```
makehdl('gmStateSpaceHDL_ee_dc_motor_control_modified/HDL Subsystem')
```

By default, HDL Coder generates VHDL code. To generate Verilog code, run this command:

```
makehdl('gmStateSpaceHDL_ee_dc_motor_control_modified/HDL Subsystem', 'TargetLanguage', 'Verilog')
```

The generated HDL code and the validation model is saved in the `hdlsrc` directory. The generated code is saved as `HDL_Subsystem_tc.vhd`. You can also verify the simulation results by running the validation model `gm_gmStateSpaceHDL_ee_dc_motor_control_modifie_vnl.slx`.

## See Also

### Functions

`checkhdl` | `makehdl`

## More About

- “Generate HDL Code for Simscape Models” on page 32-9
- “Get Started with Simscape Electrical” (Simscape Electrical)
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97

## Troubleshoot Conversion of Simscape Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model

This example shows how to modify a Simscape™ plant model that is continuous time and contains nonlinear elements to generate an HDL-compatible Simulink® model. You can then generate HDL code for this Simulink model.

### Introduction

The Simscape HDL Workflow Advisor converts the Simscape plant model to an HDL-compatible implementation model from which you generate HDL code. In some cases, the Simscape plant model might not be compatible for implementation model generation. In such cases, you first modify the Simscape plant model and then run the Advisor.

This example illustrates how to modify a permanent magnet synchronous motor model for compatibility with Simscape HDL Workflow Advisor. The model is continuous time and contains many nonlinear components. You modify this model to a discrete-time switched linear model and then run the Simscape HDL Workflow Advisor.

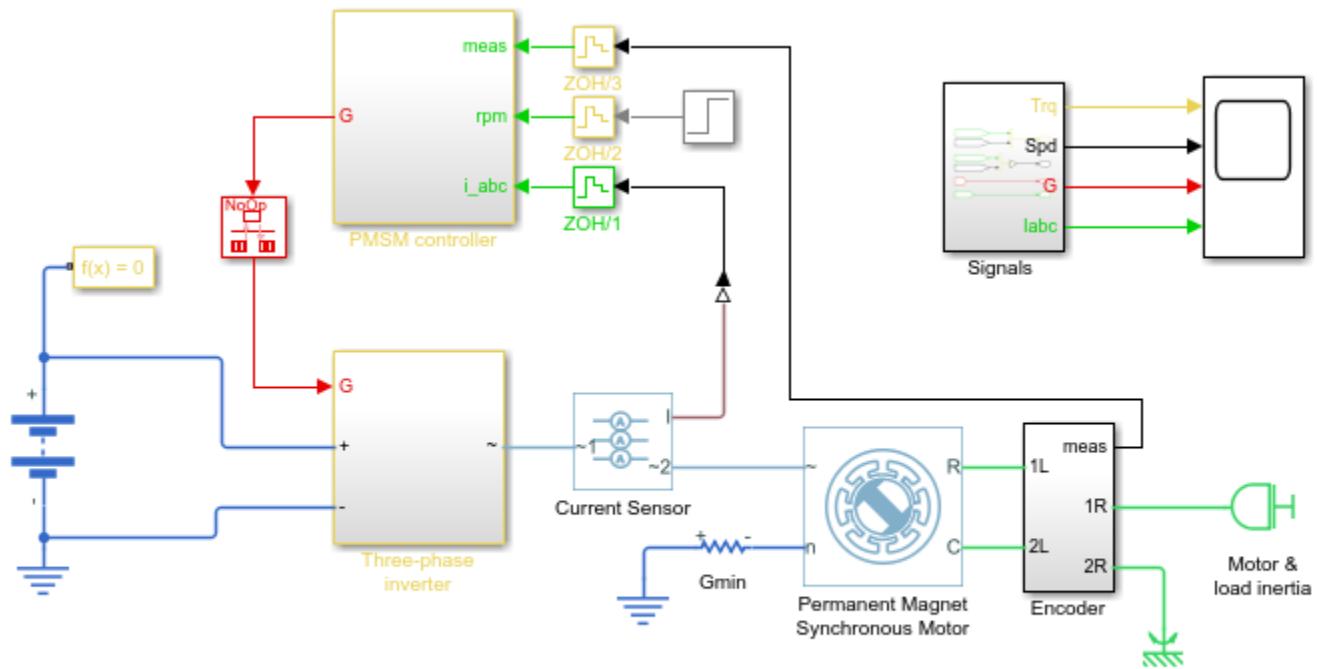
### Permanent Magnet Synchronous Motor Model

The permanent magnet synchronous motor model is a physical system in Simscape. To open the model, run this command:

```
open_system('ee_pmsm_drive')
```

Save this model as ee\_pmsm\_drive\_original.slx.

```
open_system('ee_pmsm_drive_original')
set_param('ee_pmsm_drive_original','SimulationCommand','Update')
```

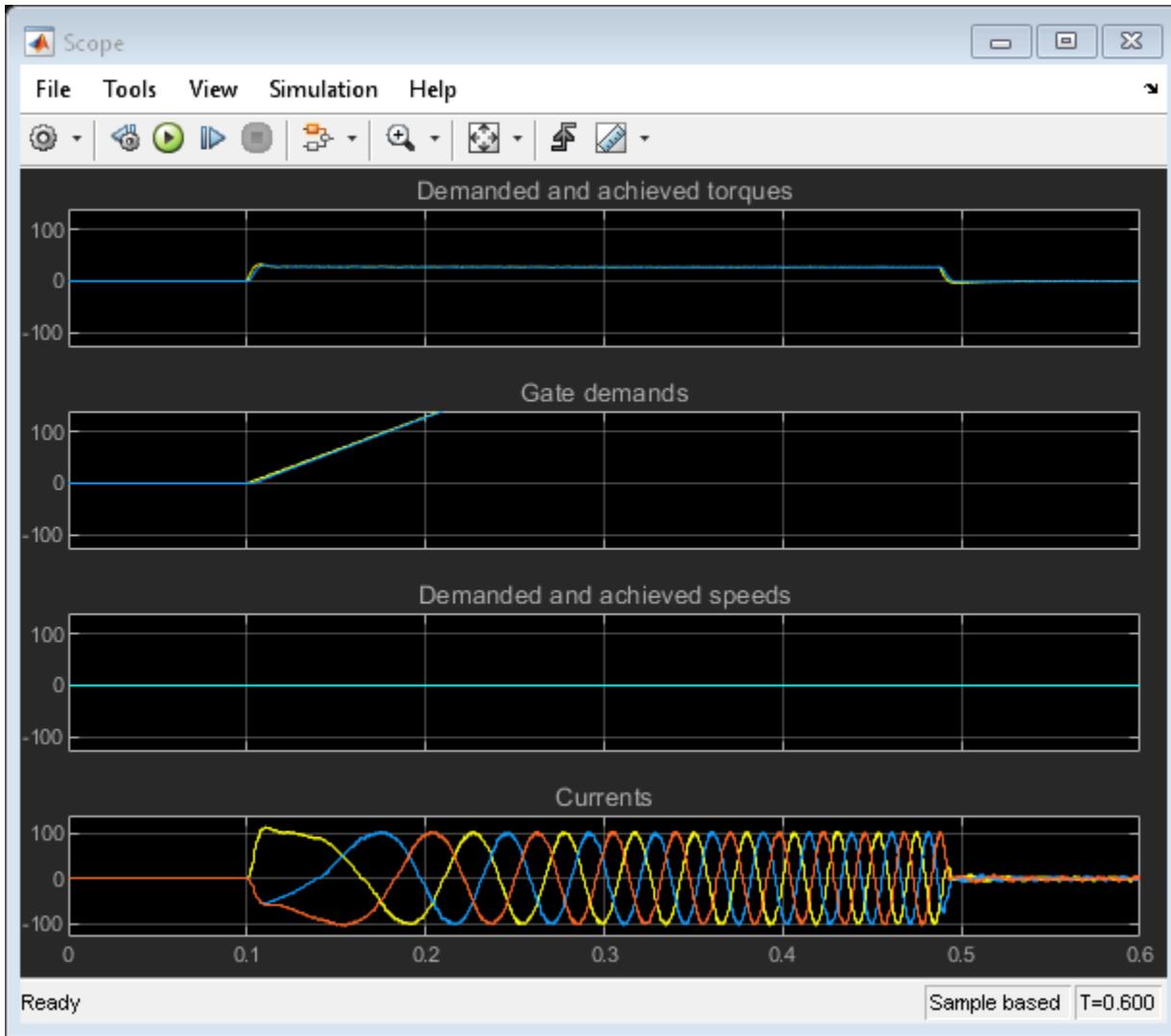


### Three-Phase PMSM Drive

1. [Explore simulation results](#) using `sscexplore`
2. [Learn more](#) about this example

The model contains a Permanent Magnet Synchronous Machine (PMSM) and inverter sized that you can use in a typical hybrid vehicle. The inverter is connected to the vehicle battery. To see how the model works, simulate the model.

```
sim('ee_pmsm_drive_original')
open_system('ee_pmsm_drive_original/Scope')
```



This model is a continuous time system. To use this model with Simscape HDL Workflow Advisor, convert the model into a discrete system. You then modify the model to use blocks that are compatible for the Simscape to HDL workflow.

### Convert Continuous-Time Model to Fixed-Step Discrete Model

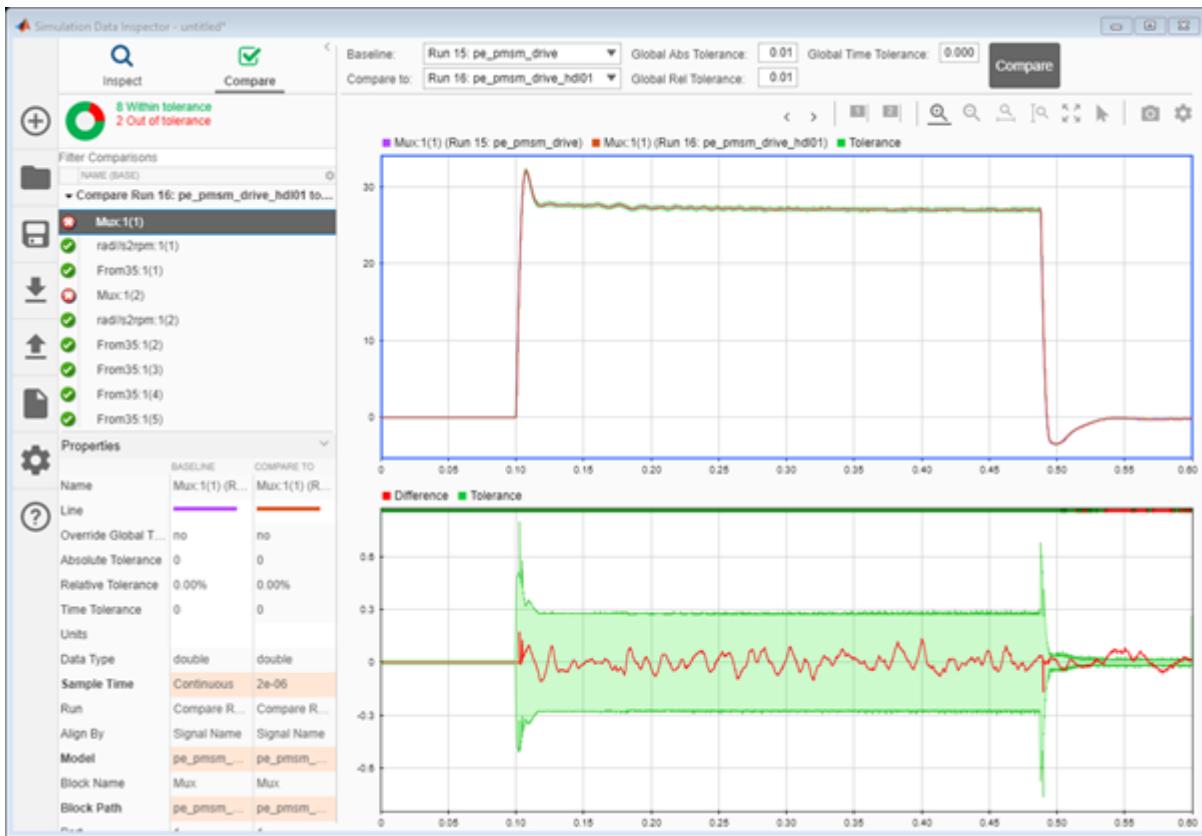
- Configure the solver options for HDL code generation by using a Solver Configuration (Simscape) block. In the block parameters:
  - Select **Use local solver**.
  - Use **Backward Euler** as the **Solver type**.
  - Specify a discrete **Sample time**,  $T_s$ .
- Modify the Solver settings of the model. On the **Modeling** tab, click **Model Settings**. On the **Solver** pane:
  - Set **Solver selection type** to **Fixed-Step**.

- Set **Solver** to discrete (no continuous states).
- Set **Fixed-step size (fundamental sample time)** to  $T_s$ .
- In the section **Tasking and sample time options**, clear **Treat each discrete rate as a separate task**.

3. Modify the display settings of your model. On the **Debug** tab, select **Information Overlays > Sample Time > Colors**. Review the Sample Time Legend for blocks that have a sample time other than  $T_s$ , or run at a continuous time scale. Double-click the Step block and set the **Sample time** to  $T_s$ .

4. For faster simulation, ignore the zero-sequence parameters of the PMSM. Double-click the Permanent Magnet Synchronous Motor block and set **Zero Sequence** to **Exclude**.

The model is now a fixed-step discrete system. Simulate the model and compare the **Torque Demand** and **Motor Torque** signals in the Simulation Data Inspector. The signals differ by more than the tolerance levels toward the end of simulation but are within acceptable limits.



You use a two-step process to convert the Simscape plant model to a HDL-compatible implementation model:

- 1 Implement a Simulink model that replaces the nonlinear part of the Simscape algorithm by using equivalent Simulink blocks.
- 2 Modify this model to use blocks that are compatible for Simscape to HDL workflow.

### Replace Nonlinear Simscape Blocks with Equivalent Simulink Implementation

1. To make the Simscape plant model HDL-Compatible, identify the presence of any nonlinear components or blocks in the model:

```
simscape.findNonlinearBlocks('ee_pmsm_drive_original')
```

```
Found network that contains nonlinear equations in the following blocks:  
{'ee_pmsm_drive_original/Permanent Magnet Synchronous Motor'}
```

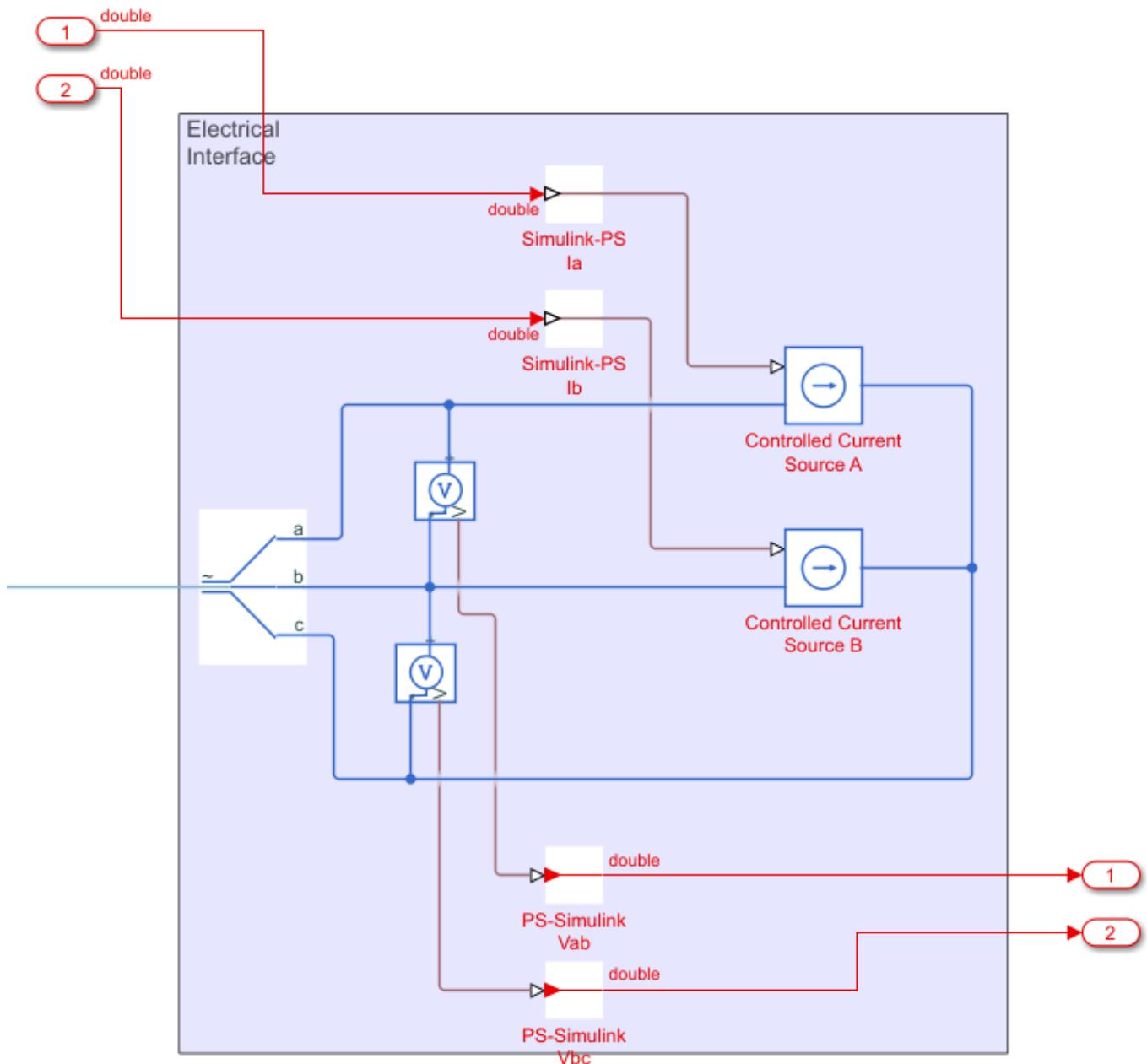
```
The number of linear or switched linear networks in the model is 0.  
The number of nonlinear networks in the model is 1.
```

```
ans =  
1x1 cell array  
{'ee_pmsm_drive_original/Permanent Magnet Synchronous Motor'}
```

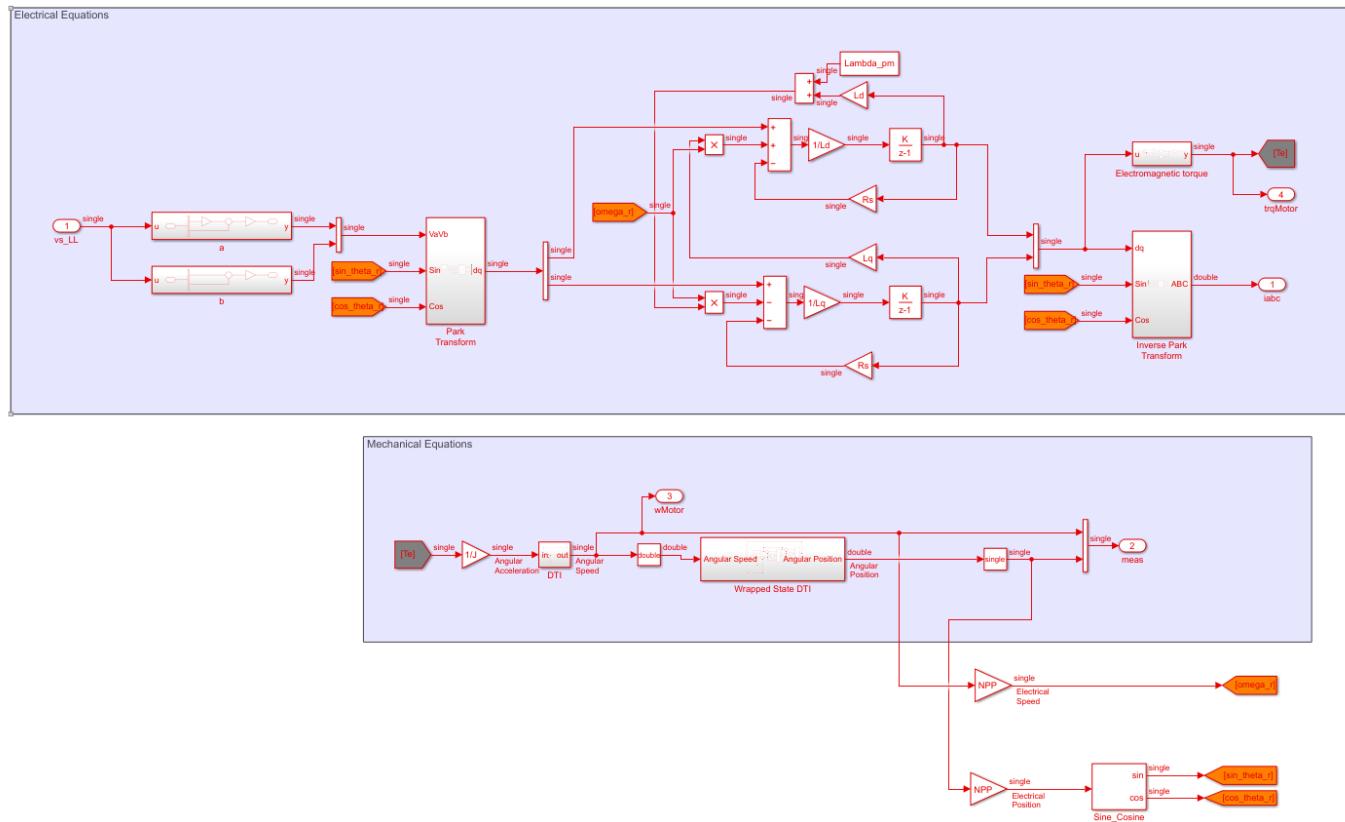
The Simscape plant model has a nonlinear block, which is the PMSM block.

2. The PMSM block, Encoder block, Gmin resistor, and Motor & Load Inertia block are replaced with Simulink blocks that perform the equivalent algorithm.

To implement the Electrical Interface block, you use Controlled Current Sources.



The interface to the PMSM is isolated from the implementation. To implement the PMSM by using Simulink blocks, you use Electrical Equations and Mechanical Equations. Inside the Park Transform and Inverse Park Transform blocks, eliminate the Sine and Cosine blocks.



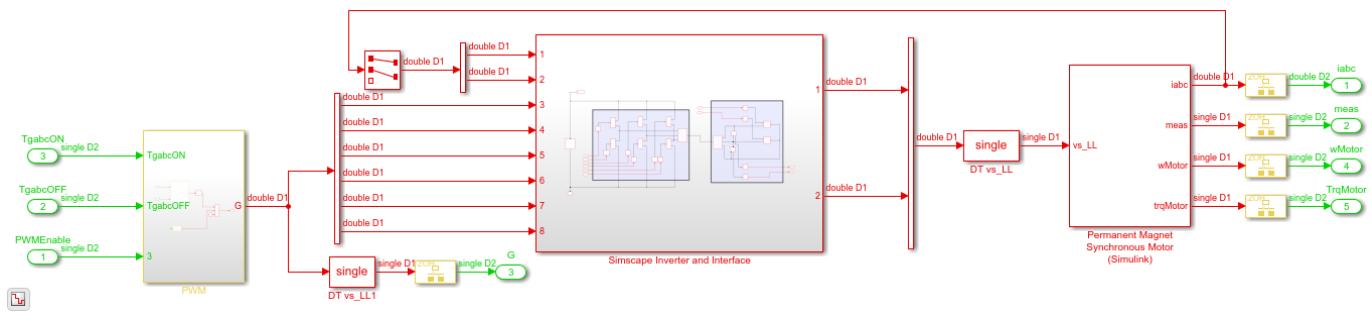
### Identify Simscape Blocks that Run on FPGA and Restructure Simscape Model

The ee\_pmsm\_drive\_singleSL model illustrates how you modify the original model ee\_pmsm\_drive\_original and prepare the model for readiness with Simscape HDL Workflow Advisor.

1. To modify the Simscape model for compatibility with HDL implementation model generation, identify the part of the Simscape algorithm that you want to run on the FPGA. In this example, you can run the three-phase inverter, electrical interface, PWM, and the Permanent Magnet Synchronous Motor (Simulink) on the FPGA.
2. After blocks to run on the FPGA have been identified, the blocks are placed inside a top-level subsystem. This subsystem is the DUT (Design Under Test) and contains blocks you run on the FPGA after generating the HDL implementation model. After running the Simscape HDL Workflow Advisor, this subsystem is replaced with the HDL algorithm. This part of the Simscape model must run at the fastest sample rate. Rate Transition blocks are added to upsample the design.
3. To save resource usage on the target hardware, Data Type Conversion blocks are added to convert the model to use **single** data types.

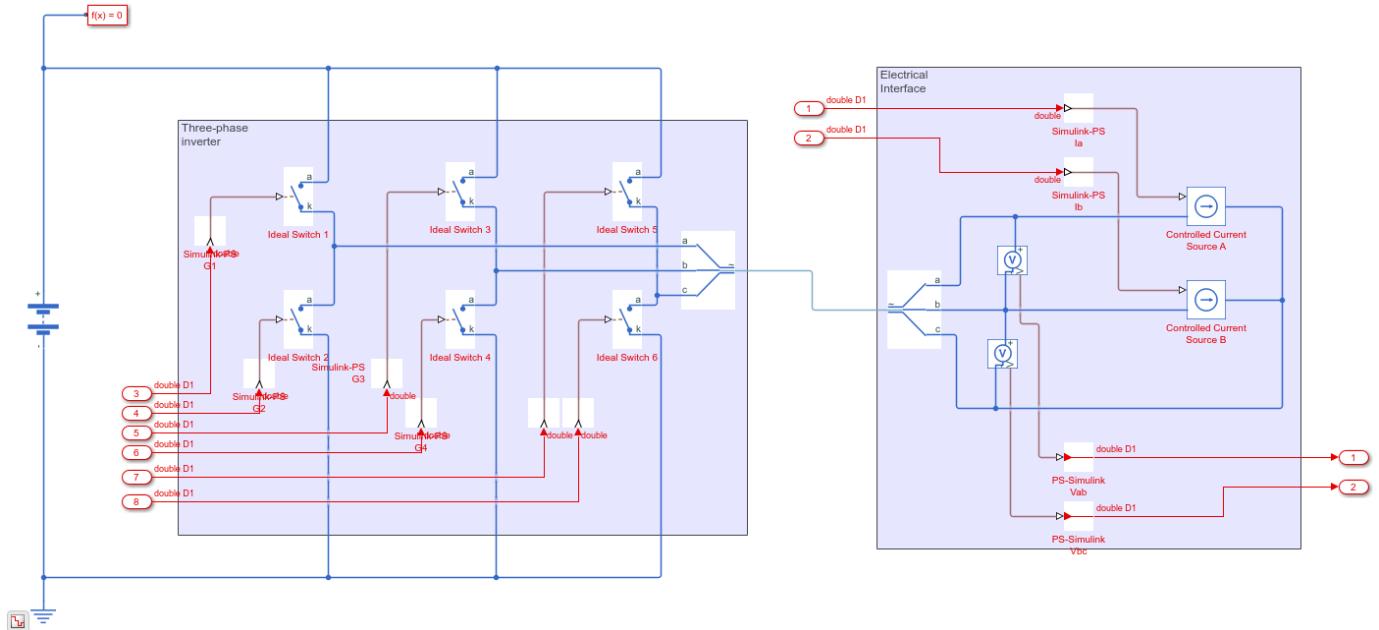
The ee\_pmsm\_drive\_singleSL model shows how these blocks are placed inside a top-level subsystem Subsystem1, which is the DUT. The blocks inside the subsystem are running at a faster rate.

```
load_system('ee_pmsm_drive_singleSL')
set_param('ee_pmsm_drive_singleSL','SimulationCommand','update')
open_system('ee_pmsm_drive_singleSL/Subsystem1')
```



In the ee\_pmsm\_drive\_singleSL model, the three-phase inverter and electrical interface are placed inside the Simscape Inverter and Interface subsystem.

```
open_system('ee_pmsm_drive_singleSL/Subsystem1/Simscape Inverter and Interface')
```

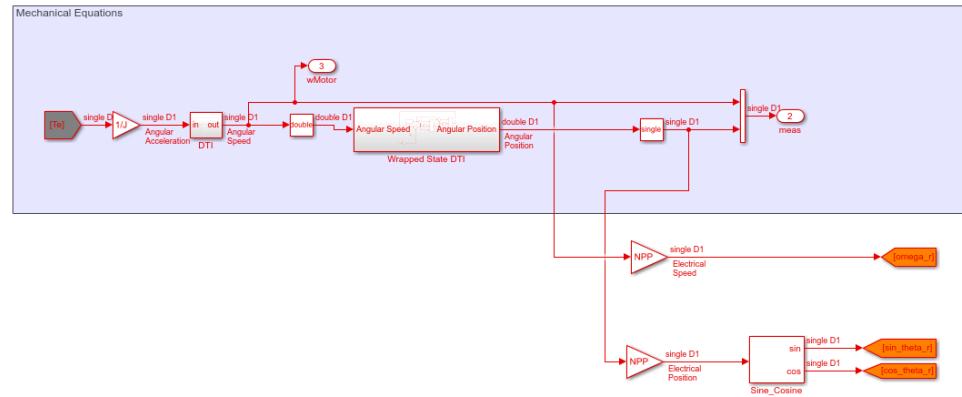
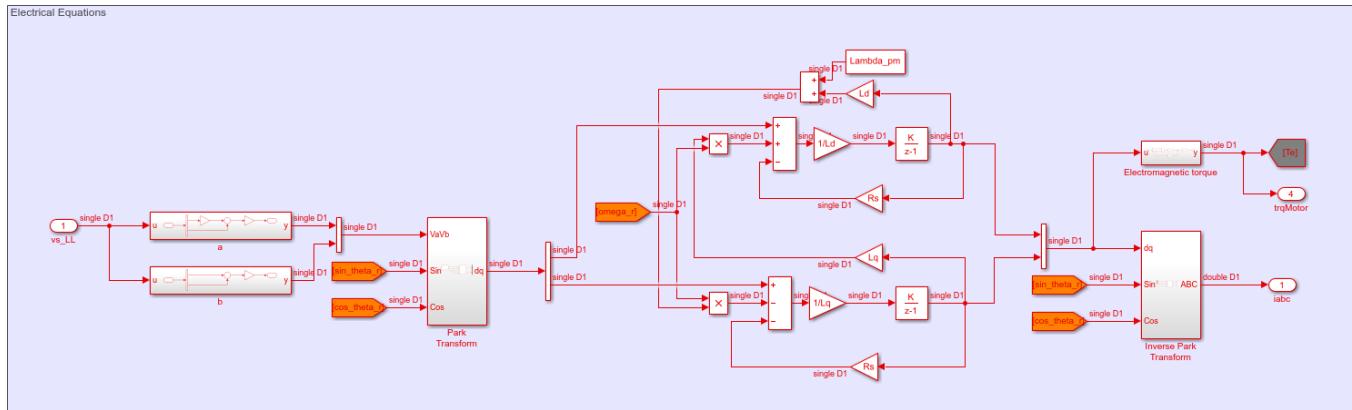


## Modify Permanent Magnet Synchronous Motor Subsystem for HDL Compatibility

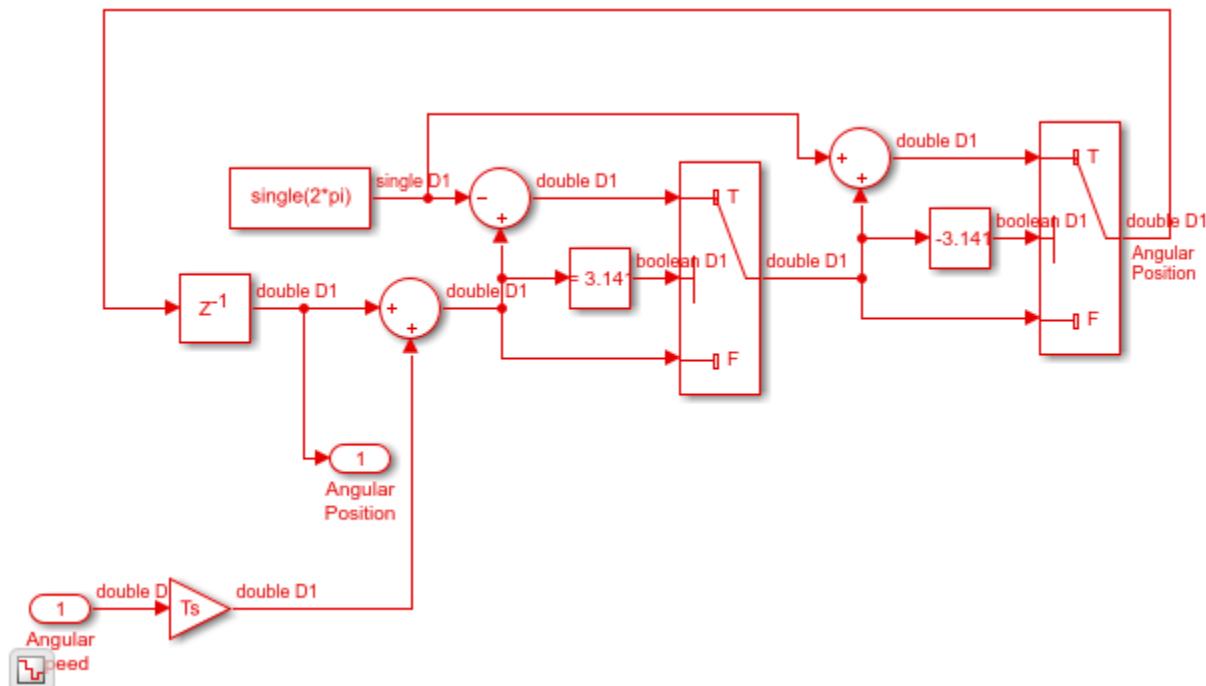
The preceding section describes the changes that have been applied to the masked subsystem, Permanent Magnet Synchronous Motor (Simulink).

1. The Integrator with Wrapped State (Discrete or Continuous) block is not compatible for HDL code generation. This block has been replaced with a Wrapped State DTI subsystem.

```
PMSMSSubsystem = 'ee_pmsm_drive_singleSL/Subsystem1/Permanent Magnet Synchronous Motor (Simulink)'
open_system(PMSMSSubsystem, 'force')
```



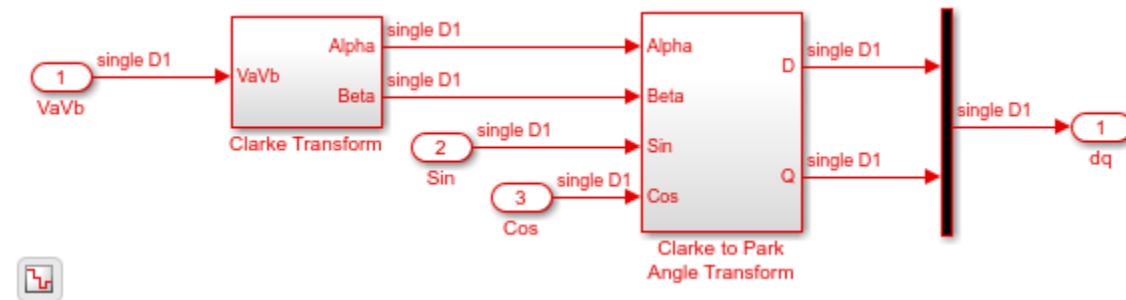
```
open_system([PMSMSubsystem, '/Wrapped State DTI'])
```



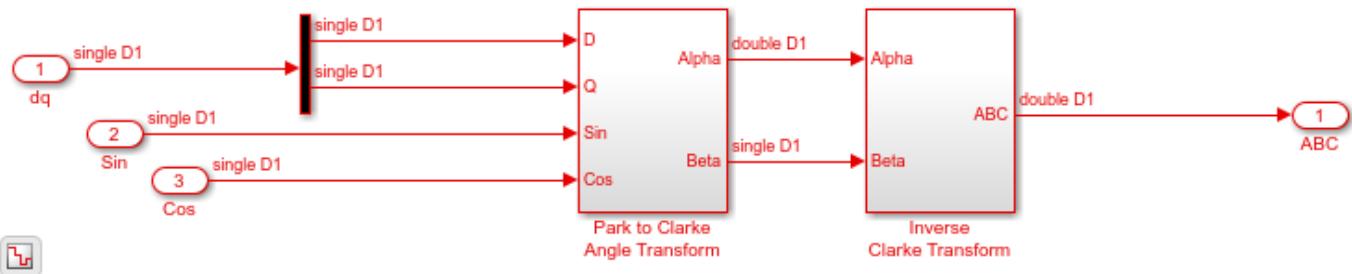
2. To reduce the FPGA area footprint for the:

- Park Transform block, Clarke Transform and Clarke to Park Angle Transform blocks are added.
- Inverse Park Transform block, Inverse Park to Clarke Angle Transform and Inverse Clarke Transform blocks are added.

```
open_system([PMSMSubsystem, '/Park Transform'])
```



```
open_system([PMSMSubsystem, '/Inverse Park Transform'])
```



3. For the Discrete-Time Integrator blocks inside this subsystem, the **Sample time** is set to -1, **Gain value** to Ts, and **Integrator method** to Accumulation:Forward Euler. You can view these block parameters programmatically by running these commands.

```
blockDTI = find_system(PMSMSubsystem, 'LookUnderMasks', 'on', ...
    'blocktype', 'DiscreteIntegrator');
for n = 1:numel(blockDTI)
    Integpath = blockDTI(n);
    Integname = get_param(Integpath, 'Name');
    stime = num2str(get_param(blockDTI{n}, 'SampleTime'));
    gval = num2str(get_param(blockDTI{n}, 'gainval'));
    integmethod = num2str(get_param(blockDTI{n}, 'IntegratorMethod'));
    disp('-----')
    disp(Integpath)
    disp(['Sample time: ', stime, ' Gain: ', gval, ...
        ' Integration method: ', integmethod])
end
disp('-----')
```

```
-----  
{'ee_pmsm_drive_singleSL/Subsystem1/Permanent Magnet...'}  
Sample time: -1      Gain: Ts      Integration method: Accumulation: Forward Euler  
-----
```

```
{'ee_pmsm_drive_singleSL/Subsystem1/Permanent Magnet...'}]
```

Sample time: -1 Gain: Ts Integration method: Accumulation: Forward Euler

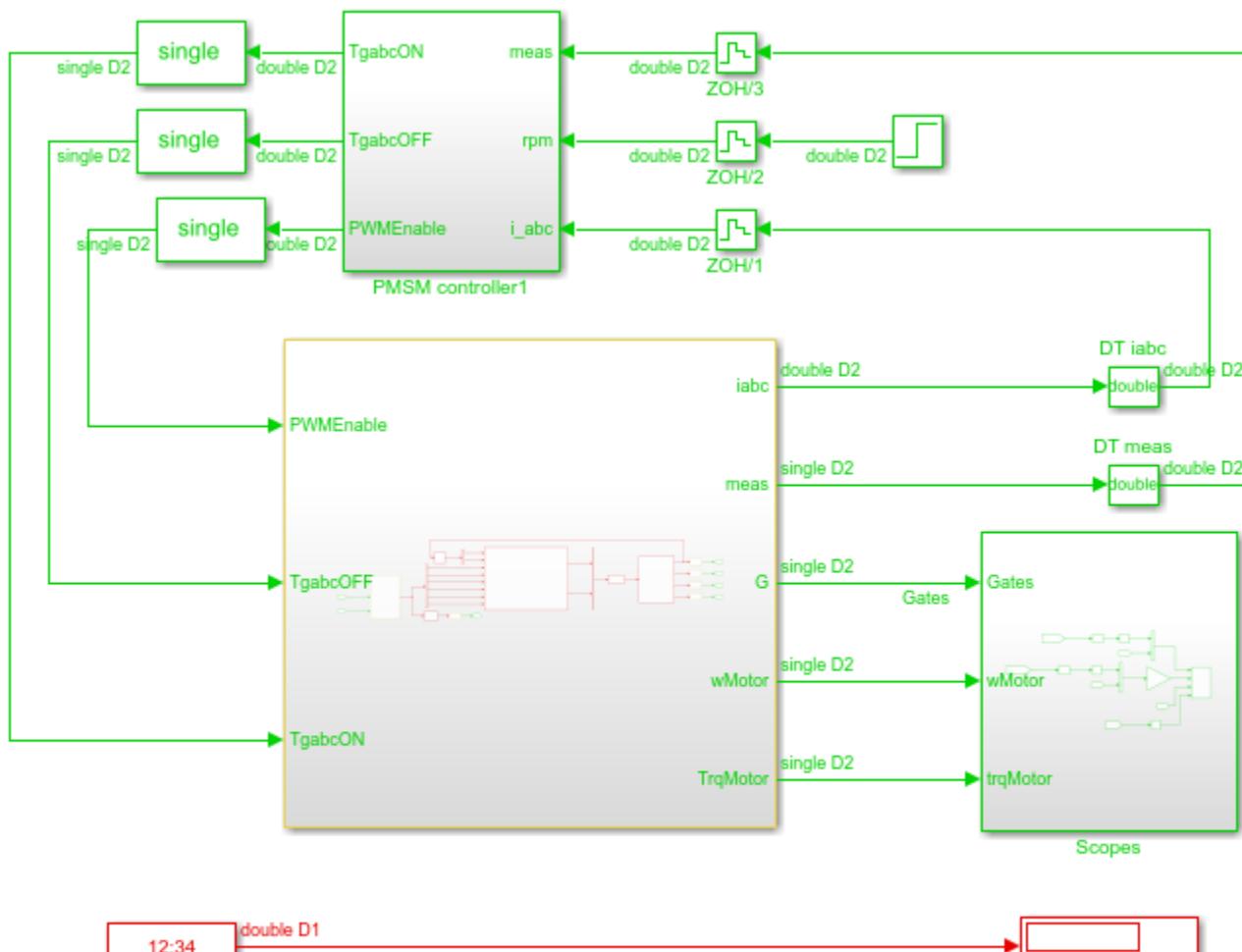
### Prepare Model and Run Simscape HDL Workflow Advisor

To the top level of the model:

- 1 A Digital Clock that has **Sample time** Ts has been added and connected to a Display block.
- 2 The Three-Phase Current Sensor Simscape block is replaced by feeding the controller with three-phase currents coming from the PMSM model.

This figure illustrates the top level of the model with the above changes.

```
open_system('ee_pmsm_drive_singleSL')
```



**Three-Phase PMSM Drive**

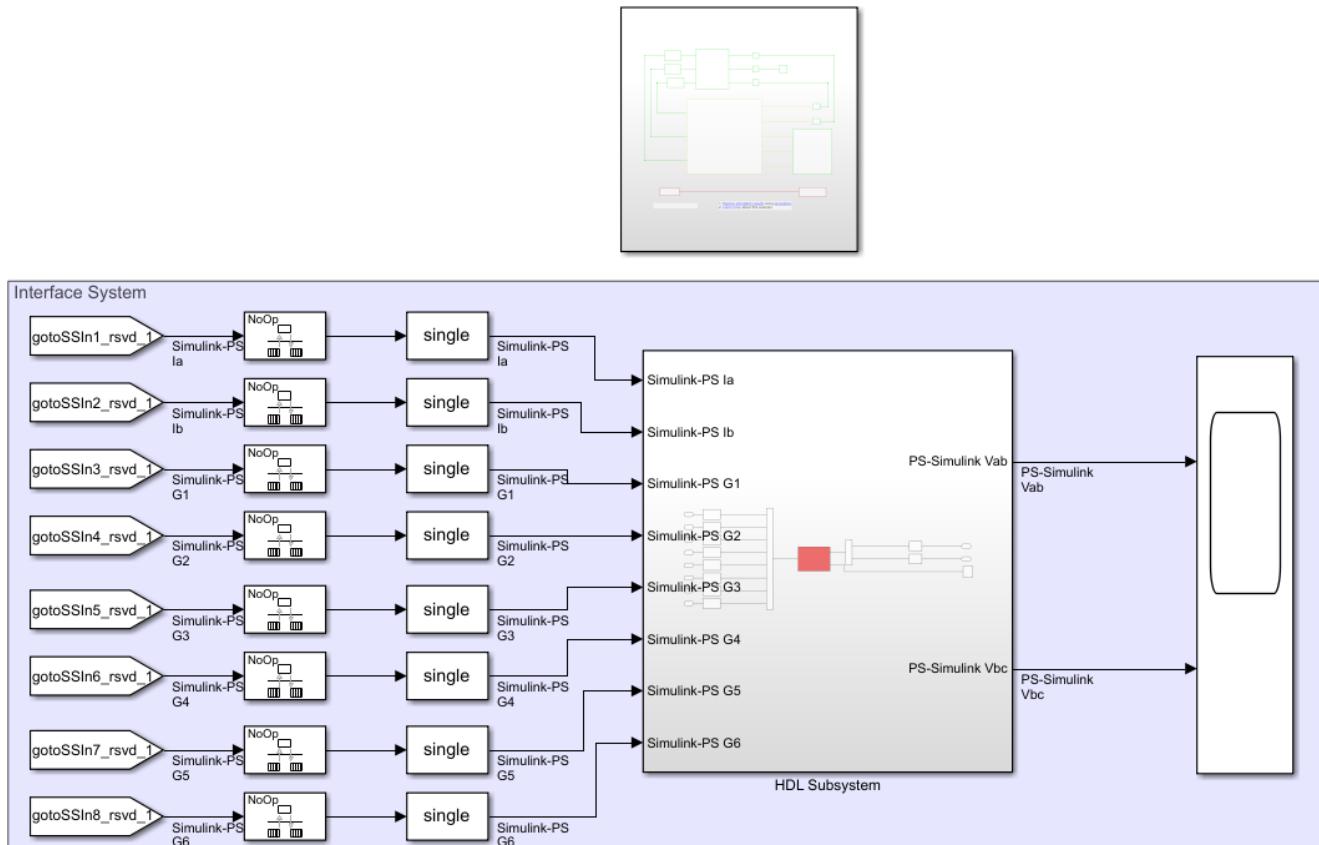


1. [Explore simulation results](#) using `sscexplore`
2. [Learn more](#) about this example

To open the Simscape HDL Workflow Advisor, run the `sschdladvisor` function for your model:

```
sschdladvisor('ee_pmsm_drive_singleSL')
```

To generate the implementation model, in the Simscape HDL Workflow Advisor, leave the default settings and then run the tasks. To open the implementation model, in the **Generate implementation model** task, click the link.



### Reconfigure Implementation Model for HDL Code Generation

In this example, the implementation model has been modified for deployment to Speedgoat FPGA I/O platforms. The model is resaved as `gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL`.

To reconfigure the single-precision implementation model for HDL code generation:

1. Run the `hdlsetup` function on the model.

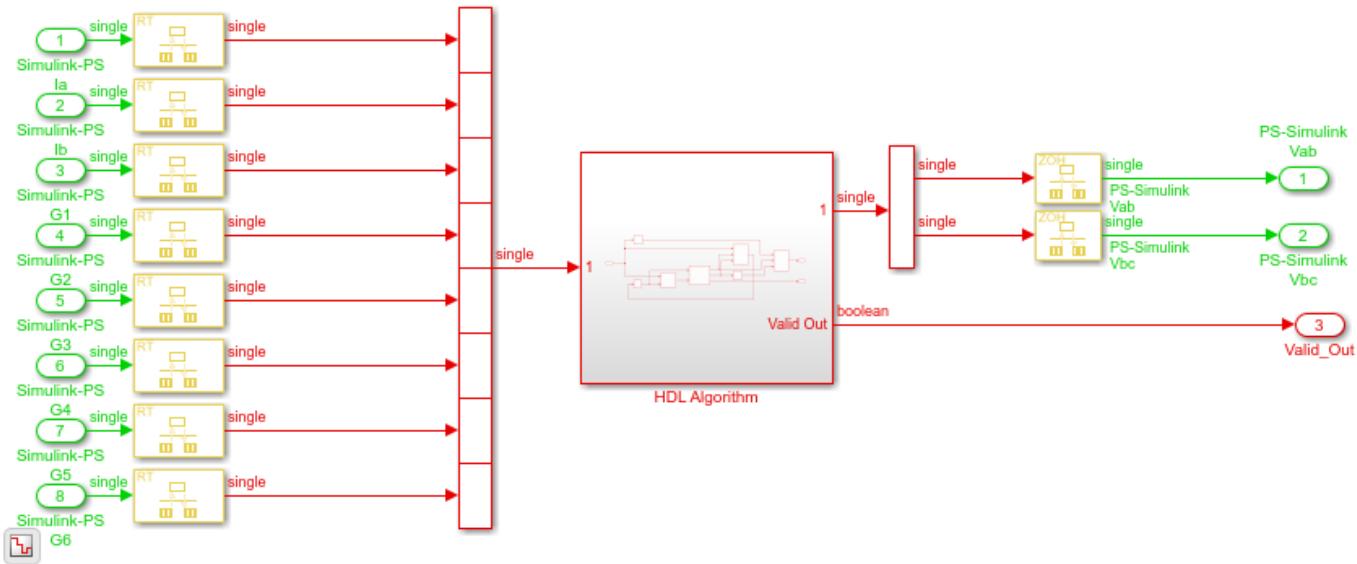
```
hdlsetup('gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL')
```

2. The model solver setting, **Fixed-step size**, is modified to `Ts/5` because the default **Number of solver iterations** is 5.

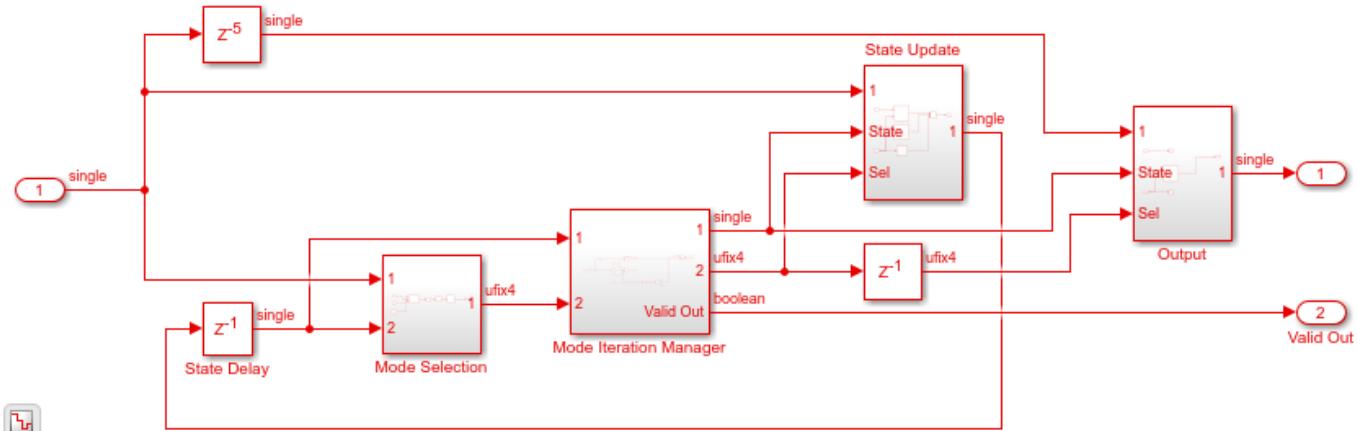
3. The Subsystem1 block contains blocks that you run on the FPGA. The Simscape Inverter and Interface subsystem is replaced with the HDL Subsystem block. The HDL Subsystem block contains the HDL algorithm that contains the HDL implementation of the Simscape algorithm. To see the HDL algorithm implementation, open this block.

```
model_name = 'gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL';
dut_name = 'gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL/Subsystem1';
```

```
load_system(model_name)
set_param(model_name, 'SimulationCommand', 'Update')
open_system([dut_name, '/HDL Subsystem'])
```

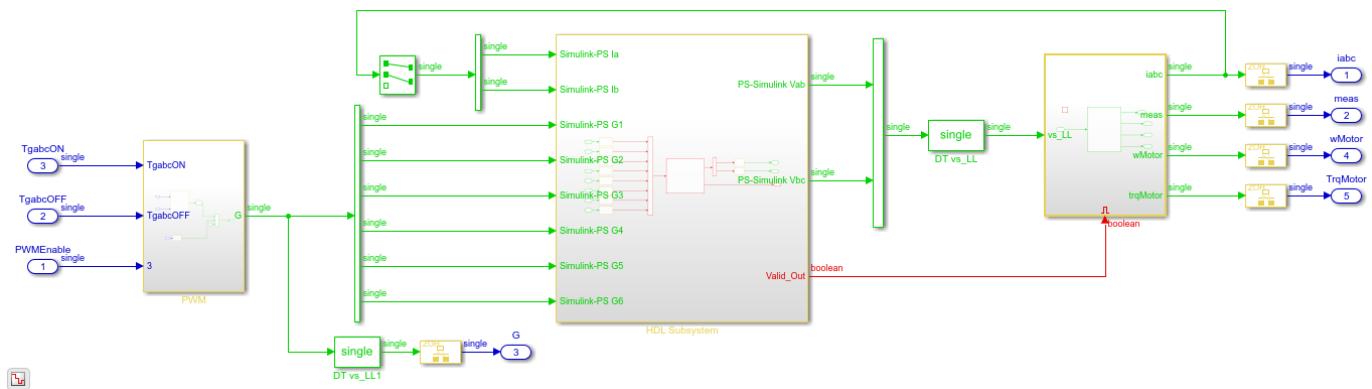


```
open_system([dut_name, '/HDL Subsystem/HDL Algorithm'])
```



4. The HDL Algorithm Subsystem has a Valid Out signal. The Permanent Magnet Synchronous Motor (Simulink) subsystem is placed inside an Enabled Subsystem and the vs\_LL input port is connected to the Valid Out signal.

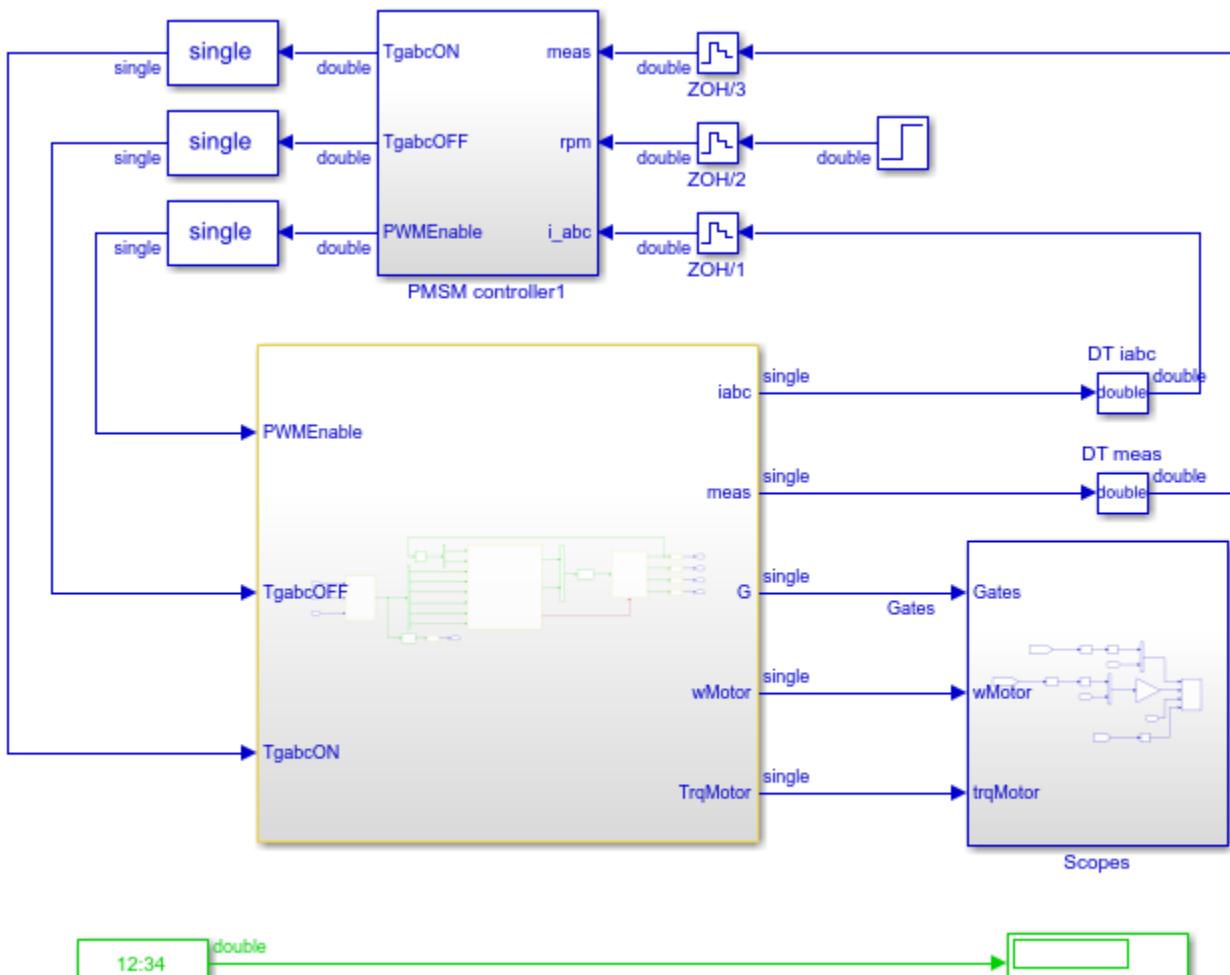
```
open_system(dut_name)
```



5. Move the block inside the subsystem that originally contained the Simscape algorithm to the top level of the model.

This figure illustrates the top level of the model with the above changes.

```
open_system(model_name)
```



### Three-Phase PMSM Drive

1. [Explore simulation results using ssceexplore](#)
2. [Learn more about this example](#)

## Generate HDL Code

Before you generate HDL code, to compare the output of the generated model after code generation with the modified Simscape plant model, specify validation model generation.

```
hdlset_param(model_name, 'GenerateValidationModel', 'on');
```

To learn more, see “Generated Model and Validation Model” on page 24-10.

To generate HDL code, run this command:

```
makehdl('gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL/HDL Subsystem')
```

By default, HDL Coder generates VHDL code. To generate Verilog code, run this command:

```
makehdl('gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL/HDL Subsystem', 'TargetLanguage', 'Verilog')
```

The code generator saves the generated HDL code and the validation model in the `hdlsrc` folder. The generated code is saved as `HDL_Subsystem_tc.vhd`. To see the resource usage information of your design, view the Code Generation Report.

To open the validation model, after you generate HDL code, open the `gm_gmStateSpaceHDL_ee_pmsm_drive_GenerateHDL_vnl.slx` model.

### Deploy Permanent Magnet Synchronous Motor to Speedgoat FPGA I/O Modules

In the HDL implementation model, Subsystem1 contains blocks you run on the FPGA. You can run the HDL Workflow Advisor on this Subsystem to deploy the HDL algorithm onto FPGA boards in Speedgoat target platforms. For an example, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 32-90.

## See Also

### Functions

`checkhdl` | `makehdl`

## More About

- “Generate HDL Code for Simscape Models” on page 32-9
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25
- “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-6
- “Get Started with Simscape Electrical” (Simscape Electrical)

## Replacing Variable Resistors

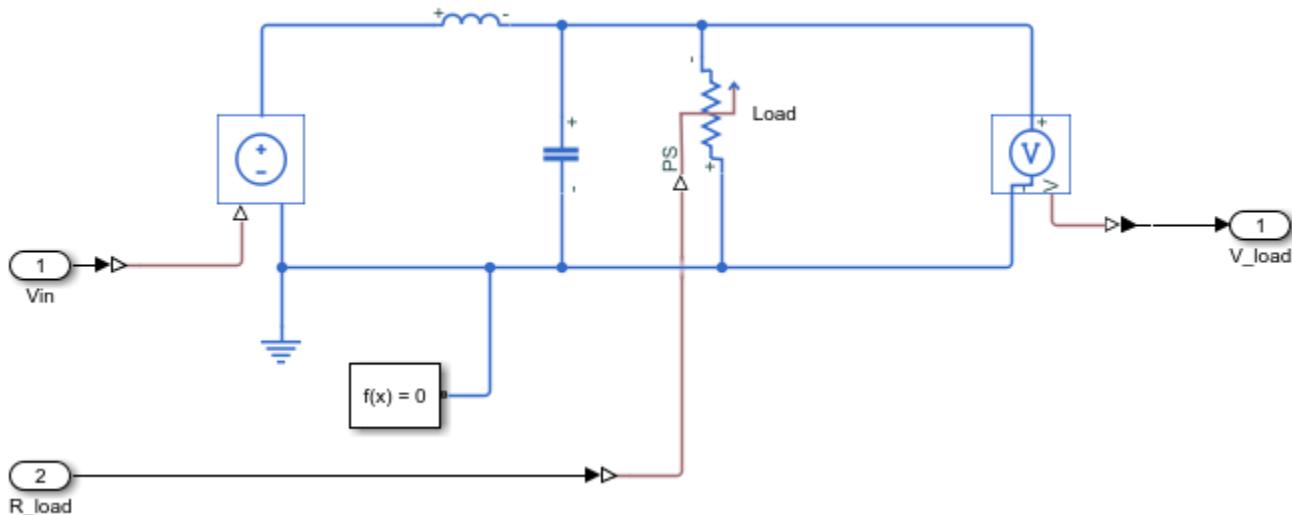
This example shows how to convert a model that is nonlinear due to a variable resistor into a switched linear model making it compatible with Simscape to HDL Workflow.

### Introduction

Simscape to HDL Workflow supports conversion of Simscape switched linear models to functionally-equivalent Simulink models that are compatible for HDL code generation. Due to the nature of the equations that result from variable resistors, blocks such as Piecewise-Constant Resistor can lead to nonlinear behavior and must be replaced with equivalent switched linear components. Specifically the Piecewise-Constant Resistor contains events that are not supported by the Simscape HDL Workflow Advisor.

Open The Simscape™ Model. In the MATLAB® command prompt, enter:

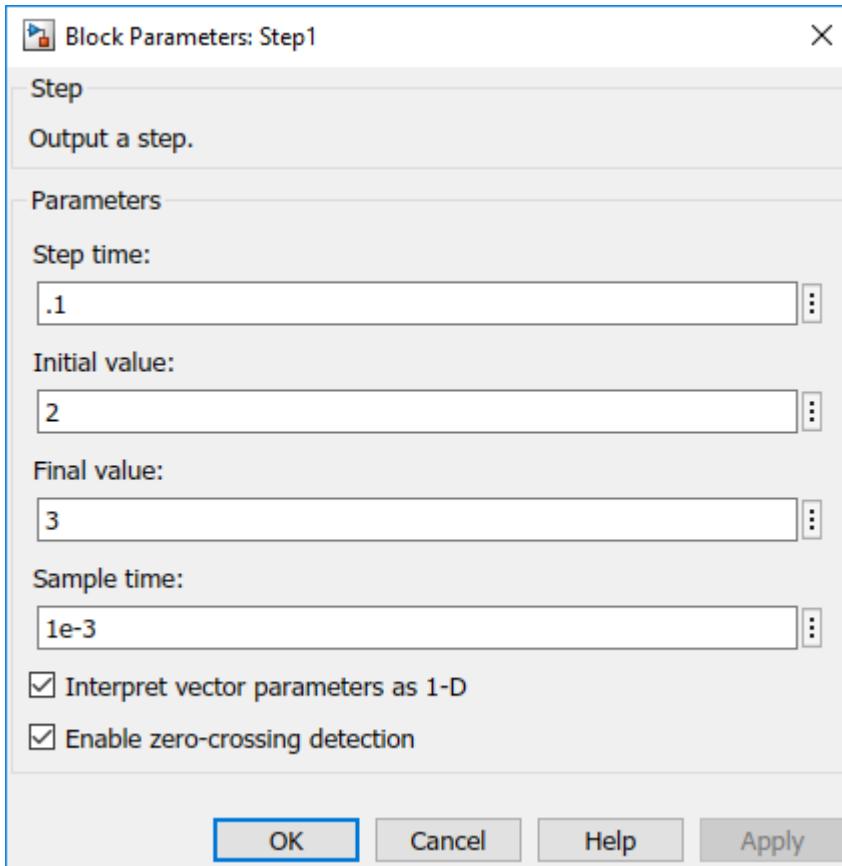
```
nonlinearModel = 'sschdlexVariableResistorExample';
load_system(nonlinearModel)
open_system([nonlinearModel, '/Simscape Subsystem'])
```



This model is an RLC circuit with a Piecewise-Constant Resistor acting as the resistor or 'load'. For the Piecewise-Constant Resistor, the relationship between voltage  $V$  and current  $I$  is  $V=I*R$  where  $R$  is the numerical value presented at the physical signal port  $R$ .

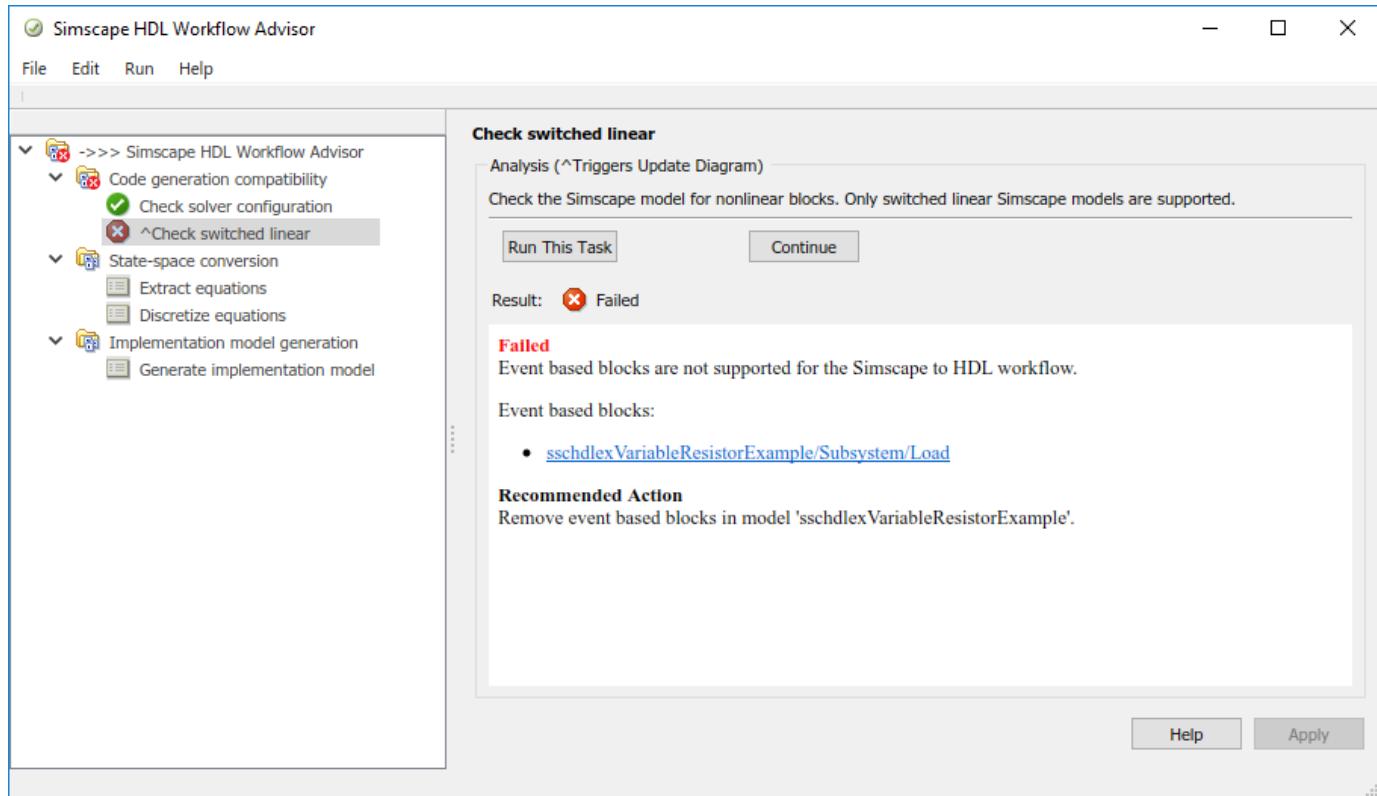
To ensure a positive value for the resistance, any value below  $1e-6$  is replaced by  $1e-6$ . This resistor is Piecewise-Constant because the resistance only changes when the input value differs from the current resistance by more than a set tolerance. Thus, a continuously changing input would be converted to a discrete set of resistances.

In this model the signal going into the Piecewise-Constant Resistor is a step function that changes from 2 to 3 at  $t=0.1$  thus changing the load resistance from  $2 \Omega$  to  $3 \Omega$ .



To open the Simscape HDL Workflow Advisor at the command-line, enter:  
sschdladvisor(nonlinearmodel)

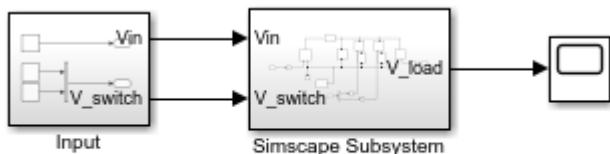
Run the workflow to the Get state-space parameters task. This task fails because of the presence of the Piecewise-Constant Resistor.



### Replace Variable Resistor with switches and constant resistors.

To convert this model to an equivalent switched linear model, replace the Piecewise-Constant Resistor with a set of switches and resistors for each desired value. To open the switched linear version in the MATLAB® command prompt, enter:

```
switchedLinearModel = 'sschdlexVariableResistorSwitchedLinearExample';
load_system(switchedLinearModel)
open_system(switchedLinearModel)
```



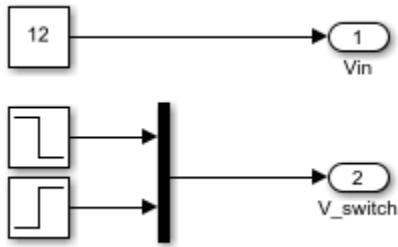
Copyright 2019 The MathWorks, Inc.

The variable resistor has been replaced by a resistor and switch for each desired resistance. To recreate the behavior of a load resistance that changes from  $2 \Omega$  to  $3 \Omega$  at  $t=0.1$  two resistors are used, one with a resistance of  $2 \Omega$  and the other with a resistance of  $3 \Omega$ . By closing and opening the switches the load resistance switches from  $2 \Omega$  to  $3 \Omega$ .

### Controlling the Switches

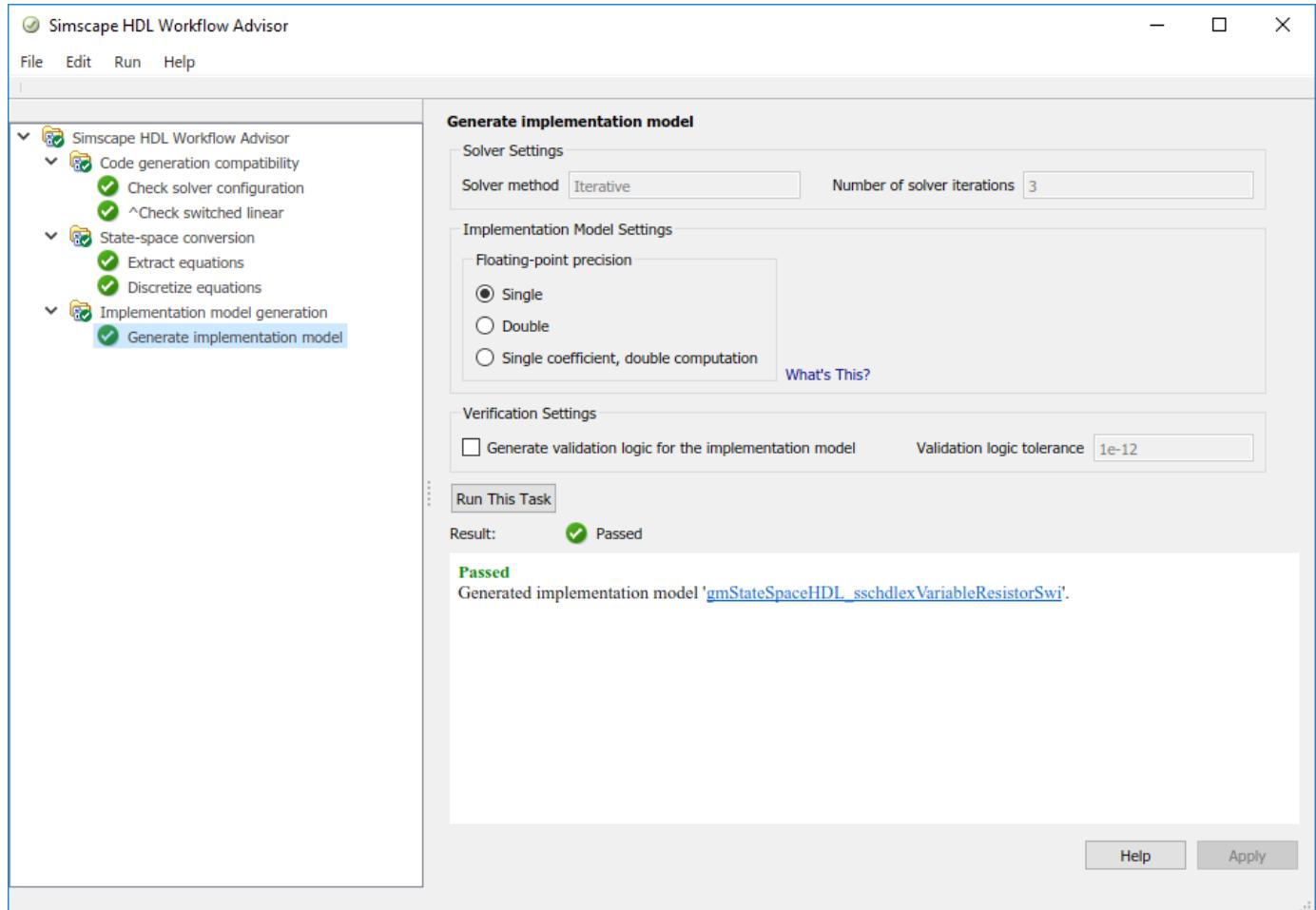
The switches must be turned on and off to provide the correct load resistance. To view the control signals for the switches in the MATLAB® command prompt, enter:

```
open_system([switchedLinearModel, '/Input'])
```



To achieve the correct resistance, create two step functions. One to open the switch in series with the  $2\ \Omega$  resistor at  $t=0.1$  and another to close the switch in series with the  $3\ \Omega$  resistor at the same time.

Now that the variable resistor has been replaced with switched linear components run the Simscape HDL Workflow Advisor and see that all the tasks run to completion.



By changing the variable piecewise resistor to a number of specified resistors that switch on and off the model has been changed to a form that is compatible with the Simscape to HDL workflow.

## Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules

This example shows how to synthesize and generate FPGA bitstream from a Simscape™ half-wave rectifier model and download the bitstream to a Speedgoat FPGA I/O 334-325K target for Hardware-in-the-Loop (HIL) implementation.

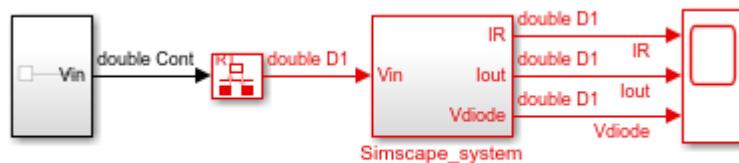
### Hardware-in-the-Loop Workflow

- 1 Generate a HDL implementation model from the Simscape model by using the Simscape HDL Workflow Advisor. The HDL implementation model is a Simulink® model that replaces the Simscape algorithm with HDL-compatible blocks
- 2 Generate FPGA bitstream for the HDL implementation model by using the HDL Workflow Advisor
- 3 Download the bitstream to the Speedgoat FPGA I/O module by using the Simulink Real-Time Explorer for Hardware-in-the-Loop Simulation.

### Half Wave Rectifier Model

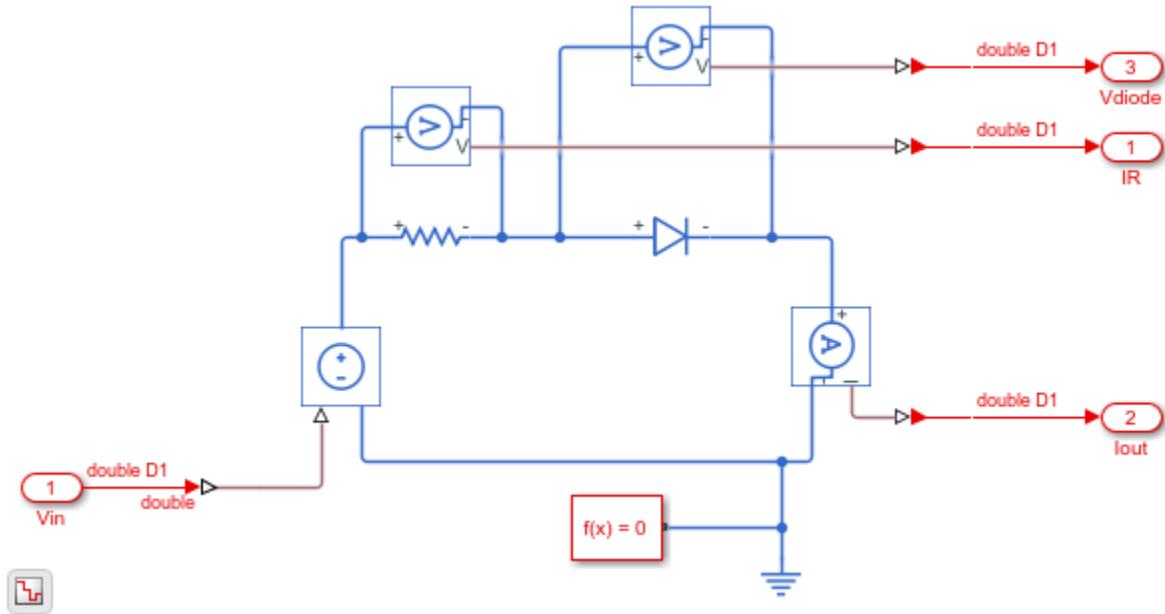
Open the Simscape half wave rectifier model. In the MATLAB® command prompt, enter:

```
modelName = 'sschdlexHalfWaveRectifierExample';
open_system(modelName)
set_param(modelName, 'SimulationCommand', 'update');
```



Copyright 2020 The MathWorks, Inc.

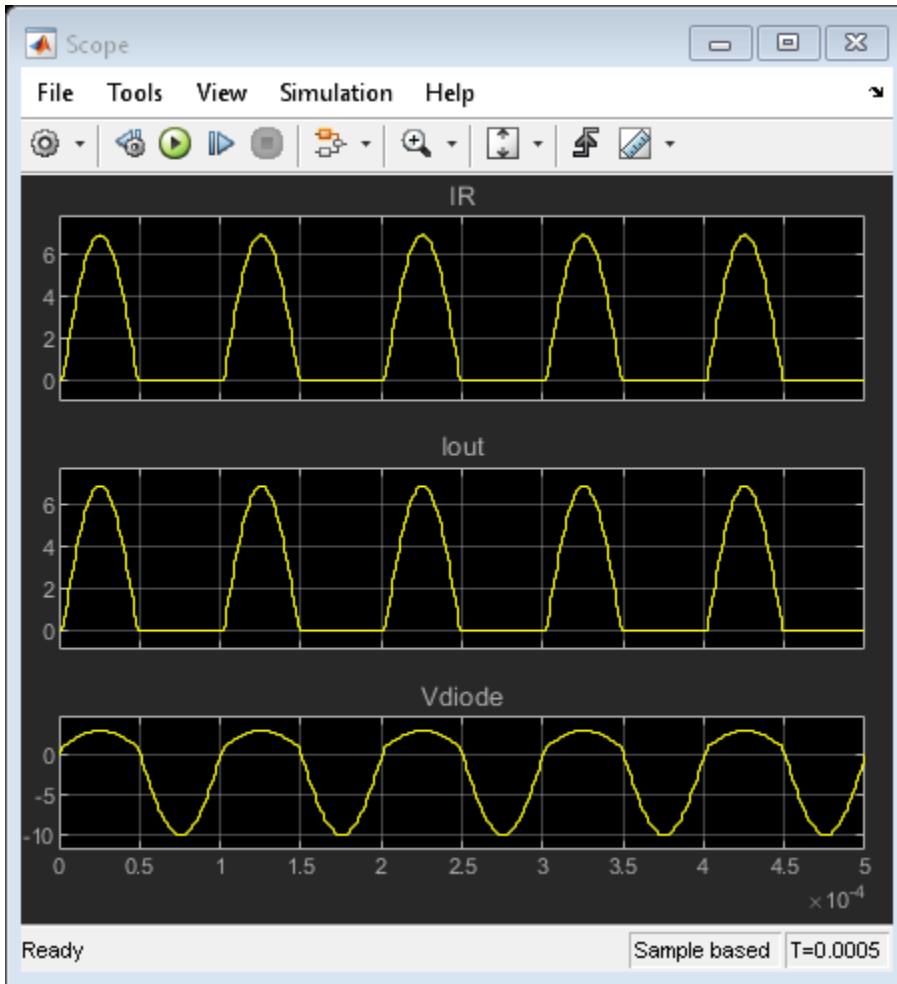
```
open_system([modelName, '/Simscape_system'])
```



The half-wave rectifier consists of a Resistor, which is a linear block, and a Diode, which is a switched linear block. At the input and output port interfaces, the model has Simulink-PS Converter and PS-Simulink Converter blocks. The solver settings are configured for compatibility with Simscape HDL Workflow Advisor. If you open the Block Parameters dialog box for the Solver Configuration block, **Use local solver** is selected and **Backward Euler** is specified as the **Solver type**. See “Get Started with Simscape Hardware-in-the-Loop Workflow” on page 32-2.

To see the algorithm functionality, simulate the model.

```
sim(ModelName)
open_system([ModelName, '/Scope'])
```



2. Configure the Simscape Model for HDL compatibility by using the `hdlsetup` function:

```
hdlsetup('sschdlexHalfWaveRectifierExample')
```

### Generate HDL Implementation Model

To generate the HDL implementation model:

1. Open the Simscape HDL Workflow Advisor:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

2. To compare functionality of the HDL implementation model with the original Simscape algorithm, select the **Generate implementation model** step, and then select the **Generate validation logic for the implementation model** check box. Use a **Validation logic tolerance** of 0.001. Right-click the **Generate implementation model** step and select **Run to Selected Task**.

The Advisor generates an HDL implementation model and a state-space validation model. To compare functionality of the HDL implementation model with the original Simscape algorithm, open and simulate the state-space validation model. The output of this model matches the original Simscape model. For a more systemic verification, see “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97.

See also “Simscape HDL Workflow Advisor Tasks” on page 33-2.

## Setup and Configuration

The Speedgoat IO334-325K FPGA module uses Xilinx® Vivado® and *IP Core Generation* workflow infrastructure. Before you deploy the HDL implementation model on the Speedgoat IO module:

### 1. Install Xilinx Vivado and Setup Tool Path

Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”. Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

### 2. Install Speedgoat Library and Speedgoat - HDL Coder Integration Packages

Install the Speedgoat Library and the Speedgoat - HDL Coder Integration packages. See [Install Speedgoat HCIP](#).

### 3. Setup I/O Module

For real-time simulation, set up the I/O module. See [Xilinx HDL Software for Speedgoat I/O Hardware](#).

## HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

To open the HDL Workflow Advisor, use the `hdladvisor` function.

```
hdladvisor('gmStateSpaceHDL_sschedlexHalfWaveRectifierEx/Simscape_system/HDL Subsystem')
```

The left pane contains folders that represent a group of related tasks. Expanding the folders and selecting a task displays information about that task in the right pane. The right pane can contain simple controls for running the task to advanced parameters and option settings that control code and test bench generation. To learn more about each task, right-click that task, and select **What's This?**. See “[Getting Started with the HDL Workflow Advisor](#)” on page 31-6.

## Generate FPGA Bitstream for Speedgoat Platform

1. Open the HDL implementation model, and then open the HDL Workflow Advisor for the implementation model.

```
open_system('gmStateSpaceHDL_sschedlexHalfWaveRectifierEx')
hdladvisor('gmStateSpaceHDL_sschedlexHalfWaveRectifierEx/HDL Subsystem')
```

2. In **Set Target Device and Synthesis Tool** task, specify **Target workflow** as [Simulink Real-Time](#) [FPGA I/O](#) and **Target platform** as [Speedgoat I0334-325K](#).

### 1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow:	Simulink Real-Time FPGA I/O		
Target platform:	Speedgoat IO334-325k	Launch Board Manager	
Synthesis tool:	Xilinx Vivado	Tool version: 2019.2.1	Refresh
Family:	Kintex7	Device:	xc7k325t
Package:	fpg676	Speed:	-2
Project folder:	hdl_prj	Browse...	

3. In the **Set Target Reference Design** task, select a value of x4 for the parameter **PCIe lanes**, and select **Run This Task**.

4. In **Set Target Interface** task, map the input and output **single** data type ports to **PCIe Interface** and select **Run This Task**.

### 1.3. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization:	Free running
---------------------------------	--------------

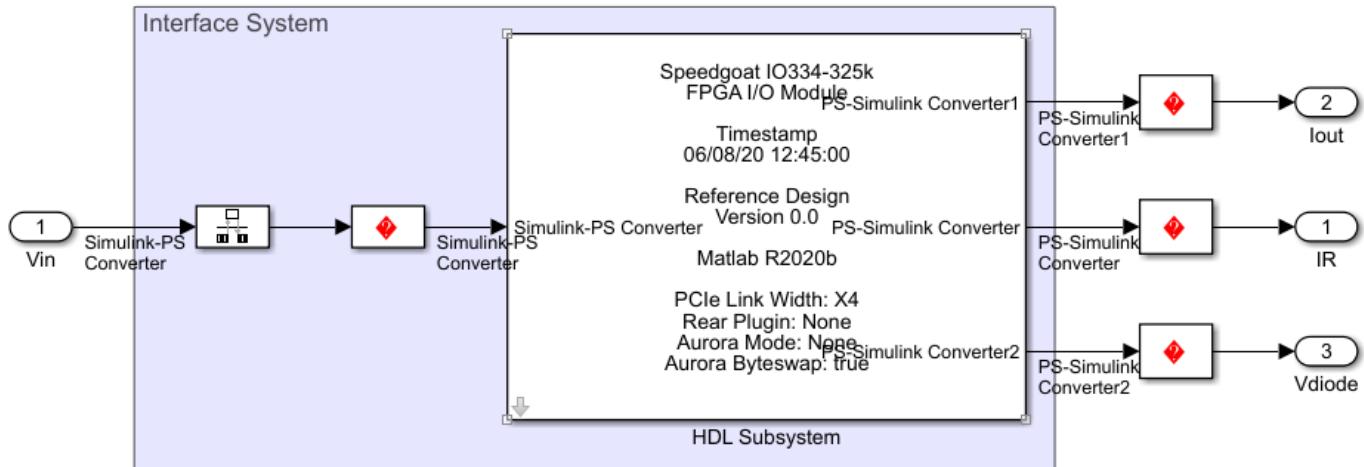
Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
Simulink-PS Converter	Import	single	PCIe Interface	x"100"	Options...
PS-Simulink Convert...	Outport	single	PCIe Interface	x"104"	
PS-Simulink Converter	Outport	single	PCIe Interface	x"108"	
PS-Simulink Convert...	Outport	single	PCIe Interface	x"10C"	

5. In the **Set Target Frequency** task, set the **Target Frequency (MHz)** as 100.

6. Right-click the **Generate Simulink Real-Time Interface** task and select **Run to Selected Task** to generate the HDL IP core, FPGA bitstream, and download the bitstream onto the Speedgoat IO334 target.

A Simulink Real-Time Interface model is generated, and named as **gm\_gmStateSpaceHDL\_sscldxHalfWaveRectifierEx\_slrt**.



For rapid prototyping, you can export the Workflow Advisor settings to a script. The script is a MATLAB file that you run from the command line. You can modify and run the script, or import the settings into the HDL Workflow Advisor User Interface. To save the workflow, in the HDL Workflow Advisor User Interface, select **File > Export to Script**. Save the file as `hdlworkflow_slrt_I0334.m`.

To import this file, in the HDL Workflow Advisor User Interface, select **File > Import from Script**. In the Import Workflow Configuration dialog box, select the `hdlworkflow_slrt_I0334.m` file. The HDL Workflow Advisor updates the tasks according to the imported script. See “Run HDL Workflow with a Script” on page 31-53.

## Deploy Bitstream to Speedgoat IO334-325k Target

### 1. Connect Development Computer to Target

Connect the development computer to the target by using a cross-over network cable. The Speedgoat Target IP address is `10.10.10.15`. Set the IP address of the communication link between the development computer and target computer to a value `10.10.10.12` because the communication link must be in the same network.

### 2. Setup and Configure Simulink Real-Time Explorer

You download the bitstream by using the Simulink Real-Time Explorer. To open the Simulink Real-Time Explorer, enter the command `slrtExplorer`. Alternatively, you can open the Explorer from the **REAL-TIME** tab of the Simulink Toolstrip.

`slrtExplorer`

a. In the **TARGET** pane click the **Add Target** button, and then click the **Properties** button on the toolbar. In the **Target Properties** Workspace, click **Host-to-Target Communication**.

- Set **IP Address** as `10.10.10.15`, **Port** as `22222`, **Subnet mask** as `255.255.255.0`, and **Gateway** as `10.10.10.10`.
- Set **Target driver** as **Auto** and **Bus type** as **PCI**.

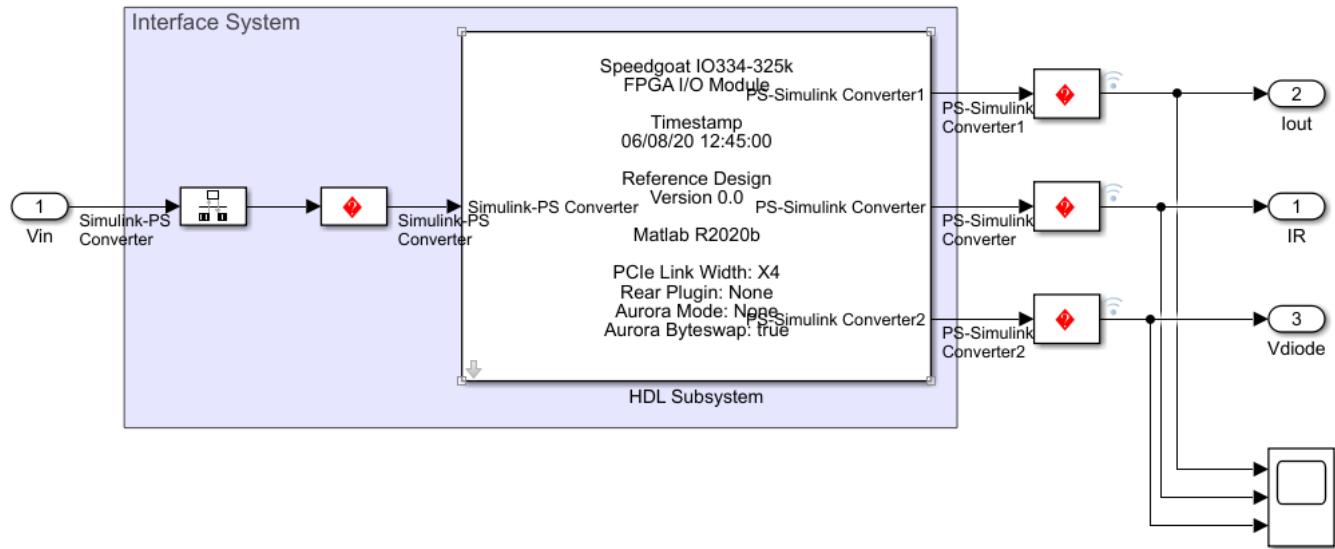
b. In the **Target Properties** Workplace, click \*Target Settings.

- Select **USB Support** and **Graphics mode**.
- Click **Boot Configuration** in **Target Properties** workspace.
- Select **Boot mode** as Network and click on **Create boot disk**. The Target MAC address appears in the **MAC address** field.

Save the configuration by clicking the **save** button.

### 3. Create Real-Time Application

Open the Simulink Real-Time Interface model. Add a Scope block to the model and connect it to the outputs. Log the output signals to view the simulation results on the Simulation Data Inspector.



### 4. Build and Run Real-Time Application

Click the **Run on Target** button on the **REAL\_TIME** tab to compile and download the model onto Speedgoat IO334-325k target.

A target object name `tg` is created in the MATLAB workspace and the model is run on the target. Observe the output simulation results on the Simulation Data Inspector. The simulation results of the downloaded model match the original Simscape model simulation.

# Validate HDL Implementation Model to Simscape Algorithm

If you design your algorithm by using Simscape switched linear blocks, you can run the Simscape HDL Workflow Advisor to generate an HDL implementation model. The HDL implementation model represents the Simscape algorithm by using Simulink blocks that are compatible for HDL code generation.

Before you prototype the implementation model on an FPGA or target Speedgoat FPGA I/O modules, you can verify the functionality of your design in the Simulink modeling environment. To verify the functionality, specify insertion of validation logic in the HDL implementation model when you run the Simscape HDL Workflow Advisor. This logic verifies whether the numeric results of the HDL implementation model match the original Simscape algorithm.

In some cases, there can be a mismatch in simulation results between the Simscape algorithm and the corresponding HDL implementation. Such mismatches generate warnings or assertions when you simulate the implementation model. To resolve the warnings, use a combination of various settings in the **Generate implementation model** task as illustrated below.

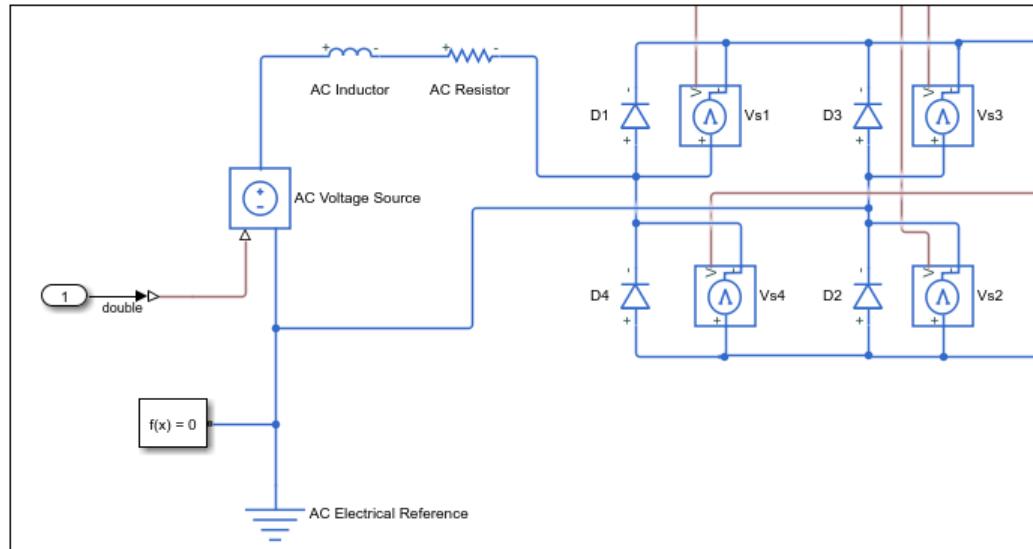
## Bridge Rectifier Model

This example uses the bridge rectifier model to illustrate how to generate an implementation model with validation logic inserted in the model, and how you can resolve any assertions that may be generated when you simulate the implementation model.

- 1 Open the bridge rectifier model. In the MATLAB Command Window, enter:

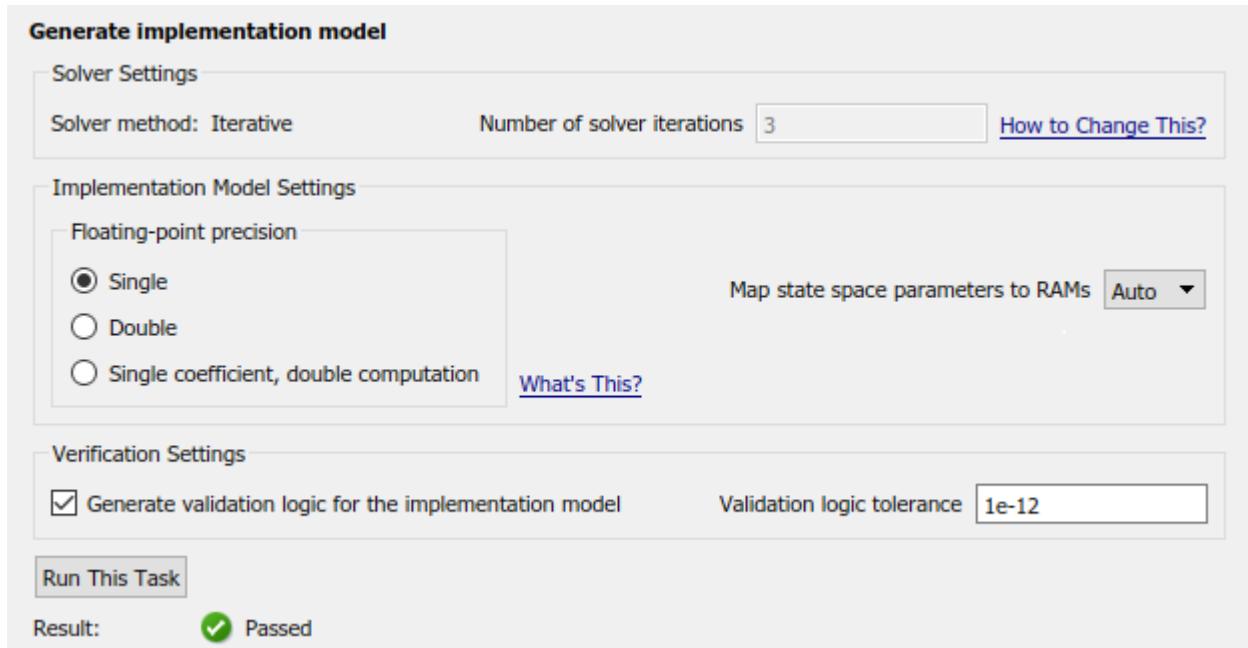
```
open_system('sschdlexBridgeRectifierExample')
open_system('sschdlexBridgeRectifierExample/Simscape_system')
```

Inside the `Simscape_system`, you see four diodes arranged in a bridge configuration. For both positive and negative input values, this configuration provides a positive, rectified output.



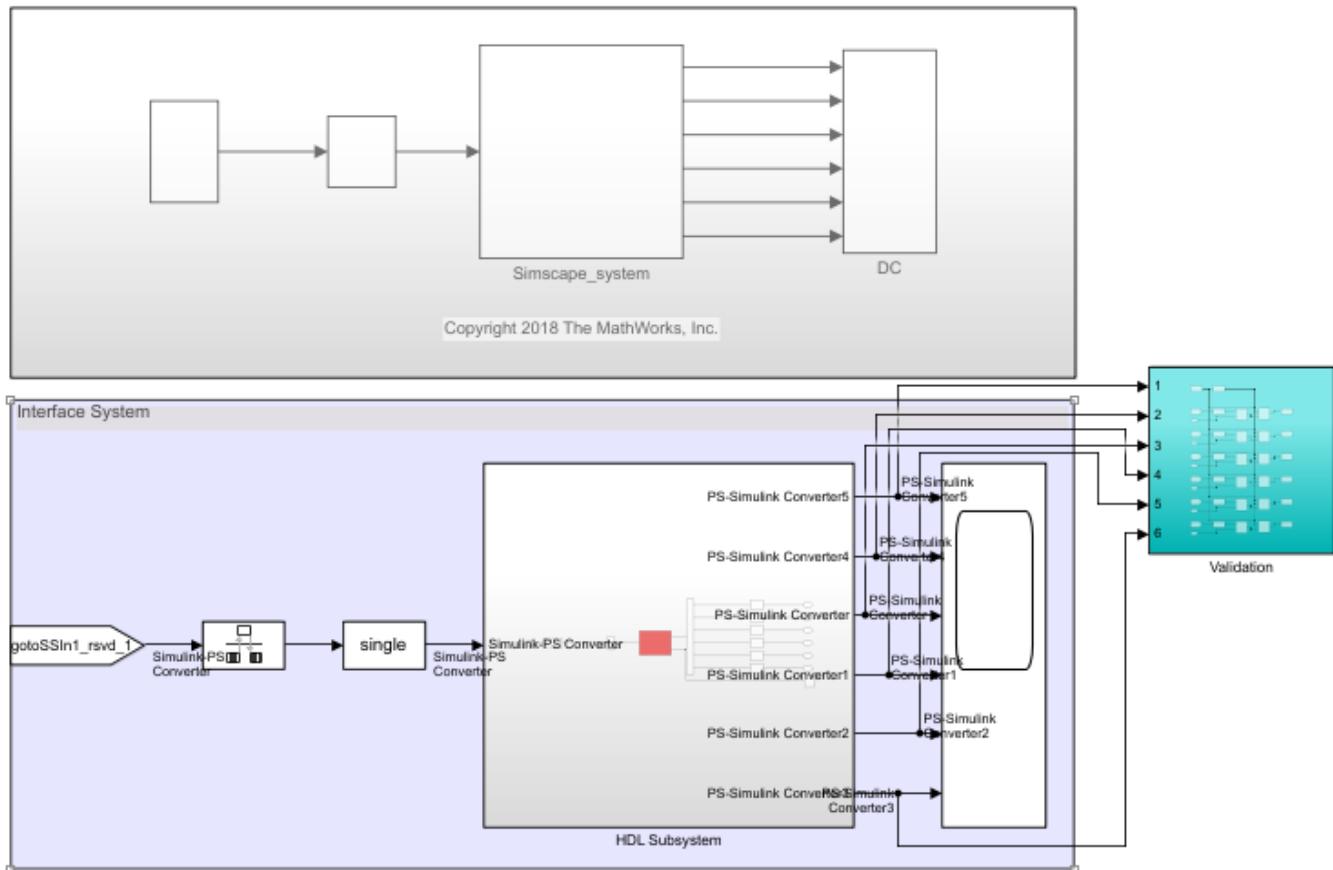
- 2 Open the Simscape HDL Workflow Advisor for your model:

- ```
sschdladvisor('sschdlexBridgeRectifierExample')
```
- 3 Right-click the **Get state-space parameters** task and select **Run to Selected Task** to run all tasks in the Advisor except for the **Generate implementation model** task.
- 4 In the **Generate implementation model** task, select the **Generate validation logic for the implementation model** check box. Leave the other options with their default values and select **Run This Task**.



After running this task, keep the UI window for this task open. If simulating the HDL implementation model generates warnings, you modify the settings in the **Generate implementation model** task and then rerun this task. You do not have to modify or rerun other tasks.

- 5 Click the link to open the HDL implementation model. You see a Validation Subsystem that compares the simulation results of the Simscape model to the HDL implementation model. Simulate the implementation model.



You see that simulating the model generates multiple assertions indicating a mismatch in the simulation results. If you open the Diagnostic Viewer, you see this message:

```
Assertion detected in 'gmStateSpaceHDL_BridgeRectifier_HDL_SimMismatch/
Validation/Check Static Range1' at time 0.04186 [4982 similar]
```

The message indicates that the Simscape™ algorithm does not match the equivalent HDL implementation. To resolve the validation mismatch, you can modify various settings in the **Generate implementation model** task until the HDL implementation model matches the Simscape algorithm. In most cases, to resolve the numeric mismatch, you may want to use a combination of these settings.

## Increase Validation Logic Tolerance

Conversion of a Simscape algorithm to an equivalent HDL implementation leads to rounding errors. The default tolerance value is relatively small and can be difficult to achieve especially with single-precision data types in the HDL implementation model. To resolve the mismatch:

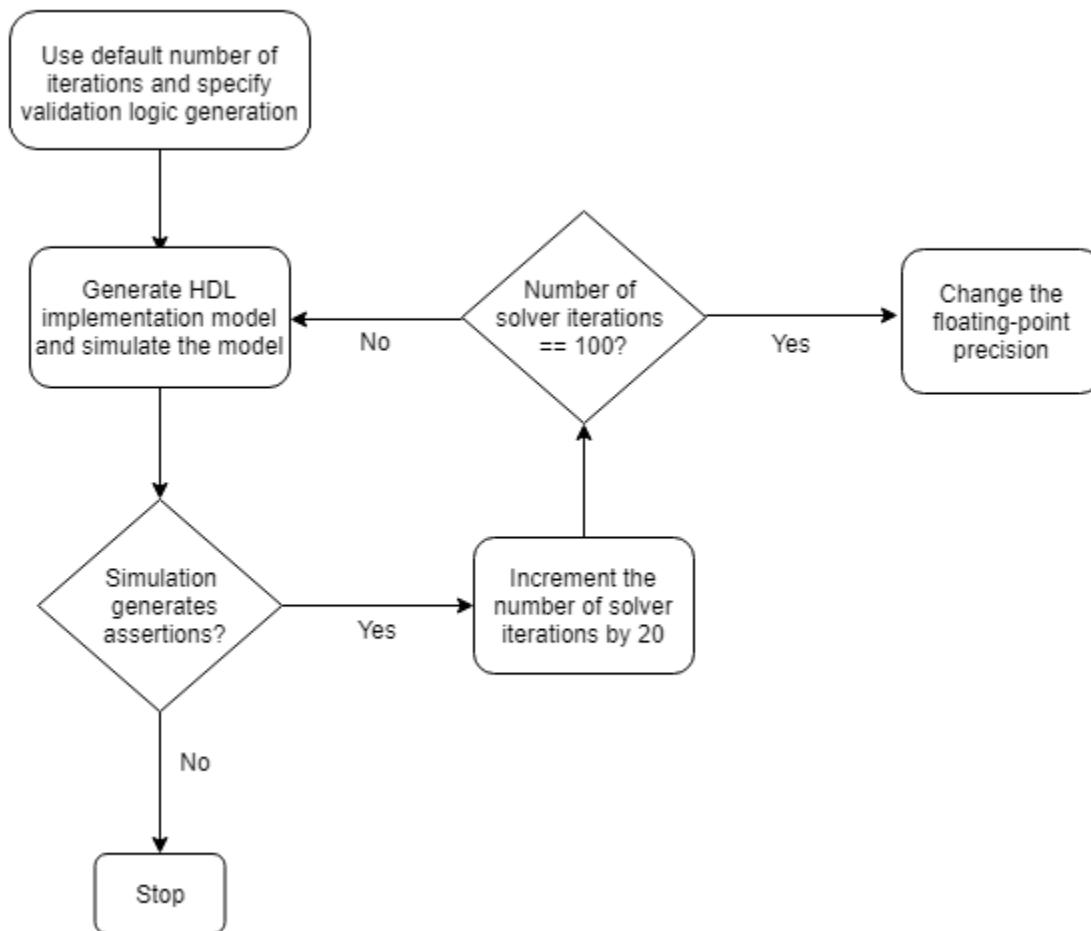
- 1 Start by increasing the **Validation logic tolerance** to an initial value such as `1e-4`.
- 2 Select **Generate validation logic for the implementation model** and run the task to generate the HDL implementation model that includes validation logic.
- 3 Simulate the model and check whether the simulation displays assertions in the Diagnostic Viewer. If the simulation results produce warnings, proceed to the next step to increase the number of solver iterations.

## Increase Number of Solver Iterations

For each mode in the physical system, the switched linear workflow arrives at a state-space representation. The solver method is iterative and performs multiple computations to determine the correct mode for the next time step. After a certain number of iterations, the output value from the next time step becomes the same as the value from the previous time step. This consistency in the output value indicates the correct number of solver iterations.

The Advisor by default chooses an optimal value for the number of solver iterations. See “Using Number of Solver Iterations” on page 33-9. If increasing the tolerance value does not improve accuracy of the HDL implementation model, you can resolve the numeric mismatch by increasing the number of solver iterations.

When you increase the number of solver iterations, the code generator changes the sample time of the generated HDL implementation model. A large number of iterations can increase the simulation time significantly. See “Reducing Number of Solver Iterations” on page 32-106. This flowchart illustrates how to change the **Number of solver iterations**.

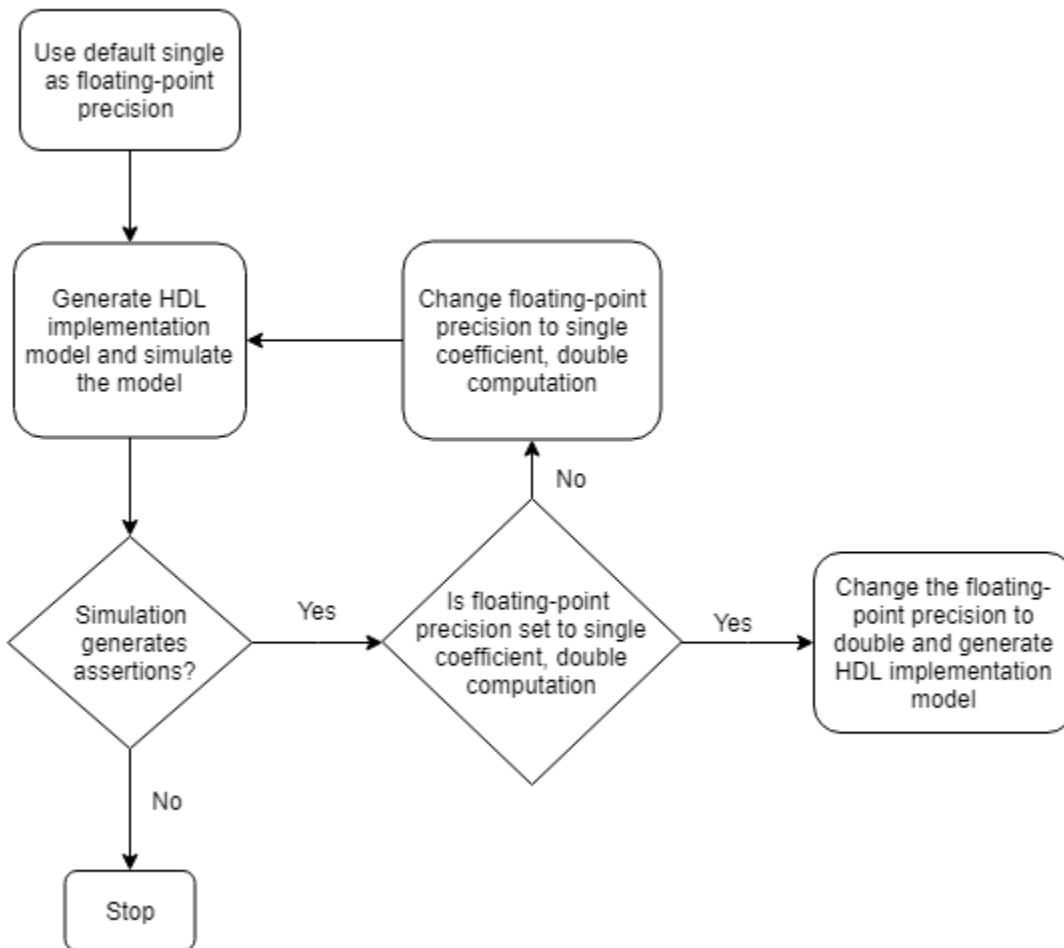


## Use Larger Floating-Point Precision

You can use the **Floating-point precision** setting in the **Generate implementation model** task to specify the floating-point data type you want to use for the algorithm inside the **HDL Subsystem**. Specify whether you want to store the matrix coefficients in **single** or **double** data types and whether to use **single** or **double** when performing the computations.

| Floating-Point Precision               | Description                                                                                                                                                                                                                                                         |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Double                                 | Using double floating-point precision increases the numerical accuracy of the generated model and the maximum achievable target frequency. However, the area consumption and pipeline latency are also increased.                                                   |
| Single                                 | This is the default setting for floating-point precision.                                                                                                                                                                                                           |
| Single coefficient, double computation | This mode offers a tradeoff between <b>Single</b> and <b>Double</b> modes of floating-point precision. To save memory usage, the coefficients that are stored in <b>single</b> . The matrix computations are then performed in <b>double</b> for improved accuracy. |

This flowchart illustrates how to change the **Floating Point Precision** and improve the numeric accuracy of the generated HDL implementation model.



**Note** Double-precision operations have large latencies and require a large **Oversampling factor** to allocate sufficient delays for the floating-point operations, which reduces the sampling frequency. For a tradeoff between accuracy and precision, use **Single coefficient, double computation** as the **Floating Point Precision**.

After specifying double data types, if the simulation results still produce warnings:

- 1 Proceed to the first step to further increase the validation logic tolerance. Use a tolerance value of  $1e-03$  and then simulate the model to see if the numeric accuracy requirements are met.
- 2 Increase the number of solver iterations if you still see warnings in the Diagnostic Viewer. Continue iterating between these steps till the HDL implementation model numerically matches the Simscape algorithm.

For the bridge rectifier model, to resolve the warnings, set the **Validation logic tolerance** to  $1e-4$  and specify the **Floating Point Precision** as double. After you generate the implementation model with the validation logic, you see that simulating the model does not display warnings in the Diagnostic Viewer.

## See Also

### Functions

`simscape.findNonlinearBlocks` | `sschdladvisor`

## More About

- “Generate HDL Code for Simscape Models” on page 32-9
- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-6
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25

## Improve Sampling Rate of HDL Implementation Model Generated from Simscape Algorithm

If you design your algorithm by using Simscape switched linear blocks, you can run the Simscape HDL Workflow Advisor to generate an HDL implementation model. When you open the HDL implementation model, you see the HDL algorithm that models the state-space representation by using Simulink blocks that are compatible for HDL code generation. To learn more about the Simscape HDL Workflow Advisor, see “Simscape HDL Workflow Advisor Tasks” on page 33-2.

### Sampling Frequency

When you generate HDL code and deploy the plant model onto an FPGA, you may want to improve the sampling frequency. The sampling frequency depends on these parameters:

- FPGA clock frequency
- Oversampling factor
- Number of solver iterations

$$\text{Sampling Frequency} = \text{FPGA clock frequency} / (\text{Oversampling factor} * \text{Number of solver iterations})$$

To improve the sampling rate, you want to maximize the FPGA clock frequency, and minimize the oversampling factor and number of solver iterations. As you improve the sampling rate, make sure that the updated sampling frequency is equivalent to the fixed sample time that you specify for your original Simscape model by using the Solver Configuration block. To learn more about how this block is used in your model before running the Simscape HDL Workflow Advisor, see “Generate HDL Code for Simscape Models” on page 32-9.

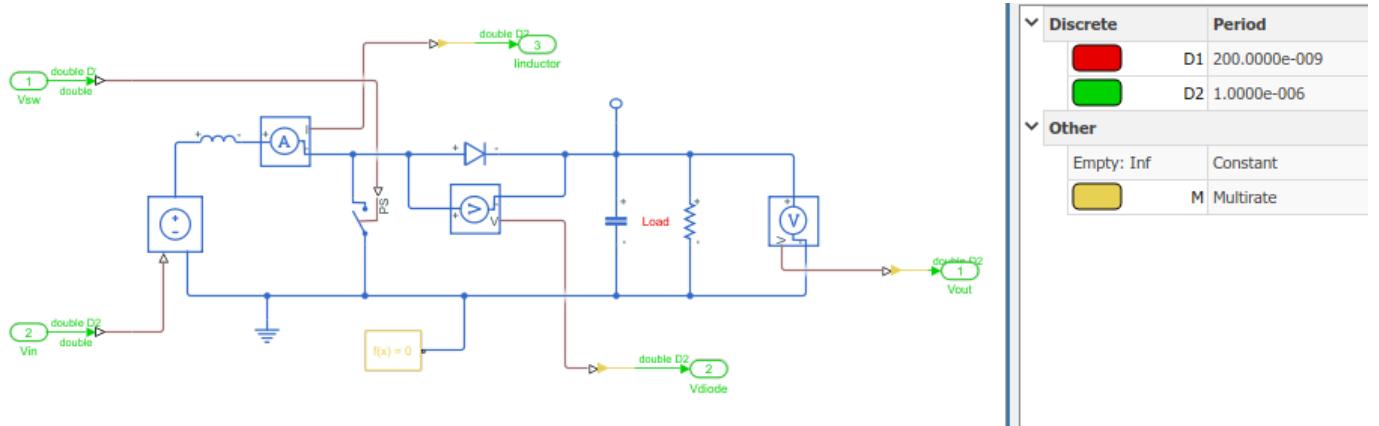
The preceding section uses the boost converter model as an example to illustrate how you can modify the oversampling factor and the number of solver iterations to improve the sampling rate.

### Boost Converter Model

This example uses the boost converter model to illustrate the change in sample time in the generated HDL implementation model and the oversampling factor that is saved on the model.

- 1 Open the boost converter model. To learn how the boost converter is implemented, open the `Simscape_system` Subsystem. To open the boost converter model, in the MATLAB Command Window, enter:

```
open_system('sschdlexBoostConverterExample')
open_system('sschdlexBoostConverterExample/Simscape_system')
```



You see that the model runs at a sample time  $1e-6$ . The sample time of  $200e-9$  corresponds to the sample time of the sources that drive the Simscape algorithm.

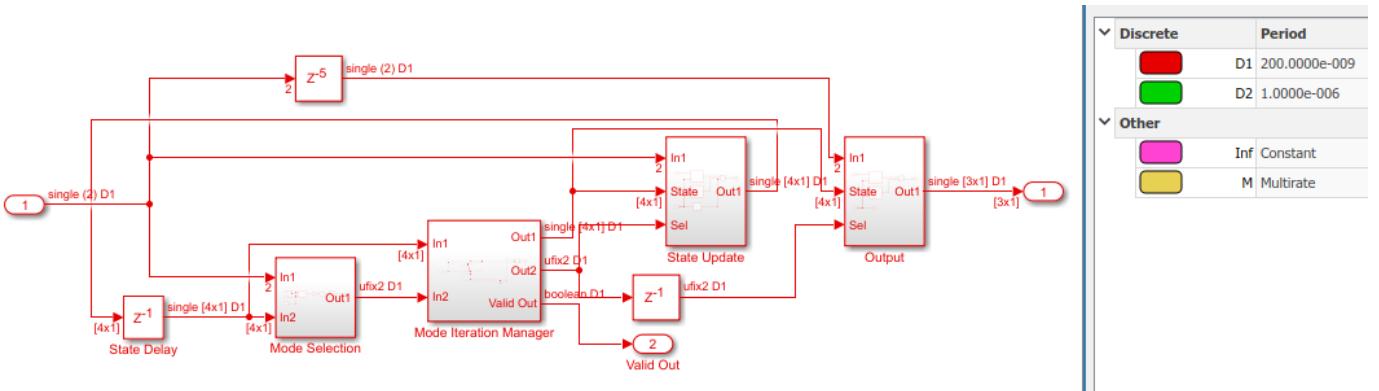
- Open the Simscape HDL Workflow Advisor for your model:

```
sschdladvisor('sschdlexBoostConverterExample')
```

- Run the workflow to the **Generate implementation model** task.

After running this task, you see a link to the generated HDL implementation model. Click the link to open the HDL implementation model.

- Simulate the HDL implementation model. When you navigate the model to the **HDL Algorithm Subsystem**, you see that the model uses **single** data types and runs at a sample time  $200e-9$ , which is 5 times faster than the original Simscape model.



- Run this command to see the HDL parameter settings that are saved on the model:

```
hdlsaveparams('gmStateSpaceHDL_sschdlexBoostConverterExamp')
```

```
% Set Model 'gmStateSpaceHDL_BoostConverter_HDL' HDL parameters
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', 'FloatingPointTargetConfiguration', ...
    hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint' ...
    , 'LatencyStrategy', 'MIN') ...
);
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', 'HDLSubsystem', ...
    'gmStateSpaceHDL_BoostConverter_HDL');
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', 'MaskParameterAsGeneric', 'on');
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL', 'Oversampling', 60);

% Set SubSystem HDL parameters
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL/HDL Subsystem', 'FlattenHierarchy', 'on');
```

```
% Set SubSystem HDL parameters
hdlset_param('gmStateSpaceHDL_BoostConverter_HDL/HDL Subsystem/HDL Algorithm/State Update/Multiply State', ...
    'SharingFactor', 1);
```

The HDL parameters that are saved indicate that the model has the native floating-point mode enabled and uses an **Oversampling factor** of 60 and has **Latency Strategy** set to MIN. This default values chosen for number of solver iterations and combination of HDL parameters offers an optimal trade-off between oversampling factor and the target FPGA clock frequency and improves the sampling frequency. To further improve the sampling frequency, reduce the number of iterations and the oversampling factor.

## Reducing Number of Solver Iterations

For each mode in the physical system, the switched linear workflow arrives at a state-space representation. The solver method is iterative and performs multiple computations to determine the correct mode for the next time step. After a certain number of iterations, the output value from the next time step becomes the same as the value from the previous time step. This consistency in the output value indicates the correct number of solver iterations.

The Advisor by default chooses an optimal value for the number of solver iterations. See “Using Number of Solver Iterations” on page 33-9. To improve the sampling rate, reduce the number of solver iterations. The number of solver iterations depends on various factors such as the complexity of your design, the number of modes in the design that the workflow calculates, and so on.

In the **Generate implementation model** task of the Simscape HDL Workflow Advisor:

- 1 Start by reducing the **Number of solver iterations** to a value such as 3
- 2 Select **Generate validation logic for the implementation model**, and then generate the HDL implementation model.
- 3 Simulate the HDL implementation model and open the Diagnostic Viewer to verify that the model does not display warnings or assertions.

If you see warnings or assertions, it indicates a simulation mismatch because the number of solver iterations that you specified is not adequate to compute the required number of modes in the state-space design. Resolve the mismatch by increasing the validation logic tolerance value or the number of solver iterations. Changing **Floating-point precision** to double is not recommended. Double-precision operations have large latencies and require a large **Oversampling factor** to allocate sufficient delays, which reduces the sampling frequency. See “Validate HDL Implementation Model to Simscape Algorithm” on page 32-97.

## Using Oversampling Factor and Latency Strategy

The **Oversampling factor** specifies the factor by which the FPGA clock rate is a multiple of the HDL implementation model base sample rate. The HDL implementation model contains feedback loops and performs multiplication of large matrices that have floating-point data types inside the feedback loops. To accommodate the large latency introduced by these floating-point operations inside the feedback loops, the code generator uses a large value of oversampling factor in conjunction with the clock-rate pipelining optimization on the model. For more information, see “Strategy 1: Global Oversampling” on page 10-68.

You vary the oversampling factor and latency strategy of the floating-point operator in conjunction. The default oversampling factor of 60 and minimum latency strategy gives an optimal sampling

frequency. To achieve the maximum FPGA clock frequency, use the maximum latency strategy. When you specify this latency strategy, the floating-point operations introduce the maximum number of delays. To allocate these delays, increase the oversampling factor. If the increase in FPGA clock frequency outweighs the increase in oversampling factor, you achieve a higher sampling frequency.

To change the latency strategy and oversampling factor in conjunction from the Configuration parameters dialog box:

- 1 On the **HDL Code Generation > Floating Point** pane, change the **Latency Strategy** to **Max**.
- 2 On the **HDL Code Generation > Global Settings** pane, increase the **Oversampling factor** to a value such as **100** depending on the complexity of your HDL design.

For the boost converter model, the default settings of **Number of solver iterations** set to **5**, **Oversampling factor** set to **60**, and **Latency Strategy** set to **Min** provides the optimal sampling frequency.

## See Also

### Functions

`simscape.findNonlinearBlocks` | `sschhdladvisor`

## More About

- “Solvers for Real-Time Simulation” (Simscape)
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-6
- “Latency Considerations with Native Floating Point” on page 10-96
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25



# Simscape HDL Workflow Advisor Tasks

---

- “Simscape HDL Workflow Advisor Tasks” on page 33-2
- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-6

## Simscape HDL Workflow Advisor Tasks

By using the Simscape HDL Workflow Advisor, you can generate an HDL implementation model. You can then generate HDL code for the implementation model and deploy the code onto FPGA platforms. To open the Advisor, run the `sschdladvisor` function. For example:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

In the Simscape HDL Workflow Advisor, for summary information on each Simscape HDL Workflow Advisor folder or task, right-click that folder or task and select **What's This?**.

### Simscape HDL Workflow Advisor folder

The Simscape HDL Workflow Advisor consists of various tasks that you can use to convert your Simscape model to an HDL implementation model. You can generate code for the HDL Subsystem in this model. The Simscape HDL Workflow Advisor consists of folders that perform these tasks:

- The **Code generation compatibility** folder consists of tasks that check whether the model is a switched linear system and uses the correct solver configuration settings.
- The **State-space conversion** folder consists of tasks that derive the state-space parameters from your model for generating the implementation model.
- The **Implementation model generation** folder consists of a task that generates the HDL implementation model from the state-space parameters.

To learn more about each folder or task, right-click that folder or task, and select **What's This?**.

### Code generation compatibility folder

The tasks in the **Code generation compatibility** folder check whether:

- You have correctly specified the solver configuration settings and the settings are consistent across Solver Configuration blocks inside each network into your Simscape model.
- Your model uses switched linear blocks.

### Check solver configuration task

The **Check solver configuration** task checks whether you have specified the correct settings and the settings are consistent across Solver Configuration blocks inside each network in your Simscape model.

The Advisor checks whether you specified these settings for all Solver Configuration blocks:

- **Use local solver** is selected
- **Solver type** is set to Backward Euler
- A discrete sample time,  $T_s$  is specified
- **Use fixed-cost runtime consistency iterations** is either selected or cleared
- **Nonlinear iterations** value is the same when **Use fixed-cost runtime consistency iterations** is selected

If you did not specify these settings, the task provides a link to the Solver Configuration block in your model and the settings to modify.

## Check switched linear task

The **Check switched linear** task checks whether you use switched linear blocks in your Simscape model.

For this task to pass, the model that you use must contain linear or switched linear blocks. Nonlinear blocks and time-varying blocks (such as Variable Inductor and Variable Capacitor blocks) are not supported.

Linear blocks are blocks that are defined by a linear relationship. For example, a resistor is a linear block since it is defined by the equation  $V = IR$ . Similarly, an inductor is linear because it is defined by  $V = \frac{d}{dt} I L$ . Switched linear blocks are blocks such as diodes or switches. These blocks are defined by a linear relationship such as  $V = IR$  where  $R$  can switch between two or more values depending on the state of the diodes or switches.

This task runs `simscape.findNonlinearBlocks` on your model to check for the presence of nonlinear blocks. For example:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

If your model contains nonlinear blocks, running this task fails. In the **Result** log, you see links provided to the nonlinear blocks in your model. To continue the workflow, replace the nonlinear blocks with switched linear blocks, and rerun the task.

When this task passes, it displays:

- A message indicating that the model is switched linear.
- Number of Simscape networks present in the model.
- The number of algebraic and differential variables for each Simscape network with links to the blocks in your Simscape model that are related to these variables.

Differential variables consume a quadratic amount of multiplier resources on the target FPGA device. Algebraic variables consume a linear amount of multiplier resources. You can use this information to determine how many multiplier resources your Simscape design consumes on the FPGA device.

- A message with links to the Simulink-PS Converter and PS-Simulink Converter blocks in your model if you use the default names for these blocks.

The input and output ports of the **HDL Subsystem** in the implementation model use the names that you specify for the Simulink-PS Converter and PS-Simulink Converter blocks. To avoid this message, use a meaningful name for these blocks.

## State-space conversion folder

Before you can generate the HDL implementation model, run the task in this folder to derive the state-space parameters from your model. The tasks in this model:

- Simulate the Simscape model to extract the differential algebraic equations.
- Discretize the differential algebraic equations to generate an abstract state space representation that represents the model in the form of linear modes.

## Extract Equations

The **Extract Equations** task simulates your Simscape model to extract the differential algebraic equations. This task derives the **Simulation stop time** value from the original Simscape model.

If this task passes, it displays the number of states, inputs, outputs, modes, and differential variables for each Simscape network present in the model. The task also displays a message if the model is purely linear and does not contain any nonlinear elements.

The number of modes is limited by the number of switches present in your Simscape model. The maximum number of modes possible are  $2^{(\text{number of switches})}$ . All the modes that the Advisor generates are executed as per the input parameters by using a switching logic. A valid number of modes are selected depending on the design of your Simscape model.

## Discretize Equations

This task discretizes the differential algebraic equations and generates an abstract discrete state-space representation. This task represents the model in the form of linear modes. Each mode is represented by a set of state-space matrices. This task derives the **Discrete sample time** value from the original Simscape model.

If this task passes, it displays the **Discrete sample time** and number of parameters and modes for each Simscape network present in the model.

### Passed

Summary of the state-space representation:

- Discrete sample time: 1e-05

| Parameter | Parameter size |
|-----------|----------------|
| A         | 7 x 7 x 3      |
| B         | 7 x 1 x 3      |
| F0        | 7 x 1 x 3      |
| C         | 6 x 7 x 3      |
| D         | 6 x 1 x 3      |
| Y0        | 6 x 1 x 3      |

## Implementation model generation folder

The task in the **Implementation model generation** folder generates an HDL implementation model from the discrete state-space representation. The implementation model represents the Simscape algorithm by using Simulink blocks that are compatible for HDL code generation. If the task **Generate implementation model** in this folder passes, it provides a link to the implementation model.

## Generate implementation model task

To generate an HDL implementation model from the discrete state-space representation, run this task. The HDL implementation model contains a **HDL Subsystem** that models the state-space equations by using the state-space parameters derived by running the **Get state-space parameters** task. The **HDL Subsystem** block represents the DUT for which you can generate HDL code.

Before you run this task, you can:

- Specify a custom value for the **Number of Solver iterations** setting. To learn more, see “Using Number of Solver Iterations” on page 33-9.
- Use the **Floating-point precision** setting to specify whether the **HDL Subsystem** in the generated implementation model stores matrix types in **single** or **double** and computes the results in **single** or **double** data types. To learn more, see “Floating-Point Precision and Numerical Accuracy” on page 33-10.
- Use the **Generate validation logic for the implementation model** to generate the logic that verifies whether the generated HDL implementation model is functionally equivalent to the original Simscape model. The logic is generated for each Simscape network present in the model. You can specify a tolerance for the numerical correctness by using the **Validation logic tolerance**. The **Validation logic tolerance** is an absolute value. For example, you can specify a tolerance value of **1e-12**.

If the task passes, you see a link to the implementation model.

## See Also

### More About

- “Simscape HDL Workflow Advisor Tips and Guidelines” on page 33-6
- “Generate HDL Code for Simscape Models” on page 32-9
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25

## Simscape HDL Workflow Advisor Tips and Guidelines

By using the Simscape HDL Workflow Advisor, you can generate an HDL implementation model. You can generate HDL code for the implementation model and deploy the generated code onto FPGA platforms. To open the Advisor, run the `sschdladvisor` function. For example:

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

The Simscape HDL Workflow Advisor consists of various tasks that convert your Simscape model to the HDL implementation model. When running various tasks in the Simscape HDL Workflow Advisor, you can follow certain tips and guidelines. You see these tips in the UI window of a particular task. For example, in the task that discretizes the equations to state-space parameters, the UI has a tip that suggests how to change the sample time. This section contains more information about each tip in the Simscape HDL Workflow Advisor UI.

### Estimating Resource Consumption Using Algebraic and Differential Variables

After you run the **Check Switched Linear** task, the task reports the number of differential and algebraic variables for each Simscape network present in the model. For example, this figure illustrates that there are two differential and two algebraic variables in the boost converter example model `sschdlexBoostConverterExample`.

```
sschdladvisor('sschdlexHalfWaveRectifierExample')
```

Run the workflow to the **Check Switched Linear** task.

#### Details

Number of Discrete Variables: 4

Number of Differential Variables: 2

| Source                                       | Value             |
|----------------------------------------------|-------------------|
| <a href="#">Simscape_system.Capacitor.vc</a> | Capacitor voltage |
| <a href="#">Simscape_system.Inductor.i_L</a> | Inductor current  |

Number of Algebraic Variables: 2

| Source                                      | Value   |
|---------------------------------------------|---------|
| <a href="#">Simscape_system.Capacitor.i</a> | Current |
| <a href="#">Simscape_system.Inductor.v</a>  | Voltage |

By viewing the number of algebraic and differential variables, you can determine how the design consumes resources on the FPGA device. If  $N_d$  is the number of differential variables and  $N_a$  is the number of algebraic variables, the resource usage on the target hardware varies according to the

relation  $Nd * (Nd + Na)$ . Differential variables consume a quadratic amount of multiplier resources on the target FPGA device. Algebraic variables consume a linear amount of multiplier resources. You can use this information to determine how many multiplier resources your Simscape design consumes on the FPGA device and whether your design is ready for conversion to state-space representation.

## Setting Simulation Stop Time for Extracting Equations

### Change Simulation Stop Time

When you run the **Extract Equations** task, the Simscape HDL Workflow Advisor reports the simulation stop time. The simulation stop time corresponds to the amount of time that the Advisor takes to run simulation on your Simscape model. The stop time must not be significantly large such that the Advisor takes a long time to run this task. Use a stop time that is sufficient to reach the required number of modes for your model. To change the stop time, navigate to the Simscape model, and then specify the **Stop Time**.

### Simulation Stop Time and Number of Modes

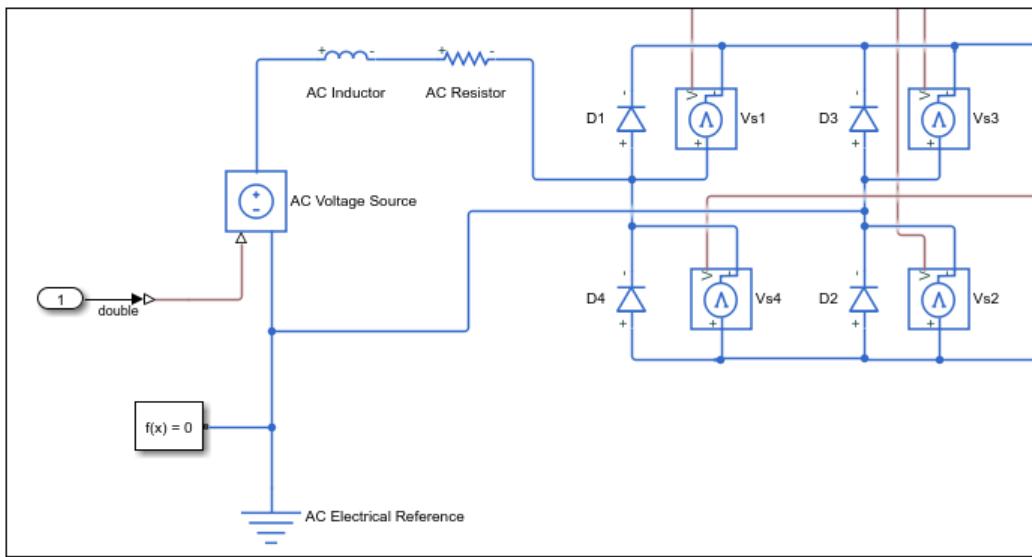
In the **Extract Equations** task, when you extract the differential algebraic equations, the Simscape HDL Workflow Advisor simulates the model to cover the nonlinear range of Simscape blocks. This task can take a long time depending on the number of switching elements in the Simscape model.

For a switched linear model, each switching element in the design has two modes. A switched linear model with  $n$  switching elements has  $2^n$  possible modes. For Simscape models with large number of switching elements, the number of modes can become significantly large. For example, the Vienna rectifier has 21 switching elements, which translates to  $2^{21}$  possible modes. The Simscape HDL Workflow Advisor can take a long time to simulate such a large model and cover such a large number of modes. In addition, the HDL implementation model that you generate for such a design can consume a large amount of resources or may not even fit on the target FPGA device.

In most cases, while simulating the model, the Advisor does not have to reach the entire  $2^n$  modes. For example, consider this bridge rectifier model. To open this model, enter:

```
open_system('sschdlexBridgeRectifierExample')
```

Inside the **Simscape\_system** Subsystem, you see the four diodes arranged in a bridge configuration.



As each diode has two states, the Simscape design can have  $2^4 = 16$  possible states. In contrast, the bridge rectifier has only three modes. The modes are:

- Diodes D1 and D2 are ON, D3 and D4 are OFF
- Diodes D1 and D2 are OFF, D3 and D4 are ON
- Diodes D1, D2, D3, and D4 are OFF

This example shows that, based on the Simscape algorithm and the input to the design, you can set the simulation stop time to a minimum value that covers the number of modes to be reached.

## Changing Sample Time for Discretizing Equations

### Change Sample Time

When you run the **Discretize Equations** task, the Simscape HDL Workflow Advisor reports the discrete sample time. The discrete sample time corresponds to the sample time that the Advisor uses to discretize the differential algebraic equations to state-space parameters. To change the sample time, in your Simscape model, open the Block Parameters dialog box for the Solver Configuration block, and then specify the **Sample time**.

### Sample Time and Discretizing Equations

In the **Discretize Equations** task, the Simscape HDL Workflow Advisor discretizes the differential algebraic equations into state-space parameters. You extract the differential algebraic equations by simulating the Simscape model in the previous task, **Extract Equations**. The **Discretize Equations** task runs much faster than the **Extract Equations** task because the Advisor only has to discretize the differential algebraic equations to state-space parameters.

The **Discretize Equations** task obtains the sample time information from the sample time that you specify for the Solver Configuration block in your model. The Advisor then discretizes the equations to state-space parameters based on this sample time information.

## Using Number of Solver Iterations

### What is Number of Solver Iterations?

In the **Generate implementation model** task, you can specify the **Number of solver iterations**. The number of solver iterations refer to the number of times the state-space model is executed per mode. The Simscape HDL Workflow Advisor generates the number of iterations that are required for executing the state-space model, automatically.

For each mode in the physical system, the switched linear workflow arrives at a state-space representation. The solver method is iterative and performs multiple computations to determine the correct mode for the next time step. After a certain number of iterations, the output value from the next time step becomes the same as the value from the previous time step. This consistency in the output value indicates the correct number of solver iterations.

By default, the **Number of Solver iterations** is 1 for linear models. For switched linear models, the **Number of solver iterations** depends on the number of mode iterations that Simscape uses during model simulation. This chosen value is optimal such that it causes the model to converge and avoids exceeding the threshold value for real-time deployment.

### Using Fixed-Cost Runtime Consistency Iterations

On the Solver Configuration block, the **Use fixed-cost runtime consistency iterations** check box is cleared by default. If you select this check box, the **Nonlinear iterations** setting on the Solver Configuration block becomes the same as the **Number of solver iterations** setting in the **Generate implementation model** task.

By default, **Nonlinear iterations** is set to 2. When you run the **Generate implementation model** task, the Advisor sets the **Number of solver iterations** to 2 and this setting cannot be modified. To modify the **Number of solver iterations**, either:

- Change **Nonlinear iterations**.
- Clear **Use fixed-cost runtime consistency iterations** and then change the **Number of solver iterations**.

To learn more about the **Use fixed-cost runtime consistency iterations** setting, see Solver Configuration. See also “Solvers for Real-Time Simulation” (Simscape).

### Change Number of Solver Iterations

By default, you can change the number of solver iterations on this task. Increasing the number of solver iterations improves the numerical accuracy of generated HDL implementation model. To achieve higher sampling frequencies, reduce the number of solver iterations. Choose a value for number of solver iterations that trades off numerical accuracy and sampling frequency.

On the Solver Configuration block, if you specify the **Use fixed-cost runtime consistency iterations** setting, you cannot change the **Number of solver iterations** setting on this task. To change the number of solver iterations, on the Solver Configuration block, change the **Nonlinear iterations** parameter and rerun the **Generate implementation model** task.

### Trading off Numerical Accuracy and Sampling Frequency

To verify whether the numeric results of the HDL implementation model matches the original Simscape model, select **Generate validation logic for the implementation model**. If the numeric

results from the HDL implementation model do not match, you can increase the number of solver iterations. To learn more, see “Increase Number of Solver Iterations” on page 32-100.

Changing the number of solver iterations trades off numerical accuracy for sampling frequency. Increasing the number of solver iterations increases the sample time of the HDL implementation model which can reduce the sampling frequency. See “Reducing Number of Solver Iterations” on page 32-106.

## Floating-Point Precision and Numerical Accuracy

Use the **Floating-point precision** setting to specify whether you want the algorithm inside the **HDL Subsystem** in the generated implementation model to use **single** or **double** data types when performing the matrix computations.

| Floating-Point Precision               | Description                                                                                                                                                                                                                                                         |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Double                                 | Using double floating-point precision increases the numerical accuracy of the generated model and the maximum achievable target frequency. However, the area consumption and pipeline latency are also increased.                                                   |
| Single                                 | This is the default setting for floating-point precision.                                                                                                                                                                                                           |
| Single coefficient, double computation | This mode offers a tradeoff between <b>Single</b> and <b>Double</b> modes of floating-point precision. To save memory usage, the coefficients that are stored in <b>single</b> . The matrix computations are then performed in <b>double</b> for improved accuracy. |

To learn more about the floating-point precision settings and tradeoffs, see “Use Larger Floating-Point Precision” on page 32-101.

## See Also

### More About

- “Generate HDL Code for Simscape Models” on page 32-9
- “Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model” on page 32-25

# Model Protection in HDL Coder

---

- “Create Protected Models to Conceal Contents and Generate HDL Code” on page 34-2
- “Test Protected Models” on page 34-9
- “Package and Share Protected Models” on page 34-11
- “Obfuscate Generated HDL Code from Simulink Models” on page 34-14

## Create Protected Models to Conceal Contents and Generate HDL Code

When you want to share a model with a third-party without revealing intellectual property, protect the model. When you create a protected model, you conceal the implementation details of the original model by compiling it into a referenced model. The protected model includes derived files to support the optional functionalities that you specify, such as support for C code generation or HDL code generation.

If you have a HDL Coder license, you can create a protected model with simulation and HDL code generation support. The protected model user can then generate HDL code for models that reference the protected model that you created. To enable C code generation support or specify additional options such as code interface, you must have a Simulink Coder or Embedded Coder® license. To learn more about that workflow, see “Protect Models to Conceal Contents”.

### How Model Protection Works

When you protect a model, you can allow the user of the protected model to:

- Open a read-only web view of the model, including model contents and block parameters.
- Simulate the model in accelerator (default), rapid accelerator, and normal modes.
- Generate HDL code for a model that includes the protected model.
- Generate C code for a model that includes the protected model, if you have Simulink Coder.
- Generate code for the protected model through the standalone interface, if you have Embedded Coder and specify an ERT-based system target file for the model.

You can optionally password-protect each option. If you choose password protection for one of these options, the software protects the supporting files by using AES-256 encryption.

### How to Create a Protected Model

Create a protected model by using one of these options:

- To create a protected model from a referenced model, select the Model block and on the Simulink Toolstrip **Model Block** tab, click the **Protect** button.
- To create a protected model from the current model:
  - On the Simulink Toolstrip **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
  - Select the Model block and on the **HDL Code > Share** tab, select **Generate Protected Model**.
- To programmatically create a protected model, use the `Simulink.ModelReference.protect` function.

When you create a protected model:

- Simulink creates and stores a protected version of the model in a file that has the same name as the source model, with an `.slxp` extension.
- The original model file, with the `.slx` extension, does not change. If you protect the model through a Model block, that Model block does not change.

- Optionally, Simulink creates a project archive (`.mlproj`) that contains the protected model, a harness model for the protected model, and additional supporting files.

If your protected model requires additional supporting files, such as base workspace definitions or a data dictionary, include these files with the model when you share the protected model. For more information, see “Package and Share Protected Models” on page 34-11.

## General Protected Model Requirements and Limitations

When you create a protected model, consider these requirements:

- You must have a HDL Coder or Simulink Coder license to create a protected model.
- The model must be available on the MATLAB path.
- The model cannot have unsaved changes.
- The model uses the configuration that is active during protection. You cannot change the configuration of a protected model.
- If the model contains variants, the protected model includes only the variant that is active during protection.
- The protected model name must not be modified. Renaming the model or changing the suffix makes the model unusable until you restore its original name and suffix.

The model must also meet all requirements listed in “Model Reference Requirements and Limitations”.

## Protected Model Restrictions for HDL Code Generation

These configurations are not supported when you create a protected model that has HDL code generation support.

- The protected model must use the same configuration parameters as the top level that it is referenced from.
- The solver settings that you specify in the **Solver** pane of the Configuration Parameters dialog box must be **Fixed-step** and **auto**.
- You must not enable these settings in the Configuration Parameters dialog box:
  - **Generate parameterized HDL code from masked subsystem**
  - **Module name prefix**
  - **Use trigger signal as clock**
  - **Minimize clock enables**
  - **Scalarize vector ports**
  - **Allow clock-rate pipelining at DUT output ports**
- Models with multiple clock signals or the **Clock inputs** set to **multiple** in the Configuration Parameters dialog box are not supported.
- Models that contain model arguments are not supported.
- HDL source code of the protected model cannot be obfuscated.
- Nested protected models are not supported.

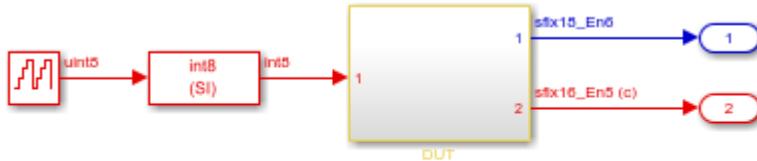
- The protected model cannot have callbacks.

To learn more about limitations for C code generation, see “Code Generation Requirements and Limitations”.

## Prepare the Parent Model

This example shows how you can protect a model that is referenced by a Model block in the parent model. Open the parent model `hdlcoder_protected_model_parent_harness`.

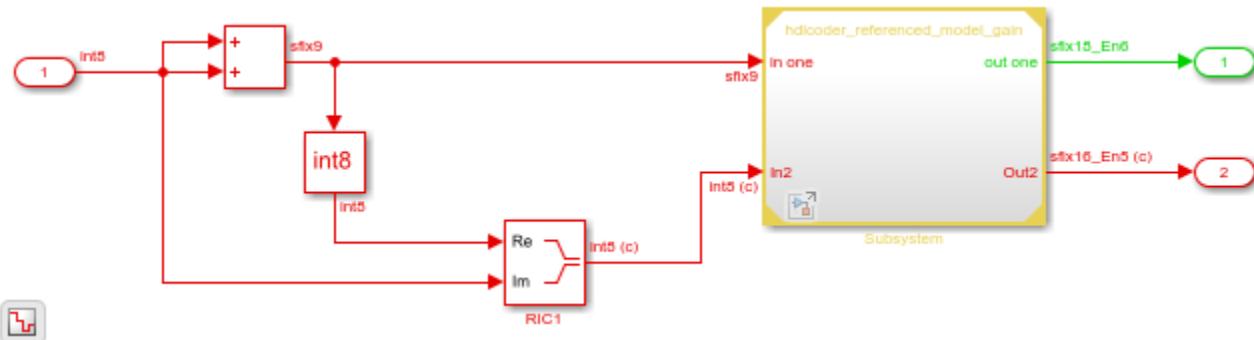
```
open_system('hdlcoder_protected_model_parent_harness')
set_param('hdlcoder_protected_model_parent_harness','SimulationCommand','Update')
```



Copyright 2015 The MathWorks, Inc.

Navigate to the Model block in your parent model. If you double-click the DUT Subsystem, and then open the `mynested` Subsystem, you see a Model block that references the model `hdlcoder_referenced_model_gain`.

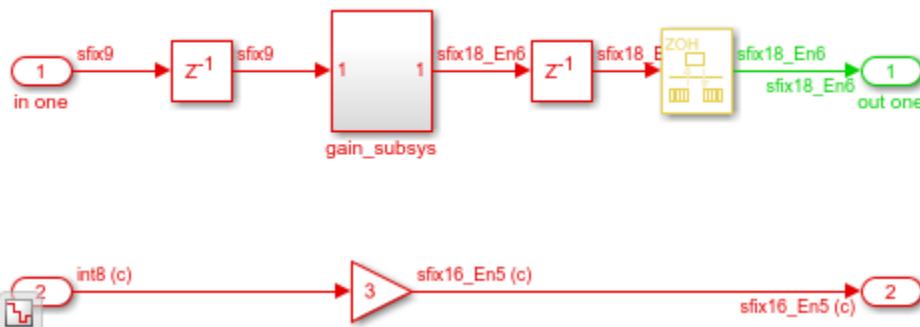
```
open_system('hdlcoder_protected_model_parent_harness/DUT/mynested')
```



Open the Model block and make sure that a *modelname* with the extension `.slx` is specified in the **Model name** field. When both the referenced model and the protected model exist in the same folder, the parent model references the protected model unless the extension is specified.

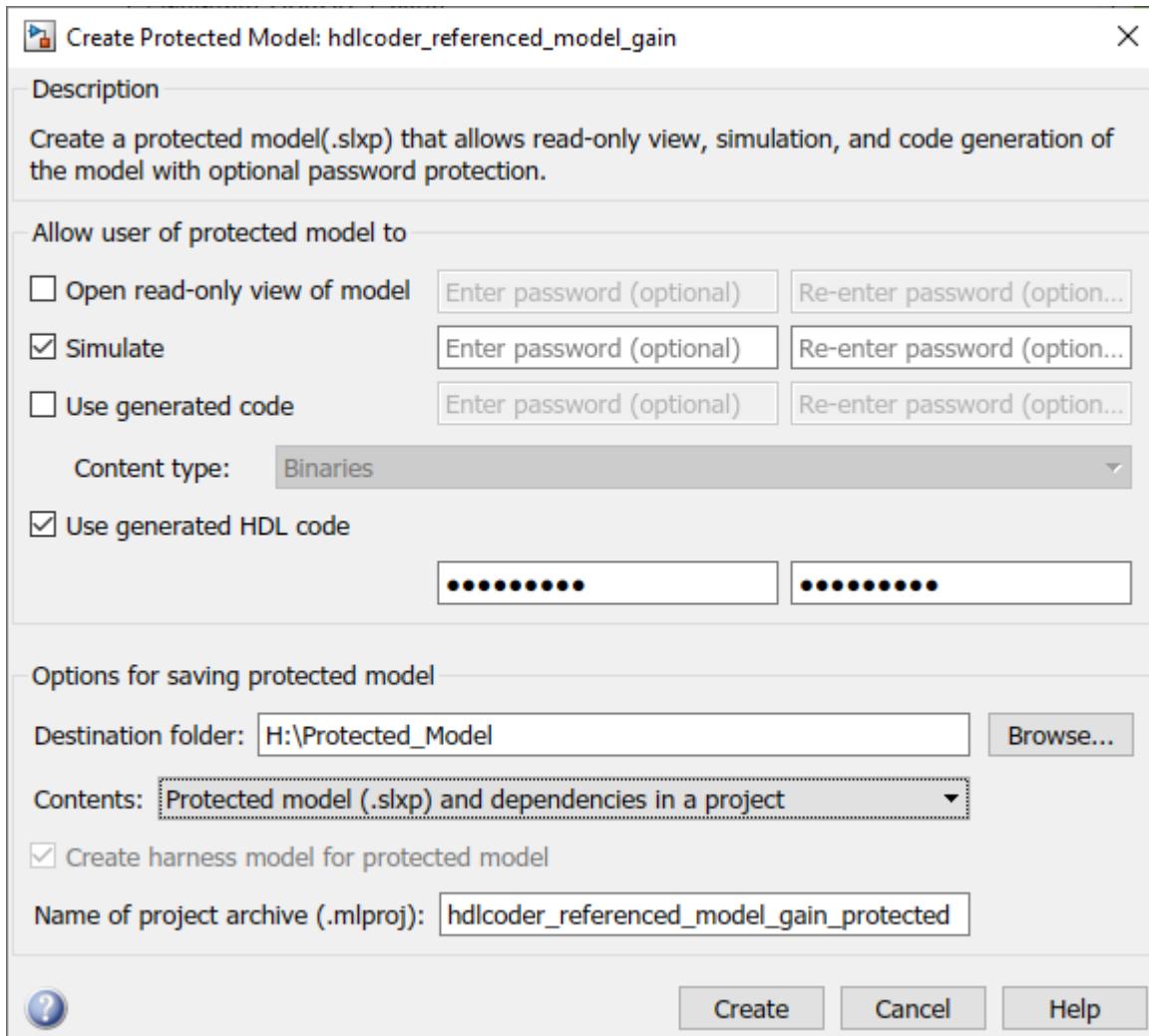
In this case, the Model block is referencing the model `hdlcoder_referenced_model_gain.slx`, the model that you want to protect. Double-click the Model block or open the model `hdlcoder_referenced_model_gain` in a separate window.

```
open_system('hdlcoder_referenced_model_gain')
set_param('hdlcoder_referenced_model_gain','SimulationCommand','Update')
```



## Protect the Referenced Model

- 1 Select the Model block.
- 2 On the Simulink Toolstrip **Model Block** tab, click **Protect**.



- 3 In the Create Protected Model dialog box, select the **Simulate** dialog box. This option allows the protected model user to simulate the model that references the protected model.
- 4 If you have Simulink Coder or Embedded Coder, you can specify additional settings such as enabling code generation support with password protection by using the **Use generated code** check box or specifying a **Code interface**. To learn more about these options, see “Protect Models to Conceal Contents” (Embedded Coder).
- 5 Select the **Use generated HDL code** check box to generate HDL code for a model that references the protected model. If you want to password-protect this functionality of the protected model, you must specify a minimum of eight characters. You can specify a unique password for this option. You cannot obfuscate the HDL source code for a protected model.
- 6 In the **Destination folder** box, specify the folder path for the protected model. The default value is the current working folder.
- 7 To automatically collect, create, and package supporting files with the protected model, set **Contents** to **Protected Model (.slxp)** and **dependencies in a project**. In this example, set **Contents** to **Protected Model (.slxp) only**.
- 8 To create a harness model for the protected model, select the **Create harness model for protected model** check box. The harness model provides an isolated environment for the Model block that references the protected model. In this example, leave the check box cleared.
- 9 Click **Create**.

HDL Coder checks compatibility of the model for HDL code generation then generates code for the model. The generated code file contents are in the `hdlsrc` folder. To learn about the files that are generated, see “Package and Share Protected Models” on page 34-11.

- 10 To use the protected model in a model hierarchy, reference it through a Model block. The **Simulation mode** for Model blocks that reference a protected model is set to **Accelerator**. You cannot change the mode. For more information, see “Reference Protected Models from Third Parties”.

To learn more about the options, see “Create Protected Model”.

To create a protected model when using the `Simulink.ModelReference.protect` function, set the Mode to `HDLCodeGeneration`. For example, run this command to protect the referenced model `hdlcoder_referenced_model_gain`:

```
Simulink.ModelReference.protect('hdlcoder_referenced_model_gain', ...
    'Mode','HDLCodeGeneration')
```

## Protected Model Report

When you create the protected model from the Simulink Editor, a protected model report is generated and included as part of the protected model. For this example, to view the protected model report, double-click the protected model or right-click the protected-model badge icon on the block in the harness model and select **Display Report**.

**Protected Model report**

Find: Match Case

**Contents**

- Summary** (selected)
- [Interface Report](#)

## Summary for hdlcoder\_referenced\_model\_gain

### Environment

Environment information for protected model "hdlcoder\_referenced\_model\_gain"

|                              |                          |
|------------------------------|--------------------------|
| Model Version                | 1.47                     |
| Simulink version             | 9.3                      |
| Simulink Coder Version       | 9.1 (R2019a) 20-Nov-2018 |
| HDL Coder version            | 3.14                     |
| Protected model generated on | Fri Dec 7 11:42:56 2018  |
| Platform                     | win64                    |

Configuration settings at the time of protected model creation: [click to open](#)

### Supported functionality

Supported functionality for protected model "hdlcoder\_referenced\_model\_gain"

|                             |                             |
|-----------------------------|-----------------------------|
| Read-only view support      | Off                         |
| Simulation support          | On                          |
| Code generation support     | Off                         |
| HDL Code generation support | On with password protection |
| Concurrent tasking support  | Off                         |

### Licenses

Licenses required to use protected model "hdlcoder\_referenced\_model\_gain"

|                      |
|----------------------|
| Simulink             |
| Fixed-Point Designer |
| HDL Coder            |

**OK** **Help**

The report contains:

- A **Summary**, including the following tables:
  - **Environment**, providing the Simulink version, the Simulink Coder version, the HDL Coder version, and platform used to create the protected model.
  - **Supported functionality**, reporting On, Off, or On with password protection for each possible functionality that the protected model supports. If you configure your protected model for multiple targets, this table includes a list of supported targets.

- **Licenses**, listing licenses required to run the protected model.
- An **Interface Report**, including model interface information such as input and output specifications, interface parameters, and data stores.

To generate a report when using the `Simulink.ModelReference.protect` function, set the 'Report' option to `True`.

## Generate HDL Code for Models Referencing Protected Model

If the protected model has simulation and HDL code generation support, the protected model user can simulate and generate HDL code from a model that references the protected model. You generate HDL code for a model that references a protected model in the same manner as how you would generate code for a regular model.

If the protected model is password-protected, before you generate code, right-click the protected model badge icon and select **Authorize**. You must then enter the password for each option. If the entered password matches the password that you specified when creating the protected model, the model is authorized. You can then generate HDL code for the model.

For example, to generate HDL code for the protected model `hdlcoder_referenced_model_gain.slxp` that is referenced by the `hdlcoder_protected_model_parent_harness` model:

- 1 Authorize the protected model `hdlcoder_referenced_model_gain.slxp` if you specified a password when creating the protected model.
- 2 Generate HDL code for the DUT Subsystem from the context menu or by using the `makehdl` function.

```
makehdl('hdlcoder_protected_model_parent_harness/DUT')
```

## See Also

### Functions

`Simulink.ModelReference.modifyProtectedModel` | `Simulink.ModelReference.protect`

## More About

- “Test Protected Models” on page 34-9
- “Package and Share Protected Models” on page 34-11

## Test Protected Models

To test a protected model that you created, compare the simulation results of the protected model to the output of the original model. As you supply the protected model from the original model, both the original and the protected model might exist on the MATLAB path.

In the parent model, if the Model block **Model name** parameter names the model without providing a suffix, the protected model takes precedence over the unprotected model. To override this default when testing the output, in the Model block **Model name** parameter, specify the file name with the extension of the unprotected model, .slx.

To compare the unprotected and protected versions of a Model block, you can use the Simulation Data Inspector. This example uses `hdlcoder_protected_model_parent_harness` and the protected model, `hdlcoder_referenced_model_gain.slxp`, which you created in “Create Protected Models to Conceal Contents and Generate HDL Code” on page 34-2.

- 1 If it is not already open, open the model `hdlcoder_protected_model_parent_harness`.

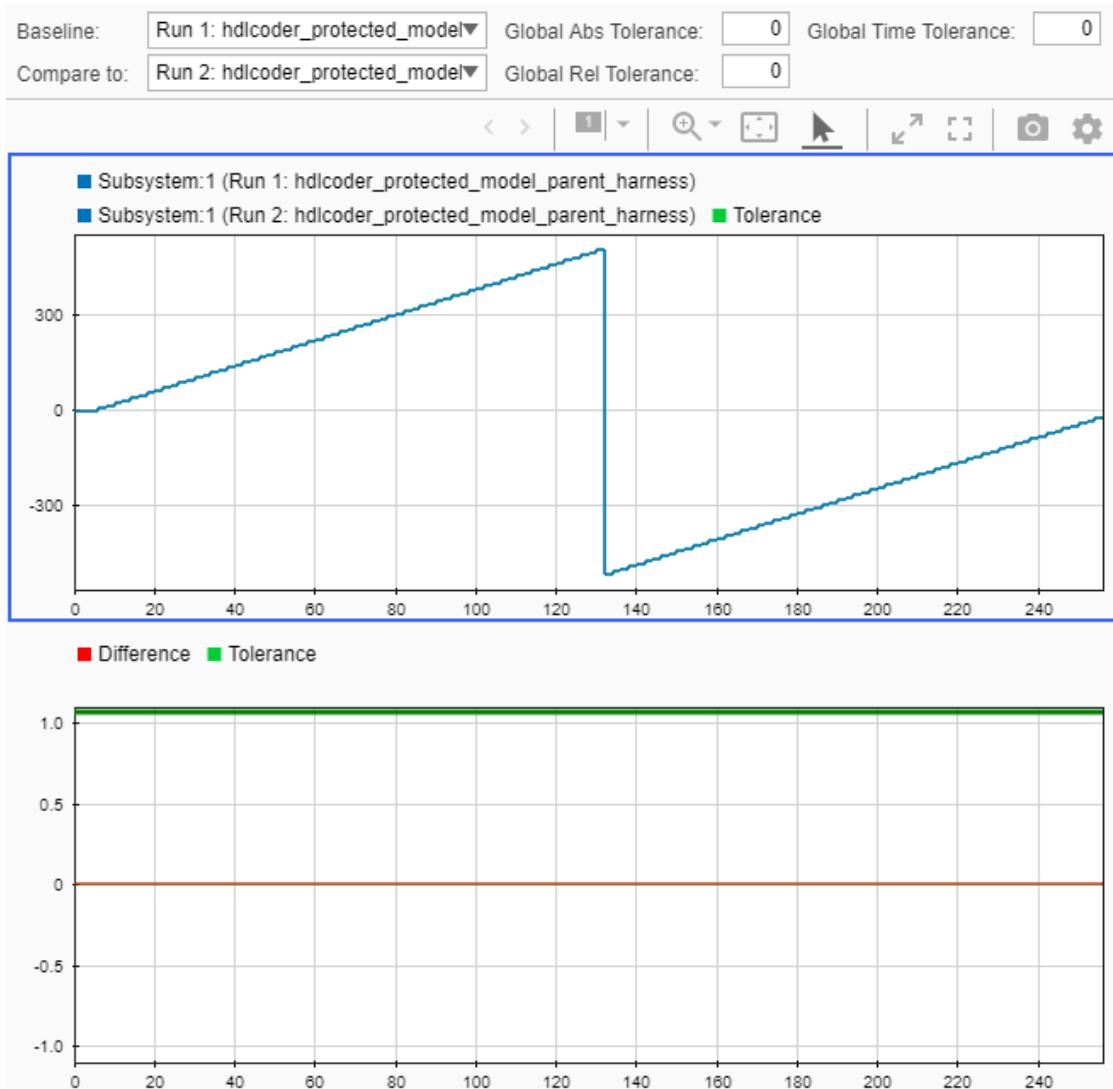
```
open_system('hdlcoder_protected_model_parent_harness')
```

- 2 Make sure that the Model block in the `hdlcoder_protected_model_parent_harness/DUT/mynested` subsystem is referencing the original model `hdlcoder_referenced_model_gain.slx` and not the protected model.
- 3 Enable logging for the output signals of the Model block. Right-click the output signals and select **Log Selected Signals**.
- 4 Simulate the model and click the **Simulation Data Inspector**. In the Simulation Data Inspector, select the signals that you logged to see the simulation results. Save this simulation run with a name such as `original_model_run`.
- 5 Now, in the Block Parameters dialog box for the Model block, change the **Model name** to `hdlcoder_referenced_model_gain.slxp`.

A badge icon appears on the Model block indicating that you are referencing the protected model. If you haven't already created the protected model, follow the steps mentioned in “Create Protected Models to Conceal Contents and Generate HDL Code” on page 34-2.

- 6 Simulate the model, which now refers to the protected model. When the simulation is complete, a new run appears in the Simulation Data Inspector. Save this run as `protected_model_run`.
- 7 In the Simulation Data Inspector, click the **Compare** tab. From the **Baseline** and **Compare To** lists, select the `original_model_run` and the `protected_model_run`. To compare the runs, click **Compare Runs**.

This figure displays the comparison between the **Baseline** and **Compare To** lists. You see that the simulation results match.



## See Also

### Functions

`Simulink.ModelReference.modifyProtectedModel` | `Simulink.ModelReference.protect`

## More About

- “Export Signal Data Using Signal Logging”
- “Compare Simulation Data”
- “Package and Share Protected Models” on page 34-11

## Package and Share Protected Models

When you protect a model, you can automatically create and package the following contents in a project archive (.mlproj) for easy sharing:

- Protected model file (.slxp)
- Harness model file
- MAT-file with base workspace definitions
- Data dictionary pruned to relevant definitions
- Other supporting files

In the Create Protected Model dialog box, set **Contents** to **Protected model (.slxp) and dependencies in a project**.

---

**Note** Before sharing the project, check whether the project contains the necessary supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed.

---

Alternatively, you can use one of these options to deliver the protected model package:

- Create a project archive to share a project that contains the protected model file and supporting files. For more information, see “Create a Project from a Model” and “Share Projects”.
- Provide the protected model file and supporting files as separate files.
- Combine the files into a ZIP or other container file.
- Provide the files in some other standard or proprietary format specified by the receiver.

Whichever approach you use to deliver a protected model, include information on how to retrieve the original files.

## Harness Model

You can create a harness model when you create your protected model. The harness model contains a Model block that references the protected model. A third-party can use the Model block to reference your protected model. The harness model is set up for simulation of the protected model.

## MAT-File with Base Workspace Definitions

Referenced models can use object definitions or tunable parameters that are defined in the MATLAB base workspace. These variables are not saved with the model. When you protect a model, you must obtain the definitions of required base workspace entities and ship them with the model.

For example, if the model uses the following base workspace variables, they must be saved to a MAT-file:

- Global tunable parameter
- Global data store
- The following objects used by a signal that connects to a root-level model Import or Outport:

- `Simulink.Signal`
- `Simulink.Bus`
- `Simulink.Alias`
- `Simulink.NumericType` that is an alias

To determine the required base workspace definitions and save them to a MAT-file, see “Protected Models for Model Reference”. Before executing the protected model as a part of a third-party model, the receiver of the protected model must load the MAT-file.

## Simulink Data Dictionary

Referenced models can use data definitions from a data dictionary, which are not saved with the model. When you protect a model that uses a data dictionary, package and ship the data dictionary with the protected model.

## Protected Model File Contents

A protected model file (`.slxp`) consists of the derived files that support the options that you selected when you created the protected model. The derived files are unpacked when you or a third-party use the protected model in simulation. You do not need to package these derived files with the protected model.

The derived files that are unpacked depends on the support that you enabled when creating the protected model. The `slprj/sim/model/*` files are deleted after they are used.

This table illustrates the files that are unpacked depending on the options that you specified. If you specified the **Use generated code** or **Code Interface** options when creating the protected model, additional files are unpacked in the derived folder. To learn about these files, see “Protected Model File Contents”.

## Protected Model Derived Files

| Supported Functionality                                                                                                      | Derived Files                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Created a protected model for simulation only and the referencing model is in <b>Normal</b> mode                             | The <i>model.mexext</i> file is placed in the build folder.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Created protected model for simulation only and referencing model is in <b>Accelerator</b> or <b>Rapid Accelerator</b> mode. | <p>These files are unpacked in the <i>slprj/sim/</i> folder:</p> <ul style="list-style-type: none"> <li>• <i>slprj/sim/model/*.h</i></li> <li>• <i>slprj/sim/model/modellib.a</i> (or <i>modellib.lib</i>)</li> <li>• <i>slprj/sim/model/tmwinternal/*</i></li> <li>• <i>slprj/sim/_sharedutils/*</i></li> </ul> <p>For the protected model report, these additional files are unpacked (but not in the build folder):</p> <ul style="list-style-type: none"> <li>• <i>slprj/sim/model/html/*</i></li> <li>• <i>slprj/sim/model/buildinfo.mat</i></li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                              |
| Created protected model with HDL code generation support.                                                                    | <p>The files are unpacked in the <i>hdlsrc</i> folder: (Additional files depend on whether you enabled support for other options such as code generation).</p> <ul style="list-style-type: none"> <li>• <i>hdlsrc/model/model.vhd</i> (or <i>model.v</i> if you specified Verilog as the <b>Target language</b>).</li> <li>• <i>hdlsrc/model/Subsystem.vhd</i> (or <i>Subsystem.v</i> if you specified Verilog as the <b>Target language</b> of the model that you protected. The additional HDL files depend on how hierarchically the referenced model was designed).</li> <li>• <i>hdlsrc/model/model_pkg.vhd</i> (This file is not generated if you specified Verilog as the <b>Target language</b> of the model that you protected).</li> <li>• <i>hdlsrc/model/model_report.html</i></li> <li>• <i>hdlsrc/model/gm_model.slxp</i> (This is a generated protected model. If you use cosimulation, HDL Coder instantiates this generated protected model).</li> </ul> |

## See Also

### Functions

[Simulink.ModelReference.modifyProtectedModel](#) | [Simulink.ModelReference.protect](#)

## More About

- “Create Protected Models to Conceal Contents and Generate HDL Code” on page 34-2
- “Test Protected Models” on page 34-9

## Obfuscate Generated HDL Code from Simulink Models

To share HDL code with a third party without revealing the intellectual property, you can generate obfuscated HDL code from Simulink models. Obfuscation reduces readability of the code. The generated HDL code does not have any comments, newlines, or spaces, and replaces identifier names with random names.

### How to Generate Obfuscated HDL Code

By default, the generated HDL code is not obfuscated. The HDL code contains newlines, comments, and is readable.

To generate obfuscated HDL code for the DUT subsystem in your model:

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Open the **HDL Code Generation** pane of the Configuration Parameters dialog box. In the **HDL Code** tab, select **Settings > HDL Code Generation Settings**.
- 3 Specify generation of obfuscated HDL code. In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Coding Style > RTL Style** section, select the **Generate obfuscated HDL code** check box.
- 4 Generate HDL code. Select the DUT subsystem as the **Code for** subsystem, and then click the **Generate HDL Code** button.

---

**Tip** By default, HDL Coder generates obfuscated VHDL code. To generate obfuscated Verilog code, in the **HDL Code Generation** pane, set **Language** to Verilog and then click the **Generate HDL Code** button.

---

To generate obfuscated HDL code from the command line, use the `ObfuscateGeneratedHDLCode` property with `hdlset_param` or `makehdl`. For example, to generate obfuscated HDL code for the `symmetric_fir` subsystem in the `sfir_fixed` model:

```
makehdl('sfir_fixed/symmetric_fir', 'ObfuscateGeneratedHDLCode', 'on')

% To generate obfuscated Verilog code, set 'Targetlanguage' to 'Verilog'
makehdl('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog', ...
        'ObfuscateGeneratedHDLCode', 'on')
```

### Generated HDL Code with Obfuscation

By default, the generated HDL code is not obfuscated. For example, this code shows the generated VHDL code for the **Complex Multiplier** model template in Simulink. To learn more about this template, see “Use Simulink Templates for HDL Code Generation” on page 10-7.

```
...
-----
-- Module: HDL_Complex_Multiplier
-- Source Path: untitled/HDL_Complex_Multiplier
-- Hierarchy Level: 0
--
-----
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

```

USE IEEE.numeric_std.ALL;

ENTITY HDL_Complex_Multiplier IS
  PORT( ...

    X_re      : IN std_logic_vector(17 DOWNTO 0); -- sfix18_En17
    ...
  );
END HDL_Complex_Multiplier;
...

```

To generate obfuscated HDL code, enable HDL code obfuscation and then generate code. For example, this code shows entity names and port names that are obfuscated in the generated VHDL code.

```
LIBRARY IEEE; ... ENTITY Q1LNc1j7NFXR IS PORT(EEY54qLw4C0j9uD:IN std_logic_vector(17 DOWNTO 0); ...
```

## Code Obfuscation Report

When you specify generation of obfuscated HDL code, and then generate code, HDL Coder produces a Code Obfuscation report. The Code Obfuscation report displays the status of HDL code obfuscation. It also displays whether the model uses configuration parameters that are incompatible with code obfuscation and provides a link to disable these parameters. These parameters are ignored during the obfuscation process.

## HDL Model Parameters Incompatible with Code Obfuscation

HDL code obfuscation is not compatible with certain Configuration Parameters and ignores these parameters if they are enabled on the model. The parameters include:

- These parameters in the **HDL Code Generation > Global Settings > General** pane:
  - “Enable prefix” on page 17-13, “Clocked process postfix” on page 17-6, and “Timing controller postfix” on page 17-7
  - “Split entity and architecture Parameters” on page 17-25
  - “Complex Signals Postfix Parameters” on page 17-28
  - “Pipeline postfix” on page 17-23
  - “Instance prefix” on page 17-33 and “Instance postfix” on page 17-33
  - “Language-Specific Identifiers and Postfix Parameters” on page 17-21
  - “Generate Statement and Vector and Component Instance Label Parameters” on page 17-32
- These parameters in the **HDL Code Generation > Global Settings > Coding Standards** tab:
  - “Choose Coding Standard and Report Option Parameters” on page 17-64
  - “Basic Coding Practices Parameters” on page 17-66
  - “RTL Description Rules for clock enables and resets Parameters” on page 17-71, “RTL Description Rules for Conditional Parameters” on page 17-74, and “Other RTL Description Rule Parameters” on page 17-77
- The “File Comment Customization Parameters” on page 17-62 parameters in the **HDL Code Generation > Global Settings > Coding Style** tab.

- The “Generate traceability report” on page 18-3 parameter in the **HDL Code Generation > Report** pane.

## Code Obfuscation Considerations and Restrictions

- Synthesizing the obfuscated HDL code might produce different synthesis results from the synthesis results of the original HDL code. For best results, perform synthesis on the original code instead of the obfuscated code.
- HDL code obfuscation replaces only names corresponding to HDL files, signals, blocks, variable names, or ports with random names. Other identifier names are not replaced, such as names of vectors or enumerations.
- For some interfaces that you use in your Simulink model, the interface information such as the port names and interface names are preserved in the obfuscated HDL code. These names are not obfuscated. The interfaces include:
  - DUT
  - Model reference
  - Black box
  - Xilinx or Intel floating-point target
- You cannot obfuscate the HDL code generated for these blocks:
  - CIC Interpolation
  - FIR Decimation
  - FIR Interpolation

## See Also

### Functions

`makehdl` | `makehdltb`

### Simulink Configuration Parameters

“Generate obfuscated HDL code” on page 17-46

## More About

- “Create HDL-Compatible Simulink Model”
- “Generate HDL Code from Simulink Model”

# HDL Test Bench

---

- “Verify Generated Code Using HDL Test Bench from Configuration Parameters” on page 35-2
- “Verify Generated Code Using HDL Test Bench at Command Line” on page 35-9
- “Test Bench Generation” on page 35-15
- “Test Bench Block Restrictions” on page 35-17

# Verify Generated Code Using HDL Test Bench from Configuration Parameters

## In this section...

- “FIR Filter Model” on page 35-2
- “Create a Folder and Copy Relevant Files” on page 35-4
- “What is a HDL Test Bench?” on page 35-5
- “How to Verify the Generated Code” on page 35-5
- “Generate HDL Test Bench” on page 35-5
- “View HDL Test Bench Files” on page 35-6
- “Run Simulation and Verify Generated HDL Code” on page 35-7

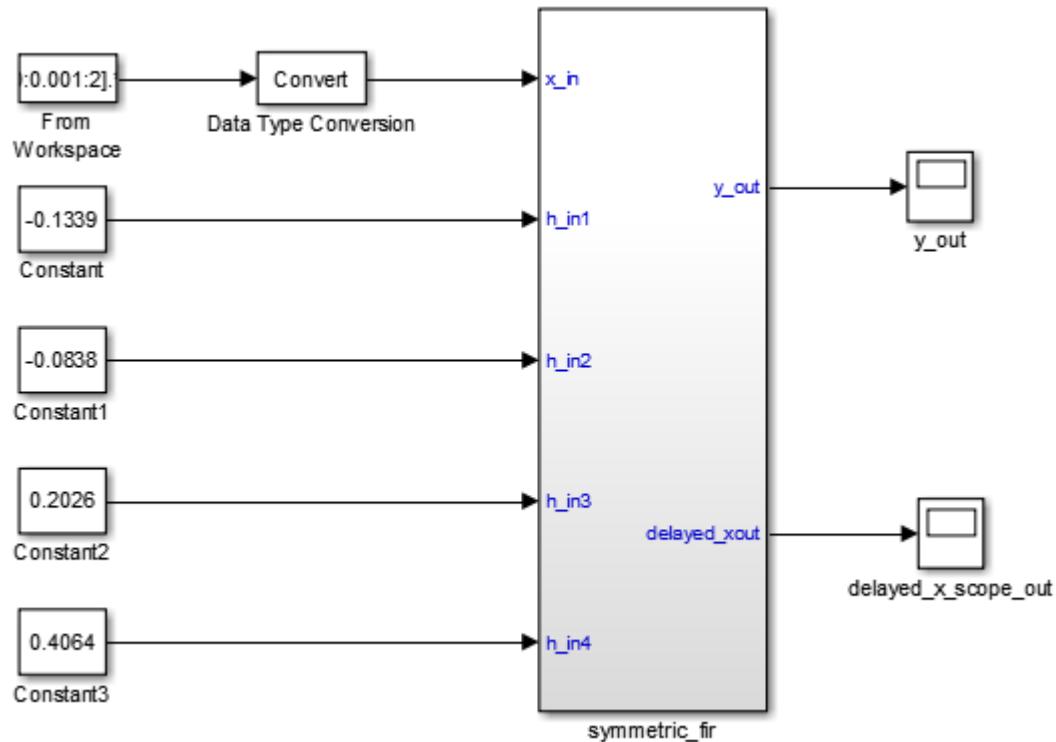
This example shows how to generate a HDL test bench and verify the generated code for your design. The example assumes that you have generated HDL code for your model. If you haven't generated HDL code, you can still open this model and generate the HDL test bench. Before generating the test bench, HDL Coder runs code generation to make sure that there is at least one successful code generation run before generating the testbench.

This example illustrates how to verify the generated code for the FIR filter model. To learn how to generate HDL code for this model, see “Generate HDL Code from Simulink Model Using Configuration Parameters” on page 12-11.

## FIR Filter Model

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



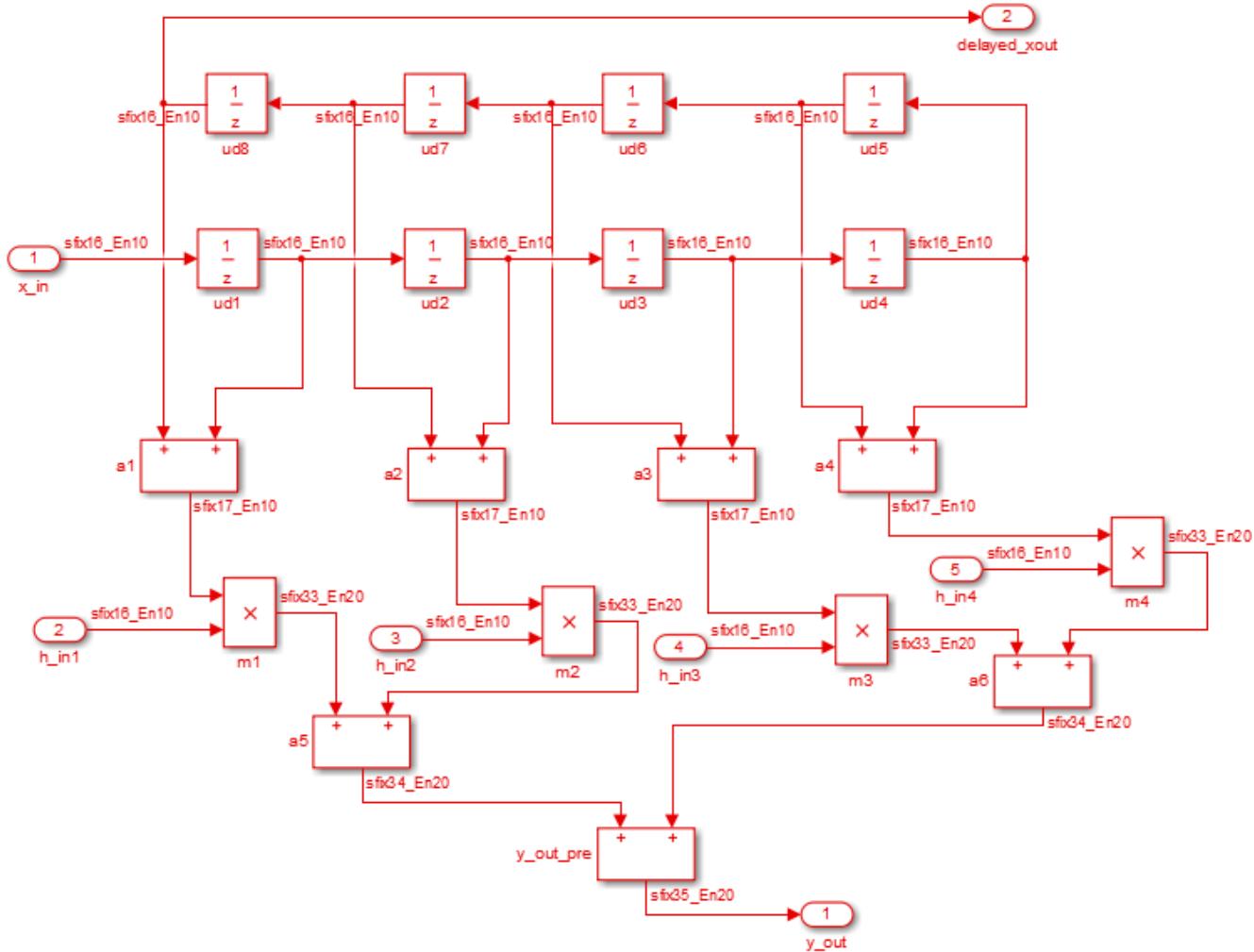
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



## Create a Folder and Copy Relevant Files

In MATLAB:

- 1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

`sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

- 2 Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

- 3 Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

## What is a HDL Test Bench?

To verify the functionality of the HDL code that you generated for the DUT, generate a HDL test bench. A test bench includes:

- Stimulus data generated by signal sources connected to the entity under test.
- Output data generated by the entity under test. During a test bench run, this data is compared to the outputs of the VHDL code, for verification purposes.
- Clock, reset, and clock enable inputs to drive the entity under test.
- A component instantiation of the entity under test.
- Code to drive the entity under test and compare its outputs to the expected data.

You can simulate the generated test bench and script files with the Mentor Graphics ModelSim simulator.

## How to Verify the Generated Code

This example illustrates how to generate a HDL test bench to simulate and verify the generated HDL code for your design. You can also verify the generated HDL code from your model using these methods:

| Verification Method                                  | For More Information                                 |
|------------------------------------------------------|------------------------------------------------------|
| Validation Model                                     | "Generated Model and Validation Model" on page 24-10 |
| HDL Cosimulation (requires HDL Verifier)             | "Cosimulation"                                       |
| SystemVerilog DPI Test Bench (requires HDL Verifier) | "SystemVerilog DPI Test Bench"                       |
| FPGA-in-the-Loop (requires HDL Verifier)             | "FPGA-in-the-Loop"                                   |

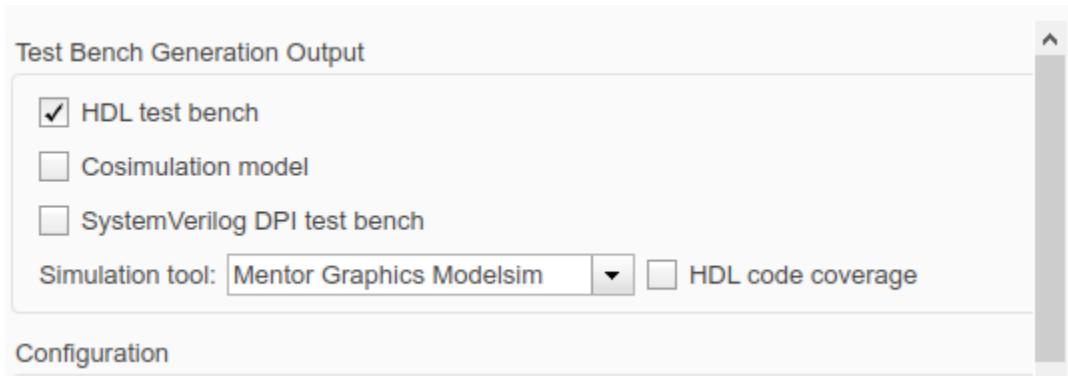
## Generate HDL Test Bench

Depending on whether you generated VHDL or Verilog code, generate VHDL or Verilog test bench code. The test bench code drives the HDL code that you generated for the DUT. By default, the HDL code and the test bench code are written to the same target folder `hdlsrc` relative to the current folder.

For the FIR filter, the `symmetric_fir` subsystem is the DUT. To generate the testbench, select this subsystem. You cannot generate a HDL testbench for an entire model.

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Select the DUT Subsystem in your model, and make sure that this Subsystem name appears in the **Code for** option. To remember the selection, you can pin this option. Click **Generate Testbench**.

By default, HDL Coder generates VHDL testbench code in the target `hdlsrc` folder.



### Generate Verilog Test Bench Code

If you want to generate Verilog test bench code, you can specify this setting in the **HDL Code Generation** pane of the Configuration Parameters dialog box.

To generate Verilog testbench code for the counter model:

- 1 In the **HDL Code** tab, click **Settings**.
- 2 In the **HDL Code Generation** pane, for **Language**, select **Verilog**. Leave other settings to the default.
- 3 In the **HDL Code Generation > Test Bench** pane, click **Generate Test Bench**.

If you haven't already generated code for your model, HDL Coder compiles the model and generates HDL code before generating the test bench. Depending on model display options such as port data types, the model can change in appearance after code generation.

As test bench generation proceeds, HDL Coder displays progress messages. The process should complete with the message

```
### HDL TestBench Generation Complete.
```

After generating the test bench, you see the generated files in the `hdlsrc` folder.

### View HDL Test Bench Files

- `symmetric_fir_tb.vhd`: VHDL test bench code, with generated test and output data. If you generated Verilog test bench code, the generated file is `symmetric_fir_tb.v`.
- `symmetric_fir_tb_pkg.vhd`: Package file for VHDL test bench code. This file is not generated if you specified Verilog as the target language.
- `symmetric_fir_tb_compile.vhd`: Compilation script (vcom commands). This script compiles and loads the entity to be tested (`symmetric_fir.vhd`) and the test bench code (`symmetric_fir_tb.vhd`).
- `symmetric_fir_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up `wave` window signal displays, and run a simulation.

To view the generated test bench code in the MATLAB Editor, double-click the `symmetric_fir_tb.vhd` or `symmetric_fir_tb.v` file in the current folder.

## Run Simulation and Verify Generated HDL Code

To verify the simulation results, you can use the Mentor Graphics ModelSim simulator. Make sure that you have already installed Mentor Graphics ModelSim.

To launch the simulator, use the `vsim` (HDL Verifier) function. This command shows how to open the simulator by specifying the path to the executable:

```
vsim('vsimdir','C:\Program Files\ModelSim\questasim\10.6b\win64\vsim.exe')
```

To compile and run a simulation of the generated model and test bench code, use the scripts that are generated by HDL Coder. Following example illustrates the commands that compile and simulate the generated test bench for the `sfir_fixed/symmetric_fir` subsystem.

- 1 Open the Mentor Graphics ModelSim software and navigate to the folder that has the previously generated code files and the scripts.

```
QuestaSim>cd C:/work/sl_hdlcoder_work/hdlsrc
```

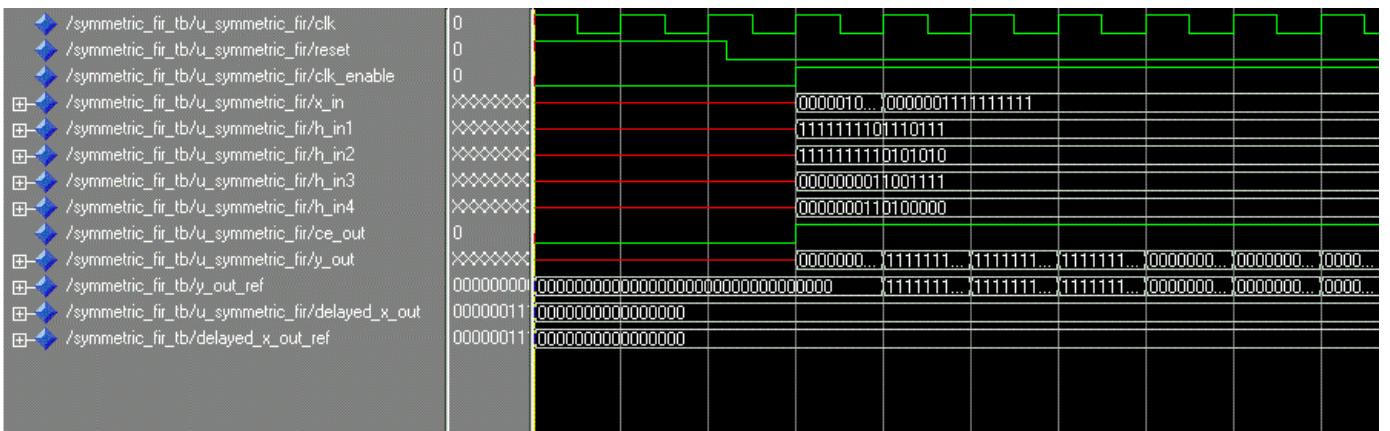
- 2 Use the generated compilation script to compile and load the generated model and text bench code. Run this command to compile the generated code.

```
QuestaSim>do symmetric_fir_tb_compile.do
```

- 3 Use the generated simulation script to execute the simulation. The following listing shows the command and responses. You can ignore any warning messages. The test bench termination message indicates that the simulation has run to completion without comparison errors. Run this command to simulate the generated code.

```
QuestaSim>do symmetric_fir_tb_sim.do
```

The simulator optimizes your design and displays the results of simulating your HDL design in a **wave** window. If you don't see the simulation results, open the **wave** window. The simulation script displays inputs and outputs in the model including the clock, reset, and clock enable signals in the **wave** window.



You can now view the signals and verify that the simulation results match the functionality of your original design. After verifying, close the Mentor Graphics ModelSim simulator, and then close the files that you have opened in the MATLAB Editor.

## See Also

`makehdl` | `makehdltb`

## More About

- “Test Bench Generation Output Parameters” on page 19-3
- “HDL Test Bench”
- “HDL Code Generation and FPGA Synthesis from Simulink Model”

# Verify Generated Code Using HDL Test Bench at Command Line

## In this section...

- “FIR Filter Model” on page 35-9
- “Create a Folder and Copy Relevant Files” on page 35-11
- “What is a HDL Test Bench?” on page 35-12
- “How to Verify the Generated Code” on page 35-12
- “Generate HDL Test Bench” on page 35-12
- “View HDL Test Bench Files” on page 35-13
- “Run Simulation and Verify Generated HDL Code” on page 35-13

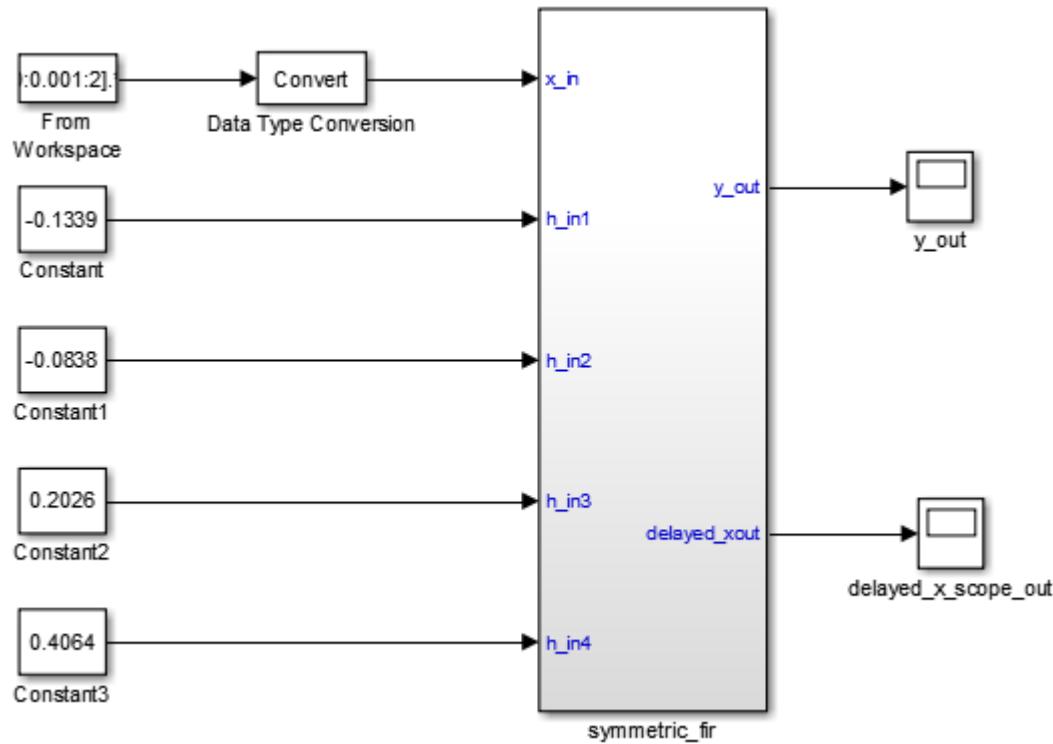
This example shows how to generate a HDL test bench and verify the generated code for your design. The example assumes that you have already generated HDL code for your model. If you haven't already generated HDL code, you can still open this model and generate the HDL test bench. Before generating the test bench, HDL Coder runs code generation to make sure that there is at least one successful code generation run before generating the testbench.

This example illustrates how to verify the generated code for the FIR filter model. To learn how to generate HDL code, see “Generate HDL Code from Simulink Model from Command Line” on page 12-15.

## FIR Filter Model

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



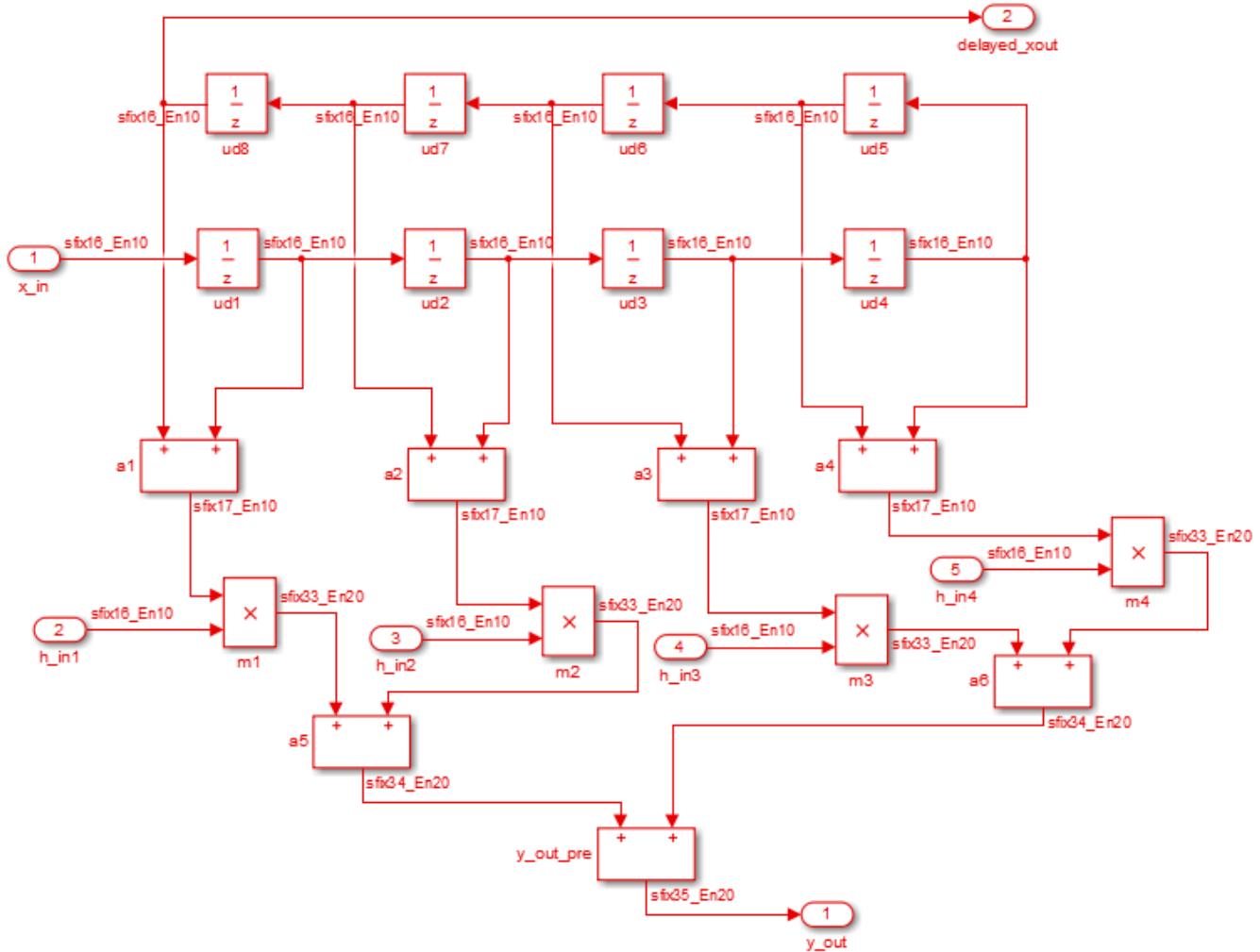
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



## Create a Folder and Copy Relevant Files

In MATLAB:

- 1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

`sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

- 2 Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

- 3 Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

## What is a HDL Test Bench?

To verify the functionality of the HDL code that you generated for the DUT, generate a HDL test bench. A test bench includes:

- Stimulus data generated by signal sources connected to the entity under test.
- Output data generated by the entity under test. During a test bench run, this data is compared to the outputs of the VHDL code, for verification purposes.
- Clock, reset, and clock enable inputs to drive the entity under test.
- A component instantiation of the entity under test.
- Code to drive the entity under test and compare its outputs to the expected data.

You can simulate the generated test bench and script files with the Mentor Graphics ModelSim simulator.

## How to Verify the Generated Code

This example illustrates how to generate a HDL test bench to simulate and verify the generated HDL code for your design. You can also verify the generated HDL code from your model using these methods:

| Verification Method                                  | For More Information                                 |
|------------------------------------------------------|------------------------------------------------------|
| Validation Model                                     | "Generated Model and Validation Model" on page 24-10 |
| HDL Cosimulation (requires HDL Verifier)             | "Cosimulation"                                       |
| SystemVerilog DPI Test Bench (requires HDL Verifier) | "SystemVerilog DPI Test Bench"                       |
| FPGA-in-the-Loop (requires HDL Verifier)             | "FPGA-in-the-Loop"                                   |

## Generate HDL Test Bench

Depending on whether you generated VHDL or Verilog code, generate VHDL or Verilog test bench code. The test bench code drives the HDL code that you generated for the DUT. By default, the HDL code and the test bench code are written to the same target folder `hdlsrc` relative to the current folder.

To generate test bench code and the scripts for compilation and simulation, use the `makehdltb` function. At the command line, enter:

```
makehdltb('sfir_fixed/symmetric_fir')
```

To specify the customizations before you generate testbench code, use the `hdlset_param` function. You can also specify various name-value pair arguments with the `makehdltb` function to customize HDL code generation options while generating HDL code. For example, to generate Verilog test bench code, use the `TargetLanguage` property.

```
makehdltb('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog')
```

Alternatively, if you are using `hdlset_param`, set this parameter on the model and then run the `makehdltb` function.

```
hdlset_param('sfir_fixed', 'TargetLanguage', 'Verilog')
makehdl('sfir_fixed/symmetric_fir')
```

If you haven't already generated code for your model, HDL Coder compiles the model and generates HDL code before generating the test bench. Depending on model display options such as port data types, the model can change in appearance after code generation.

As test bench generation proceeds, HDL Coder displays progress messages. The process should complete with the message

```
### HDL TestBench Generation Complete.
```

After generating the test bench, you see the generated files in the `hdlsrc` folder.

## View HDL Test Bench Files

- `symmetric_fir_tb.vhd`: VHDL test bench code, with generated test and output data. If you generated Verilog test bench code, the generated file is `symmetric_fir_tb.v`.
- `symmetric_fir_tb_pkg.vhd`: Package file for VHDL test bench code. This file is not generated if you specified Verilog as the target language.
- `symmetric_fir_tb_compile.vhd`: Compilation script (vcom commands). This script compiles and loads the entity to be tested (`symmetric_fir.vhd`) and the test bench code (`symmetric_fir_tb.vhd`)..
- `symmetric_fir_tb_sim.do`: Mentor Graphics ModelSim script to initialize the simulator, set up `wave` window signal displays, and run a simulation.

To view the generated test bench code in the MATLAB Editor, double-click the `symmetric_fir_tb.vhd` or `symmetric_fir_tb.v` file in the current folder.

## Run Simulation and Verify Generated HDL Code

To verify the simulation results, you can use the Mentor Graphics ModelSim simulator. Make sure that you have already installed Mentor Graphics ModelSim.

To launch the simulator, use the `vsim` (HDL Verifier) function. This command shows how to open the simulator by specifying the path to the executable:

```
vsim('vsimdir','C:\Program Files\ModelSim\questasim\10.6b\win64\vsim.exe')
```

To compile and run a simulation of the generated model and test bench code, use the scripts that are generated by HDL Coder. Following example illustrates the commands that compile and simulate the generated test bench for the `sfir_fixed/symmetric_fir` subsystem.

- 1 Open the Mentor Graphics ModelSim software and navigate to the folder that has the previously generated code files and the scripts.

```
QuestaSim>cd C:/work/sl_hdlcoder_work/hdlsrc
```

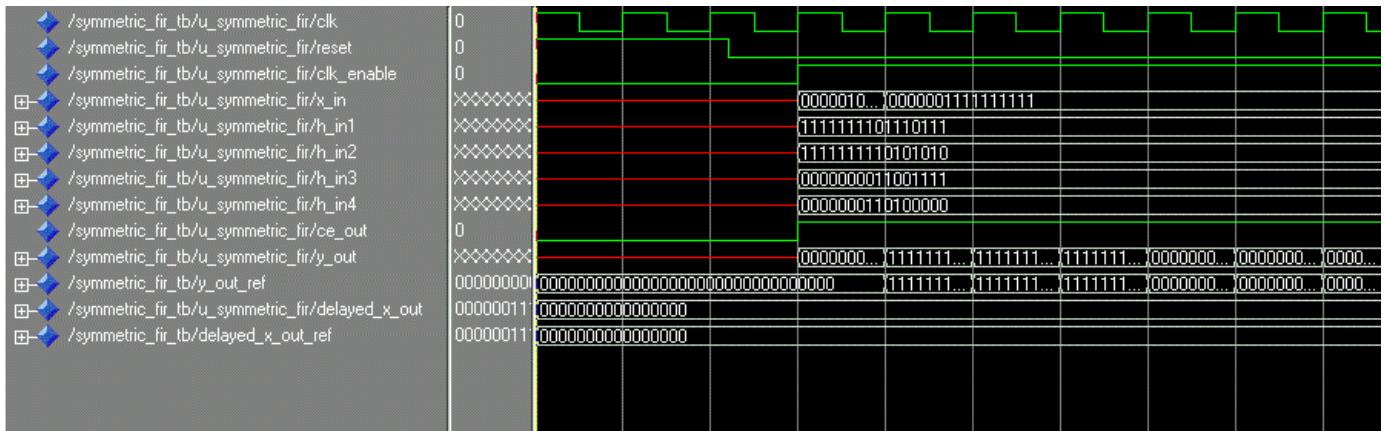
- 2 Use the generated compilation script to compile and load the generated model and test bench code. Run this command to compile the generated code.

```
QuestaSim>do symmetric_fir_tb_compile.do
```

- 3** Use the generated simulation script to execute the simulation. The following listing shows the command and responses. You can ignore any warning messages. The test bench termination message indicates that the simulation has run to completion without comparison errors. Run this command to simulate the generated code.

```
QuestaSim>do symmetric_fir_tb_sim.do
```

The simulator optimizes your design and displays the results of simulating your HDL design in a **wave** window. If you don't see the simulation results, open the **wave** window. The simulation script displays inputs and outputs in the model including the clock, reset, and clock enable signals in the **wave** window.



You can now view the signals and verify that the simulation results match the functionality of your original design. After verifying, close the Mentor Graphics ModelSim simulator, and then close the files that you have opened in the MATLAB Editor.

## See Also

`makehdl` | `makehdltb`

## More About

- “Test Bench Generation Output Parameters” on page 19-3
- “HDL Test Bench”
- “HDL Code Generation and FPGA Synthesis from Simulink Model”

# Test Bench Generation

You can generate a HDL Testbench for a subsystem or model reference that you specify in your Simulink model. The coder generates an HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT.

## In this section...

- ["How Test Bench Generation Works" on page 35-15](#)
- ["Test Bench Data Files" on page 35-15](#)
- ["Test Bench Data Type Limitations" on page 35-15](#)
- ["Use Constants Instead of File I/O" on page 35-15](#)

## How Test Bench Generation Works

HDL Coder writes the DUT stimulus and reference data from your MATLAB or Simulink simulation to data files (.dat).

During HDL simulation, the HDL test bench reads the saved stimulus from the .dat files. The test bench compares the actual DUT output with the expected output, which is also saved in .dat files. After you generate code, the message window displays links to the test bench data files.

Reference data is delayed by one clock cycle in the waveform viewer compared to default test bench generation due to the delay in reading data from files.

## Test Bench Data Files

The coder saves stimulus and reference data for each DUT input and output in a separate test bench data file (.dat), with the following exceptions:

- Two files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants.

Vector input or output data is saved as a single file.

## Test Bench Data Type Limitations

If you have double, single, or enumeration data types at the DUT inputs and outputs, the simulation data is generated as constants in the test bench code, instead of writing the simulation data to files.

## Use Constants Instead of File I/O

You can generate test bench stimulus and reference data as constants in the test bench code instead of using file I/O. Simulating a long running test bench that uses constants requires more memory than a test bench that uses file I/O.

If your DUT inputs or outputs use data types that are not supported for file I/O, test bench generation automatically generates data as constants. For details, see "Test Bench Data Type Limitations" on page 35-15.

## Using the HDL Workflow Advisor

To generate a test bench that uses constants:

- 1 In the **HDL Code Generation > Set Code Generation Options > Set Testbench Options** task, clear **Use file I/O to read/write test bench data** and click **Apply**.
- 2 In the **HDL Code Generation > Generate RTL Code and Testbench** task, select **Generate RTL testbench** and click **Apply**.

## Using the Command Line

To generate a test bench that uses constants, use the `UseFileIOInTestBench` parameter with `makehdltb`.

For example, to generate a Verilog test bench by using constants for a DUT subsystem, `sfir_fixed/symmetric_fir`, enter:

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','Verilog',...
    'UseFileIOInTestBench','off');
```

## See Also

`makehdltb`

## More About

- “Test Bench Block Restrictions” on page 35-17
- “Choose a Test Bench for Generated HDL Code” on page 27-39

## Test Bench Block Restrictions

Blocks that belong to the blocksets and toolboxes in the following list should not be directly connected to the DUT. Instead, place them in a subsystem, and connect the subsystem to the DUT. This restriction applies to all blocks in the following products:

- RF Blockset™
- Simscape Driveline™
- SimEvents®
- Simscape Multibody
- Simscape Electrical Power Systems
- Simscape



# FPGA Board Customization

---

- “FPGA Board Customization” on page 36-2
- “Create Custom FPGA Board Definition” on page 36-6
- “Create Xilinx KC705 Evaluation Board Definition File” on page 36-7
- “FPGA Board Manager” on page 36-18
- “New FPGA Board Wizard” on page 36-21
- “FPGA Board Editor” on page 36-32

# FPGA Board Customization

## In this section...

- “Feature Description” on page 36-2
- “Custom Board Management” on page 36-2
- “FPGA Board Requirements” on page 36-2

## Feature Description

Both HDL Coder and HDL Verifier software include a set of predefined FPGA boards you can use with the Turnkey or FPGA-in-the-loop (FIL) workflows. You can view the lists of these supported boards in the HDL Workflow Advisor or in the FIL wizard. With the FPGA Board Manager, you can add additional boards to use either of these workflows. To add a board, you need the relevant information from the board specification documentation.

The FPGA Board Manager is the hub for accessing wizards and dialog boxes that take you through the steps necessary to create a custom board configuration. You can also access options for:

- Importing a custom board
- Copying a board definition file for further modification
- Verifying a new board

## Custom Board Management

You manage FPGA custom boards through the following user interfaces:

- “FPGA Board Manager” on page 36-18: portal to adding, importing, deleting, and otherwise managing board definition files.
- “New FPGA Board Wizard” on page 36-21: This wizard guides you through creating a custom board definition file with information you obtain from the board specification documentation.
- “FPGA Board Editor” on page 36-32: user interface for viewing or editing board information.

To begin, review the “FPGA Board Requirements” on page 36-2 and then follow the steps described in “Create Custom FPGA Board Definition” on page 36-6.

## FPGA Board Requirements

- “FPGA Device” on page 36-2
- “FPGA Design Software” on page 36-3
- “General Hardware Requirements” on page 36-3
- “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 36-3
- “JTAG Connection Requirements for FPGA-in-the-Loop” on page 36-5

### FPGA Device

Select one of the following links to view a current list of supported FPGA device families:

- For use with FPGA-in-the-loop (FIL), see “Supported FPGA Device Families for Board Customization” (HDL Verifier).
- For use with FPGA Turnkey, see “Supported FPGA Device Families for Board Customization”.

## FPGA Design Software

Altera Quartus II or Xilinx ISE is required. See product documentation for HDL Coder or HDL Verifier for the specific software versions required.

The following MathWorks tools are required to use FIL or FPGA Turnkey.

| Workflow         | Required Tools                                                                                                    |
|------------------|-------------------------------------------------------------------------------------------------------------------|
| FPGA-in-the-loop | <ul style="list-style-type: none"> <li>• HDL Verifier</li> <li>• Fixed-Point Designer</li> </ul>                  |
| FPGA Turnkey     | <ul style="list-style-type: none"> <li>• HDL Coder</li> <li>• Simulink</li> <li>• Fixed-Point Designer</li> </ul> |

## General Hardware Requirements

To use an FPGA development board, make sure that you have the following FPGA resources:

- **Clock:** An external clock connected to the FPGA is required. The clock can be differential or single-ended. The accepted clock frequency is from 5 MHz to 300 MHz. When used with FIL, there are additional requirements to the clock frequency (see “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 36-3).
- **Reset:** An external reset signal connected to the FPGA is optional. When supplied, this signal functions as the global reset to the FPGA design.
- **JTAG download cable:** A JTAG download cable that connects host computer and FPGA board is required for the FPGA programming. The FPGA must be programmable using Xilinx iMPACT or Altera Quartus II.

## Ethernet Connection Requirements for FPGA-in-the-Loop

- “Supported Ethernet PHY Device” on page 36-3
- “Ethernet PHY Interface” on page 36-4
- “Special Timing Considerations for RGMII” on page 36-4
- “Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface” on page 36-4

## Supported Ethernet PHY Device

On the FPGA board, the Ethernet MAC is implemented in FPGA. An Ethernet PHY chip is required to be on the FPGA board to connect the physical medium to the Media Access (MAC) layer in the FPGA.

---

**Note** When programming the FPGA, HDL Verifier assumes that there is only one download cable connected to the Host computer. It also assumes that the FPGA programming software automatically recognizes the cable. If not, use FPGA programming software to program your FPGA with the correct options.

The FIL feature is tested with the following Ethernet PHY chips and may not work with other Ethernet PHY devices.

| Ethernet PHY Chip               | Test                                                  |
|---------------------------------|-------------------------------------------------------|
| Marvell® Alaska 88E1111         | For GMII, RGMII, SGMII, and 100 Base-T MII interfaces |
| National Semiconductor DP83848C | For 100 Base-T MII interface only                     |

### Ethernet PHY Interface

The Ethernet PHY chip must be connected to the FPGA using one of the following interfaces:

| Interface                                           | Note                                                       |
|-----------------------------------------------------|------------------------------------------------------------|
| Gigabit Media Independent Interface (GMII)          | Only 1000 Mbits/s speed is supported using this interface. |
| Reduced Gigabit Media Independent Interface (RGMII) | Only 1000 Mbits/s speed is supported using this interface. |
| Serial Gigabit Media Independent Interface (SGMII)  | Only 1000 Mbits/s speed is supported using this interface. |
| Media Independent Interface (MII)                   | Only 100 Mbits/s speed is supported using this interface.  |

**Note** For GMII, the TXCLK (clock signal for 10/100 Mbits signal) signal is not required because only 1000 Mbits/s speed is supported.

In addition to the standard GMII/RGMII/SGMII/MII interface signals, FPGA-in-the-loop also requires an Ethernet PHY chip reset signal (ETH\_RESET\_n). This active-low reset signal performs the PHY hardware reset by FPGA. It is active-low.

### Special Timing Considerations for RGMII

When the RGMII interface is used, the MAC on the FPGA assumes that the data are aligned with the edges of reference clock as specified in the original RGMII v1.3 standard. In this case, PC board designs provide additional trace delay for clock signals.

The RGMII v2.0 standard allows the transmitter to integrate this delay so that PC board delay is not required. Marvell Alaska 88E1111 has internal registers to add internal delays to RX and TX clocks. The internal delays are not added by default, which means that you must use the MDIO module to configure Marvell 88E1111 to add internal delays. For more information on the MDIO module, see “FIL I/O” on page 36-25.

### Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface

When GMII/RGMII/SGMII interfaces are used, the FPGA requires an exact 125 MHz clock to drive the 1000 Mbits/s communication. This clock is derived from the user supplied external clock using the clock module or PLL.

Not all external clock frequencies can derive an exact 125 MHz clock frequency. The acceptable clock frequencies vary depending on the FPGA device family. The recommended clock frequencies are 50, 100, 125, and 200 MHz.

## JTAG Connection Requirements for FPGA-in-the-Loop

| Vendor    | Required Hardware                                                                                                                                                                                                                                                    | Required Software                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Intel     | USB Blaster I or USB Blaster II download cable                                                                                                                                                                                                                       | <ul style="list-style-type: none"> <li>• USB Blaster I or II driver</li> <li>• For Windows® operating systems: Quartus Prime executable directory must be on system path.</li> <li>• For Linux® operating systems: versions below Quartus II 13.1 are not supported. Quartus II 14.1 is not supported. Only 64-bit Quartus is supported. Quartus library directory must be on <code>LD_LIBRARY_PATH</code> before starting MATLAB. Prepend the Linux distribution library path before the Quartus library on <code>LD_LIBRARY_PATH</code>. For example, <code>/lib/x86_64-linux-gnu:\$QUARTUS_PATH</code>.</li> </ul> |
| Xilinx    | Digilent® download cable. <ul style="list-style-type: none"> <li>• If your board has an onboard Digilent USB-JTAG module, use a USB cable.</li> <li>• If your board has a standard Xilinx 14 pin JTAG connector, use with HS2 or HS3 cable from Digilent.</li> </ul> | <ul style="list-style-type: none"> <li>• For Windows operating systems: Xilinx Vivado executable directory must be on system path.</li> <li>• For Linux operating systems: Digilent Adept2</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                 |
|           | FTDI USB-JTAG cable <ul style="list-style-type: none"> <li>• Supported for boards with onboard FT4232H, FT232H, or FT2232H devices implementing USB-to JTAG</li> </ul>                                                                                               | Supported for Windows operating systems.<br><b>Note</b> FTDI USB JTAG support is only available for MATLAB as AXI Master and for FPGA Data Capture.                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Microsemi | JTAG connection not supported                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## Create Custom FPGA Board Definition

- 1 Be ready with the following:
  - a Board specification document. Any format you are comfortable with is fine. However, if you have it in an electronic version, you can search for the information as it is required.
  - b If you plan to validate (test) your board definition file, set up FPGA design software tools:  
For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.
- 2 Open the FPGA Board Manager by typing `fpgaBoardManager` in the MATLAB command window. Alternatively, if you are using the HDL Workflow Advisor, you can click **Launch Board Manager** at Step 1.1.
- 3 Open the New FPGA Board wizard by clicking **Create New Board**. For a description of all the tasks you can perform with the FPGA Board Manager, see “FPGA Board Manager” on page 36-18.
- 4 The wizard guides you through entering all board information. At each page, fill in the required fields. For assistance in entering board information, see “New FPGA Board Wizard” on page 36-21.
- 5 Save the board definition file. This step is the last and is automatically instigated when you click **Finish** in the New FPGA Board wizard. See “Save Board Definition File” on page 36-13.

Your custom board definition now appears in the list of available FPGA Boards in the FPGA Board Manager. If you are using HDL Workflow Advisor, it also shows in the **Target platform** list.

Follow the example “Create Xilinx KC705 Evaluation Board Definition File” on page 36-7 for a demonstration of adding a custom FPGA board with the New FPGA Board Manager.

# Create Xilinx KC705 Evaluation Board Definition File

## In this section...

- “Overview” on page 36-7
- “What You Need to Know Before Starting” on page 36-7
- “Start New FPGA Board Wizard” on page 36-7
- “Provide Basic Board Information” on page 36-8
- “Specify FPGA Interface Information” on page 36-9
- “Enter FPGA Pin Numbers” on page 36-10
- “Run Optional Validation Tests” on page 36-12
- “Save Board Definition File” on page 36-13
- “Use New FPGA Board” on page 36-14

## Overview

For FPGA-in-the-loop, you can use your own qualified FPGA board, even if it is not in the pre-registered FPGA board list supplied by MathWorks. Using the New FPGA Board wizard, you can create a board definition file that describes your custom FPGA board.

In this example, you can follow the workflow of creating a board definition file for the Xilinx KC705 evaluation board to use with FIL simulation.

## What You Need to Know Before Starting

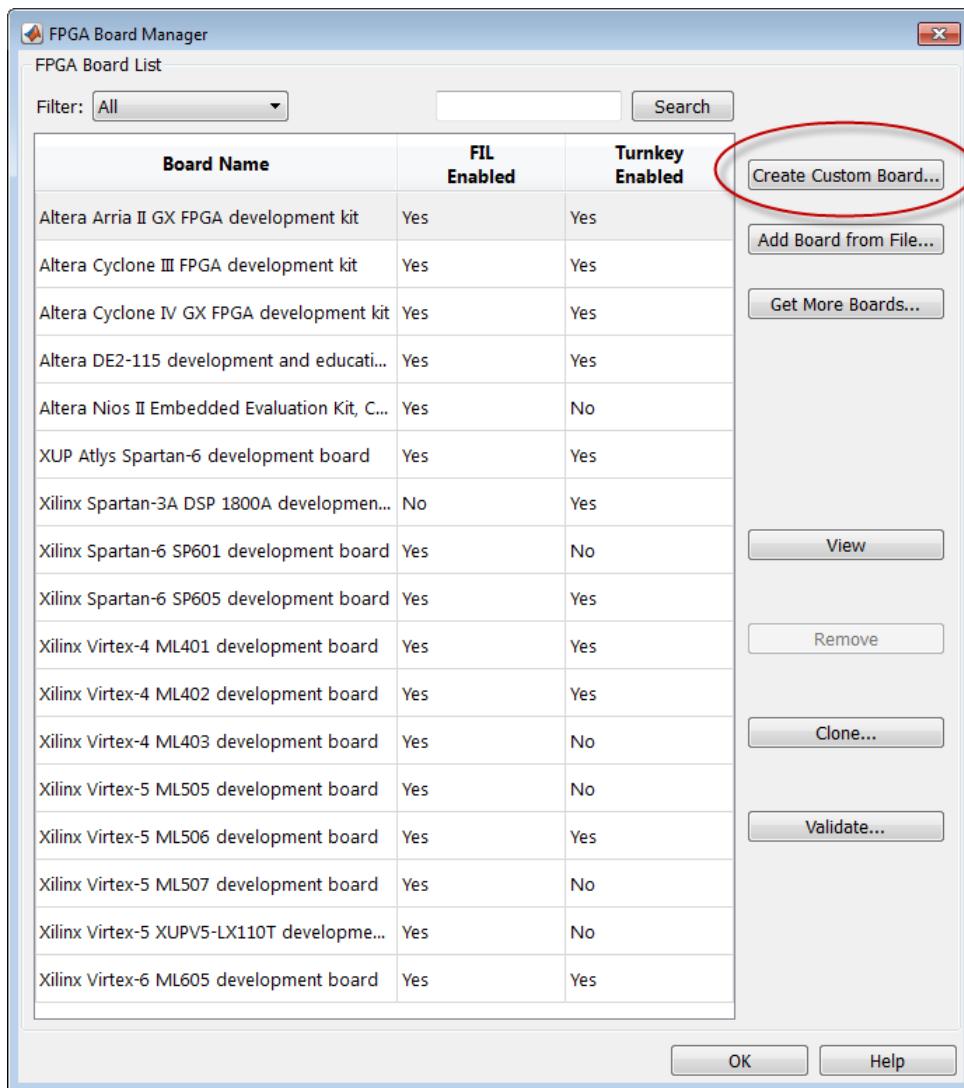
- Check the board specification so that you have the following information ready:
  - FPGA interface to the Ethernet PHY chip
  - Clock pins names and numbers
  - Reset pins names and numbers

In this example, the required information is supplied to you. In general, you can find this type of information in the board specification file. This example uses the KC705 Evaluation Board for the Kintex-7 FPGA User Guide, published by Xilinx.

- For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.
- To verify programming the FPGA board after you add its definition file, attach the custom board to your computer. However, having the board connected is not necessary for creating the board definition file.

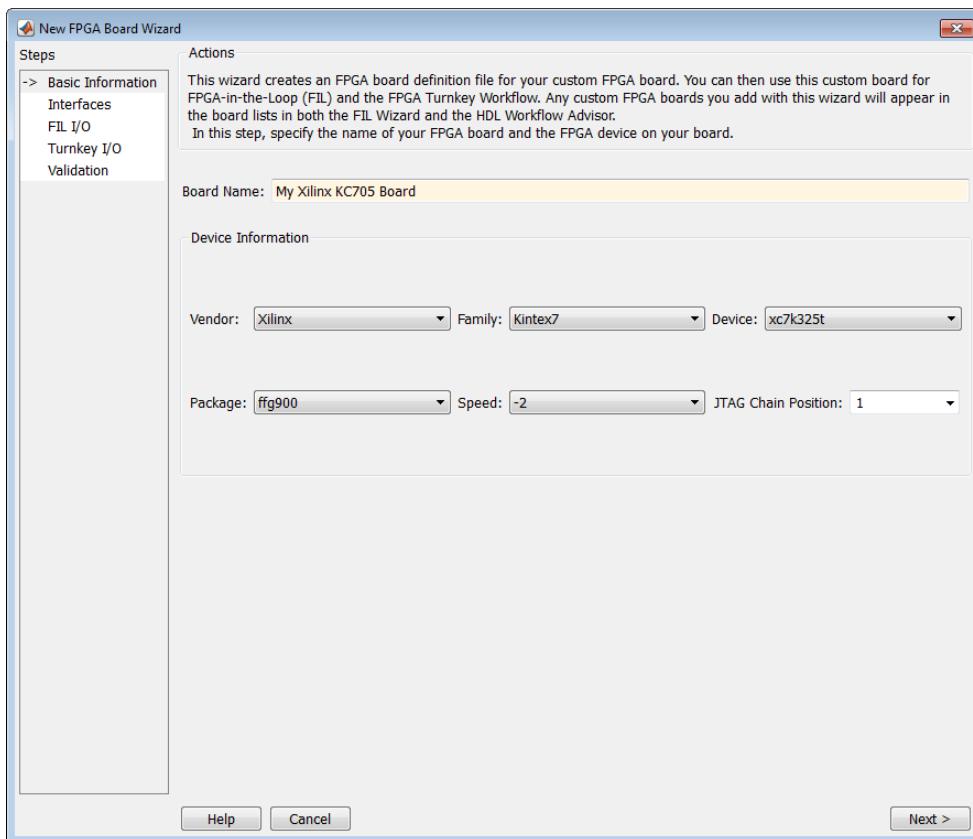
## Start New FPGA Board Wizard

- 1 Start the FPGA Board Manager by entering the following command at the MATLAB prompt:  
`>>fpgaBoardManager`
- 2 Click **Create Custom Board** to open the New FPGA Board wizard.



## Provide Basic Board Information

- In the Basic Information pane, enter the following information:
  - Board Name:** Enter "My Xilinx KC705 Board"
  - Vendor:** Select Xilinx
  - Family:** Select Kintex7
  - Device:** Select xc7k325t
  - Package:** Select ffg900
  - Speed:** Select -2
  - JTAG Chain Position:** Select 1



The information you just entered can be found in the KC705 Evaluation Board for the Kintex-7 FPGA User Guide.

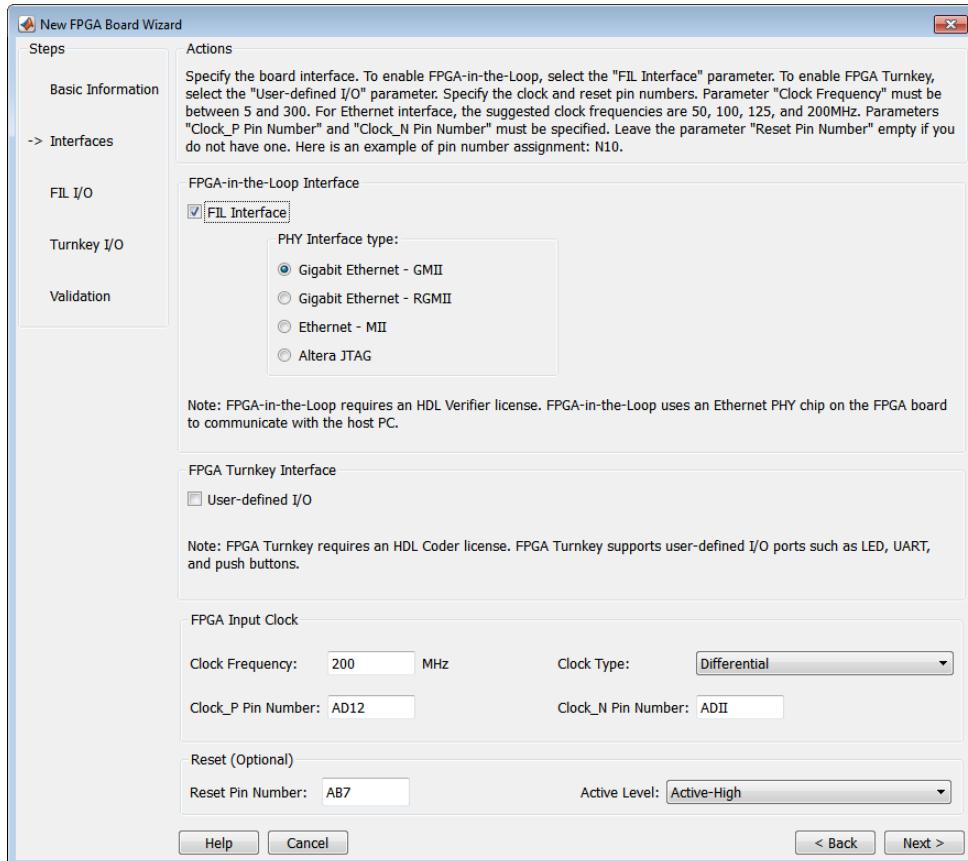
- 2 Click **Next**.

## Specify FPGA Interface Information

- 1 In the Interfaces pane, perform the following tasks.
  - a Select **FIL Interface**. This option is required for using your board with FPGA-in-the-loop.
  - b Select **GMII** in the PHY Interface Type. This option indicates that the onboard FPGA is connected to the Ethernet PHY chip via a GMII interface.
  - c Leave the **User-defined I/O** option in the FPGA Turnkey Interface section cleared. FPGA Turnkey workflow is not the focus of this example.
  - d **Clock Frequency:** Enter 200. This Xilinx KC705 board has multiple clock sources. The 200 MHz clock is one of the recommended clock frequencies for use with Ethernet interface (50, 100, 125, and 200 MHz).
  - e **Clock Type:** Select **Differential**.
  - f **Clock\_P Pin Number:** Enter AD12.
  - g **Clock\_N Pin Number:** Enter AD11.
  - h **Clock IO Standard** — Leave blank.
  - i **Reset Pin Number:** Enter AB7. This value supplies a global reset to the FPGA.

- j **Active Level:** Select Active-High.
- k **Reset IO Standard** — Leave blank.

You can obtain all necessary information from the board design specification.



- 2 Click **Next**.

## Enter FPGA Pin Numbers

- 1 In the FILI/O pane, enter the numbers for each FPGA pin. This information is required.

Pin numbers for RXD and TXD signals are entered from the least significant digit (LSD) to the most significant digit (MSB), separated by a comma.

| For signal name... | Enter FPGA pin number... |
|--------------------|--------------------------|
| ETH_COL            | W19                      |
| ETH_CRS            | R30                      |
| ETH_GTXCLK         | K30                      |
| ETH_MDC            | R23                      |
| ETH_MDIO           | J21                      |
| ETH_RESET_n        | L20                      |

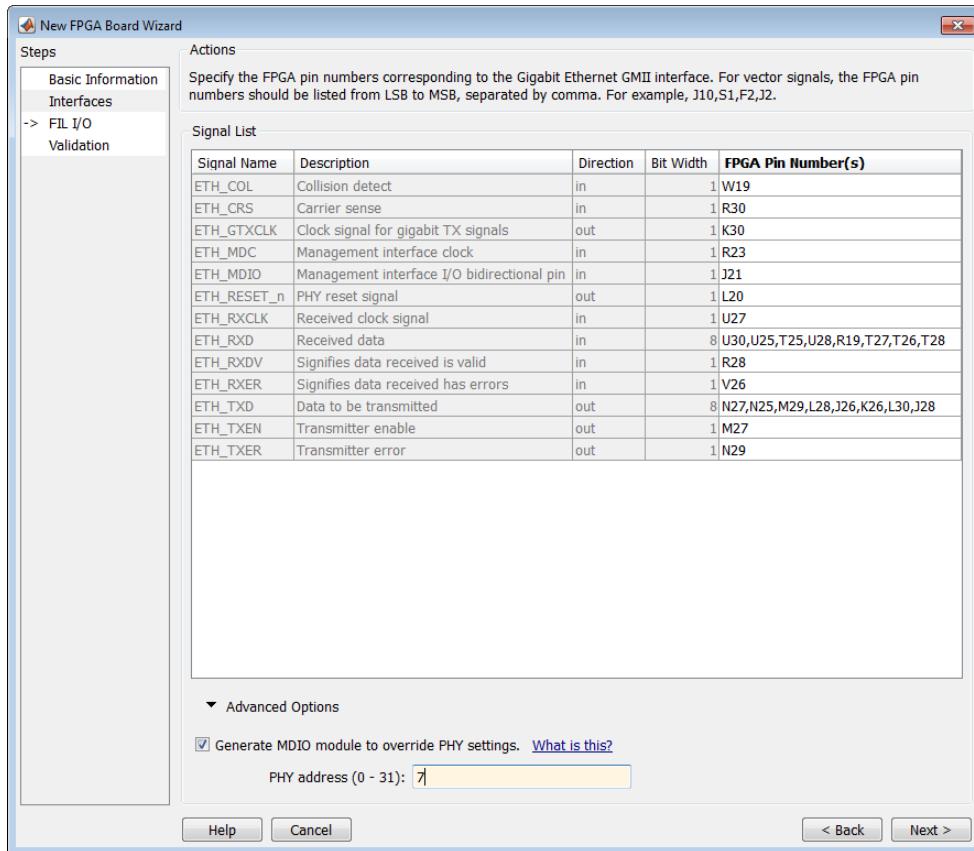
| For signal name... | Enter FPGA pin number...        |
|--------------------|---------------------------------|
| ETH_RXCLK          | U27                             |
| ETH_RXD            | U30,U25,T25,U28,R19,T27,T26,T28 |
| ETH_RXDV           | R28                             |
| ETH_RXER           | V26                             |
| ETH_TXD            | N27,N25,M29,L28,J26,K26,L30,J28 |
| ETH_TXEN           | M27                             |
| ETH_TXER           | N29                             |

- 2 Click Advanced Options to expand the section.
- 3 Check the **Generate MDIO module to override PHY settings** option.

This option is selected for the following reasons:

- There are jumpers on the Xilinx KC705 board that configure the Ethernet PHY device to MII, GMII, RGMII, or SGMII mode. Since this example uses the GMII interfaces, the FPGA board does not work if the PHY devices are set to the wrong mode. When the **Generate MDIO module to override PHY settings** option is selected, the FPGA uses the Management Data Input/Output (MDIO) bus to override the jumper settings and configure the PHY chip to the correct GMII mode.
- This option currently only applies to Marvell Alaska PHY device 88E1111 and this KC705 board is using the Marvel device.

- 4 **PHY address (0 - 31):** Enter 7.



5 Click **Next**.

## Run Optional Validation Tests

This step provides a validation test for you to verify if the entered information is correct by performing FPGA-in-the-loop cosimulation. You need Xilinx ISE 13.4 or higher versions installed on the same computer. This step is optional and you can skip it, if you prefer.

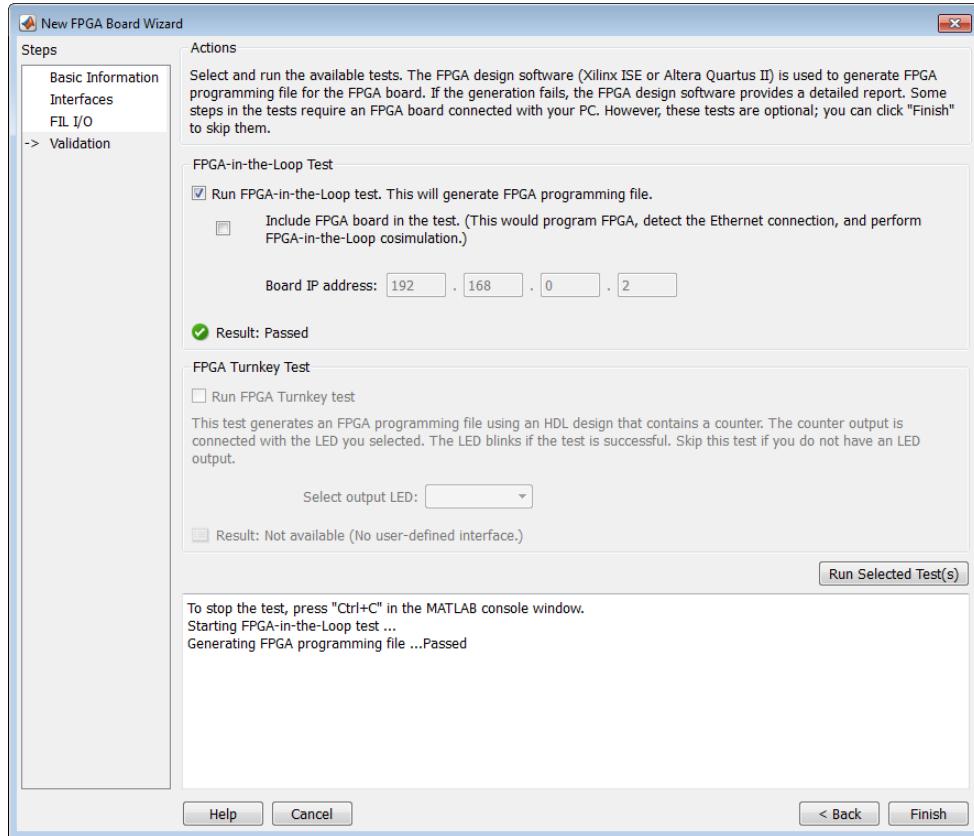
---

**Note** For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

---

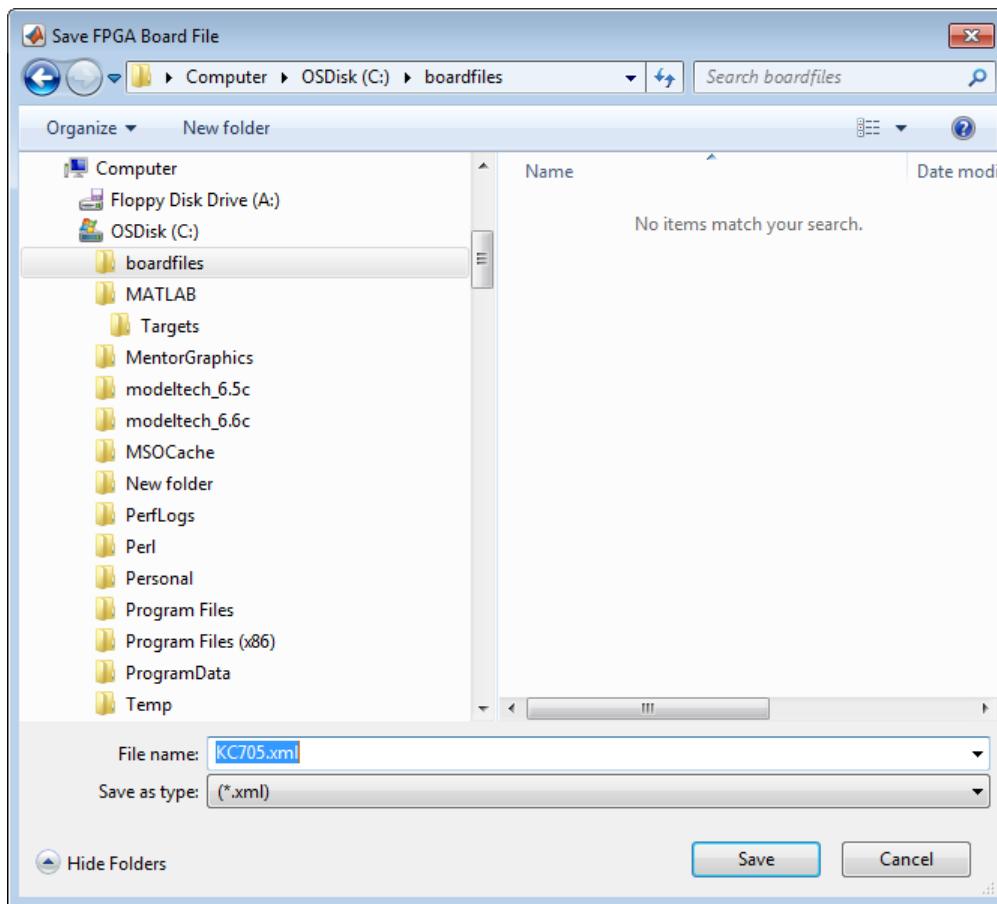
To run this test, perform the following actions.

- 1 Check the **Run FPGA-in-the-Loop test** option.
- 2 If you have the board attached, check the **Include FPGA board in the test** option. You need to supply the IP address of the FPGA Board. This example assumes that the Xilinx KC705 board is attached to your host computer and it has an IP address of 192.168.0.2.
- 3 Click **Run Selected Test(s)**. The tests take about 10 minutes to complete.



## Save Board Definition File

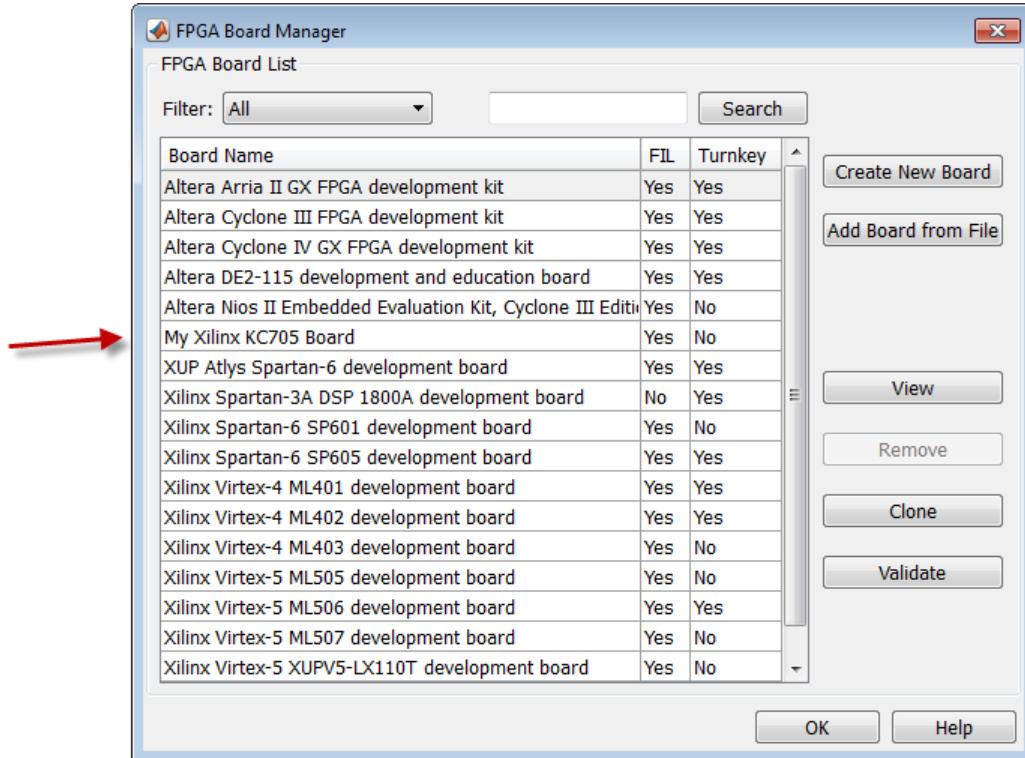
- 1 Click **Finish** to exit the New FPGA Board wizard. A **Save As** dialog box pops up and asks for the location of the FPGA board definition file. For this example, save as C:\boardfiles\KC705.xml.



- 2** Click **Save** to save the file and exit.

## Use New FPGA Board

- 1** After you save the board definition file, you are returned to the FPGA Board Manager. In the FPGA Board List, you can now see the new board you defined.

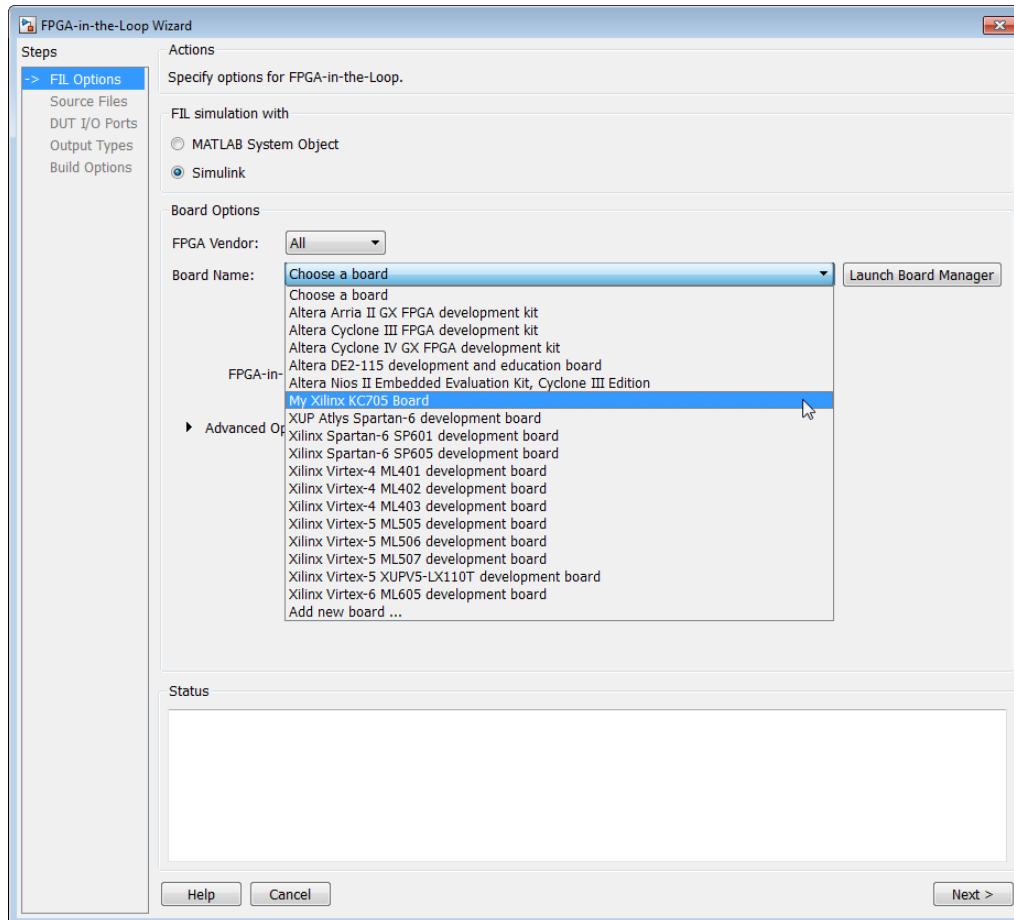


Click **OK** to close the FPGA Board Manager.

- 2 You can view the new board in the board list from either the FIL wizard or the HDL Workflow Advisor.
  - a Start the FIL wizard from the MATLAB prompt.

```
>>filWizard
```

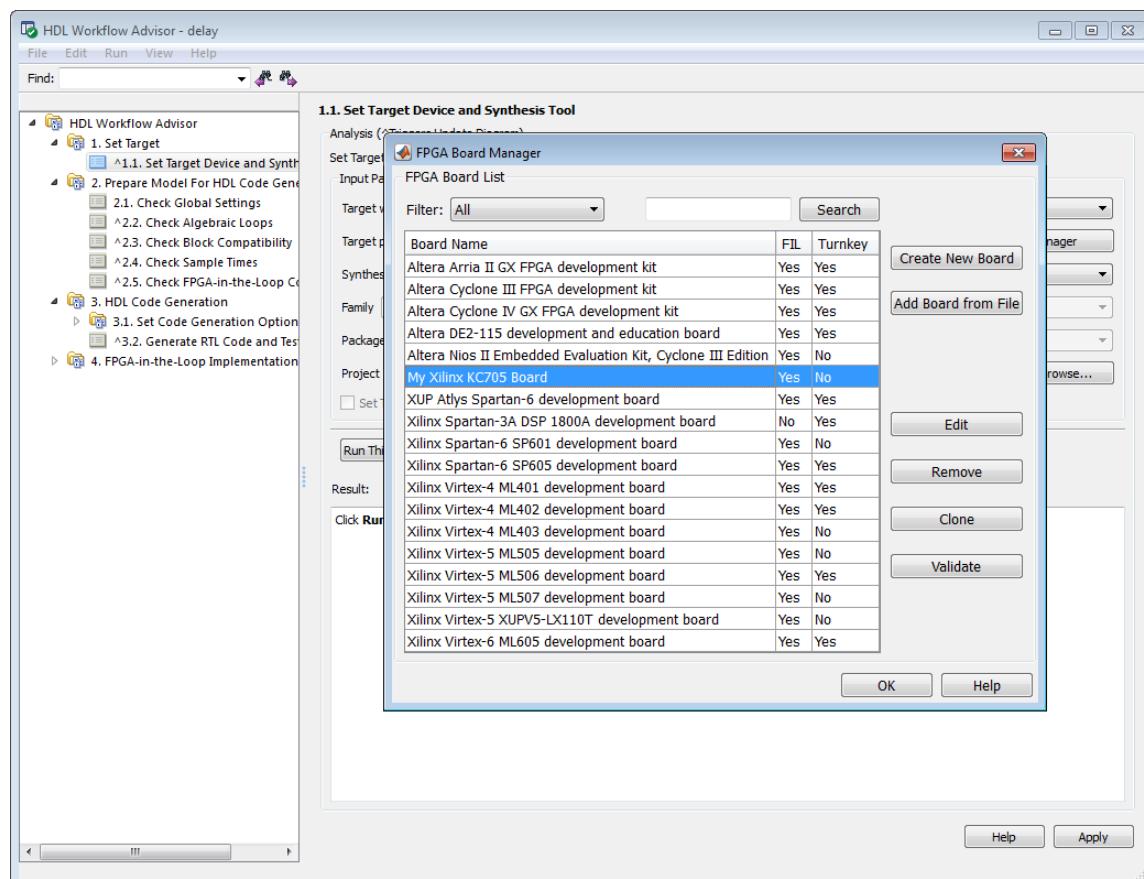
The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



**b** Start HDL Workflow Advisor.

In step 1.1, select **FPGA-in-the-Loop** and click **Launch Board Manager**.

The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



## FPGA Board Manager

### In this section...

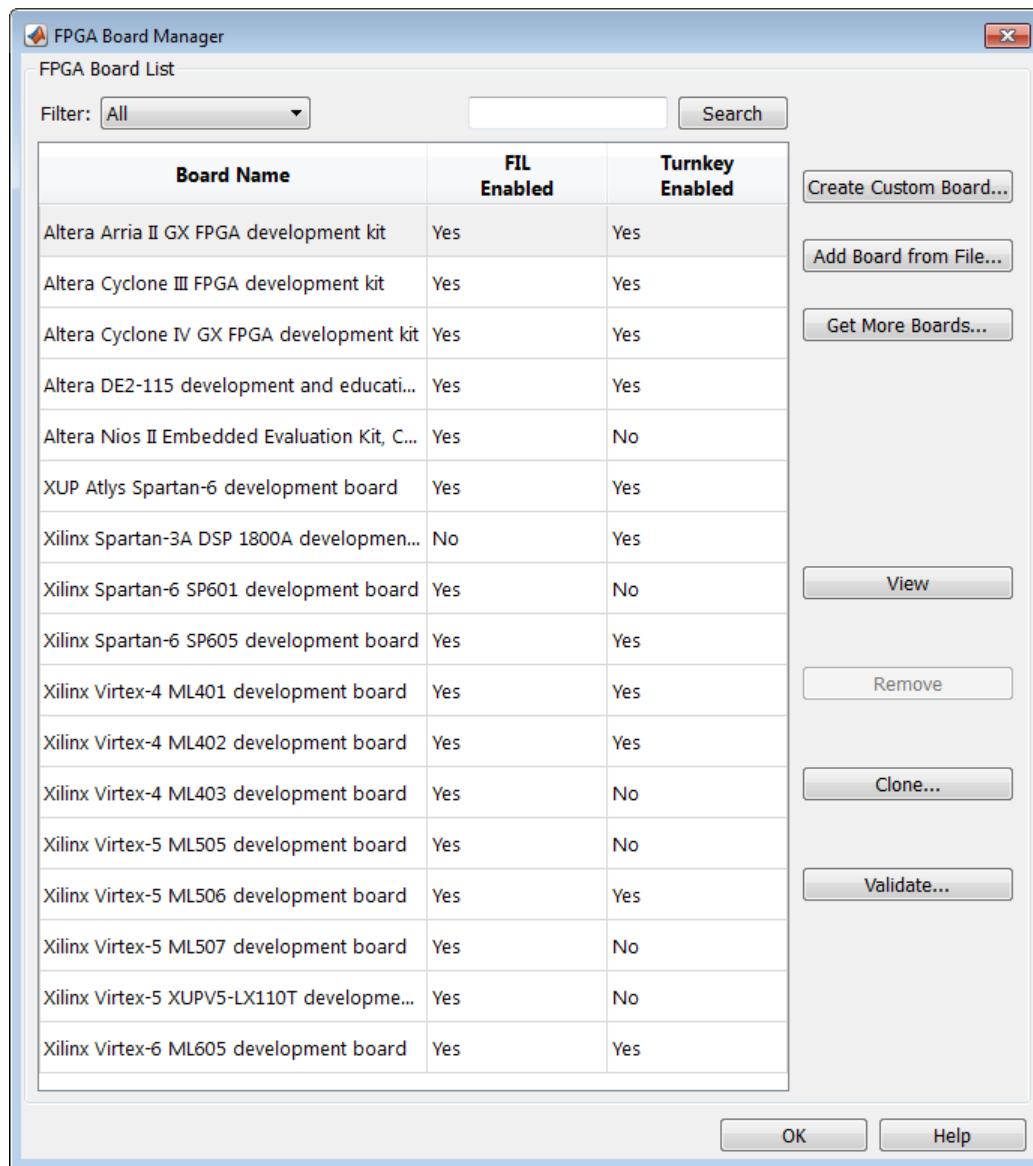
- “Introduction” on page 36-18
- “Filter” on page 36-19
- “Search” on page 36-19
- “FIL Enabled/Turnkey Enabled” on page 36-20
- “Create Custom Board” on page 36-20
- “Add Board from File” on page 36-20
- “Get More Boards” on page 36-20
- “View/Edit” on page 36-20
- “Remove” on page 36-20
- “Clone” on page 36-20
- “Validate” on page 36-20

### Introduction

The FPGA Board Manager is the portal to managing custom FPGA boards. You can create a board definition file or edit an existing one. You can even import a custom board from an existing board definition file.

You start the FPGA Board Manager by one of the following methods:

- By typing `fpgaBoardManager` in the MATLAB command window
- From the FIL wizard by clicking **Launch Board Manager** on the first page
- From the HDL Workflow Advisor (when using HDL Coder) at Step 1.1



## Filter

Choose one of the following views:

- All boards
- Only those boards that were preinstalled with HDL Verifier or HDL Coder
- Only custom boards

## Search

Find a specific board in the list or those boards that fully or partially match your search string.

## FIL Enabled/Turnkey Enabled

These columns indicate whether the specified board is supported for FIL or Turnkey operations.

## Create Custom Board

Start New FPGA Board wizard. See “New FPGA Board Wizard” on page 36-21. You can find the process for creating a board definition file in “Create Custom FPGA Board Definition” on page 36-6.

## Add Board from File

Import a board definition file (.xml).

## Get More Boards

Download FPGA board support packages for use with FIL

- 1 Click **Get more boards**.
- 2 Follow the prompts in the Support Package Installer to download an FPGA board support package.
- 3 When the download is complete, you can see the new boards in the board list in the FPGA Board Manager.

---

**Offline Support Package Installation** You can install an FPGA board support package without an internet connection. See “Install Support Package Offline” (HDL Verifier).

---

## View/Edit

View board configurations and modify the information. You can view a read-only file but not edit it. See “FPGA Board Editor” on page 36-32.

## Remove

Remove custom board from the list. This action does not delete the board definition XML file.

## Clone

Makes a copy of an existing custom board for further modification.

## Validate

Runs the validation tests for FIL See “Run Optional Validation Tests” on page 36-12.

## New FPGA Board Wizard

Using the New FPGA Board wizard, you can enter all the required information to add a board to the FPGA board list. This list applies to both FIL and Turnkey workflows. Review “FPGA Board Requirements” on page 36-2 before adding an FPGA board to make sure that it is compatible with the workflow for which you want to use it.

Several buttons in the New FPGA Board wizard help with navigation:

- **Back:** Go to a previous page to review or edit data already entered.
- **Next:** Go to next page when all requirements of current page have been satisfied.
- **Help:** Open Doc Center, and display this topic.
- **Cancel:** Exit New FPGA Board wizard. You can exit with or without saving the information from your session.

---

**Adding Boards Once for Multiple Users** To add new boards globally, follow these instructions. To access a board added globally, all users must be using the same MATLAB installation.

- 1 Create the following folder:

*matlabroot/toolbox/shared/eda/board/boardfiles*

- 2 Copy the board description XML file to the **boardfiles** folder.
- 3 After copying the XML file, restart MATLAB. The new board appears in the FPGA board list for either or both the FIL and Turnkey workflows.

All boards under this folder show-up in the FPGA board list automatically for users with the same MATLAB installation. You do not need to use FPGA Board Manager to add these boards again.

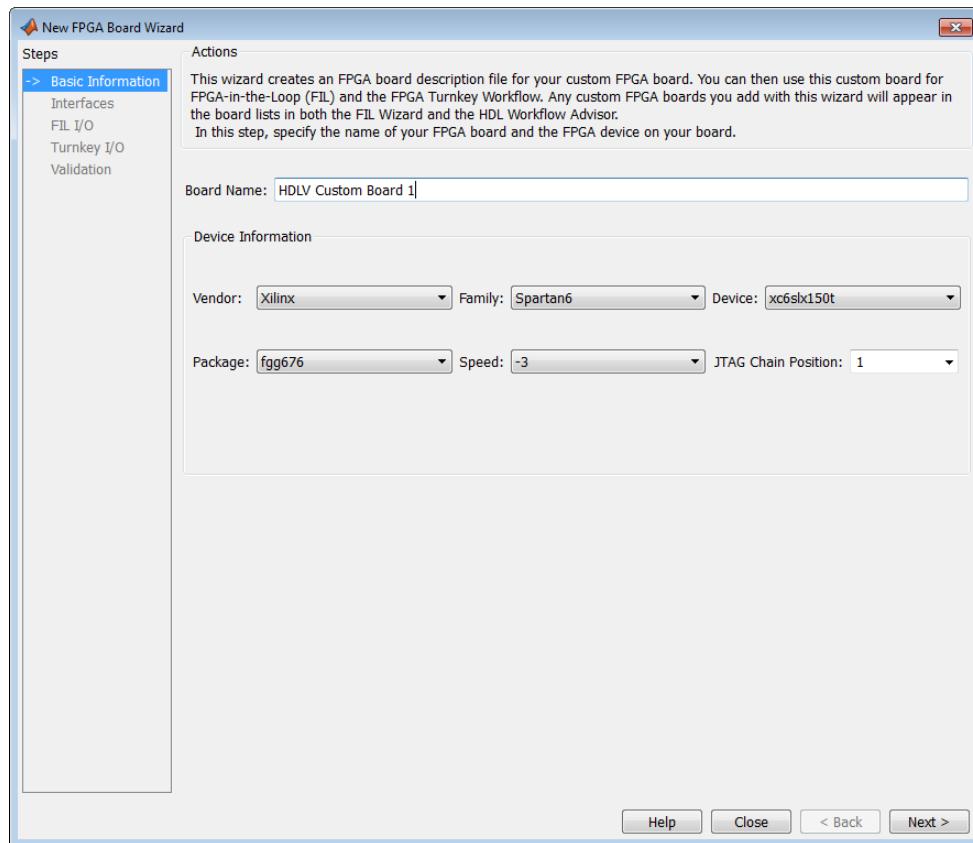
---

The workflow for adding an FPGA board contains these steps:

|                           |
|---------------------------|
| <b>In this section...</b> |
|---------------------------|

- |                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>“Basic Information” on page 36-22</li> <li>“Interfaces” on page 36-22</li> <li>“FIL I/O” on page 36-25</li> <li>“Turnkey I/O” on page 36-27</li> <li>“Validation” on page 36-30</li> <li>“Finish” on page 36-31</li> </ul> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Basic Information



**Board Name:** Enter a unique board name.

### Device Information:

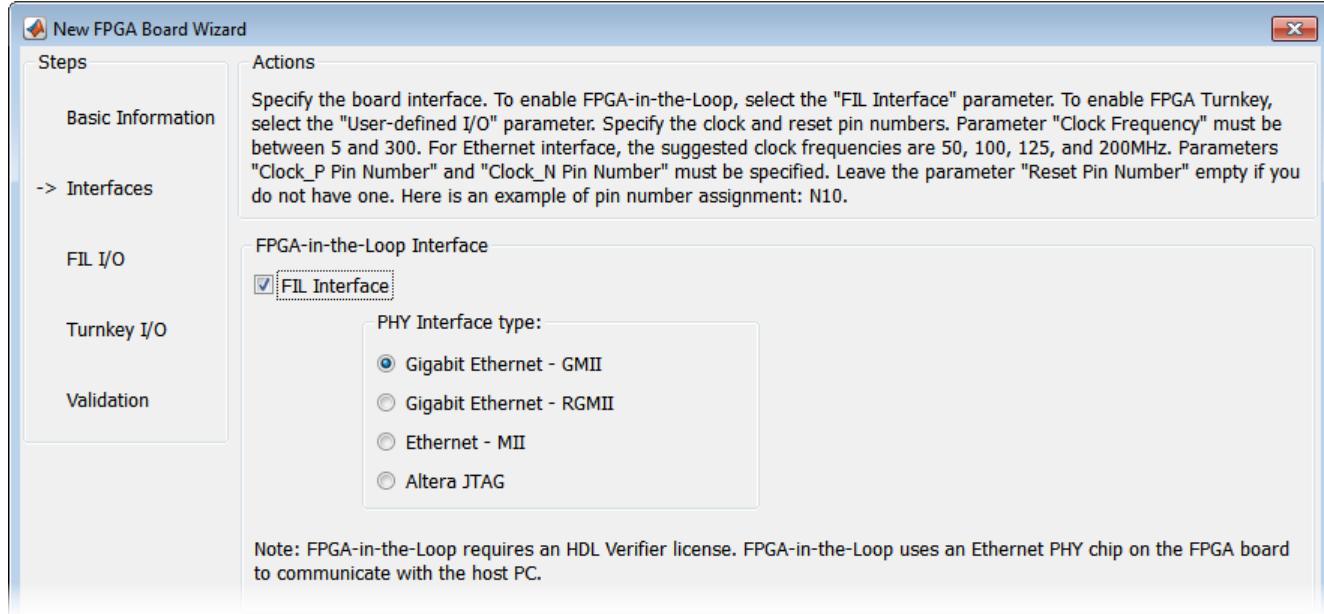
- **Vendor:** Xilinx or Altera
- **Family:** Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device:** Use the board specification file to select the correct device.
- For Xilinx boards only:
  - **Package:** Use the board specification file to select the correct package.
  - **Speed:** Use the board specification file to select the correct speed.
  - **JTAG Chain Position:** Value indicates the starting position for JTAG chain. Consult the board specification file for this information.

## Interfaces

- “FIL Interface for Altera Boards” on page 36-23
- “FIL Interface for Xilinx Boards” on page 36-24
- “FPGA Turnkey Interface” on page 36-24

- “FPGA Input Clock and Reset” on page 36-25

## FIL Interface for Altera Boards

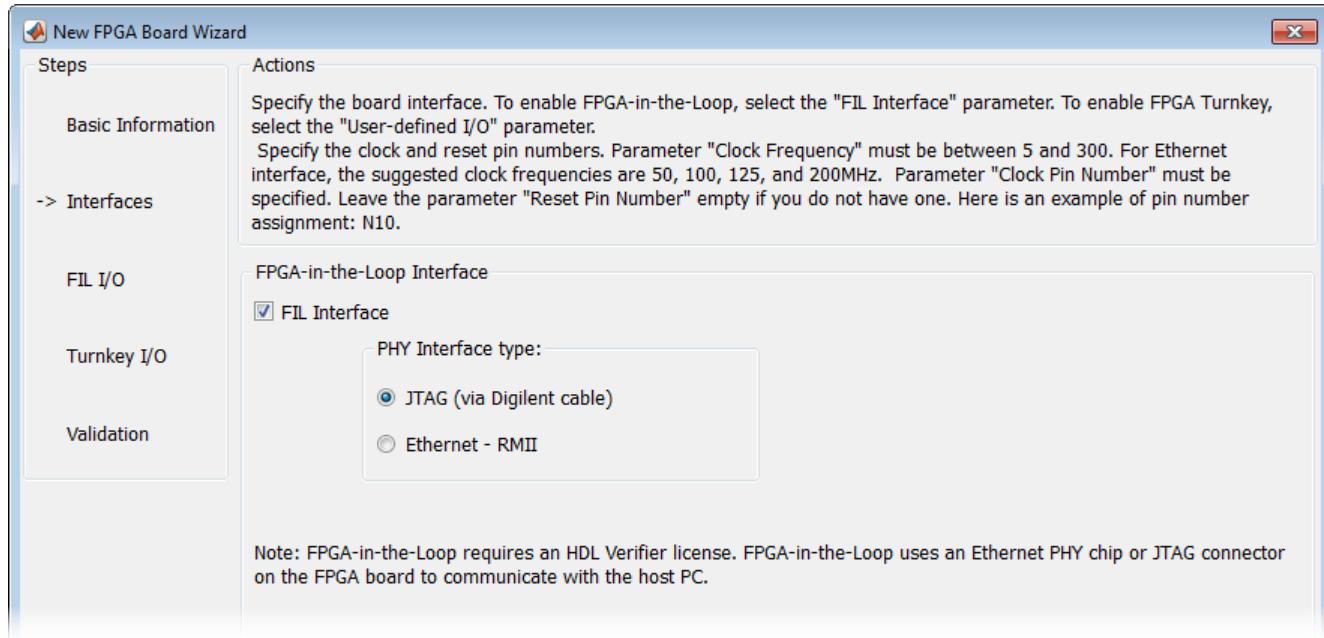


- FPGA-in-the-Loop:** To use this board with FIL, select **FIL Interface**.
- Select one of the following **PHY Interface types**:
  - Gigabit Ethernet — GMII**
  - Gigabit Ethernet — RGMII**
  - Gigabit Ethernet — SGMII** (the SGMII option appears if you select a board from the Stratix V or Stratix IV device families)
  - Ethernet — MII**
  - Altera JTAG** (Altera boards only)

---

**Note** Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

## FIL Interface for Xilinx Boards



Note: FPGA-in-the-Loop requires an HDL Verifier license. FPGA-in-the-Loop uses an Ethernet PHY chip or JTAG connector on the FPGA board to communicate with the host PC.

**1** **FPGA-in-the-Loop Interface:** To use this board with FIL, select **FIL Interface**.

**2** Select one of the following **PHY Interface types**:

- **JTAG (via Digilent cable)** (Xilinx boards only)
- **Ethernet — RMII**

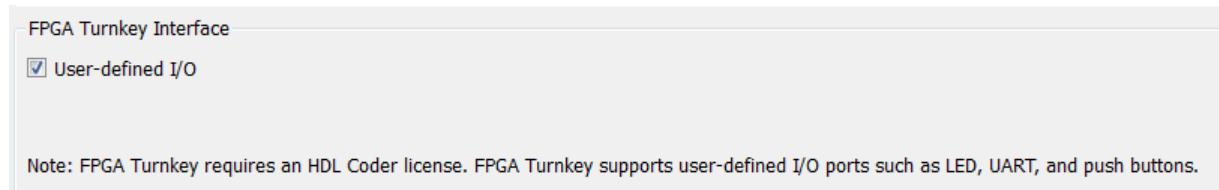
**Note** Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

For more information on how to set up the JTAG connection for Xilinx boards, see “JTAG with Digilent Cable Setup” on page 36-34.

### Limitations

When you simulate your FPGA design through a Digilent JTAG cable, you cannot use any other debugging feature that requires access to the JTAG; for example, the Vivado Logic Analyzer.

### FPGA Turnkey Interface



**FPGA Turnkey Interface:** If you want to use with board with the HDL Coder FPGA Turnkey workflow, select **User-defined I/O**.

## FPGA Input Clock and Reset

**FPGA Input Clock**

Clock Frequency: 200 MHz      Clock Type: Differential

Clock\_P Pin Number: E19      Clock\_N Pin Number: E18

Clock IO Standard: LVDS

**Reset (Optional)**

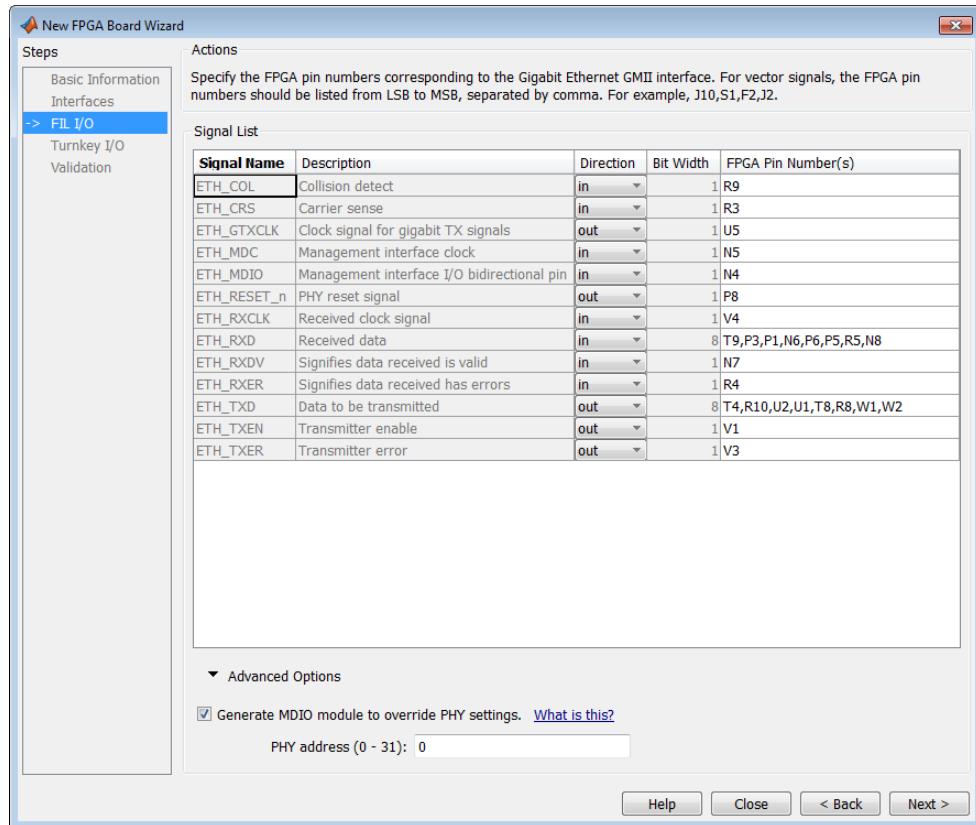
Reset Pin Number: AV40      Active Level: Active-High

Reset IO Standard: LVCMOS18

- 1 **FPGA Input Clock** — Clock details are required for both workflows. You can find all necessary information in the board specification file.
  - **Clock Frequency** — Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
  - **Clock Type** — Single\_Ended or Differential.
  - **Clock Pin Number** (Single\_Ended) — Must be specified. Example: N10.
  - **Clock\_P Pin Number** (Differential) — Must be specified. Example: E19.
  - **Clock\_N Pin Number** (Differential) — Must be specified. Example: E18.
  - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVDS.
- 2 **Reset (Optional)** — If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.
  - **Reset Pin Number** — Leave empty if you do not have one.
  - **Active Level** — Active-Low or Active-High.
  - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVCMOS33.

## FIL I/O

When you select an Ethernet connection to your board, you must specify pins for the Ethernet signals on the FPGA.



**Signal List:** Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas.

**Note** If your PHY chip does not have the optional TX\_ER pin, tie ETH\_TXER to one of the unused pins on the FPGA.

**Generate MDIO module to override PHY settings:** See the next section on FPGA Board Management Data Input/Output Bus (MDIO) to determine when to use this feature. If you do select this option, enter the PHY address.

### What Is the Management Data Input/Output Bus?

Management Data Input/Output (MDIO) is a serial bus, defined in the IEEE 802.3 standard, that connects MAC devices and Ethernet PHY devices. The FPGA MAC uses the MDIO bus to set control registers in the Ethernet PHY device on the board.

Currently only the Marvell 88E1111 PHY chip is supported by this MDIO module implementation. Do not select this check box if you are not using Marvell 88E1111.

The generated MDIO module is used to perform the following operations:

- **GMII mode:** The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.

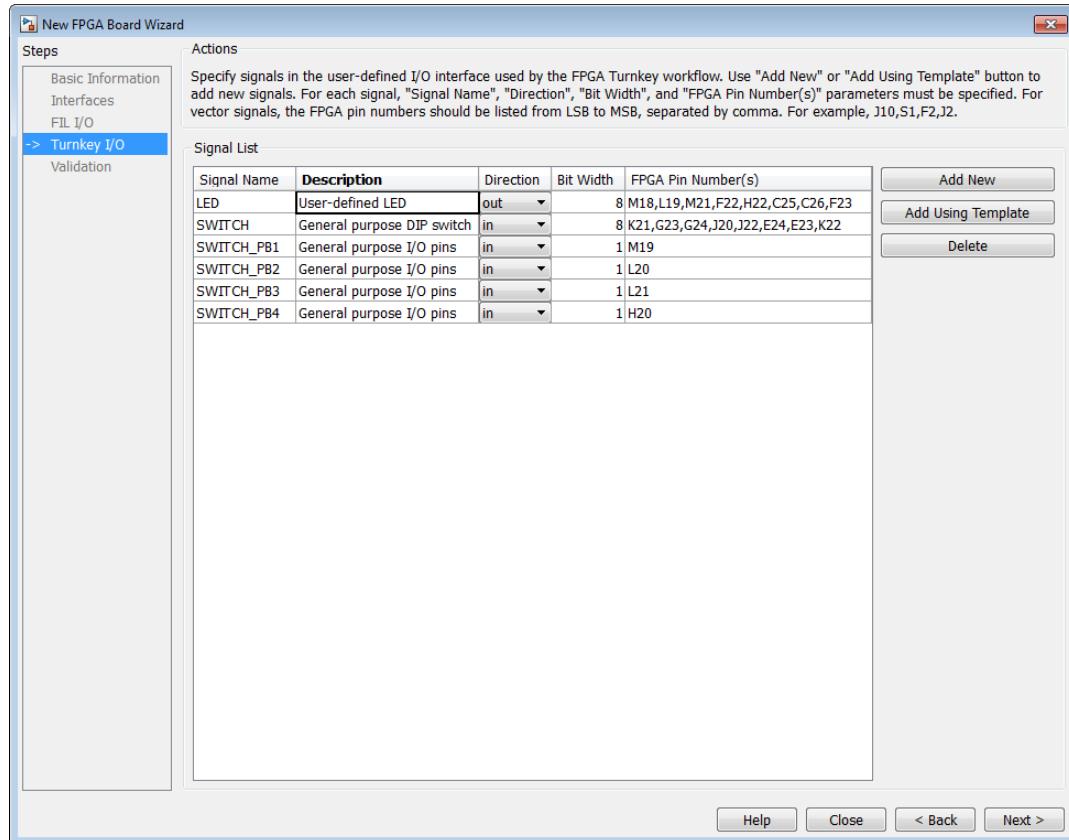
- **RGMII mode:** The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMII mode. In addition, the module sets the PHY chip to add internal delay for RX and TX clocks.
- **SGMII mode:** The PHY device can start up using other modes, such as RGMII/GMII. The generated MDIO module sets the PHY chip in SGMII mode.
- **MII mode:** The generated MDIO module sets the PHY device in GMII compatible mode. The module also sets the autonegotiation register to remove the 1000 Base-T capability advertisement. This reset ensures that the autonegotiation process does not select 1000 Mbits/s speed, which is not supported in MII mode.

**When To Select MDIO:** Select the **Generate MDIO module to override PHY settings** option when both the following conditions are met:

- The onboard Ethernet PHY device is Marvell 88E1111.
- The PHY device startup settings are not compatible with the FPGA MAC. The MDIO modules for different PHY modes must override these settings, as previously described.

**Specifying the PHY Address:** The PHY address is a 5-bit integer. The value is determined by the CONFIG[0] and CONFIG[1] pin on Marvell 88E1111 PHY device. See the board manual for this value.

## Turnkey I/O



**Note** Provide FIL I/O for an Ethernet connection only. Define at least one output port for the Turnkey I/O interface.

**Signal List:** Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas. The number of pin numbers must match the bit width of the corresponding signal.

**Add New:** You are prompted to enter all entries in the signal list manually.

**Add Using Template:** The wizard prepopulates a new signal entry for UART, LED, GPIO, or DIP Switch signals with the following:

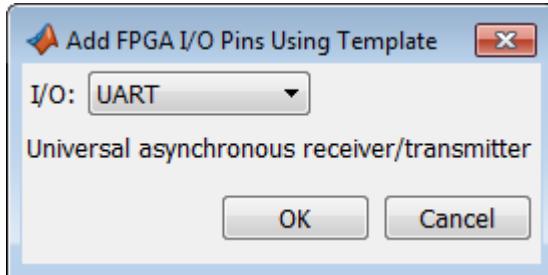
- A generic signal name
- Description
- Direction
- Bit width

You can change the values in any of these prepopulated fields.

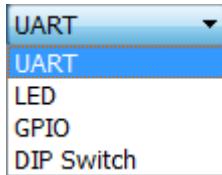
**Delete:** Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

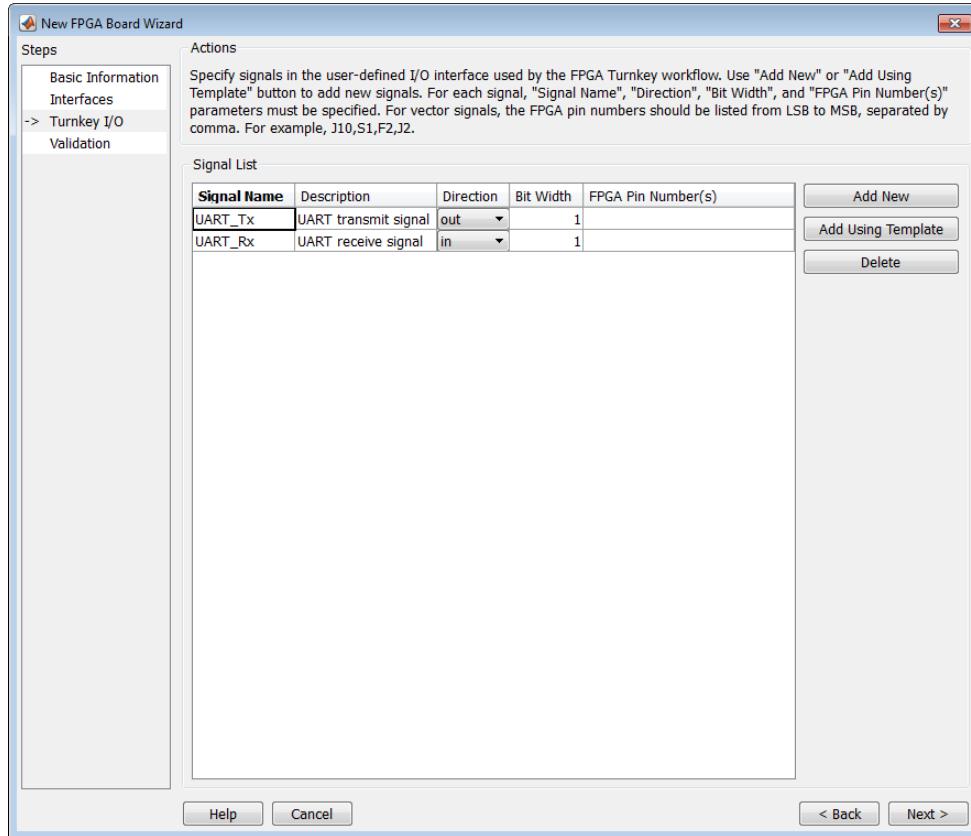
- 1 In the Turnkey I/O dialog box, click **Add Using Template**.
- 2 You can now view the template dialog box.



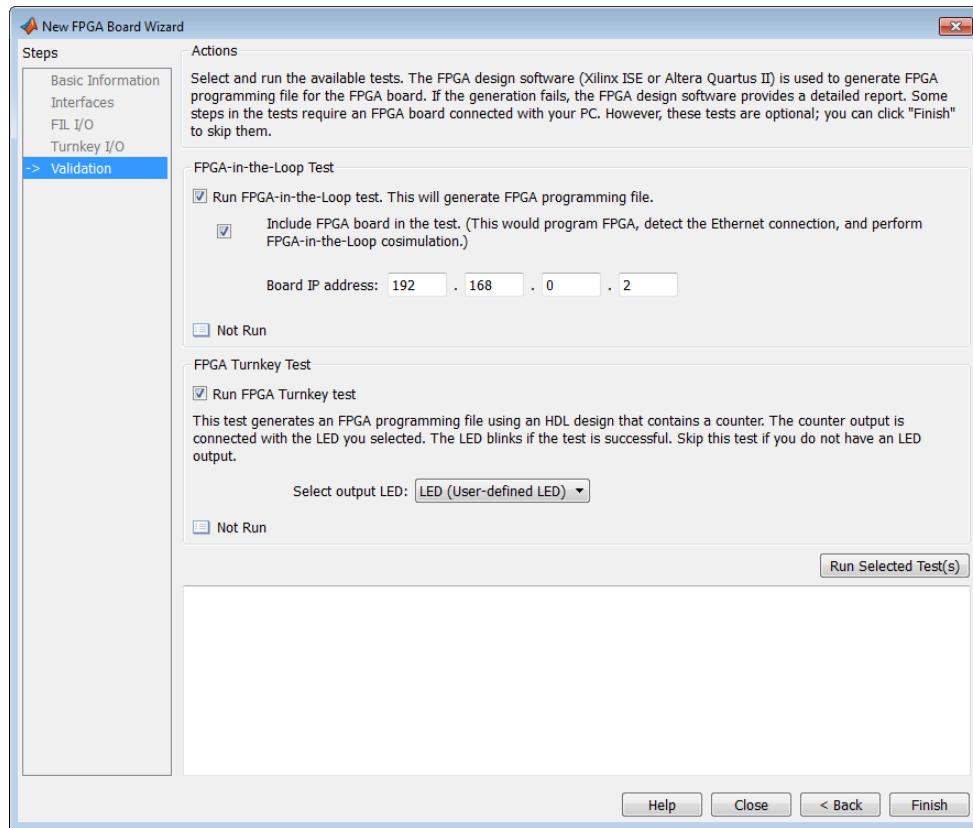
- 3 Pull down the I/O list and select from the following options:



- 4 Click **OK**.
- 5 The wizard adds the specified signal (or signals) to the I/O list.



## Validation



### FPGA-in-the-Loop Test

- **Run FPGA-in-the-Loop test:** Select to generate an FPGA programming file.
- **Include FPGA board in the test:** (Optional) This selection programs the FPGA with the generated programming file, detects the Ethernet connection (if selected), and performs FPGA-in-the-loop simulation.
- **Board IP address:** (Ethernet connection only) Use this option for setting the board IP address if it is not the default IP address (192.168.0.2).

If necessary, change the computer IP address to a different subnet from 192.168.0.x when you set up the network adapter. If the default board IP address 192.168.0.2 is in use by another device, change the Board IP address according to the following guidelines:

- The subnet address, typically the first 3 bytes of board IP address, must be the same as the host IP address.
- The last byte of the board IP address must be different from the host IP address.
- The board IP address must not conflict with the IP addresses of other computers.

For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

### FPGA Turnkey Test

- **Run FPGA Turnkey test:** Select to generate an FPGA programming file using an HDL design that contains a counter. You must have a board attached.
- **Select output LED:** The counter's output is connected with the LED you select. Skip this test if you do not have an LED output.

## Finish

When you have completed validation, click **Finish**. See “Save Board Definition File” on page 36-13.

## FPGA Board Editor

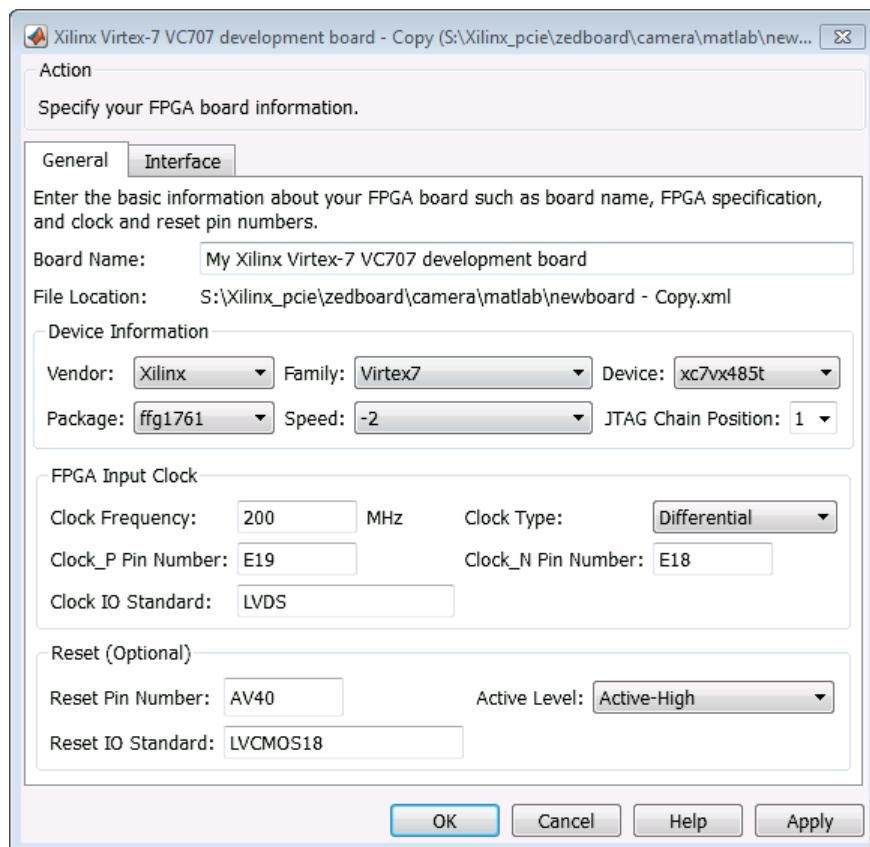
### In this section...

"General Tab" on page 36-32

"Interface Tab" on page 36-34

To edit a board definition XML file, first make it writeable. If the file is read-only, the FPGA Board Editor only lets you view the board configuration information. You cannot modify that information.

### General Tab



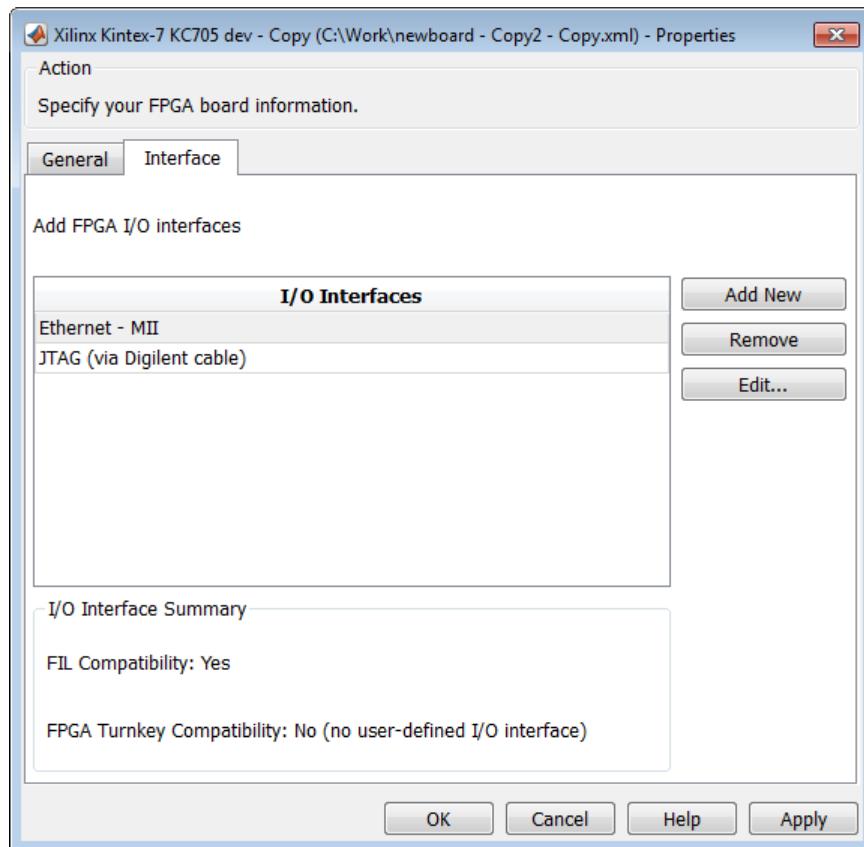
**Board Name:** Unique board name

#### Device Information:

- **Vendor:** Xilinx or Altera
- **Family:** Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device:** Device depends on the specified vendor and family. See the board specification file for applicable settings.
- For Xilinx boards only:

- **Package:** Package depends on specified vendor, family, and device. See the board specification file for applicable settings.
- **Speed:** Speed depends on package. See the board specification file for applicable settings.
- **JTAG Chain Position:** Value indicates the starting position for JTAG chain. Consult the board specification file for this information.
- **FPGA Input Clock.** Clock details are required for both the FIL and Turnkey workflows. You can find all necessary information in the board specification file.
  - **Clock Frequency.** Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
  - **Clock Type:** Single\_Ended or Differential.
  - **Clock Pin Number** (Single\_Ended) — Must be specified. Example: N10.
  - **Clock\_P Pin Number** (Differential) — Must be specified. Example: E19.
  - **Clock\_N Pin Number** (Differential) — Must be specified. Example: E18.
  - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVDS.
- **Reset (Optional).** If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.
  - **Reset Pin Number.** Leave empty if you do not have one.
  - **Active Level :** Active-Low or Active-High.
  - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, LVCMOS33.

## Interface Tab



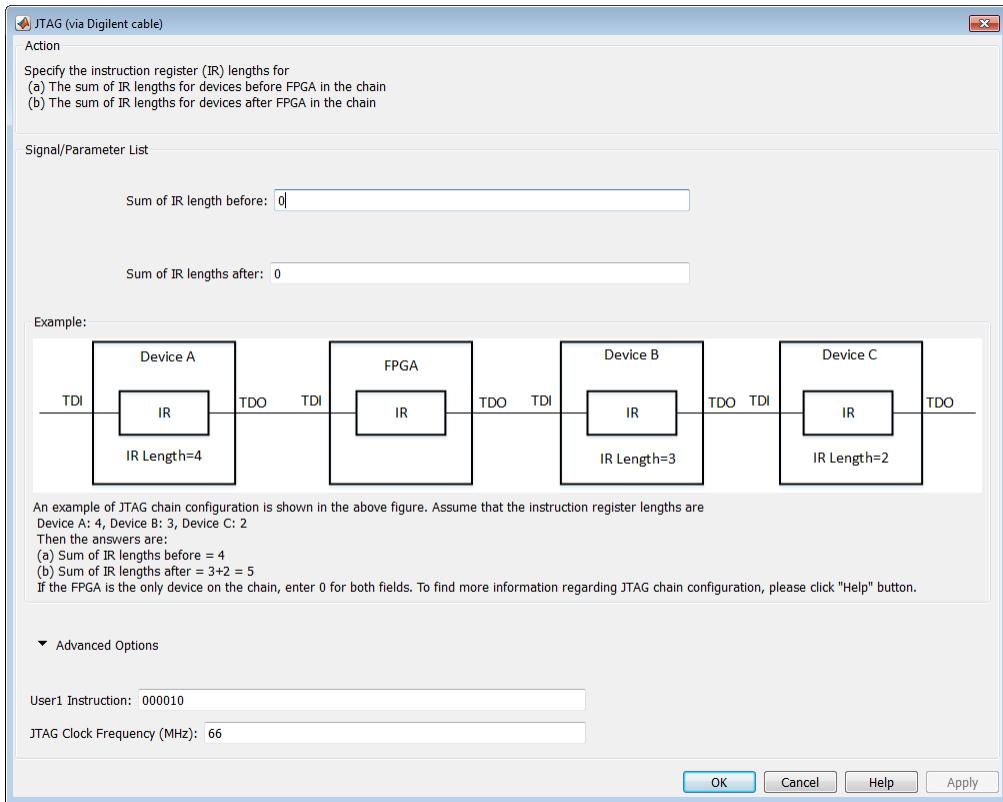
The Interface page describes the supported FPGA I/O Interfaces. Select any listed interface and click **View** to see the **Signal List**. If the board definition file has write permission, you can also **Add New** interface, **Edit** the interface, or **Remove** an interface.

### JTAG with Digilent Cable Setup

---

**Note** Enter information for the JTAG cable setup carefully. If the settings are incorrect, the simulation errors out and does not work. If you are still unsure about how to setup your JTAG cable after reading these instructions, contact MathWorks technical support with detailed information about your board.

---



- 1 Signal/Parameter List** — Provide the sum of the lengths of the instruction registers (IR) for all devices before and after the FPGA in the chain.
  - If the FPGA is the only item in the device chain, use zeros in both **Sum of IR length before** and **Sum of IR length after**.
  - If you are using a Zynq device, and it is the only item in the device chain, enter 4 in **Sum of IR length before** and 0 in **Sum of IR length after**.

If your board does not meet either of those conditions, follow these instructions to obtain the IR lengths:

- Connect the FPGA board to your computer using the JTAG cable. Turn on the board.
- Make sure that you installed the cable drivers during Vivado installation.
- Open Vivado Hardware Manager and select **Open a new hardware target**. In the dialog box is a summary of the IR lengths for all devices for that target.
- Sum the IR lengths before the FPGA and enter the total in **Sum of IR length before**. Sum the IR lengths after the FPGA and enter the total in **Sum of IR length after**.

Vivado Hardware Manager cannot recognize the IR length of less common devices. For these devices, consult the device manual for instruction register length.

- 2 Advanced Options** — If the default values are not the same as the most common settings for many devices, set the **User1 Instruction** and **JTAG Clock Frequency (MHz)** parameters. The most common settings are 000010 and 66, respectively.
  - User1 Instruction** — The JTAG USER1 Instruction defined in the Xilinx Bscane2 primitive. This binary instruction number, defined by Xilinx, varies from device to device. For most of the

7-series devices, this instruction is **000010**. If your device has a different value, enter it in this parameter.

To find this value, look at the **bsd** file for your specific device, found in your Vivado installation. For example, for the XA7A32T-CPG236 device, the **bsd** file is located in **Vivado \2014.2\data\parts\xilinx\artix7\artix7\xa7a32t\cpg236**.

Open this file. The **USER1** value is **000010**. Enter this value at **User1 Instruction**.

"USER1 (000010), "

- **JTAG Clock Frequency (MHz)** — Clock frequency used by the JTAG circuit. This value varies by device. You can find this value in the same **bsd** file described under **User1 Instruction**. For example, the JTAG clock frequency is 66 MHz for device XA7A32T-CPG236:

```
attribute TAP_SCAN_CLOCK of TCK : signal is (66.0e6, BOTH);
```

# HDL Workflow Advisor Tasks

---

## HDL Workflow Advisor Tasks

### In this section...

- "HDL Workflow Advisor Tasks Overview" on page 37-3
- "Set Target Overview" on page 37-4
- "Set Target Device and Synthesis Tool" on page 37-4
- "Set Target Reference Design" on page 37-5
- "Set Target Interface" on page 37-5
- "Set Target Frequency" on page 37-6
- "Set Target Interface" on page 37-6
- "Set Target Interface" on page 37-7
- "Prepare Model For HDL Code Generation Overview" on page 37-8
- "Check Global Settings" on page 37-9
- "Check Algebraic Loops" on page 37-9
- "Check Block Compatibility" on page 37-9
- "Check Sample Times" on page 37-10
- "Check FPGA-In-The-Loop Compatibility" on page 37-10
- "HDL Code Generation Overview" on page 37-10
- "Set Code Generation Options Overview" on page 37-11
- "Set Basic Options" on page 37-11
- "Set Report Options" on page 37-11
- "Set Advanced Options" on page 37-12
- "Set Optimization Options" on page 37-12
- "Set Testbench Options" on page 37-12
- "Generate RTL Code" on page 37-12
- "Generate RTL Code and Testbench" on page 37-12
- "Verify with HDL Cosimulation" on page 37-13
- "Generate RTL Code and IP Core" on page 37-13
- "FPGA Synthesis and Analysis Overview" on page 37-14
- "Create Project" on page 37-15
- "Perform Synthesis and P/R Overview" on page 37-15
- "Perform Logic Synthesis" on page 37-16
- "Perform Mapping" on page 37-16
- "Perform Place and Route" on page 37-16
- "Run Synthesis" on page 37-17
- "Run Implementation" on page 37-17
- "Annotate Model with Synthesis Result" on page 37-17
- "Download to Target Overview" on page 37-18
- "Generate Programming File" on page 37-19

### In this section...

- "Program Target Device" on page 37-19
- "Generate Simulink Real-Time Interface" on page 37-19
- "Save and Restore HDL Workflow Advisor State" on page 37-19
- "FPGA-In-The-Loop Implementation" on page 37-19
- "Set FPGA-In-The-Loop Options" on page 37-19
- "Build FPGA-In-The-Loop" on page 37-20
- "Check USRP Compatibility" on page 37-20
- "Generate FPGA Implementation" on page 37-20
- "Check SDR Compatibility" on page 37-20
- "SDR FPGA Implementation" on page 37-21
- "Set SDR Options" on page 37-21
- "Build SDR" on page 37-22
- "Embedded System Integration" on page 37-22
- "Create Project" on page 37-22
- "Generate Software Interface" on page 37-23
- "Build FPGA Bitstream" on page 37-23
- "Program Target Device" on page 37-23

## HDL Workflow Advisor Tasks Overview

The HDL Workflow Advisor is a tool that supports a suite of tasks covering the stages of the FPGA design process. Some tasks perform model validation or checking; others run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run.

For summary information on each HDL Workflow Advisor folder or task, select the folder or task icon and then click the HDL Workflow Advisor **Help** button.

- **Set Target:** The tasks in this category enable you to select the desired target device and map its I/O interface to the inputs and outputs of your model.
- **Prepare Model For HDL Code Generation:** The tasks in this category check your model for HDL code generation compatibility. The tasks also report on model settings, blocks, or other conditions (such as algebraic loops) that would impede code generation, and provide advice on how to fix such problems.
- **HDL Code Generation:** This category supports all HDL-related options of the Configuration Parameters dialog, including setting HDL code and test bench generation parameters, and generating code, test bench, or a cosimulation model.
- **FPGA Synthesis and Analysis:** The tasks in this category support:
  - Synthesis and timing analysis through integration with third-party synthesis tools
  - Back annotation of the model with critical path and other information obtained during synthesis
- **FPGA-in-the-Loop Implementation:** This category implements the phases of FIL, including providing block generation, synthesis, logical mapping, PAR (place-and-route), programming file

generation, and a communications channel. These capabilities are designed for a particular board and tailored to your RTL code. An HDL Verifier license is required for FIL.

- **Download to Target:** The tasks in this category depend on the selected target device and potentially include:
  - Generation of a target-specific FPGA programming file
  - Programming the target device
  - Generation of a model that contains a Simulink Real-Time interface subsystem

#### See Also

"Getting Started with the HDL Workflow Advisor" on page 31-6

## Set Target Overview

The tasks in the **Set Target** folder enable you to select a target FPGA device and define the interface generated for the device.

- **Set Target Device and Synthesis Tool:** Select a target FPGA device and synthesis tools.
- **Set Target Reference Design:** For IP Core Generation workflow, select a reference design for your target device.
- **Set Target Interface:** For IP Core Generation, FPGA Turnkey, and Simulink Real-Time FPGA I/O workflows, use the Target Platform Interface Table to assign each port on your DUT to an I/O resource on the target device.
- **Set Target Frequency:** Select the target clock rate for the FPGA implementation of your design.

For summary information on each **Set Target** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

#### See Also

"Getting Started with the HDL Workflow Advisor" on page 31-6

## Set Target Device and Synthesis Tool

The **Set Target Device and Synthesis Tool** task enables you to select an FPGA target device and an associated synthesis tool from a pulldown menu that lists the devices that HDL Workflow Advisor currently supports.

#### Description

This task displays the following options:

- **Target Workflow:** A pulldown menu that lists the possible workflows that HDL Workflow Advisor supports. Choose from:
  - Generic ASIC/FPGA
  - FPGA-in-the-loop
  - FPGA Turnkey
  - Simulink Real-Time FPGA I/O

- IP Core Generation
- Customization for the USRP device
- Software Defined Radio
- **Target platform:** A pulldown menu that lists the devices the HDL Workflow Advisor currently supports. Not available for the Generic ASIC/FPGA workflow.
- **Synthesis tool:** Select a synthesis tool, then select the **Family**, **Device**, **Package**, and **Speed** for your synthesis target.

If your synthesis tool is not one of the **Synthesis tool** options, see “[Synthesis Tool Path Setup](#)”. After you set up your synthesis tool path, click **Refresh** to make the tool available in the HDL Workflow Advisor.

- **Project folder:** Specify the project folder name.
- **Tool version:** This check box displays the current synthesis tool version.

---

**Note** If you select Intel Quartus Pro or Microsemi Libero SoC as the **Synthesis tool**, you can only run the Generic ASIC/FPGA workflow. When you use these tools, the **Annotate Model with Synthesis Result** task is not available. In this case, you can run the workflow to synthesis and then view the timing reports to see the critical path.

---

## Set Target Reference Design

The **Set Target Reference Design** task displays the reference design input parameters and the tool version. A **Reference design parameters** section displays any custom parameters that you specify for the reference design.

### Description

The task displays the following options:

- **Reference design:** A pulldown menu that lists the reference designs that HDL Coder supports and any custom reference designs that you specify. To learn more about creating a custom board and reference design, see “[Board and Reference Design Registration System](#)” on page 41-39.
- **Reference design tool version:** A text box that displays the current reference design tool version. It is recommended to use a reference design tool version that is compatible with the supported tool version. If there is a tool version mismatch, HDL Coder generates an error when you run this task. The tool version mismatch can potentially cause the **Create Project** task to fail.

If you select the **Ignore tool version mismatch** check box, HDL Coder generates a warning instead of an error. You can attempt to continue with creating the reference design project.

- **Reference design parameters:** Lists the parameters of the reference design. These can be parameters available with the default reference designs that HDL Coder supports, or parameters that you define for your custom reference design. For more information, see “[Define Custom Parameters and Callback Functions for Custom Reference Design](#)” on page 41-48.

## Set Target Interface

The **Set Target Interface** task displays properties of input and output ports on your DUT, and enables you to map these ports to I/O resources on the target device.

## Description

**Set Target Interface** displays the Target Platform Interface Table, which shows:

- The name, port type (input or output), and data type for each port on your DUT.
- A pulldown menu listing the available I/O resources for the target device.

These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

## Set Target Frequency

Specify the target frequency for these workflows:

- **Generic ASIC/FPGA:** To specify the target frequency that you want your design to achieve. HDL Coder generates a timing constraint file for that clock frequency and adds the constraint to the FPGA synthesis tool project that you create in the **Create Project** task. If the target frequency is not achievable, the synthesis tool generates an error. Target frequency is not supported with Microsemi Libero SoC.
- **IP Core Generation:** To specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- **Simulink Real-Time FPGA I/O:** For Speedgoat boards that are supported with Xilinx ISE, specify the target frequency to generate the clock module to produce the clock signal with that frequency.

The Speedgoat boards that are supported with Xilinx Vivado use the IP Core Generation workflow infrastructure. Specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.

- **FPGA Turnkey:** To generate the clock module to produce the clock signal with that frequency automatically.

## See Also

"Target Frequency Parameter" on page 14-8

## Set Target Interface

Select a processor-FPGA synchronization mode, and map your DUT input and output ports to I/O resources on the target device.

## Description

For **Processor/FPGA synchronization**, select:

- **Free running** if you do not want your processor and FPGA to be automatically synchronized.
- **Coprocessing - blocking** if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when FPGA

execution time is short relative to the processor sample time, and you want the FPGA to complete before the processor continues.

- **Coprocessing - nonblocking with delay** (not supported for IP Core Generation workflow) if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues.

This setting is saved with the model as the `ProcessorFPGASynchronization` HDL block property for the DUT block.

The Target Platform Interface Table shows:

- The name, port type (input or output), and data type for each port on your DUT.
- A pulldown menu listing the available I/O resources for the target device.

These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

## See Also

- “Processor and FPGA Synchronization” on page 40-23
- “Custom IP Core Generation” on page 40-10
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63

## Set Target Interface

Select a processor-FPGA synchronization mode, and map your DUT input and output ports to I/O resources on the target device. Optionally, specify a reference design.

### Description

**Reference design:** Select the predefined embedded system integration project into which HDL Coder inserts your generated IP core.

**Reference design path:** Enter the path to your downloaded reference design components. This field is available only if the specified **Reference design** requires downloadable components.

For **Processor/FPGA synchronization**, select:

- **Free running** if you do not want your processor and FPGA to be automatically synchronized.
- **Coprocessing - blocking** if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when FPGA execution time is short relative to the processor sample time, and you want the FPGA to complete before the processor continues.
- **Coprocessing - nonblocking with delay** (not supported for IP Core Generation workflow) if you want HDL Coder to generate synchronization logic for the FPGA automatically, so that the processor and FPGA run in tandem. Select this mode when the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues.

This setting is saved with the model as the **ProcessorFPGASynchronization** HDL block property for the DUT block.

The Target Platform Interface Table shows:

- The name, port type (input or output), and data type for each port on your DUT.
- A dropdown menu listing the available I/O resources for the target device.

These resources are device-specific. For detailed information on each resource, see the documentation for your FPGA development board.

### See Also

- “Processor and FPGA Synchronization” on page 40-23
- “Custom IP Core Generation” on page 40-10
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63

## Prepare Model For HDL Code Generation Overview

The tasks in the **Prepare Model For HDL Code Generation** folder check the model for compatibility with HDL code generation. If a check encounters a condition that would raise a code generation warning or error, the right pane of the HDL Workflow Advisor displays information about the condition and how to fix it. The **Prepare Model For HDL Code Generation** folder contains the following checks:

- **Check Global Settings:** Check model parameters for compatibility with HDL code generation.
- **Check Algebraic Loops:** Check the model for algebraic loops.
- **Check Block Compatibility:** Check that blocks in the model support HDL code generation.
- **Check Sample Times:** Check the solver options, tasking mode, and rate transition diagnostic settings, given the model's sample times.
- **Check FPGA-in-the-Loop Compatibility:** Check model compatibility with FPGA-in-the-loop, specifically:
  - Not allowed: sink/source subsystems, single/double data types, zero sample time
  - Must be present: HDL Verifier license

This option is available only if you select **FPGA-in-the-Loop** for Target workflow.

- **Check USRP Compatibility:** The model must have two input ports and two output ports of signed 16-bit signals.

This option is available only if you select **Customization for the USRP Device** for Target workflow.

For summary information on each **Prepare Model For HDL Code Generation** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

### See Also

- “Getting Started with the HDL Workflow Advisor” on page 31-6

## Check Global Settings

**Check Global Settings** checks model-wide parameter settings for HDL code generation compatibility.

### Description

This check examines the model parameters for compatibility with HDL code generation and flags conditions that would raise an error or a warning during code generation. The HDL Workflow Advisor displays a table with the following information about each condition detected:

- *Block*: Hyperlink to the model configuration dialog box page that contains the error or warning condition
- *Settings*: Name of the model parameter that caused the error or warning condition
- *Current*: Current value of the setting
- *Recommended*: Recommended value of the setting
- *Severity*: Severity level of the warning or error condition. Minimally, you should fix settings that are tagged as `error`.

### Tip

To set reported settings to their recommended values, click the **Modify All** button. You can then run the check again and proceed to the next check.

### See Also

“Model configuration checks” on page 39-12

## Check Algebraic Loops

Detect algebraic loops in the model.

### Description

The HDL Coder software does not support HDL code generation for models in which algebraic loop conditions exist. **Check Algebraic Loops** examines the model and fails the check if it detects an algebraic loop. Eliminate algebraic loops from your model before proceeding with further HDL Workflow Advisor checks or code generation.

### See Also

- “Algebraic Loop Concepts”
- “Check algebraic loops” on page 38-9

## Check Block Compatibility

Check the DUT for unsupported blocks.

## Description

**Check Block Compatibility** checks blocks within the DUT for compatibility with HDL code generation. The check fails if it encounters blocks that HDL Coder does not support. The HDL Workflow Advisor reports incompatible blocks, including the full path to each block.

## See Also

- “View HDL-Supported Blocks and HDL-Specific Block Documentation” on page 22-2
- “Check for unsupported blocks” on page 38-17
- “Check for unsupported blocks with Native Floating Point” on page 38-30

## Check Sample Times

Check the solver, sample times, and tasking mode settings for the model.

## Description

**Check Sample Times** checks the solver options, sample times, tasking mode, and rate transition diagnostics for HDL code generation compatibility. Solver options that the HDL Coder software requires or recommends are:

- **Type:** Fixed-step. (The coder currently supports variable-step solvers under limited conditions. See `hdlsetup` for details.)
- **Solver:** Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually the best one for simulating discrete systems.
- **Tasking mode:** SingleTasking. The coder does not currently support models that execute in multitasking mode. Do not set **Tasking mode** to Auto.
- **Multitask rate transition** and **Single task rate transition** diagnostic options: set to Error.

## Check FPGA-In-The-Loop Compatibility

HDL Verifier checks model for compatibility with FPGA-in-the-loop processing.

## See Also

“Prepare DUT For FIL Interface Generation” (HDL Verifier).

## HDL Code Generation Overview

The tasks in the **HDL Code Generation** folder enable you to:

- Set and validate HDL code and test bench generation parameters. Most parameters of the **HDL Code Generation** pane of the Configuration Parameters dialog box and the Model Explorer are supported.
- Generate any or all of:
  - RTL code
  - RTL test bench
  - Cosimulation model

- SystemVerilog DPI test bench

To run the tasks in the **HDL Code Generation** folder automatically, select the folder and click **Run All**.

---

**Tip** After each task in this folder runs, HDL Coder updates the Configuration Parameters dialog box and the Model Explorer.

---

## Set Code Generation Options Overview

The tasks in the **Set Code Generation Options** folder enable you to set and validate HDL code and test bench generation parameters. Each task of the **Set Code Generation Options** folder supports options of the **HDL Code Generation** pane of the Configuration Parameters dialog box and the Model Explorer. The tasks are:

- **Set Basic Options**: Set parameters that affect overall code generation.
- **Set Report Options**: Set parameters that affect the code generation report.
- **Set Advanced Options**: Set parameters that specify detailed characteristics of the generated code, such as HDL element naming and whether certain optimizations apply.
- **Set Optimization Options**: Set parameters that specify optimizations such as resource sharing and pipelining to improve area and timing.
- **Set Testbench Options**: Set options that determine characteristics of generated test bench code.

To run the tasks in the **Set Code Generation Options** folder automatically, select the folder and click **Run All**.

## Set Basic Options

Set parameters that affect overall code generation.

### Description

The **Set Basic Options** task sets options that are fundamental to HDL code generation. These options include selecting the DUT and selecting the target language. The basic options are the same as those found in the top-level **HDL Code Generation** pane of the Configuration Parameters dialog box, except that the **Code generation output** group is omitted.

### See Also

- “Target Language and Folder Selection Parameters” on page 13-3
- “Code Generation Output Parameter” on page 17-94

## Set Report Options

Set parameters that specify the sections that you want to see in the Code Generation Report.

The options are same as those found in the **HDL Code Generation > Report** pane of the Configuration Parameters dialog box and the Model Explorer.

## Set Advanced Options

Set parameters that specify detailed characteristics of the generated code.

### Description

The advanced options are the same as those found in the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box and the Model Explorer.

## Set Optimization Options

Set parameters that specify optimizations such as resource sharing and pipelining to improve area and timing.

### Description

The optimization options are the same as those found in the **HDL Code Generation > Target and Optimizations** pane of the Configuration Parameters dialog box and the Model Explorer.

## Set Testbench Options

Set options that determine characteristics of generated test bench code.

### Description

The test bench options are the same as those found in the **HDL Code Generation > Test Bench** pane of the Configuration Parameters dialog box and the Model Explorer.

## Generate RTL Code

Generate RTL code and HDL top-level wrapper.

### Description

The **Generate RTL Code** task generates RTL code and an HDL top-level wrapper for the DUT subsystem. It also generates a constraint file that contains pin mapping information and clock constraints.

## Generate RTL Code and Testbench

Select and initiate generation of RTL code, RTL test bench, and cosimulation model.

### Description

The **Generate RTL Code and Testbench** task enables choosing what type of code or model that you want to generate. You can select any combination of the following:

- **Generate RTL code:** Generate RTL code in the target language.
- **Generate test bench:** Generate the test bench(es) selected in **Set Testbench Options**.
- **Generate validation model:** Generate a validation model that highlights generated delays and other differences between your original model and the generated cosimulation model. With a validation model, you can observe the effects of streaming, resource sharing, and delay balancing.

The validation model contains the DUT from the original model and the DUT from the generated cosimulation model. Using the validation model, you can verify that the output of the optimized DUT is bit-true to the results produced by the original DUT.

## See Also

"Generating a Simulink Model for Cosimulation with an HDL Simulator" (Filter Design HDL Coder).

## Verify with HDL Cosimulation

Run this step to verify the generated HDL using cosimulation between the HDL Simulator and the Simulink test bench. This step shows only if you selected **Cosimulation model**, and specified an HDL simulator, in **Set Testbench Options**.

## Generate RTL Code and IP Core

Select and initiate generation of RTL code and custom IP core.

### Description

In the **Generate RTL Code and IP Core** task, specify characteristics of the generated IP core:

- **IP core name:** Enter the IP core name.

This setting is saved with the model as the **IPCoreName** HDL block property for the DUT block.

- **IP core version:** Enter the IP core version number. HDL Coder appends the version number to the IP core name to generate the output folder name.

This setting is saved with the model as the **IPCoreVersion** HDL block property for the DUT block.

- **IP core folder** (not editable): HDL Coder generates the IP core files in the output folder shown, including the HTML documentation.
- **IP repository:** If you have an IP repository folder, enter its path manually or by using the **Browse** button. The coder copies the generated IP core into the IP repository folder.
- **Additional source files:** If you are using a black box interface in your design to include existing Verilog or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (;), or by using the **Add** button. The source file language must match your target language.

This setting is saved with the model as the **IPCoreAdditionalFiles** HDL block property for the DUT block.

- **FPGA Data Capture buffer size:** The buffer size uses values that are  $128*2^n$ , where  $n$  is an integer. By default, the buffer size is 128 ( $n=0$ ). The maximum value of  $n$  is 13, which means that the maximum value for buffer size is 1048576 ( $=128*2^{13}$ ).

This setting is saved with the model as the **IPDataCaptureBufferSize** HDL block property for the DUT block.

- **Generate IP core report:** Leave this option selected to generate HTML documentation for the IP core.
- **Enable readback on AXI4 slave write registers:** Select this option if you want to read back the value that is written to the AXI4 slave registers by using the AXI4 slave interface. When you run

this task, the code generator adds a mux for each AXI4 register in the address decoder logic. This mux compares the address that the data is written to when reading the values. If you are reading from multiple AXI4 slave registers, the readback logic becomes a long mux chain that can affect synthesis frequency.

This setting is saved with the model as the `AXI4RegisterReadback` HDL block property for the DUT block.

- **Generate default AXI4 slave interface:** Leave this option selected if you want to generate an HDL IP core with the AXI4 slave interface for signals such as clock, reset, ready, timestamp, and so on. If you want to generate a generic HDL IP core without any AXI4 slave interfaces, clear this check box. In addition, make sure that you do not map any of the DUT ports to AXI4 or AXI4-Lite interfaces. You can only map the ports to External or Internal IO interfaces, or AXI4-Stream interface with TLAST mapping.

This setting is saved with the model as the `GenerateDefaultAXI4Slave` HDL block property for the DUT block.

#### See Also

- “Custom IP Core Generation” on page 40-10
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-19
- “Custom IP Core Report” on page 40-13

## FPGA Synthesis and Analysis Overview

Create projects for supported FPGA synthesis tools, perform FPGA synthesis, mapping, and place/route tasks, and annotate critical paths in the original model

#### Description

The tasks in the **FPGA Synthesis and Analysis** folder enable you to:

- Create FPGA synthesis projects for supported FPGA synthesis tools.
- Launch supported FPGA synthesis tools, using the project files to perform synthesis, mapping, and place/route tasks.
- Annotate your original model with critical path information obtained from the synthesis tools.

For a list of supported third-party synthesis tools, see “Third-Party Synthesis Tools and Version Support”.

The tasks in the folder are:

- **Create Project**
- **Perform Synthesis and P/R**
- **Annotate Model with Synthesis Result**

#### See Also

“HDL Code Generation and FPGA Synthesis from Simulink Model”

## Create Project

Create FPGA synthesis project for supported FPGA synthesis tool.

### Description

This task creates a synthesis project for the selected synthesis tool and loads the project with the HDL code generated for your model.

When the project creation completes, the HDL Workflow Advisor displays a link to the project in the right pane. Click this link to view the project in the synthesis tool project window.

### Synthesis objective

Select a synthesis objective to generate tool-specific optimization Tcl commands for your project. If you specify **None**, no Tcl commands are generated.

To learn how the synthesis objectives map to Tcl commands, see “Synthesis Objective to Tcl Command Mapping” on page 31-51.

### Additional source files

Enter additional HDL source files you want included in your synthesis project. Enter each file name manually, separated with a semicolon (;), or by using the **Add Source** button.

For example, you can include HDL source files (.vhd or .v) or a constraint file (.ucf or .sdc).

### Additional project creation Tcl files

Enter additional project creation Tcl files you want to include in your synthesis project. Enter each file name manually, separated with a semicolon (;), or by using the **Add Tcl** button.

For example, you can include a Tcl script (.tcl) to execute after creating the project.

### See Also

- “Third-Party Synthesis Tools and Version Support”
- “HDL Code Generation and FPGA Synthesis from Simulink Model”
- “Synthesis Objective to Tcl Command Mapping” on page 31-51

## Perform Synthesis and P/R Overview

Launch supported FPGA synthesis tools to perform synthesis, mapping, and place/route tasks.

### Description

The tasks in the **Perform Synthesis and P/R** folder enable you to launch supported FPGA synthesis tool and:

- Synthesize the generated HDL code.
- Perform mapping and timing analysis.
- Perform place and route functions.

For a list of supported third-party synthesis tools, see “Third-Party Synthesis Tools and Version Support”.

## See Also

"HDL Code Generation and FPGA Synthesis from Simulink Model"

## Perform Logic Synthesis

Launch supported FPGA synthesis tool and synthesize the generated HDL code.

### Description

The **Perform Logic Synthesis** task:

- Launches the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design, and emits netlists and related files.
- Displays a synthesis log in the **Result** subpane.

## See Also

"HDL Code Generation and FPGA Synthesis from Simulink Model"

## Perform Mapping

Launches supported FPGA synthesis tool and maps the synthesized logic design to the target FPGA.

### Description

The **Perform Mapping** task:

- Launches the synthesis tool in the background.
- Runs a mapping process that maps the synthesized logic design to the target FPGA.
- Emits a circuit description file for use in the place and route phase.
- Also emits pre-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

Enable **Skip pre-route timing analysis** if your tool does not support early timing estimation. When this option is enabled, the **Annotate Model with Synthesis Result** task sets **Critical path source** to **post-route**.

## See Also

"HDL Code Generation and FPGA Synthesis from Simulink Model"

## Perform Place and Route

Launches the synthesis tool in the background and runs a Place and Route process.

### Description

The **Perform Place and Route** task:

- Launches the synthesis tool in the background.
- Runs a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.
- Also emits post-routing timing information for use in critical path analysis and back annotation of your source model.
- Displays a log in the **Result** subpane.

### Tips

If you select **Skip this task**, the HDL Workflow Advisor executes the workflow, but omits the **Perform Place and Route** task, marking it Passed. You might want to select **Skip this task** if you prefer to do place and route work manually.

If **Perform Place and Route** fails, but you want to use the post-mapping timing results to find critical paths in your model, you can select **Ignore place and route errors** and continue to the **Annotate Model with Synthesis Result** task.

### See Also

"HDL Code Generation and FPGA Synthesis from Simulink Model"

## Run Synthesis

Launches Xilinx Vivado and executes the Vivado **Synthesis** step.

Enable **Skip pre-route timing analysis** if you do not want to do early timing estimation.

## Run Implementation

Launches Xilinx Vivado and executes the Vivado **Implementation** step.

If you select **Skip this task**, the HDL Workflow Advisor omits the **Run Implementation** task, marking it Passed. Select **Skip this task** if you prefer to do place and route work manually.

If **Run Implementation** fails, you can select **Ignore place and route errors** and continue to the **Annotate Model with Synthesis Result** task.

### Check Timing Report

If there are timing failures during this task, the task does not fail. You must check the timing report for timing failures.

## Annotate Model with Synthesis Result

Analyzes pre- or post-routing timing information and visually highlights critical paths in your model

### Description

The **Annotate Model with Synthesis Result** task helps you to identify critical paths in your model. At your option, the task analyzes pre- or post-routing timing information produced by the **Perform Synthesis and P/R** task group, and visually highlights one or more critical paths in your model.

**Note** If you select Intel Quartus Pro or Microsemi Libero SoC as the **Synthesis tool**, the **Annotate Model with Synthesis Result** task is not available. In this case, you can run the workflow to synthesis and then view the timing reports to see the critical path.

If **Generate FPGA top level wrapper** is selected in the **Generate RTL Code and Testbench** task, **Annotate Model with Synthesis Result** is not available. To perform back-annotation analysis, clear the check box for **Generate FPGA top level wrapper**.

### Input Parameters

#### Critical path source

Select **pre-route** or **post-route**.

The **pre-route** option is unavailable when **Skip pre-route timing analysis** is enabled in the previous task group.

#### Critical path number

You can annotate up to 3 critical paths. Select the number of paths you want to annotate.

#### Show all paths

Show critical paths, including duplicate paths.

#### Show unique paths

Show only the first instance of a path that is duplicated.

#### Show delay data

Annotate the cumulative timing delay on each path.

#### Show ends only

Show the endpoints of each path, but omit the connecting signal lines.

### Results and Recommended Actions

When the **Annotate Model with Synthesis Result** task runs to completion, HDL Coder displays the DUT with critical path information highlighted.

### See Also

"HDL Code Generation and FPGA Synthesis from Simulink Model"

## Download to Target Overview

The **Download to Target** folder supports the following tasks:

- **Generate Programming File**: Generate an FPGA programming file.
- **Program Target Device**: Download generated programming file to the target development board.
- **Generate Simulink Real-Time Interface** (for Speedgoat target devices only): Generate a model that contains a Simulink Real-Time interface subsystem.

For summary information on each **Download to Target** task, select the task icon and then click the HDL Workflow Advisor **Help** button.

## See Also

“Getting Started with the HDL Workflow Advisor” on page 31-6

## Generate Programming File

The **Generate Programming File** task generates an FPGA programming file that is compatible with the selected target device.

## Program Target Device

The **Program Target Device** task downloads the generated FPGA programming file to the selected target device.

Before executing the **Program Target Device** task, make sure that your host PC is properly connected to the target development board via the required programming cable.

## Generate Simulink Real-Time Interface

The **Generate Simulink Real-Time Interface** task generates a model containing an interface subsystem that you can plug in to a Simulink Real-Time model.

The naming convention for the generated model is:

`gm_fpgamodelname_slrt`

where `fpgamodelname` is the name of the original model.

## Save and Restore HDL Workflow Advisor State

You can save the current settings of the HDL Workflow Advisor to a named restore point. Later, you can restore the same settings by loading the restore point data into the HDL Workflow Advisor.

## See Also

“Getting Started with the HDL Workflow Advisor” on page 31-6.

## FPGA-In-The-Loop Implementation

Set FIL options and run FIL processing.

## Set FPGA-In-The-Loop Options

Set connection type, board IP, and MAC addresses and select additional files, if required.

### Connection

Select either JTAG (Altera boards only) or Ethernet.

### Board IP Address

Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2).

### Board MAC Address

Under most circumstances, you do not need to change the Board MAC address. You will need to do so if you connect more than one FPGA development board to a single computer (for which you must have a separate NIC for each board). You must change the Board MAC address for additional boards so that each address is unique.

### Additional Source Files

Select additional source files for the HDL design that is to be verified on the FPGA board, if required. HDL Workflow Advisor attempts to identify the file type; change the file type in the **File Type** column if it is incorrect.

## Build FPGA-In-The-Loop

During the build process, the following actions occur:

- FPGA-in-the-loop generates a FIL block named after the top-level module and places it in a new model.
- After new model generation, FIL opens a command window. In this window, the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation. When the process completes, a message in the command prompts you to close the window.
- FPGA-in-the-loop builds a testbench model around the generated FIL block.

### Check USRP Compatibility

The model must have two input ports and two output ports of signed 16-bit signals.

## Generate FPGA Implementation

This step initiates FPGA programming file creation. For Input Parameters, enter the path to the Ettus Research USRP™ FPGA files you previously downloaded. If you have not yet downloaded these files, see the Support Package for USRP Radio documentation.

When this step completes, see the instructions for downloading the programming file to the FPGA and running the simulation in the Support Package for USRP Radio documentation for FPGA Targeting.

### Check SDR Compatibility

The DUT must adhere to certain signal interface requirements. During Check SDR Compatibility, the following interface checks are performed (Inputs and Outputs go through the same checks).

- Must include single complex signal, two scalar signals, or single vectored signal of size 2
- Must have a bitwidth of 16
- Must be signed
- Must be single rate
- If have vectored ports must use Scalarize Vectors option

- If have multiple rates, must use Single clock
- Must use synchronous reset
- Must use active-high reset
- Must use a user overclocking factor of 1

All error checks are done for a given task run and reported in a table. This allows a single iteration to fix all errors.

## SDR FPGA Implementation

The SDR FPGA integrates customer logic as generated in previous steps as well as SDR-specific code to provide data and control paths between an RF board and the host.

This step consists of the following tasks:

- Set SDR Options: Choose customization options
- Build SDR: Generate FPGA programming file for an SDR target.

### Set SDR Options

Choose customization options for the completion of the SDR FPGA implementation.

#### SDR FPGA Component Options

- **RF board for target**

Choose one of the following:

- Epic Bitshark FMC-1Rx RevB
- Epic Bitshark FMC-1Rx RevC

- **Folder with vendor HDL source code**

Specify the folder that contains the RF interface HDL downloaded from the vendor support site. Use **Browse** to navigate to the correct folder.

- **User logic synthesis frequency**

Specify the maximum frequency at which you want to run your design. This value must be greater than the sampling frequencies for ADC and DAC as specified in the ADI FMCOMMS or Epiq Bitshark™ block.

- **User logic data path**

Select either the Receiver data path or the Transmitter data path.

#### Radio IP Addresses

- **Board IP address**

Set the board's IP address in this field if it is not the default IP address (192.168.10.1).

- **Board MAC address**

Under most circumstances, you do not need to change the Board MAC address. However, you need to do so if you connect more than one FPGA development board to a single computer (for

which you must have a separate NIC for each board). You must change the Board MAC address for additional boards so that each address is unique.

### **Additional Source and Project Files for the HDL Design**

Specify files you want included in the ISE or Vivado project. You should include only file types supported by ISE or Vivado. If an included file does not exist, the HDL Workflow Advisor cannot create the project.

- **File:** Name of file added to design (with **Add**).
- **File Type:** File type. The software will attempt to determine the file type automatically, but you may override the selection. Options are VHDL, Verilog, EDIF netlist, VQM netlist, QSF file, Constraints, and Others.
- **Add:** Add a new file to the list.
- **Remove:** Removes the currently selected file from the list.
- **Up:** Moves the currently selected file up the list.
- **Down:** Moves the currently selected file down the list.

**Show full paths to source files** (checkbox) triggers a full path display. Leaving this box unchecked displays only the file name.

### **Build SDR**

The HDL Workflow Advisor creates a new Xilinx ISE or Vivado project and adds the following:

- All the necessary files from the FPGA repository
- The generated HDL files for the selected subsystem and algorithm

If no errors are found during FPGA project generation and syntax checking, the FPGA programming file generation process starts. You can view this process in an external command shell and monitor its progress. When the process is finished, a message in the command window prompts you to close the window.

## **Embedded System Integration**

Tasks in this folder integrate your generated HDL IP core with the embedded processor.

### **Create Project**

Create project for embedded system tool.

In the message window, after the project is generated, you can click the project link to open the generated embedded system tool project.

#### **Embedded system tool**

Embedded design tool.

#### **Project folder**

Folder where your generated project files are saved.

### Synthesis objective

Select a synthesis objective to generate tool-specific optimization Tcl commands for your project. If you specify **None**, no Tcl commands are generated.

To learn how the synthesis objectives map to Tcl commands, see “[Synthesis Objective to Tcl Command Mapping](#)” on page 31-51.

### Generate Software Interface

Generate a software interface model or script or both with IP core driver blocks for embedded C code generation.

After you generate the software interface model, you can generate C code from it using Embedded Coder. The script contains commands that enable you to connect to the target hardware, and to write to or read from the generated IP core by using AXI driver blocks.

When you clear both the **Generate Simulink software interface model** and **Generate MATLAB software interface script** check boxes, this task is skipped.

**Operating system:** Select your target operating system.

### Build FPGA Bitstream

Generate bitstream for embedded system.

#### Run build process externally

Enable this option to run the build process in parallel with MATLAB. If this option is disabled, you cannot use MATLAB until the build is finished.

#### Tcl file for synthesis build

To customize your synthesis build, save your custom Tcl commands in a file and select **Custom**. Enter the file path manually or by using the **Browse** button. The contents of your custom Tcl file are inserted between the Tcl commands that open and close the project.

If you select **Custom** and want to generate a bitstream, the bitstream generation Tcl command must refer to the top file wrapper name and location either directly or implicitly. For example, the following Xilinx Vivado Tcl command generates a bitstream and implicitly refers to the top file name and location:

```
launch_runs impl_1 -to_step write_bitstream
```

### Program Target Device

Program the connected target SoC device. Specify the **Programming method** for the target device:

- **JTAG:** Uses a JTAG cable to program the target SoC device.
- **Download:** This is the default **Programming method**. Copies the generated FPGA bitstream, device tree, and system initialization scripts to the SD card on the Zynq board, and keeps the bitstream on the SD card persistently. To use this programming method, you do not require an Embedded Coder license. You can create an SSH object by specifying the **IP Address**, **SSH Username**, and **SSH Password**. HDL Coder uses the SSH object to copy the bitstream to the SD card and reprogram the board.

To define your own function to program the target device in your custom reference design, you can use the **Custom Programming method**. To use the custom programming, register the function handle of the custom programming function using the `CallbackCustomProgrammingMethod` method of the `hdlcoder.ReferenceDesign` class. For example:

```
hRD.CallbackCustomProgrammingMethod = ...
@parameter_callback.callback_CustomProgrammingMethod;
```

For more information, see “Program Target FPGA Boards or SoC Devices” on page 40-49.

# HDL Code Advisor

---

- “HDL Coder Checks in Model Advisor / HDL Code Advisor Overview” on page 38-3
- “Model configuration checks overview” on page 38-4
- “Check for model parameters suited for HDL code generation” on page 38-5
- “Check for global reset setting for Xilinx and Altera devices” on page 38-7
- “Check inline configurations setting” on page 38-8
- “Check algebraic loops” on page 38-9
- “Check for visualization settings” on page 38-10
- “Check delay balancing setting” on page 38-11
- “Check for ports and subsystems overview” on page 38-12
- “Check for invalid top level subsystem” on page 38-13
- “Check initial conditions of enabled and triggered subsystems” on page 38-14
- “Check for blocks and block settings overview” on page 38-15
- “Check for infinite and continuous sample time sources” on page 38-16
- “Check for unsupported blocks” on page 38-17
- “Check for large matrix operations” on page 38-18
- “Check for MATLAB Function block settings” on page 38-19
- “Check for Stateflow chart settings” on page 38-20
- “Check for obsolete Unit Delay Enabled/Resettable Blocks” on page 38-21
- “Check for blocks that have nonzero output latency” on page 38-22
- “Check for unsupported storage class for signal objects” on page 38-23
- “Native Floating Point Checks Overview” on page 38-24
- “Check for single datatypes in the model” on page 38-25
- “Check for double datatypes in the model with Native Floating Point” on page 38-26
- “Check for Data Type Conversion blocks with incompatible settings” on page 38-27
- “Check for HDL Reciprocal block usage” on page 38-28
- “Check for Relational Operator block usage” on page 38-29
- “Check for unsupported blocks with Native Floating Point” on page 38-30
- “Check blocks with nonzero ulp error” on page 38-31
- “Industry standard checks overview” on page 38-32
- “Check VHDL file extension” on page 38-33
- “Check naming conventions” on page 38-34
- “Check top-level subsystem/port names” on page 38-35
- “Check module/entity names” on page 38-36
- “Check signal and port names” on page 38-37
- “Check package file names” on page 38-38

- “Check generics” on page 38-39
- “Check clock, reset, and enable signals” on page 38-40
- “Check architecture name” on page 38-41
- “Check entity and architecture” on page 38-42
- “Check clock settings” on page 38-43

# HDL Coder Checks in Model Advisor / HDL Code Advisor Overview

The **HDL Coder** checks in the Model Advisor or the HDL Code Advisor verify and update your Simulink model or subsystem for compatibility with HDL code generation. Running the checks produces a report that lists suboptimal conditions or settings, and then proposes better model configuration settings.

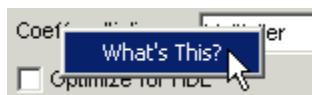
To learn about:

- HDL Code Advisor UI, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2.
- Model Advisor, see “Run Model Advisor Checks for HDL Coder” on page 39-6.

The left pane displays folders that perform various checks:

- **Model configuration checks:** Prepare your model for compatibility with HDL code generation. This folder contains checks that verify whether model parameters are HDL-compatible, whether your design contains algebraic loops, and so on.
- **Checks for ports and subsystems:** Verify whether ports and subsystems in your model have settings that are compatible for HDL code generation. The checks include whether you have a valid top-level DUT Subsystem and whether you have specified an initial condition for Enabled Subsystem and Triggered Subsystem blocks.
- **Checks for blocks and block settings:** Verify whether blocks in your model are supported for HDL code generation, and whether the supported blocks have HDL-compatible settings. The checks include whether source blocks in your model have a continuous sample time and whether Stateflow Charts and MATLAB Function blocks have HDL-compatible settings, and so on.
- **Native Floating Point checks:** Verify whether the block is compatible for HDL code generation in Native Floating Point mode. The checks include whether the blocks in your Simulink model are supported for HDL code generation with Native Floating Point, and whether the model uses single data types, and so on. Native floating-point support in HDL Coder generates target-independent HDL code from your single-precision floating-point model. For more information, see “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103.
- **industry-standard checks:** Verify whether your Simulink model conforms to the industry-standard rules. Industry-standard rules recommend using certain HDL coding guidelines. When generating code, HDL Coder displays an HDL coding standard report that shows how well the generated code adheres to the industry-standard guidelines. For more information, see “HDL Coding Standards” on page 26-4.

To learn more about each individual check, right-click that check, and select **What's This?**



See also “HDL Code Advisor Checks” on page 39-11.

## Model configuration checks overview

Use the checks in this folder to prepare your model for compatibility with HDL code generation. This folder contains checks that verify whether:

- The model parameters and visualization settings are compatible with HDL code generation.
- Your model uses foreign characters that are incompatible with the current encoding.
- The global reset setting is asynchronous for Altera devices and synchronous for Xilinx devices.
- The `InlineConfigurations` setting is enabled on the model.
- The design contains algebraic loops.

# Check for model parameters suited for HDL code generation

**Check ID:** com.mathworks.HDL.ModelChecker.runModelParamsChecks

Check for model parameters set up for HDL code generation.

## Description

This check verifies whether the model parameters that you specify are compatible for HDL code generation. This check ensures that you use these settings in the Configuration Parameters dialog box.

| Command-Line Parameter Setting                                                                                                                      | Configuration Parameter Setting                                                                                                                                                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Set Solver to FixedStepDiscrete.                                                                                                                    | Set <b>Type</b> to Fixed-step and <b>Solver</b> to Discrete (no continuous states).                                                                                                                                                                                                                                                                                  |
| Set FixedStep to auto.                                                                                                                              | Set <b>Fixed-step size (fundamental sample time)</b> to auto.                                                                                                                                                                                                                                                                                                        |
| Set EnableMultiTasking to off.                                                                                                                      | Disable the <b>Treat each discrete rate as a separate task</b> check box.                                                                                                                                                                                                                                                                                            |
| Set AlgebraicLoopMsg to error                                                                                                                       | Set <b>Algebraic loop</b> to error.                                                                                                                                                                                                                                                                                                                                  |
| Set SingleTaskRateTransMsg to error.                                                                                                                | Set <b>Single task rate transition</b> to error.                                                                                                                                                                                                                                                                                                                     |
| Set MultiTaskRateTransMsg to error.                                                                                                                 | Set <b>Multitask rate transition</b> to error.                                                                                                                                                                                                                                                                                                                       |
| Set BlockReduction to off                                                                                                                           | Disable the <b>Block Reduction</b> check box.                                                                                                                                                                                                                                                                                                                        |
| Set ConditionallyExecuteInputs to off.                                                                                                              | Disable <b>Conditional input branch execution</b> .                                                                                                                                                                                                                                                                                                                  |
| Set DefaultParameterBehaviour to Inlined. You can set this parameter at the command line by using <code>set_param</code> or <code>hdlsetup</code> . | Set <b>Default parameter behavior</b> to Inlined. If you want to set this parameter in the Configuration Parameters dialog box, you must have Simulink Coder.<br><br><b>Note</b> Enabling this parameter is the same as setting the <b>InlineParams</b> property to on. Setting <b>InlineParams</b> to off changes <b>DefaultParameterBehavior</b> value to Tunable. |
| Set DataTypeOverride to off.                                                                                                                        | No dialog box prompt.                                                                                                                                                                                                                                                                                                                                                |
| Set ProdHWDeviceType to ASIC/FPGA->ASIC/FPGA.                                                                                                       | Set <b>Device vendor</b> to ASIC/FPGA.                                                                                                                                                                                                                                                                                                                               |
| Set ShowLineDimensions and ShowPortDataTypes to on.                                                                                                 | In the <b>Debug</b> tab, on the <b>Information Overlays &gt; Ports</b> section, select <b>Base data types</b> .                                                                                                                                                                                                                                                      |
| Set SignalLoggingSaveFormat to Dataset.                                                                                                             | No dialog box prompt.                                                                                                                                                                                                                                                                                                                                                |

If there are incompatible model parameters, HDL Coder displays a warning and lists the model parameters that have to be fixed.

## Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator runs the `hdlsetup` command to set up the model parameters for HDL code generation. You can then rerun the check.

### See Also

- `hdlsetup`
- “Create HDL-Compatible Simulink Model”

# Check for global reset setting for Xilinx and Altera devices

**Check ID:** com.mathworks.HDL.ModelChecker.runGlobalResetChecks

Check asynchronous reset setting for Altera devices and synchronous reset setting for Xilinx devices.

## Description

This check verifies whether you use a global synchronous reset for a Xilinx device or a global asynchronous reset for an Altera device. You can improve the performance of your design by adhering to this recommended global reset setting depending on whether you target a Xilinx device or an Altera device.

---

**Note** If you do not specify a target device, this check passes successfully.

---

## Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator updates the **Reset type** setting to **Synchronous** if you use a Xilinx device and **Asynchronous** if you use an Altera device. You can then rerun the check.

## See Also

"Reset type" on page 17-8

## Check inline configurations setting

**Check ID:** com.mathworks.HDL.ModelChecker.runInlineConfigurationsChecks

Check `InlineConfigurations` is enabled.

### Description

This check verifies whether you have the `InlineConfigurations` property enabled. By default, this property is enabled, and HDL Coder includes VHDL configurations for the model inline with the rest of the VHDL code. If you disable this property, the code generator displays a warning.

### Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator updates the `InlineConfigurations` setting to on.

# Check algebraic loops

**Check ID:** com.mathworks.HDL.ModelChecker.runAlgebraicLoopChecks

Check model for algebraic loops.

## Description

HDL Coder does not support code generation for models in which algebraic loop conditions exist. This check examines the model and fails the check if it detects an algebraic loop.

## Results and Recommended Actions

To fix this warning, eliminate algebraic loops from your model and then run this check again.

## See Also

"Algebraic Loop Concepts"

## Check for visualization settings

**Check ID:** com.mathworks.HDL.ModelChecker.runVisualizationChecks

Check for display settings: port data types and sample time color coding.

### Description

This check determines whether you have:

- Enabled the **Port Data Types** setting on the model.
- Set the **Sample Time to Colors** on the model.

This check displays a message if your Simulink model doesn't have either or both of these settings.

### Results and Recommended Actions

Click **Modify Settings**, and the code generator:

- Enables the **Port Data Types** setting on the model.
- Sets the **Sample Time to Colors**.

You can then rerun the check.

### See Also

- “Display Port Data Types”
- “View Sample Time Information”

# Check delay balancing setting

**Check ID:** com.mathworks.HDL.ModelChecker.runBalanceDelaysChecks

Check Balance Delays is enabled.

## Description

This check reports a warning if the **Balance delays** setting is disabled on the model. When you generate HDL code, certain optimizations or block implementations can introduce delays along some signal paths in your model. If **Balance delays** is disabled, the code generator does not introduce equivalent delays on other parallel signal paths, which can result in a numerical mismatch between the original model and the generated model.

## Results and Recommended Actions

To fix this warning, click **Modify Settings**, and the code generator enables the **Balance delays** setting on the model. You can then rerun the check.

## See Also

- “Delay Balancing” on page 24-63
- “Balance delays” on page 15-3

## Check for ports and subsystems overview

This folder contains checks that verify whether ports and subsystems in your model have settings that are compatible for HDL code generation. You can verify whether:

- The subsystem in your Simulink model is a valid top level subsystem.
- The initial condition of Enabled Subsystem or Triggered Subsystem blocks in your model is zero.

## Check for invalid top level subsystem

**Check ID:** com.mathworks.HDL.ModelChecker.runInvalidDUTChecks

Check for subsystems that cannot be at the top level for HDL code generation.

### Description

This check verifies whether your Subsystem is a valid DUT for generating HDL code. For example, if you use an invalid DUT such as an Enabled Subsystem or a For Each Subsystem block, this check displays a warning and provides a link to the Subsystem.

### Results and Recommended Actions

To fix this warning, place this Subsystem inside another Subsystem, and then use that Subsystem as the DUT. You can then rerun the check.

## Check initial conditions of enabled and triggered subsystems

**Check ID:** com.mathworks.HDL.ModelChecker.runEnTrigInitConChecks

Check for initial condition of enabled and triggered subsystems.

### Description

This check verifies that any Enabled Subsystem blocks or Triggered Subsystem blocks in your Simulink model have a zero initial condition. If you have output ports with a nonzero initial condition, running this check displays a warning and provides links to the output ports.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** to set the initial condition of the output ports of the Enabled and Triggered Subsystem blocks to zero. You can then rerun the check.

### See Also

- Triggered Subsystem
- Enabled Subsystem

## Check for blocks and block settings overview

These checks verify whether blocks in your model are supported for HDL code generation, and whether the supported blocks have HDL-compatible settings. You can verify whether:

- There are source blocks with infinite sample time in your model.
- The blocks in your Simulink model are compatible for HDL code generation.
- There are unconnected lines, input ports, or output ports in your model.
- There are unresolved or disabled library links.
- MATLAB Function and Stateflow Chart blocks in your model have HDL-compatible settings.
- There are Delay, Unit Delay, and Zero-Order Hold blocks in the model that perform rate transition and replace them with Rate Transition blocks.

## Check for infinite and continuous sample time sources

**Check ID:** com.mathworks.HDL.ModelChecker.runSampleTimeChecks

Check source blocks with infinite or continuous sample time.

### Description

By default, the sample time of a Constant block is `inf`. When you connect a Constant block with sample time of `inf` to other blocks in your design, it hinders speed and area optimizations. Optimizations such as retiming, sharing, and streaming use the clock rate information to improve the speed and area of your design. For more details, see the “HDL Code Generation” section of the Constant page.

This check reports a warning if your design contains source blocks that have an infinite or continuous sample time.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** to update the **Sample time** of these source blocks to inherit through back propagation. That is, HDL Coder sets **Sample time** of these blocks to -1. You can then rerun the check.

### See Also

- “What Is Sample Time?”
- “Usage of Rate Change and Constant Blocks” on page 21-82

## Check for unsupported blocks

**Check ID:** com.mathworks.HDL.ModelChecker.runBlockSupportChecks

Check for unsupported blocks for HDL code generation.

### Description

This check displays a warning if your model uses blocks that are not supported for HDL code generation.

### Results and Recommended Actions

To fix this warning, update your design to use blocks that are supported with HDL Coder. You can then rerun the check.

## Check for large matrix operations

**Check ID:** com.mathworks.HDL.ModelChecker.runMatrixSizesChecks

Check for large matrix operations.

### Description

This check displays a warning if your design contains:

- Signals with matrix types that have more than two dimensions. HDL code generation supports matrix types that have a maximum of two dimensions.
- Large matrix operations with Add, Sum, or Product blocks that result in a matrix output with more than ten elements. Synthesizing the generated HDL code from such a design can result in the utilization of large number of resources on the target FPGA. For more details, see the "HDL Code Generation" sections of the block reference pages.

### Results and Recommended Actions

To fix this warning, update your design so that there are no matrix types with more than two dimensions and the result of a matrix operation does not produce an output with more than ten elements. To verify that the check passes, compile the design and rerun the check.

### See Also

- Product
- “Signal and Data Type Support” on page 10-2

# Check for MATLAB Function block settings

**Check ID:** com.mathworks.HDL.ModelChecker.runMLFcnBlkChecks

Check HDL compatible settings for MATLAB Function blocks.

## Description

This check displays a warning if your model uses MATLAB Function blocks that have settings not recommended for HDL code generation. The settings include checking whether the MATLAB Function block has:

- `fimath` settings defined as per `hdlfimath`. The `hdlfimath` function uses `fimath` settings that are compatible for HDL code generation.
- **Saturate on integer overflow** check box cleared.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the MATLAB Function block settings to be compatible with HDL code generation.

## See Also

“Design Guidelines for the MATLAB Function Block” on page 29-29

## Check for Stateflow chart settings

**Check ID:** com.mathworks.HDL.ModelChecker.runStateflowChartSettingsChecks

Check HDL compatible settings for Stateflow Chart blocks.

### Description

This check displays a warning if your model uses Stateflow Chart blocks that have settings incompatible for HDL code generation. The settings include checking whether the Stateflow Chart has **Execute (enter) Chart At Initialization** set, whether the **State Machine Type** is **Moore**, and so on.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the Stateflow Chart settings to be compatible with HDL code generation.

### See Also

Chart

# Check for obsolete Unit Delay Enabled/Resettable Blocks

**Check ID:** com.mathworks.HDL.ModelChecker.run0obsoleteDelaysChecks

Check if the DUT contains obsolete Unit Delay Enabled/Resettable blocks

## Description

This check displays a warning if the DUT Subsystem contains any of these blocks:

- Unit Delay Enabled
- Unit Delay Resettable
- Unit Delay Enabled Resettable

These blocks have been obsoleted. The code generator does not recommend usage of these blocks in your Simulink model.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces these blocks with the corresponding synchronous counterparts:

- Unit Delay Enabled is replaced by Unit Delay Enabled Synchronous.
- Unit Delay Resettable is replaced by Unit Delay Resettable Synchronous.
- Unit Delay Enabled Resettable is replaced by Unit Delay Enabled Resettable Synchronous.

These blocks are recommended because they use the State Control block for synchronous simulation behavior and generate hardware-friendly HDL code. For more information, see “Synchronous Subsystem Behavior with the State Control Block” on page 27-85.

## Check for blocks that have nonzero output latency

**Check ID:** com.mathworks.HDL.ModelChecker.runNFPLatencyChecks

Check for blocks that introduce latency in the generated code but do not simulate with latency in original model

### Description

Native floating-point operators and certain fixed-point blocks introduce latency in the generated HDL code. This check detects blocks in your Simulink model that introduce latency in the generated HDL code when you use fixed point and floating-point types. If your model uses floating-point types, set the “Floating Point IP Library” on page 16-3 to Native Floating Point.

When you run the check, the **Result** subpane displays hyperlinks to the blocks that have a nonzero output latency, and the latency value. When you generate code, HDL Coder figures out this latency.

### Results and Recommended Actions

By using the latency information reported in the **Result** subpane, you can add the appropriate number of delays adjacent to those blocks in your original model, and therefore simulate the original model with latency. The code generator absorbs the delays you added to your model, and does not have to introduce additional latency in the generated model.

For blocks with nonzero latency that are reported by this check, consider the effect this latency has in the validation model. In the generated model and validation model, you see the additional delays that the code generator adds to account for the latency.

### See Also

“Latency Considerations with Native Floating Point” on page 10-96

# Check for unsupported storage class for signal objects

**Check ID:** com.mathworks.HDL.ModelChecker.runSignalObjectStorageClassChecks

Check whether signal object storage class is 'ExportedGlobal' or 'ImportedExtern' or 'ImportedExternPointer'

## Description

This check displays a warning if your model contains signals that have the signal object storage class set to 'ExportedGlobal', 'ImportedExtern', or 'ImportedExternPointer'. The warning message also provides links to those signals that have the signal object storage class set to one of these signal object storage class specifications.

HDL code generation ignores these storage class specifications that you specify in your design, which may sometimes result in conflicting signal names. When you simulate the validation model, HDL Coder may generate errors.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces those signals that have the signal object storage class specified as ExportedGlobal, ImportedExtern, or ImportedExternPointer to Auto.

## See Also

[coder.storageClass](#)

## More About

- “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder)
- “Storage Classes for Code Generation from MATLAB Code” (Embedded Coder)

## Native Floating Point Checks Overview

These checks verify whether the model is compatible for HDL code generation in Native Floating Point mode. Native floating-point support in HDL Coder generates target-independent HDL code from your single-precision floating-point model. For more information, see “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103.

Use the checks in this folder to verify whether:

- You use the Native Floating Point mode when your design contains `single` data types.
- Your model uses `double` data types in Native Floating Point mode.
- Blocks in your model are supported for HDL code generation in Native Floating Point mode.
- Blocks in your model have a nonzero ulp error and a nonzero output latency.

# Check for single datatypes in the model

**Check ID:** com.mathworks.HDL.ModelChecker.runNFPSuggestionsChecks

Check for single data types in the model.

## Description

This check detects whether your Simulink model uses single data types and displays a warning if the **Floating Point IP Library** is not set to Native Floating Point.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator sets Native Floating Point as the **Floating Point IP Library**. You can then rerun the check.

## See Also

"Getting Started with HDL Coder Native Floating-Point Support" on page 10-80

## Check for double datatypes in the model with Native Floating Point

**Check ID:** com.mathworks.HDL.ModelChecker.runDoubleDatatypeChecks

Check for double data types in the model.

### Description

This check displays a warning when your Simulink model uses double data types with **Floating Point IP Library** set to Native Floating Point. The check passes when your model uses double data types but does not have the **Floating Point IP Library** set to Native Floating Point.

### Results and Recommended Actions

To fix this warning, update your design by converting double data types to `single` and then set the **Floating Point IP Library** to Native Floating Point. You can then rerun the check.

---

**Note** The check passes when your model uses double data types but does not have the **Floating Point IP Library** set to Native Floating Point. However, double data types in your model can generate reals in the HDL code which is not synthesizable. To generate synthesizable HDL code, convert double data types to `single` and set the **Floating Point IP Library** to Native Floating Point.

---

### See Also

"Getting Started with HDL Coder Native Floating-Point Support" on page 10-80

# Check for Data Type Conversion blocks with incompatible settings

**Check ID:** com.mathworks.HDL.ModelChecker.runNFPDTCChecks

Check conversion mode of Data Type Conversion blocks.

## Description

This check displays a warning when Data Type Conversion blocks in your model convert from a floating-point data type to a fixed-point data type or vice-versa, and has **Input and output to have equal** parameter set to **Stored Integer (SI)**.

HDL Coder does not support Data Type Conversion blocks that use the **Stored Integer (SI)** conversion mode and convert between floating point and fixed point data types. During this conversion, the **Stored Integer (SI)** mode does not preserve the underlying stored integer bits of the floating point input signal.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces Data Type Conversion blocks in **Stored Integer (SI)** mode with **Float Typecast** blocks.

Using the **Float Typecast** block, you can access the stored integer bits of a floating point input signal when converting between floating point and fixed point data types. The block works similar to the **typecast** function.

## See Also

"Getting Started with HDL Coder Native Floating-Point Support" on page 10-80

## Check for HDL Reciprocal block usage

**Check ID:** com.mathworks.HDL.ModelChecker.runNFPHDLRecipChecks

Check HDL Reciprocal blocks are not using floating point types

### Description

This check displays a warning if your Simulink model contains HDL Reciprocal blocks that use floating-point data types. HDL Reciprocal blocks with floating point data types can have potential numerical mismatches with the Simulink simulation results, and can use more resources on the target hardware.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator replaces the HDL Reciprocal blocks with Math Reciprocal blocks.

### See Also

HDL Reciprocal

# Check for Relational Operator block usage

**Check ID:** com.mathworks.HDL.ModelChecker.runNFPRelopChecks

Check Relational Operator blocks which use floating point types have boolean outputs.

## Description

This check displays a warning if your Simulink model contains Relational Operator blocks that compare floating-point data types and produce a nonboolean output. IBy default, the **Output data type** of the block is **boolean**. If you specify a nonboolean output, the block implementation after HDL code generation is not optimal, and can increase the resource usage on the target hardware device.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator changes the **Output data type** of the block to **boolean**.

## See Also

Relational Operator

## Check for unsupported blocks with Native Floating Point

**Check ID:** com.mathworks.HDL.ModelChecker.runNFPSupportedBlocksChecks

Check for unsupported blocks with Native Floating Point.

### Description

This check displays a warning if your Simulink model contains blocks that use `single` data types, and are not supported for HDL code generation in the native floating-point mode.

### Results and Recommended Actions

To fix this warning, make sure that your design uses blocks that are supported for HDL code generation with Native Floating Point.

### See Also

"Simulink Blocks Supported with Native Floating-Point" on page 10-120

## Check blocks with nonzero ulp error

**Check ID:** com.mathworks.HDL.ModelChecker.runNFPULPErrorChecks

Check for blocks that have nonzero ulp error with Native Floating Point.

### Description

This check detects blocks in your Simulink model that have a nonzero ULP error in native floating-point mode. When you run the check, the **Result** subpane displays hyperlinks to the blocks that have nonzero ULP error, and the ulp values.

### Results and Recommended Actions

To fix this warning, look for instances of blocks that have nonzero ULP error and specify a **Tolerance Value** by setting **Floating point tolerance check based on the ulp error**. You can then rerun the check.

---

**Note** Fixing warnings that are reported by this check does not guarantee that your Simulink model has a zero ulp error. Make sure that you verify the ulp of your design by using multiple methods, such as by generating HDL code and test benches.

---

### See Also

"Numeric Considerations with Native Floating-Point" on page 10-84

## Industry standard checks overview

These checks verify whether your Simulink model conforms to the industry-standard rules. Industry-standard rules recommend using certain HDL coding guidelines. When generating code, HDL Coder displays an HDL coding standard report that shows how well the generated code adheres to the industry-standard guidelines. For more information, see “HDL Coding Standards” on page 26-4

Use the checks in this folder to verify whether:

- Names in your design adhere to the standard naming conventions.
- Subsystem names, top-level subsystem and port names, and signal and port names have the recommended number of characters in length.
- The generated VHDL code from your design follows recommended guidelines. The guidelines recommend that the file name extension is `.vhd`, the architecture name is `rtl`, the package file postfix is `_pkg`, and that the generated code does not use generics at the top level.
- Clock settings adhere to the industry-standard guidelines, and whether clock, reset, and enable signals adhere to the naming conventions.

## Check VHDL file extension

**Check ID:** com.mathworks.HDL.ModelChecker.runFileExtensionChecks

Check file extensions of VHDL files containing entities.

### Description

This check detects whether the file extension is .vhd when you generate code with VHDL as the target language. This check corresponds to rule 1.A.A.1 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the file name extension to .vhd.

### See Also

Rule 1.A.A.1 of “Basic Coding Practices” on page 26-9.

## Check naming conventions

**Check ID:** com.mathworks.HDL.ModelChecker.runNameConventionChecks

Check standard keywords used by EDA tools.

### Description

This check detects whether names in the design are adhering to the standard naming convention and not using names starting with VDD, VSS, and so on. This check corresponds to rule 1.A.A.4 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the names by replacing the reserved keywords with rsvd to adhere to the industry-standard rule.

### See Also

Rule 1.A.A.4 of “Basic Coding Practices” on page 26-9.

## Check top-level subsystem/port names

**Check ID:** com.mathworks.HDL.ModelChecker.runToplevelNameChecks

Check top-level module/entity and port names.

### Description

This check verifies whether top-level module/entity and port names are of the same case and less than or equal to 16 characters in length. This check corresponds to rule 1.A.A.9 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the top-level module/entity and port names to be of the same case and less than or equal to 16 characters. Names longer than 16 characters are truncated to fit within 16 characters in length.

### See Also

Rule 1.A.A.9 of “Basic Coding Practices” on page 26-9.

## Check module/entity names

**Check ID:** com.mathworks.HDL.ModelChecker.runSubsystemNameChecks

Check module/entity names.

### Description

This check verifies whether subsystems in your model have names between 2 and 32 characters in length. This check corresponds to rule 1.A.B.1 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the Subsystem names such that the module/entity and port names are between 2 and 32 characters.

### See Also

Rule 1.A.B.1 of “Basic Coding Practices” on page 26-9.

## Check signal and port names

**Check ID:** com.mathworks.HDL.ModelChecker.runPortSignalNameChecks

Check signal and port name lengths.

### Description

This check verifies whether ports and signals from the blocks have names between 2 and 40 characters in length. This check corresponds to rule 1.A.C.3 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the signal and port names to be between 2 and 40 characters in length.

### See Also

Rule 1.A.C.3 of “Basic Coding Practices” on page 26-9.

## Check package file names

**Check ID:** com.mathworks.HDL.ModelChecker.runPackageNameChecks

Check file name containing packages.

### Description

This check verifies whether the package file postfix is `_pac` when you generate code with VHDL as the target language. By default, the generated package file postfix is `_pkg`. This check corresponds to rule 1.A.D.1 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the package file name to `_pac`.

### See Also

Rule 1.A.D.1 of “Basic Coding Practices” on page 26-9.

# Check generics

**Check ID:** com.mathworks.HDL.ModelChecker.runGenericChecks

Check generics at top level subsystem.

## Description

This check verifies whether you have generics at the top-level subsystem. If your design uses mask parameters and has the `MaskParameterAsGeneric` setting enabled, then the top-level subsystem can have generics. This check corresponds to rule 1.A.D.9 of the industry-standard rules.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator disables the `MaskParameterAsGeneric` setting so that there are no generics at the top-level subsystem.

## See Also

Rule 1.A.D.9 of “Basic Coding Practices” on page 26-9.

## Check clock, reset, and enable signals

**Check ID:** com.mathworks.HDL.ModelChecker.runClockResetEnableChecks

Check naming convention for clock, reset, and enable signals.

### Description

This check verifies whether clock, reset, and clock enable signals follow the recommended naming convention. Clock signal names must contain `clk` or `ck`, reset signal names must contain `rstx`, `reset_x`, or `reset_x`, and clock enable signal names must contain `en`. This check corresponds to rule 1.A.E.2 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the clock, reset, and enable signals to adhere to the naming conventions.

### See Also

Rule 1.A.E.2 of “Basic Coding Practices” on page 26-9.

# Check architecture name

**Check ID:** com.mathworks.HDL.ModelChecker.runArchitectureNameChecks

Check VHDL architecture name in the generated HDL code.

## Description

This check verifies whether the architecture name is `rtl` when you generate code with VHDL as the target language. This check corresponds to rule 1.A.F.1 of the industry-standard rules.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator updates the `VHDLArchitectureName` setting to `rtl` to adhere to the industry-standard rule.

## See Also

Rule 1.A.F.1 of “Basic Coding Practices” on page 26-9.

## Check entity and architecture

**Check ID:** com.mathworks.HDL.ModelChecker.runSplitEntityArchitectureChecks

Check whether the VHDL entity and architecture are described in the same file.

### Description

This check detects when you have the entity and architecture descriptions in separate files when you generate code with VHDL as the target language. The entity and architecture descriptions can be in separate files if you enable the `SplitEntityArch` setting. This check corresponds to rule 1.A.F.4 of the industry-standard rules.

### Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator disables the `SplitEntityArch` setting so that the entity and architecture descriptions are in the same file.

### See Also

Rule 1.A.F.4 of “Basic Coding Practices” on page 26-9.

# Check clock settings

**Check ID:** com.mathworks.HDL.ModelChecker.runClockChecks

Check constraints on clock signals.

## Description

This check detects multiple constraints on clock signals that correspond to these industry-standard rules:

- Rule 1.B.A.1: Design should have only a single clock and use only one edge of the clock. This rule may be violated if you have the `ClockInputs` property set to `Multiple`.
- Rule 1.D.C.6: Do not use flip-flops with inverted edges.
- Rule 1.D.D.2: One hierarchical level should have a single clock only. This rule can be violated if you set `ClockInputs` to `Multiple`, or your design uses trigger signals and enabling `TriggerAsClock` can result in clock signals at various levels in the hierarchy.

## Results and Recommended Actions

To fix this warning, click **Modify Settings** and the code generator:

- Updates the `ClockInputs` property to `Single`.
- Disables the `TriggerAsClock` setting.

## See Also

Rules 1.B.A.1, 1.D.C.6, and 1.D.D.2 of “Basic Coding Practices” on page 26-9.



# Using the HDL Code Advisor

---

- “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2
- “Run Model Advisor Checks for HDL Coder” on page 39-6
- “HDL Code Advisor Checks” on page 39-11

# Check HDL Compatibility of Simulink Model Using HDL Code Advisor

## In this section...

- “Open the HDL Code Advisor” on page 39-2
- “Run Checks In the HDL Code Advisor” on page 39-3
- “Fix HDL Code Advisor Warnings or Failures” on page 39-3
- “View and Save HDL Code Advisor Reports” on page 39-4

The HDL Code Advisor verifies and updates your Simulink model or subsystem for compatibility with HDL code generation. The Model Checker checks for model configuration settings, ports and subsystem settings, block settings, support for native floating point, and conformance to the industry-standard rules. The Code Advisor produces a report that lists suboptimal conditions or settings, and then proposes better model configuration settings.

The HDL Code Advisor has these caveats:

- If you reference one model in another by using a Model block, the HDL Code Advisor checks the model configurations or settings of the parent model. To check whether the referenced model is compatible with HDL code generation, open the HDL Code Advisor for the referenced model, and then run the checks.
- If you run the checks on masked library blocks in your Simulink model, the Code Advisor cannot verify whether the blocks inside the library blocks have HDL-compatible settings.
- When you apply Model Advisor checks to your model, it increases the likelihood that your model does not violate certain modeling standards or guidelines. However, it does not guarantee that the design is ready for HDL code generation. Make sure that you verify the design by using multiple methods for HDL code generation readiness.

## Open the HDL Code Advisor

To open the HDL Code Advisor:

- From the UI, in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the DUT Subsystem and then click **HDL Code Advisor**.
- To run the checks for the Subsystem you want to analyze, right-click that Subsystem, and in the context menu, select **HDL Code > Check Model Compatibility**.
- At the command line, enter `hdlmodelchecker('system')`. *system* is a handle or name of the model or subsystem that you want to check. For more information, see `hdlmodelchecker`.

In the HDL Code Advisor, the left pane lists the folders in the hierarchy. Each folder represents a group or category of related checks. Expanding the folders shows available checks in each folder. From the left pane, you can select a folder or an individual check. The HDL Code Advisor displays information about the selected folder or check in the right pane. The content of the right pane depends on the selected folder or check. The right pane has a **Result** subpane that contains a display area for status messages and other task results.

To learn more about each individual check, right-click that check, and select **What's This?**.



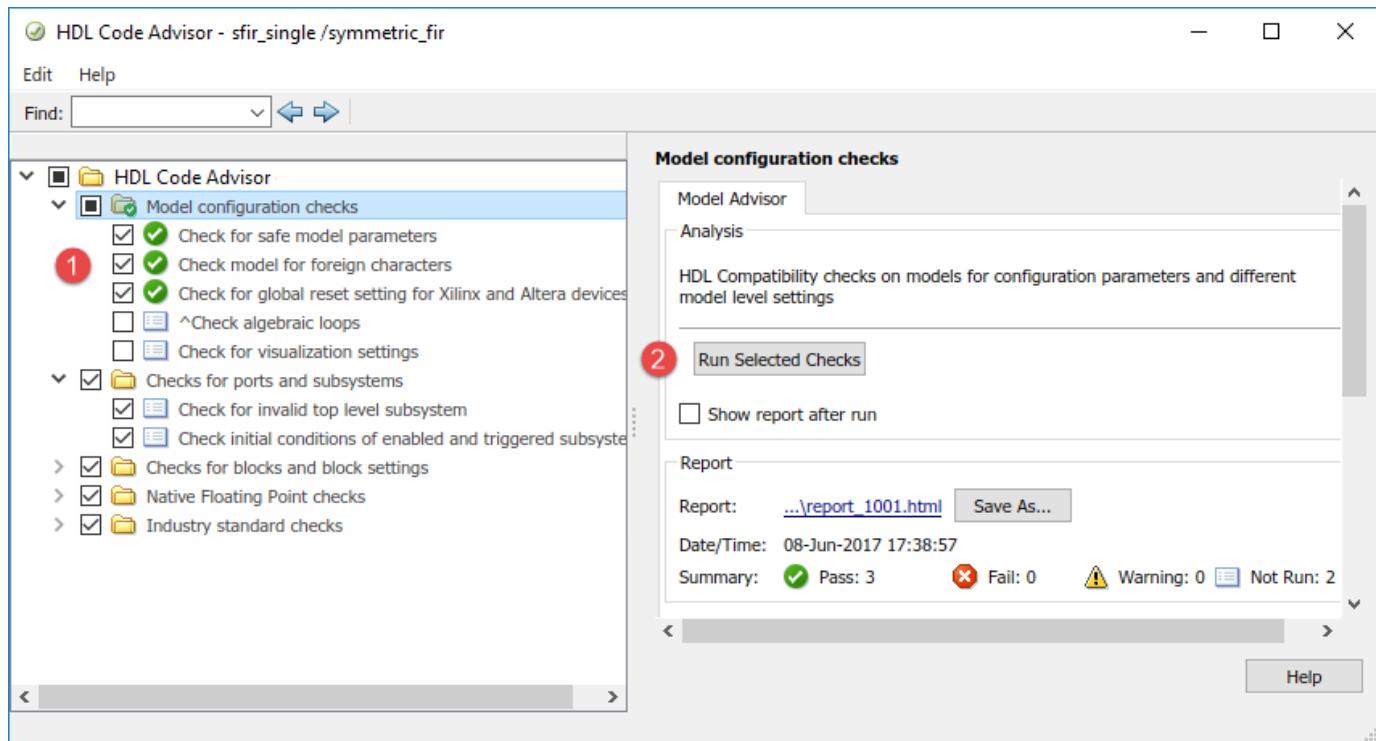
## Run Checks In the HDL Code Advisor

In the HDL Code Advisor window, you can run individual checks or a group of checks. To run a check, **Select** that check and then click **Run This Check**. For example, to run the **Check for safe model parameters**, select the check box, and then click **Run This Check**.

In the HDL Code Advisor window, you can run a group of checks within a folder.

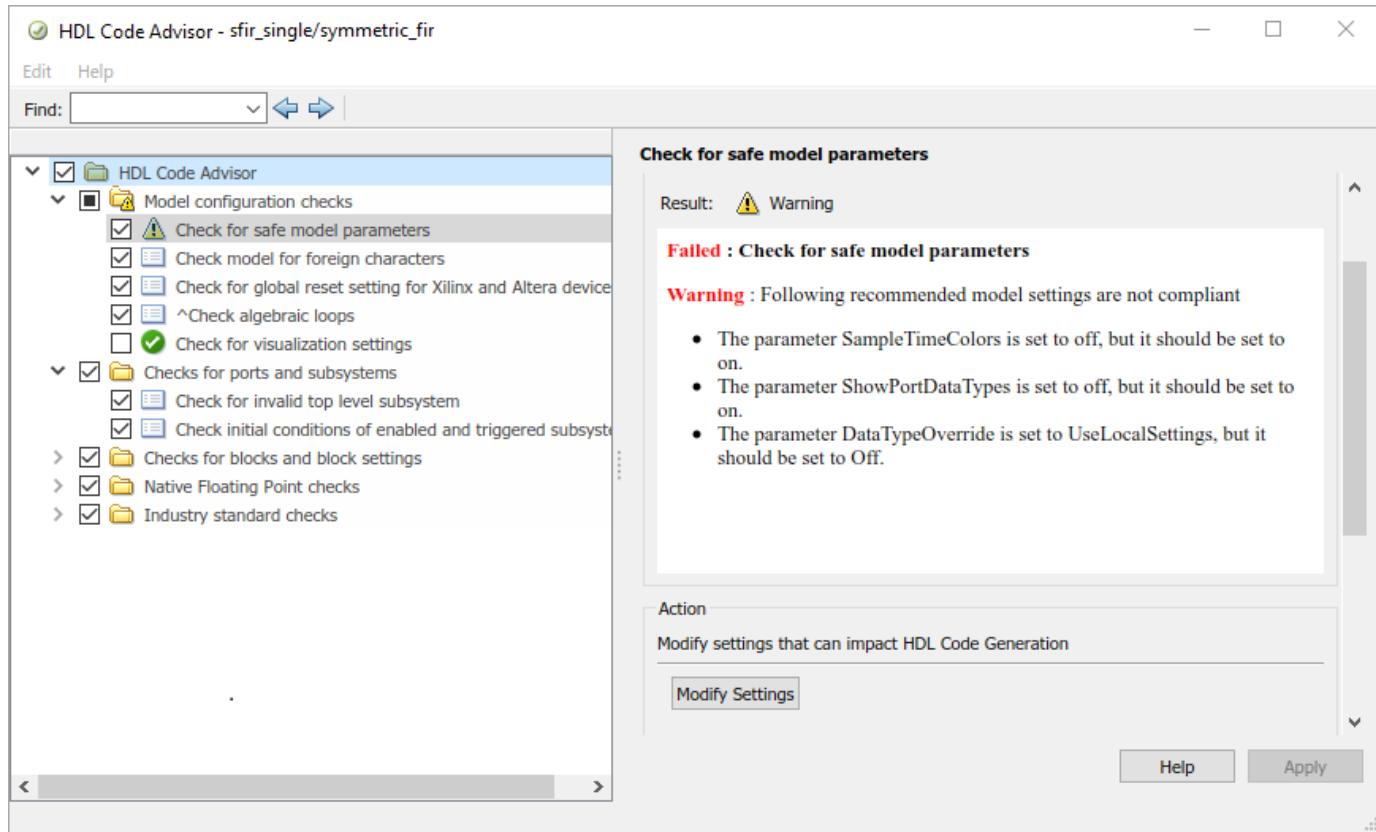
- 1 Select the checks that you want to run.
- 2 Select the folder that contains these checks and then click **Run Selected Checks**.

This example shows how to run selected checks in the **Model configuration checks** folder.



## Fix HDL Code Advisor Warnings or Failures

In the HDL Code Advisor, if a check fails, the right pane shows the warning or failure information in a **Result** subpane. The **Result** subpane displays model settings that are not compliant. For some tasks, use the **Action** subpane to apply the Code Advisor recommended settings. This example displays the incorrect model settings that caused the **Check for safe model parameters** to fail.



To apply the correct model configuration settings that the code generator reported in the **Result** subpane, click the **Modify Settings** button. After you click **Modify Settings**, the **Result** subpane reports the changes that were applied. You can now run this check.

## View and Save HDL Code Advisor Reports

When you run checks in the HDL Code Advisor, HDL Coder generates an HTML report of the check results. Each folder in the HDL Code Advisor contains a report for the checks within that folder and its subfolders. To access reports, select a folder such as **Model configuration checks**, and in the **Report** subpane, click **Save As**. If you rerun the HDL Code Advisor, the report is updated in the working folder, not in the save location.

This report shows typical results for a run of the **Model configuration checks** folder.

**Model Advisor Report - sfir\_single.slx**

Simulink version: 9.0 Model version: 1.82  
 System: sfir\_single/symmetric\_fir Current run: 08-Jun-2017 17:38:57  
 Treat as Referenced Model: off

**Run Summary**

| Pass | Fail | Warning | Not Run | Total |
|------|------|---------|---------|-------|
| 3    | 0    | 0       | 2       | 5     |

**Model configuration checks**

- Check for safe model parameters**  
**Passed** : Check for safe model parameters
- Check model for foreign characters**  
 Check that the characters in the model can be represented in the current encoding.  
**Passed** All the characters in the model can be represented in the current encoding.

As you run the checks, the HDL Code Advisor updates the reports with the latest information for each check in the folder. When you run the checks at different times, timestamps appear at the top right of the report to indicate when checks have been run. Checks that occurred during previous runs have a timestamp following the check name. You can filter checks in the report such that tasks that are **Not Run** do not appear or show tasks that **Passed**, and so on. To view the report for a folder each time the tasks for the folder have been run, select **Show report after run**.

## See Also

### More About

- “HDL Code Advisor Checks” on page 39-11

## Run Model Advisor Checks for HDL Coder

The Model Advisor checks a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation. The Model Advisor produces a report that lists the suboptimal conditions or settings that it finds, and proposes better model configuration settings where appropriate. HDL Coder integrates the checks in the HDL Code Advisor with the Model Advisor.

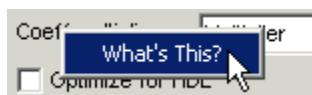
### Open the Model Advisor Checks

You can open the Model Advisor in either of these ways:

- In the **Modeling** tab, select **Model Advisor**. In the System Selector dialog box, select the model or Subsystem that you want to analyze, and click **OK**.
- To run the model advisor checks for the Subsystem that you want to analyze, right-click that Subsystem, and select **Model Advisor > Open Model Advisor**.
- At the command line, enter `modeladvisor('system')`. *system* is the name of the model or Subsystem that you want to analyze. For more information, see `modeladvisor`.

When you open the Model Advisor in Simulink, you see the checks in the **HDL Coder** subfolder of the **By Product** folder. Each subfolder in the **HDL Coder** folder represents a group or category of related checks. Expanding the folders display available checks in each folder. From the left pane, you can select a folder or an individual check. The Model Advisor displays information about the selected folder or check in the right pane. The content of the right pane depends on the selected folder or check. The right pane has a **Result** subpane that contains a display area for status messages and other task results.

To learn more about each individual check, right-click that check, and select **What's This?**.



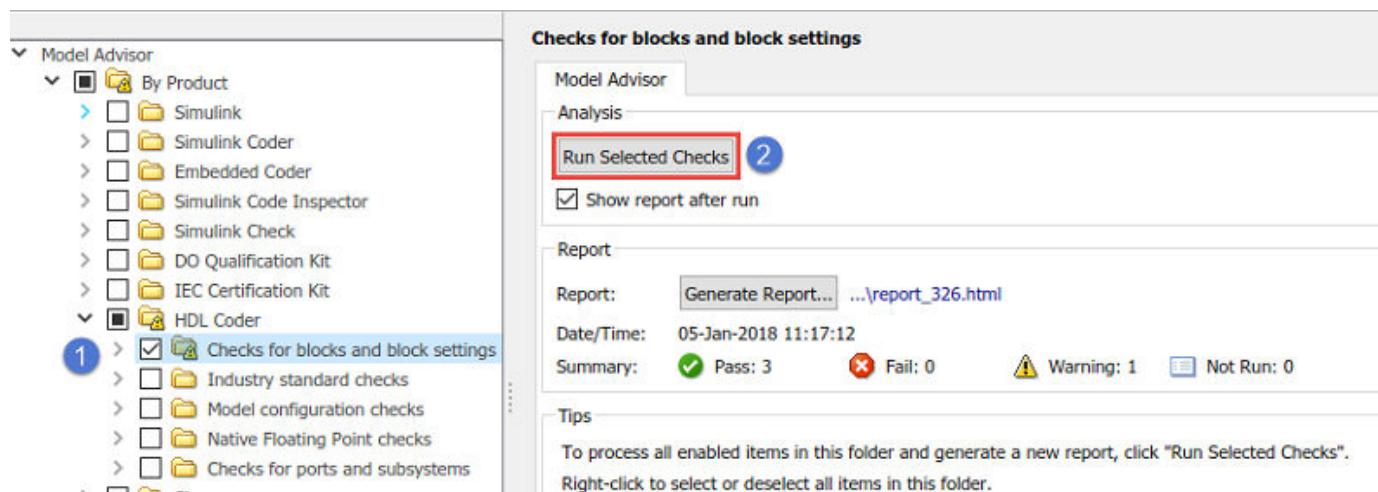
### Run Checks in the Model Advisor

In the Model Advisor window, you can run individual checks or a group of checks. To run a check, **Select** that check, and then click **Run This Check**.

To run a group of checks within a folder:

- 1 Select the checks that you want to run.
- 2 Select the folder that contains these checks and then click **Run Selected Checks**

For example, to run all the checks in the **Checks for blocks and block settings** folder, select the folder, and then click **Run Selected Checks**.



You can also click the button to run the selected checks in the Model Advisor.

## Run Checks In Background

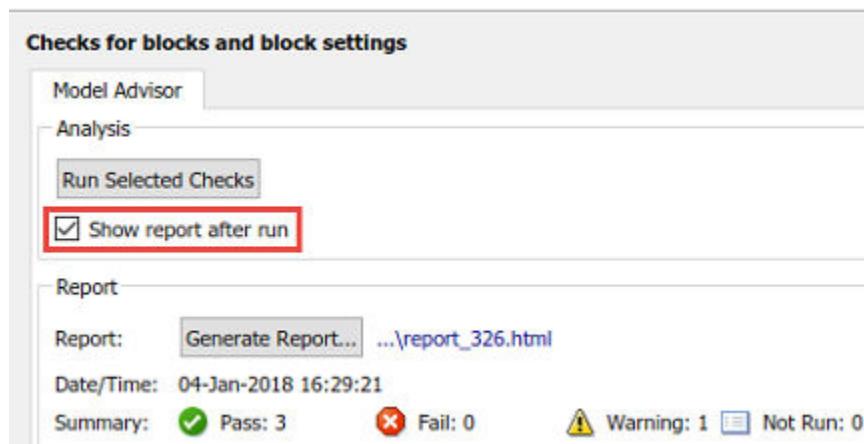
If you have Parallel Computing Toolbox™, you can run the checks in the background. You can continue working on the model during analysis. The analysis does not reflect changes that you make to your model while Model Advisor is running in the background.

To run the Model Advisor checks in background, before you select and run the checks, click the **Run checks in background** toggle, .

When you run the checks, the Model Advisor starts an analysis on a parallel processor. To stop running checks in the background, in the Model Advisor Window, click **Stop background run**, .

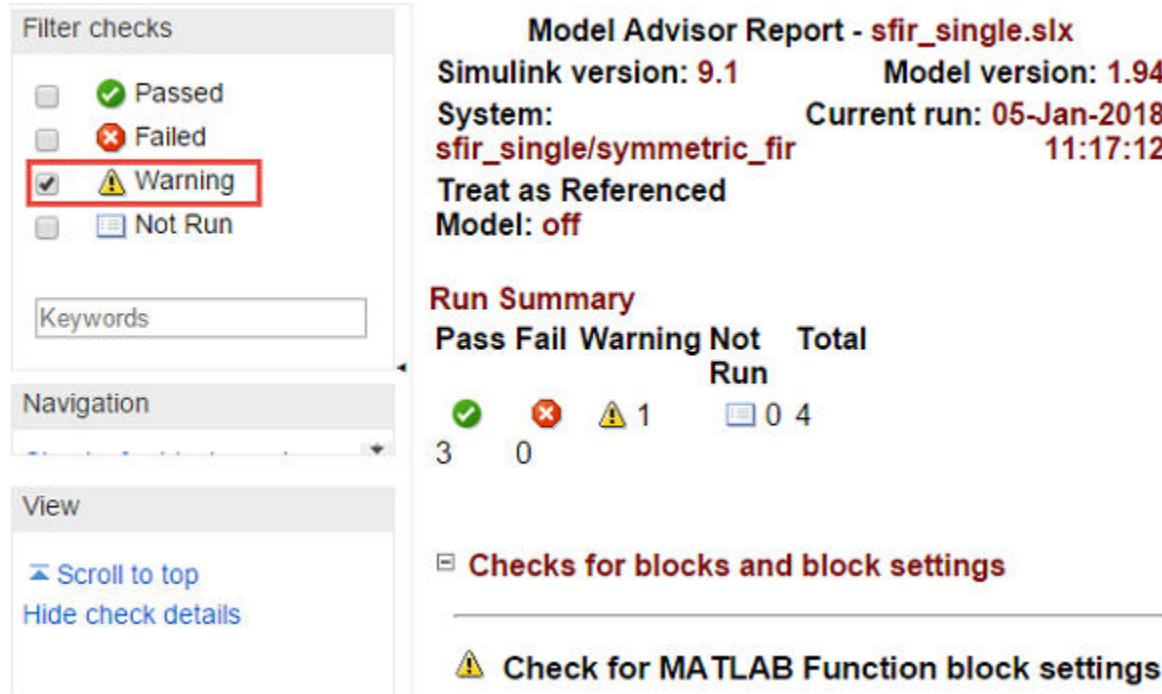
## Display Check Results in the Model Advisor Report

To display an HTML report of the check results, before you run the checks, select **Show report after run**. You can specify this setting to generate a report for all the checks in the **HDL Coder** folder, or for all checks within a subfolder, such as the **Checks for blocks and block settings**.



If you did not select **Show report after run**, you can generate a report after you run the checks by selecting **Generate Report**. Specify the **Directory**, **Filename**, and **Format** of the HTML report that want to generate.

This report shows typical results for a run of the **Checks for blocks and block settings** folder.



The report displays a run summary of the checks in the folder that you generated the report for. As you run the checks, the Model Advisor updates the reports with the latest information for each check in the folder. When you run the checks at different times, timestamps appear at the top right of the report to indicate when checks have been run. Checks that occurred during previous runs have a timestamp following the check name. You can filter checks in the report to show checks that display a **Warning**, or show checks that **Passed**, and so on.

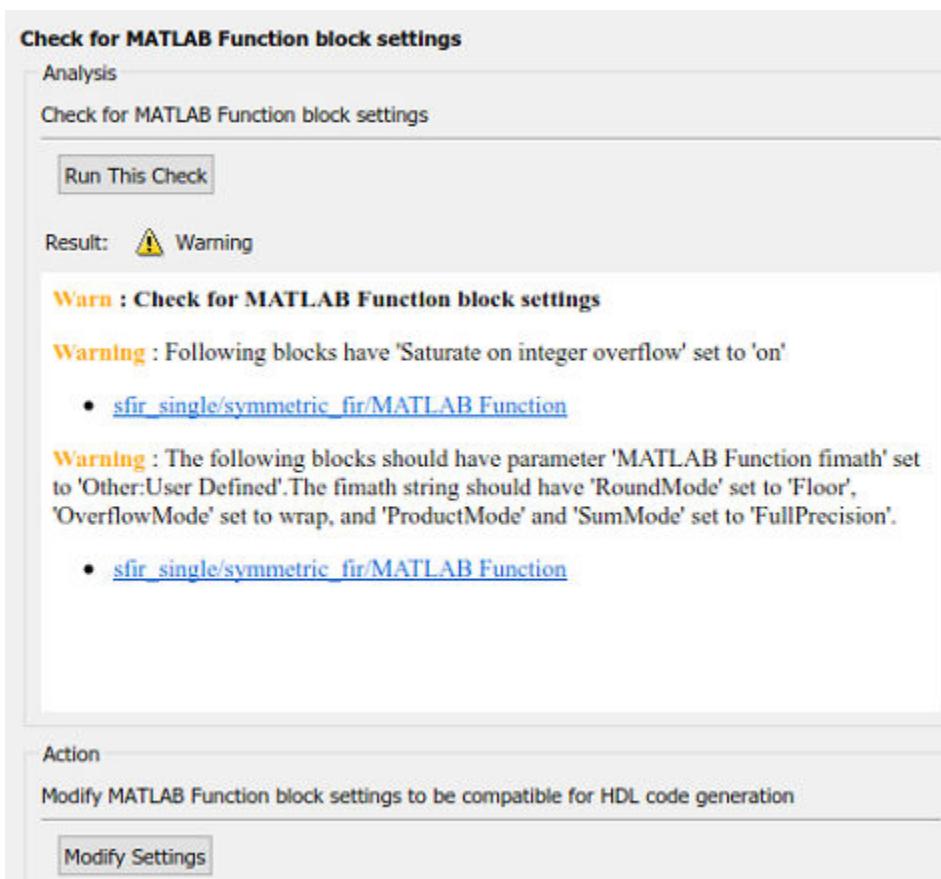
## Fix Warnings or Failures

When a model or referenced model has a suboptimal condition, checks can fail. After you run a Model Advisor analysis, indicates checks that have warnings. A warning result is informational. You can fix the reported issue or move on to the next task.

You can also use the Model Advisor highlighting capability to color-highlight Simulink blocks and Stateflow charts in your model, which indicates the analysis results. To highlight blocks, in the Model Advisor window, select **Highlighting > Enable Highlighting**.

To fix warnings or failures, in the **Result** subpane, review the recommended actions to make changes to your model. When you fix a warning or failure, to verify that the check passes, rerun the check.

Some checks have an **Action** subpane. This example displays the incorrect MATLAB Function block settings that caused the check to display a warning.



When you select **Modify Settings**, the **Result** subpane shows the changes that were applied. To verify that the check passes, rerun the check. If you use the Model Advisor dashboard, you see that the analysis is faster when you rerun the check because the Model Advisor does not reload the checks before executing them.

## Save and Restore Model Advisor State

By default, the Simulink software saves the state of the most recent Model Advisor session. The next time that you activate the Model Advisor, it returns to that state. You can also save the current settings of the Model Advisor to a named restore point. A *restore point* is a snapshot in time of the model, base workspace, and Model Advisor. Later, you can restore the same settings by loading the restore point data into the Model Advisor.

You can use this data restore point to revert changes to your model in response to recommendations from the Model Advisor. For example, you can save a model and restore point to undo your changes if the Model Advisor reports a warning after running a certain check. You can also restore the default configuration of the Model Advisor. In the Model Advisor window, select **Settings > Restore Default Configuration**.

To save the Model Advisor state, in the Model Advisor Window, select **File > Save Restore Point As**. Enter a **Name** and **Description**, and then click **Save**. You can save more than one restore point.

To restore a Model Advisor state, in the Model Advisor Window, select **File > Load Restore Point**. Select the restore point and click **Load**. When you load a restore point, the Model Advisor warns that the restoration overwrites the current settings.

## See Also

### More About

- “HDL Code Advisor Checks” on page 39-11
- “Run Model Advisor Checks”

# HDL Code Advisor Checks

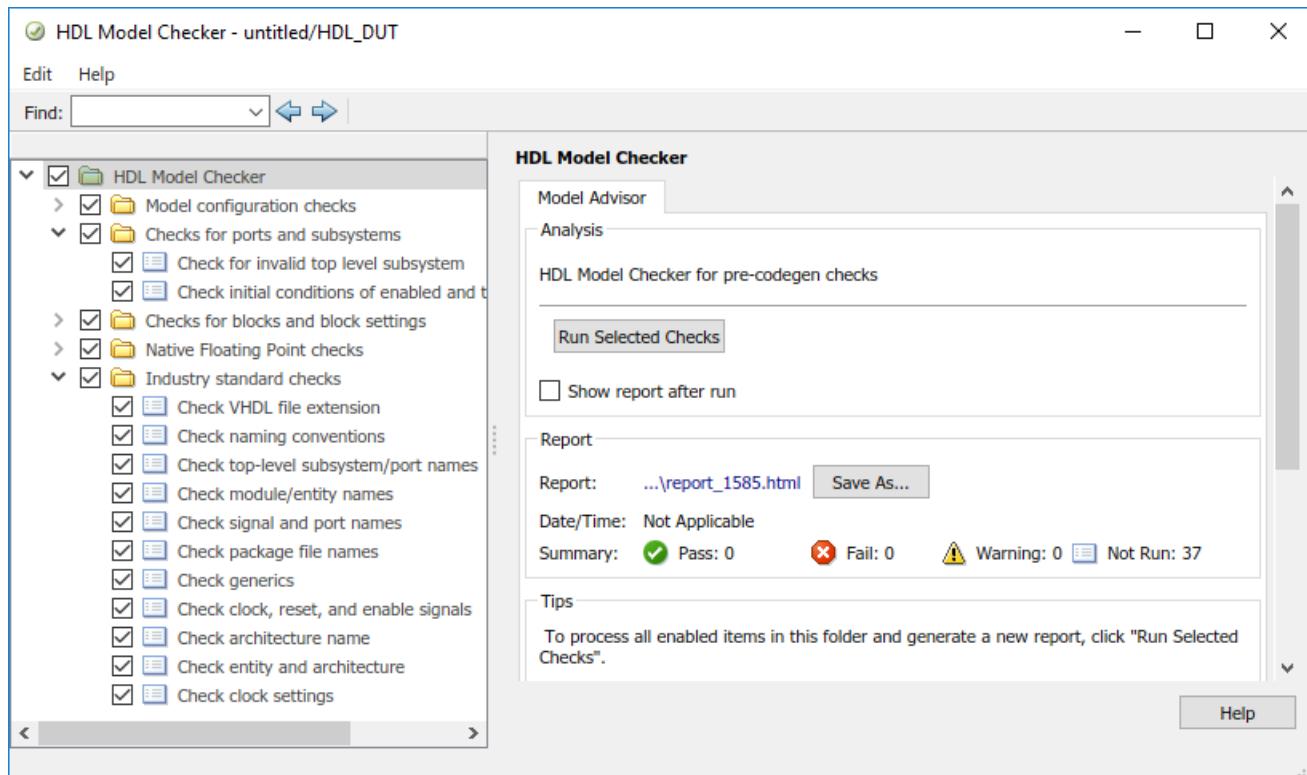
## In this section...

- “Model configuration checks” on page 39-12
- “Checks for ports and subsystems” on page 39-12
- “Checks for blocks and block settings” on page 39-12
- “Native Floating Point checks” on page 39-13
- “industry standard checks” on page 39-14

The HDL Code Advisor and the Model Advisor checks in HDL Coder verify and update your Simulink model or subsystem for compatibility with HDL code generation. The Code Advisor has checks for:

- Model configuration settings
- Ports and Subsystem settings
- Blocks and block settings
- Native Floating Point support
- Industry standard guidelines

When you run a check, the Code Advisor displays the result as a pass or a failure. You can fix warnings or failures by using the Model Advisor recommended settings.



## Model configuration checks

Use the checks in this folder to prepare your model for compatibility with HDL code generation. This folder contains checks that verify whether model parameters are HDL-compatible, whether your design contains algebraic loops, and so on.

| Check Name                                                                  | Description                                                                                           |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| "Check for model parameters suited for HDL code generation" on page 38-5    | Check for model parameters set up for HDL code generation.                                            |
| "Check model for foreign characters"                                        | Search the model for unresolved library links, where the specified library block cannot be found.     |
| "Check for global reset setting for Xilinx and Altera devices" on page 38-7 | Check asynchronous reset setting for Altera devices and synchronous reset setting for Xilinx devices. |
| "Check inline configurations setting" on page 38-8                          | Check whether you have <b>InlineConfigurations</b> enabled.                                           |
| "Check algebraic loops" on page 38-9                                        | Check model for algebraic loops.                                                                      |
| "Check for visualization settings" on page 38-10                            | Check model for display settings: port data types and sample time color coding.                       |
| "Check delay balancing setting" on page 38-11                               | Check Balance Delays is enabled.                                                                      |

**Note** If you use the Model Advisor, you see the "Check model for foreign characters" in the **Simulink** folder.

## Checks for ports and subsystems

This folder contains checks that verify whether ports and subsystems in your model have settings that are compatible for HDL code generation. The checks include whether you have a valid top-level DUT Subsystem and whether you have specified an initial condition for Enabled Subsystem and Triggered Subsystem blocks.

| Check Name                                                                   | Description                                                                   |
|------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| "Check for invalid top level subsystem" on page 38-13                        | Check for subsystems that cannot be at the top level for HDL code generation. |
| "Check initial conditions of enabled and triggered subsystems" on page 38-14 | Check for initial condition of enabled and triggered subsystems.              |

## Checks for blocks and block settings

These checks verify whether blocks in your model are supported for HDL code generation, and whether the supported blocks have HDL-compatible settings. The checks include whether source blocks in your model have a continuous sample time and whether Stateflow Charts and MATLAB Function blocks have HDL-compatible settings, and so on.

| Check Name                                                               | Description                                                                                                                                       |
|--------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| “Check for infinite and continuous sample time sources” on page 38-16    | Check source blocks with continuous sample time.                                                                                                  |
| “Check for unsupported blocks” on page 38-17                             | Check for unsupported blocks for HDL code generation.                                                                                             |
| “Check for large matrix operations” on page 38-18                        | Check for large matrix operations.                                                                                                                |
| “Identify unconnected lines, input ports, and output ports”              | Check for unconnected lines or ports.                                                                                                             |
| “Identify disabled library links”                                        | Search model for disabled library links.                                                                                                          |
| “Identify unresolved library links”                                      | Search the model for unresolved library links, where the specified library block cannot be found.                                                 |
| “Check for MATLAB Function block settings” on page 38-19                 | Check HDL compatible settings for MATLAB Function blocks.                                                                                         |
| “Check for Stateflow chart settings” on page 38-20                       | Check HDL compatible settings for Stateflow Chart blocks.                                                                                         |
| “Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition” | Identify Delay, Unit Delay, or Zero-Order Hold blocks that are used for rate transition. Replace these blocks with actual Rate Transition blocks. |
| “Check for blocks that have nonzero output latency” on page 38-22        | Check for blocks that have nonzero output latency with fixed point and native floating point.                                                     |
| “Check for unsupported storage class for signal objects” on page 38-23   | Check whether signal object storage class is 'ExportedGlobal' or 'ImportedExtern' or 'ImportedExternPointer'                                      |

**Note** If you use the Model Advisor, you see the “Identify unconnected lines, input ports, and output ports”, “Identify disabled library links”, “Identify unresolved library links”, and “Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition” in the **Simulink** folder.

## Native Floating Point checks

These checks verify whether the model is compatible for HDL code generation in Native Floating Point mode. The checks include whether the blocks in your Simulink model are supported for HDL code generation with Native Floating Point, and whether the model uses single data types, and so on. Native floating-point support in HDL Coder generates target-independent HDL code from your single-precision floating-point model. For more information, see “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103.

| Check Name                                                                         | Description                               |
|------------------------------------------------------------------------------------|-------------------------------------------|
| “Check for single datatypes in the model” on page 38-25                            | Check for single data types in the model. |
| “Check for double datatypes in the model with Native Floating Point” on page 38-26 | Check for double data types in the model. |

| Check Name                                                                       | Description                                                                           |
|----------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| "Check for Data Type Conversion blocks with incompatible settings" on page 38-27 | Check conversion mode of Data Type Conversion blocks.                                 |
| "Check for HDL Reciprocal block usage" on page 38-28                             | Check HDL Reciprocal blocks are not using floating point types.                       |
| "Check for Relational Operator block usage" on page 38-29                        | Check Relational Operator blocks which use floating point types have boolean outputs. |
| "Check for unsupported blocks with Native Floating Point" on page 38-30          | Check for unsupported blocks with native floating-point.                              |
| "Check blocks with nonzero ulp error" on page 38-31                              | Check for blocks that have nonzero ulp error with native floating-point.              |

## industry standard checks

These checks verify whether your Simulink model conforms to the industry-standard rules. Industry-standard rules recommend using certain HDL coding guidelines. When generating code, HDL Coder displays an HDL coding standard report that shows how well the generated code adheres to the industry-standard guidelines.

| Check Name                                             | Description                                                                    |
|--------------------------------------------------------|--------------------------------------------------------------------------------|
| "Check VHDL file extension" on page 38-33              | Check file extensions of VHDL files containing entities.                       |
| "Check naming conventions" on page 38-34               | Check standard keywords used by EDA tools.                                     |
| "Check top-level subsystem/port names" on page 38-35   | Check top-level module/entity and port names.                                  |
| "Check module/entity names" on page 38-36              | Check module/entity names.                                                     |
| "Check signal and port names" on page 38-37            | Check signal and port name lengths.                                            |
| "Check package file names" on page 38-38               | Check file name containing packages.                                           |
| "Check generics" on page 38-39                         | Check generics at top-level subsystem.                                         |
| "Check clock, reset, and enable signals" on page 38-40 | Check naming convention for clock, reset, and enable signals.                  |
| "Check architecture name" on page 38-41                | Check VHDL architecture name in the generated HDL code.                        |
| "Check entity and architecture" on page 38-42          | Check whether the VHDL entity and architecture are described in the same file. |
| "Check clock settings" on page 38-43                   | Check constraints on clock signals.                                            |

For more information, see:

- "HDL Coding Standards" on page 26-4
- "Basic Coding Practices" on page 26-9
- "RTL Description Techniques" on page 26-18
- "RTL Design Methodology Guidelines" on page 26-41

**See Also**

`hdlmodelchecker`

**More About**

- “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2
- “Run Model Advisor Checks for HDL Coder” on page 39-6



# **Hardware-Software Codesign**



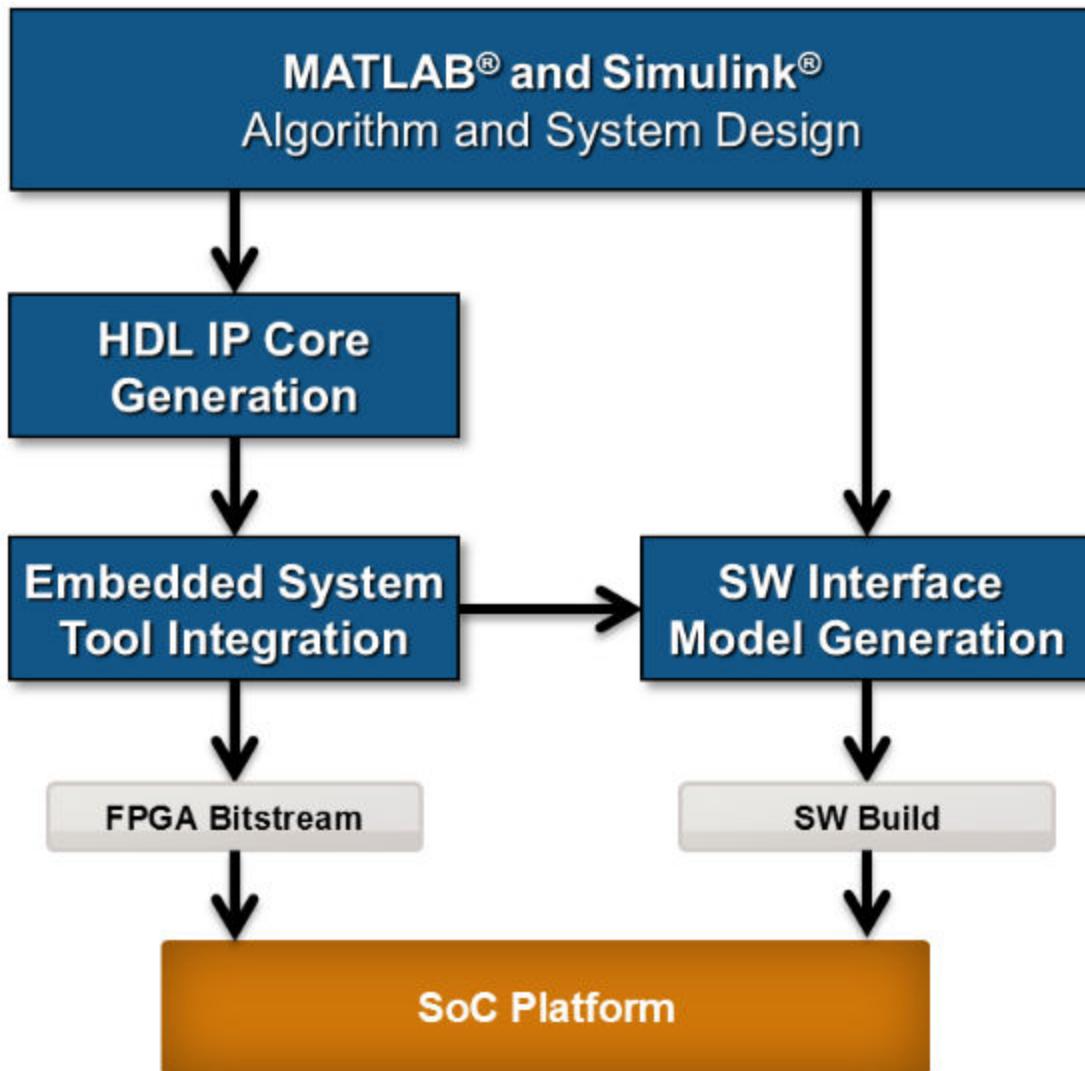
# Hardware-Software Co-Design Basics

---

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Speedgoat FPGA Support with HDL Workflow Advisor” on page 40-8
- “Custom IP Core Generation” on page 40-10
- “Custom IP Core Report” on page 40-13
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-19
- “Processor and FPGA Synchronization” on page 40-23
- “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 40-25
- “IP Caching for Faster Reference Design Synthesis” on page 40-29
- “Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows” on page 40-34
- “Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface” on page 40-45
- “Program Target FPGA Boards or SoC Devices” on page 40-49
- “Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core” on page 40-53
- “Create Software Interface Script to Control and Rapidly Prototype HDL IP Core” on page 40-60
- “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65
- “Getting Started with Targeting Zynq UltraScale+ MPSoC Platform” on page 40-84
- “Getting Started with Targeting Intel SoC Devices” on page 40-104
- “Getting Started with Targeting Intel Quartus Pro based Devices” on page 40-122
- “Save Target Hardware Settings in Model” on page 40-137
- “Using IP Core Generation Workflow from MATLAB: LED Blinking” on page 40-143
- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705” on page 40-153
- “IP Core Generation Workflow Without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit” on page 40-162
- “IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705” on page 40-172

## Hardware-Software Co-Design Workflow for SoC Platforms

The HDL Coder hardware-software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Zynq-7000 platform or Intel SoC platform. You can explore the best ways to partition and deploy your design by iterating through the following workflow.



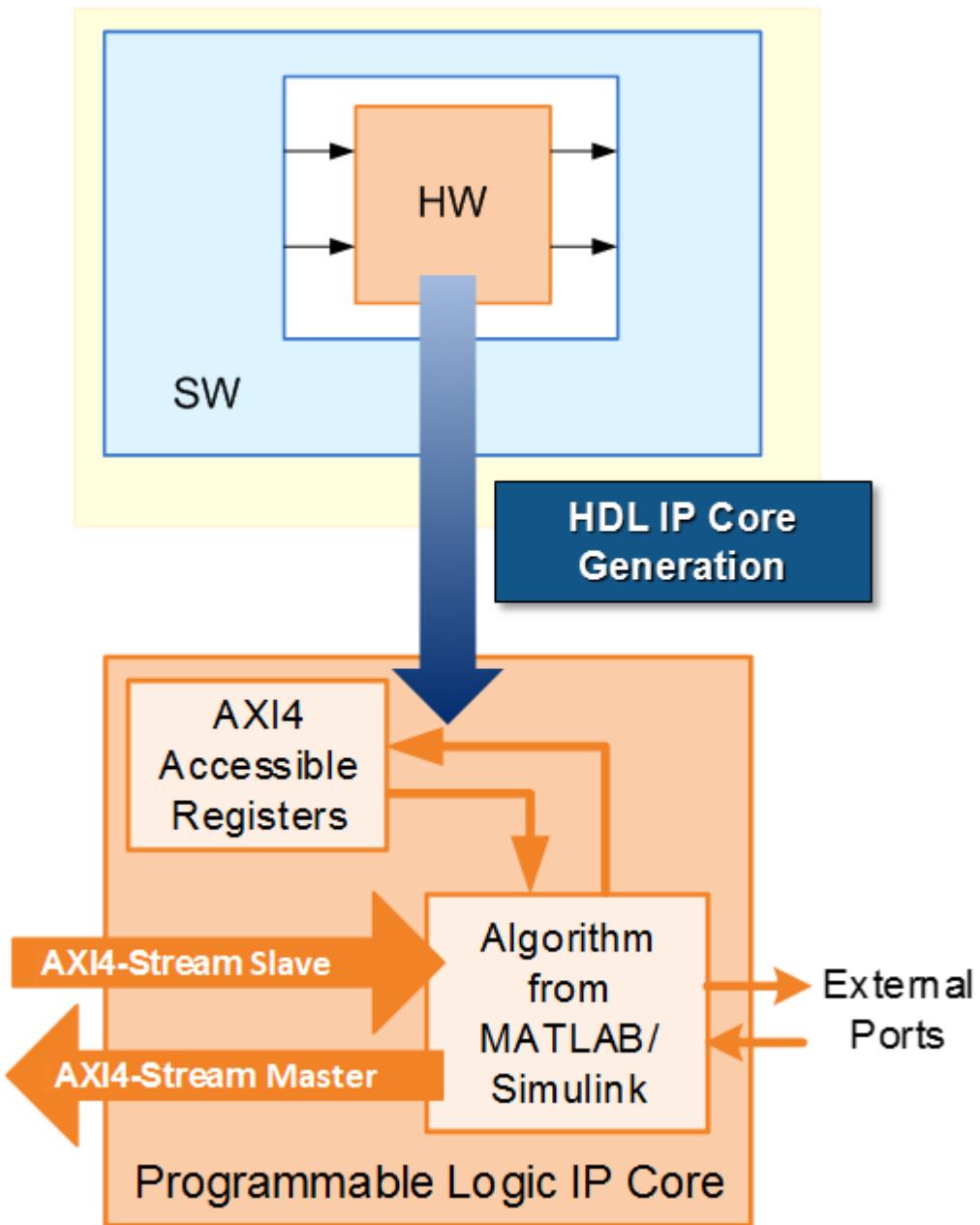
- 1 *MATLAB and Simulink Algorithm and System Design:* You begin by implementing your design in MATLAB or Simulink. When the design behavior meets your requirements, decide how to partition your design: which parts you want to run in hardware, and which parts you want to run in embedded software.

The part of the design that you want to run in hardware must use MATLAB syntax or Simulink blocks that are supported and configured for HDL code generation. See:

- “MATLAB Algorithm Design”
- “Model and Architecture Design”

**2** *HDL IP Core Generation:* Enclose the hardware part of your design in an atomic Subsystem block or MATLAB function, and use the HDL Workflow Advisor to define and generate an HDL IP core.

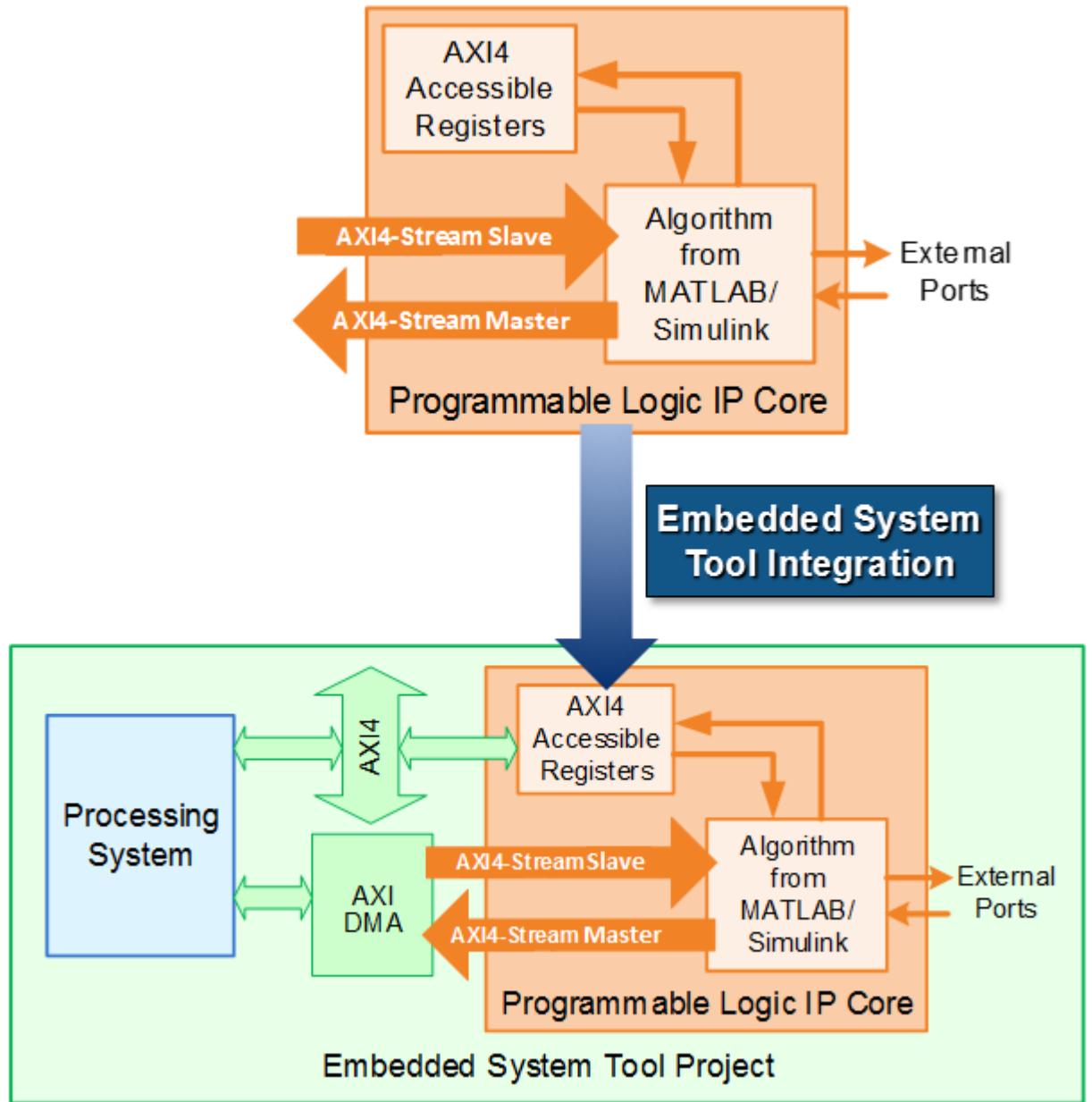
The following diagram shows a design that has been partitioned into a hardware part, in orange, and software part, in blue. HDL IP core generation creates an IP core from the hardware part of the model. The IP core includes hardware interface components such as AXI4 accessible registers, AXI4 or AXI4-Lite interfaces, AXI4-Stream or AXI4-Stream Video interfaces, AXI4 Master interfaces, and external ports.



- 3 *Embedded System Tool Integration:* As part of the HDL Workflow Advisor IP core generation workflow, you insert your generated IP core into a *reference design*, and generate an FPGA bitstream for the SoC hardware.

The *reference design* is a predefined embedded system integration project. It contains all elements the Intel or Xilinx software needs to deploy your design to the SoC platform, except for the custom IP core and embedded software that you generate.

The following diagram illustrates the relationship between the reference design, in green, and the generated IP core, in orange.

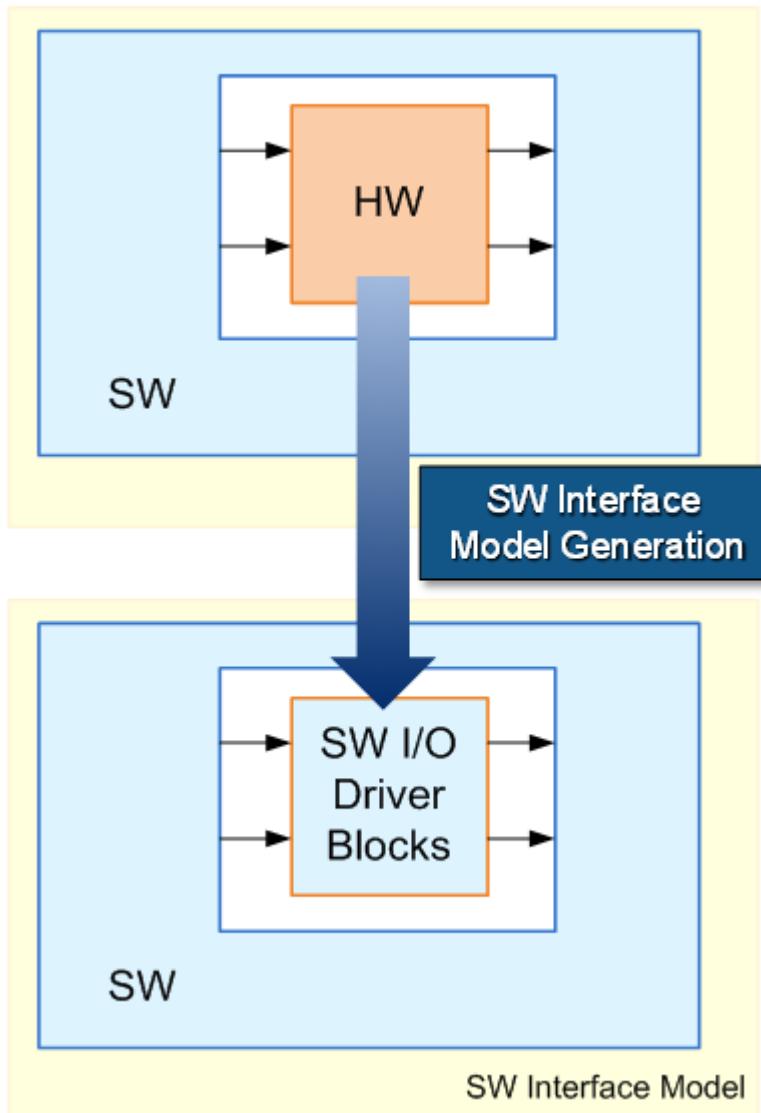


- 4 SW Interface Generation** (requires a Simulink license and Embedded Coder license): In the HDL Workflow Advisor, after you generate the IP core and insert it into the reference design, you can optionally generate a software interface model and a software interface script. The software interface model is your original model with AXI driver blocks replacing the hardware part. The script is a MATLAB file that is generated based on the reference design and Target platform interface table settings. It contains commands that enable you to connect to the target hardware, and to write to or read from the generated IP core by using AXI driver blocks.

If you have an Embedded Coder license, you can automatically generate the software interface model and script, generate embedded code from it, and build and run the executable on the Linux kernel on the ARM® processor. The generated embedded software includes AXI driver code generated from the AXI driver blocks that controls the HDL IP core.

If you do not have an Embedded Coder license or Simulink license, you can write the embedded software and manually build it for the ARM processor. See “Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core” on page 40-53

The following diagram shows the difference between the original model and the software interface model.



- 5 *SoC Platform and External Mode PIL:* Using the HDL Workflow Advisor, you program your FPGA bitstream to the SoC platform. You can then run the software interface model in external mode, or processor-in-the-loop (PIL) mode, to test your deployed design.

If your deployed design does not meet your design requirements, you can repeat the workflow with a modified model, or a different hardware-software partition.

## See Also

### Related Examples

- “Xilinx Zynq Platform”
- “Intel SoC Devices”

## Speedgoat FPGA Support with HDL Workflow Advisor

Use Simulink Real-Time and HDL Coder to implement Simulink algorithms and configure I/O functionality on Speedgoat Simulink-Programmable I/O modules. For an example that shows the development workflow for FPGA I/O modules, see “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63.

When you open the HDL Workflow Advisor in HDL Coder and run the **Simulink Real-Time** **FPGA I/O** workflow, you generate a Simulink Real-Time interface subsystem. The subsystem mask controls the block parameters. Do not edit the parameters directly. The FPGA I/O board block descriptions are for informational purposes only.

## Speedgoat Simulink-Programmable I/O Module Support

Speedgoat Simulink-Programmable I/O modules are part of Speedgoat target computer systems. To run the **Simulink Real-Time** **FPGA I/O** workflow, install the Speedgoat Library and the Speedgoat HDL Coder Integration Packages. You can then choose the **Target platform** and run the workflow to generate a Simulink Real-Time interface subsystem. To see the documentation for the integration packages, enter this command at the MATLAB command prompt.

```
speedgoat.hdlc.doc
```

| To learn about                                                                                                                                 | See links                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| The integration packages and how you can install them.                                                                                         | See Speedgoat - HDL Coder Integration Packages.                  |
| Speedgoat I/O modules that are supported with the HDL Workflow Advisor.                                                                        | See Speedgoat Real-Time FPGA Application Support from HDL Coder. |
| Applications and use cases                                                                                                                     | See Common Use Cases and Applications.                           |
| Supported interfaces for various types of I/O connectivity and protocols as well as fundamental functionality such as PCIe read/write and DMA. | See Supported Interfaces.                                        |
| Provided examples for all supported I/O modules and functionality                                                                              | See Speedgoat I/O Examples.                                      |

## Prepare for FPGA Workflow

To work with FPGAs in the Simulink Real-Time environment, install:

- HDL Coder and Simulink Real-Time.
- Xilinx design tools with specific tool and version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”. You must also set up the path to the tool by using the `hdlsetuptoolpath` function.
- Speedgoat Library and the Speedgoat HDL Coder Integration Packages.
- Speedgoat FPGA I/O module in the Speedgoat target machine.

You can use the workflow in HDL Coder to generate HDL code for your FPGA target device.

## See Also

### Related Examples

- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63

### More About

- “HDL Language Support and Supported Third-Party Tools and Hardware”
- “Tool Setup”

### External Websites

- [www.speedgoat.com](http://www.speedgoat.com)

## Custom IP Core Generation

### In this section...

- “Custom IP Core Architectures” on page 40-10
- “Target Platform Interfaces” on page 40-10
- “Processor/FPGA Synchronization” on page 40-11
- “Custom IP Core Generated Files” on page 40-11
- “Restrictions” on page 40-11

Using the HDL Workflow Advisor, you can generate a custom IP core from a model or algorithm. The generated IP core is sharable and reusable. You can integrate it with a larger design by adding it in an embedded system integration environment, such as Intel Qsys, Xilinx EDK, or Xilinx IP Integrator.

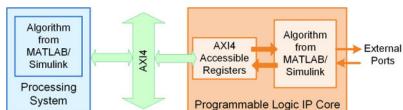
To learn how to generate a custom IP core, see:

- “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-19
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-35

## Custom IP Core Architectures

You can generate an IP core:

- With an AXI4 or AXI4-Lite interface.



- With an AXI4 or AXI4-Lite interface and AXI4-Stream Video interfaces.



- Without any AXI4 or AXI4-Lite interfaces. To learn more, see “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-19.

The *Algorithm from MATLAB/Simulink* block represents your DUT. HDL Coder generates the rest of the IP core based on your target platform interface settings and processor/FPGA synchronization mode.

## Target Platform Interfaces

You can map each port in your DUT to one of the following target platform interfaces in the IP core:

- AXI4-Lite: Use this slave interface to access control registers or for lightweight data transfer. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.

- AXI4: Use this slave interface to connect to components that support burst data transmission. HDL Coder generates memory-mapped registers and allocates address offsets for the ports you map to this interface.

**Note** Interfaces AXI4 and AXI4-Lite are also referred to as AXI4 slave interfaces. In the generated HDL IP core, you can have either AXI4 or AXI4-Lite interface but not both interfaces.

- AXI4-Stream Video: Use this interface to send or receive a 32-bit scalar video data stream.
- External ports: Use external ports to connect to FPGA external IO pins, or to other IP cores with external ports.

To learn more about the AXI4, AXI4-Lite and AXI4-Stream Video protocols, refer to your target hardware documentation.

## Processor/FPGA Synchronization

HDL Coder generates synchronization logic in the IP core based on the processor/FPGA synchronization mode you choose.

When generating a custom IP core, the following processor/FPGA synchronization options are available:

- Free running (default)
- Coprocessing – blocking

To learn more about the processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 40-23.

## Custom IP Core Generated Files

After you generate a custom IP core, the IP core files are in the `ipcore` folder within your project folder. In the HDL Workflow Advisor, you can view the IP core folder name in the **IP core folder** field of the **HDL Code Generation > Generate RTL Code and IP Core** task.

The IP core folder contains the following generated files:

- IP core definition files.
- HDL source files (.vhd or .v).
- A C header file with the register address map.
- (Optional) An HTML report with instructions for using the core and integrating the IP core in your embedded system project.

## Restrictions

IP Core Generation workflow does not support :

- **RAM Architecture** set to Generic RAM without clock enable.
- Using different clocks for the IP core and the AXI interface. The `IPCore_Clk` and `AXILite_ACLK` must be synchronous and connected to the same clock source. The `IPCore_RESETN` and `AXILite_RESETN` must be connected to the same reset source. See “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 40-25.

## See Also

### Related Examples

- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705” on page 40-153
- “IP Core Generation Workflow Without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit” on page 40-162

### More About

- “Multirate IP Core Generation” on page 41-35

# Custom IP Core Report

## In this section...

- “Summary” on page 40-13
- “Target Interface Configuration” on page 40-13
- “Register Address Mapping” on page 40-14
- “IP Core User Guide” on page 40-15
- “IP Core File List” on page 40-18

You generate an HTML custom IP core report by default when you generate a custom IP core. The report describes the behavior and contents of the generated custom IP core.

## Summary

The Summary section shows your coder settings when you generated the custom IP core.

The following figure is an example of a Summary section.

### Summary

|                       |                                             |
|-----------------------|---------------------------------------------|
| IP core name          | DUT_ip                                      |
| IP core version       | 1.0                                         |
| IP core folder        | <a href="#">hdl_prj\ipcore\DU T_ip_v1_0</a> |
| Target platform       | Arrow SoCKit development board              |
| Target tool           | Altera QUARTUS II                           |
| Target language       | Verilog                                     |
| Reference Design      | Default system                              |
| Model                 | <a href="#">axi4_vec</a>                    |
| Model version         | 1.91                                        |
| HDL Coder version     | 3.10                                        |
| IP core generated on  | 10-Dec-2016 21:06:26                        |
| IP core generated for | <a href="#">DUT</a>                         |

## Target Interface Configuration

The Target Interface Configuration section shows how your DUT ports map to the target hardware interface and the processor/FPGA synchronization mode.

The following figure is an example of a Target Interface Configuration section.

## Target Interface Configuration

You chose the following target interface configuration for [axi4\\_vec](#) :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

| Port Name | Port Type | Data Type    | Target Platform Interfaces | Bit Range / Address / FPGA Pin |
|-----------|-----------|--------------|----------------------------|--------------------------------|
| In1       | Import    | uint8 (3)    | AXI4                       | x"100"                         |
| In2       | Import    | uint16 (100) | AXI4                       | x"200"                         |
| In3       | Import    | single       | AXI4                       | x"114"                         |
| In4       | Import    | uint32       | AXI4                       | x"118"                         |
| Out1      | Outport   | uint8 (3)    | AXI4                       | x"160"                         |
| Out2      | Outport   | uint16 (100) | AXI4                       | x"A00"                         |
| Out3      | Outport   | single       | AXI4                       | x"11C"                         |
| Out4      | Outport   | uint32       | AXI4                       | x"120"                         |

To learn more about processor/FPGA synchronization modes, see “Processor and FPGA Synchronization” on page 40-23.

To learn more about target platform interfaces, see “Custom IP Core Generation” on page 40-10.

## Register Address Mapping

The Register Address Mapping section shows the address offsets for AXI4-Lite bus accessible registers in your custom IP core, and the name of the C header file that contains the same address offsets.

The following figure is an example of a Register Address Mapping section.

## Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

| Register Name | Address Offset | Description                                                                     |
|---------------|----------------|---------------------------------------------------------------------------------|
| IPCore_Reset  | 0x0            | write 0x1 to bit 0 to reset IP core                                             |
| IPCore_Enable | 0x4            | enabled (by default) when bit 0 is 0x1                                          |
| In1_Data      | 0x100          | data register for Import In1, vector with 3 elements, address ends at 0x108     |
| In1_Strobe    | 0x110          | strobe register for port In1                                                    |
| In3_Data      | 0x114          | data register for Import In3                                                    |
| In4_Data      | 0x118          | data register for Import In4                                                    |
| Out3_Data     | 0x11C          | data register for Outport Out3                                                  |
| Out4_Data     | 0x120          | data register for Outport Out4                                                  |
| Out1_Data     | 0x160          | data register for Outport Out1, vector with 3 elements, address ends at 0x168   |
| Out1_Strobe   | 0x170          | strobe register for port Out1                                                   |
| In2_Data      | 0x200          | data register for Import In2, vector with 100 elements, address ends at 0x38C   |
| In2_Strobe    | 0x400          | strobe register for port In2                                                    |
| Out2_Data     | 0xA00          | data register for Outport Out2, vector with 100 elements, address ends at 0xB8C |
| Out2_Strobe   | 0xC00          | strobe register for port Out2                                                   |

The register address mapping is also in the following C header file for you to use when programming the processor:  
[include\DUT\\_ip\\_addr.h](#)

The IP core name is appended to the register names to avoid name conflicts.

## IP Core User Guide

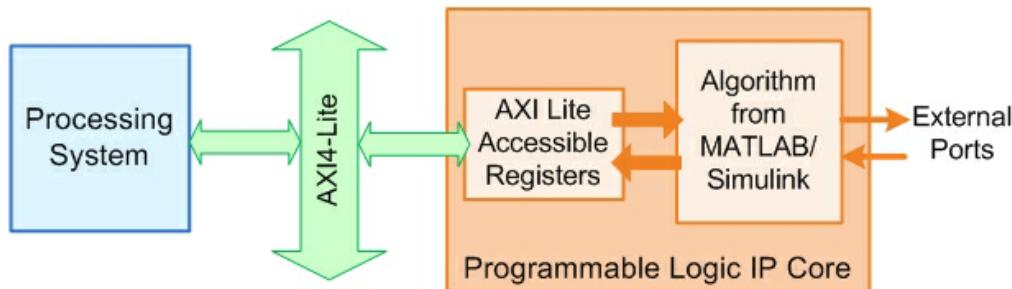
The IP Core User Guide section gives a high-level overview of the system architecture, describes the processor and FPGA synchronization mode, and gives instructions for integrating the IP core in your embedded system integration environment.

The following figure is an example of an IP Core User Guide system architecture description.

### Theory of Operation

This IP core is designed to be connected to an embedded processor with an **AXI4-Lite bus**. The processor acts as bus master, and the IP core acts as slave. By accessing the generated registers via the AXI4-Lite bus, the processor can control the IP core, and read and write data from and to the IP core.

For example, to reset the IP core, write 0x1 to the bit 0 of `IPCore_Reset` register. To enable or disable the IP core, write 0x1 or 0x0 to the `IPCore_Enable` register. To access the data ports of the MATLAB/Simulink algorithm, read or write to the associated data registers.

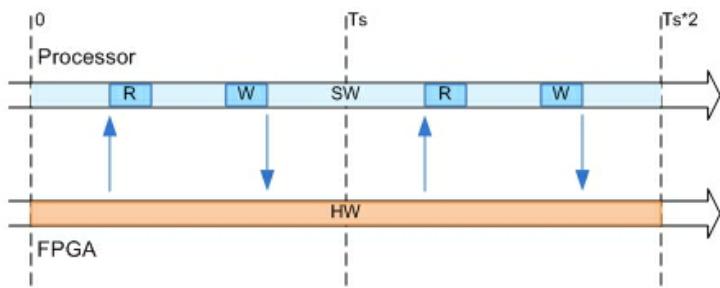


This IP core also supports the **External Port** interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Xilinx EDK environment.

The following figure is an example of a processor/FPGA synchronization description.

### Processor/FPGA Synchronization

The **Free running** mode means there is no explicit synchronization between embedded processor software execution (SW) and the IP core (HW). SW and HW runs independently. The data written from the processor to IP core takes effect immediately, and the data read from the IP core is the latest data available on the IP core output ports.

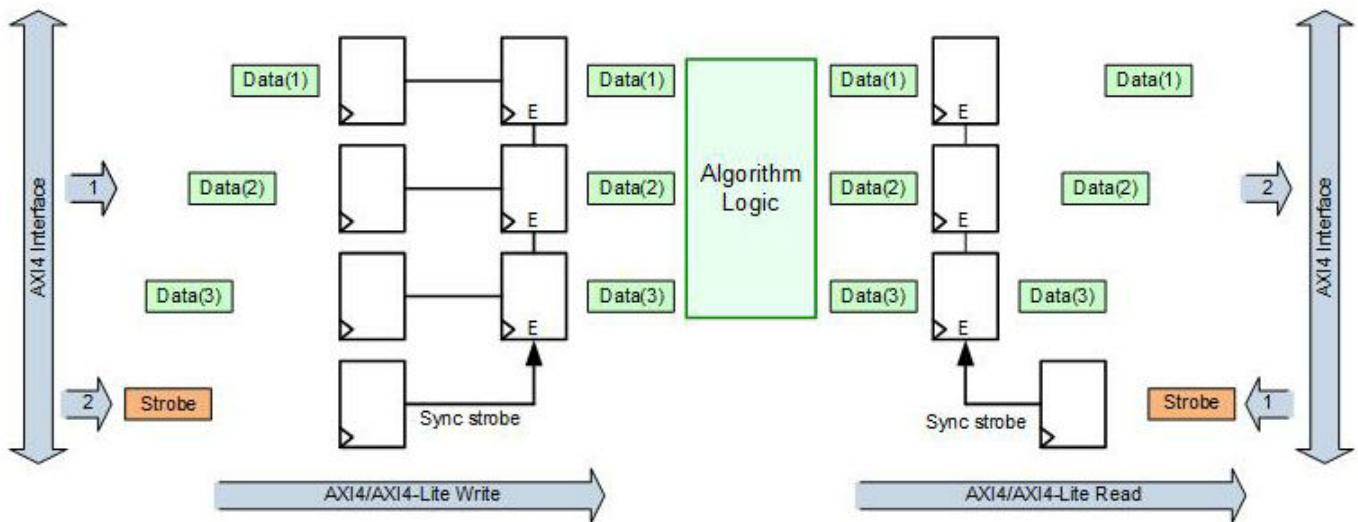


If you use vector data signals at the DUT interface, the IP core report displays this section that shows how the code generator synchronizes vector data across the AXI4 interface.

## Vector Data Read/Write with Strobe Synchronization

All the elements of vector data are treated as synchronous to the IP core algorithm logic. Additional strobe registers added for each vector input and output port maintain this synchronization across multiple sequential AXI4 reads/writes. For input ports, the strobe register controls the enables on a set of shadow registers, allowing the IP core logic to see all the updated vector elements simultaneously. For output ports, the strobe register controls the synchronous capturing of vector data to be read.

To read a vector data port, first write the strobe address with 0x1, then read each desired data element from corresponding address range. To write a vector data port, first write each desired data element, then write 0x1 to the strobe address to complete the transaction.



The following figure is an example of instructions for integrating the IP core into your embedded system integration environment on the Xilinx platform. If you are targeting an Altera platform, the report displays similar instructions for integrating the IP core into the Altera Qsys environment.

### EDK Environment Integration

This IP Core is generated for the Xilinx EDK environment. The following steps are an example showing how to add the IP core into the EDK environment:

1. Copy the IP core folder into the "pcores" folder in your Xilinx Platform Studio (XPS) project. This step adds the IP core into the XPS project user library.
2. In the XPS project, find the IP core in the user library and add the IP core to the design.
3. Connect the S\_AXI port of the IP core to the embedded processor's AXI master port.
4. Connect the clock and reset ports of the IP core to the global clock and reset signals.
5. Assign a base address for the IP core.
6. Connect external ports and add FPGA pin assignment constraints.
7. Generate FPGA bitstream and download the bitstream to target device.

## IP Core File List

The IP Core File List section lists the files and file folders that comprise your custom IP core.

The following figure is an example of an IP core file list.

### IP Core File List

The IP core folder is located at:

[hdl\\_prj\ipcore\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地 v1\\_00\\_a](#)

Following files are generated under this folder:

#### IP core definition files

[data\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地 v2\\_1\\_0.mpd](#)

[data\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地 v2\\_1\\_0.pao](#)

#### IP core report

[doc\hdlcoder\\_led\\_blinking\\_ip\\_core\\_report.html](#)

#### IP core HDL source files

[hdl\vhdl\led\\_counter\\_pkg.vhd](#)

[hdl\vhdl\led\\_counter.vhd](#)

[hdl\vhdl\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地 dut.vhd](#)

[hdl\vhdl\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地 axi\\_lite\\_module.vhd](#)

[hdl\vhdl\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地 addr\\_decoder.vhd](#)

[hdl\vhdl\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地 axi\\_lite.vhd](#)

[hdl\vhdl\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地.vhd](#)

#### IP core C header file

[include\hdlcoder\\_led\\_blinking\\_led\\_counter\\_pc当地\\_addr.h](#)

## See Also

### More About

- “Custom IP Core Generation” on page 40-10
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Multirate IP Core Generation” on page 41-35

# Generate Board-Independent HDL IP Core from Simulink Model

## In this section...

- “Generate Board-Independent IP Core” on page 40-19
- “IP Core without AXI4 Slave Interfaces” on page 40-21
- “Requirements and Limitations for IP Core Generation” on page 40-22

When you open the HDL Workflow Advisor and run the IP Core Generation workflow for your Simulink model, you can specify a generic Xilinx platform or a generic Intel platform. The workflow then generates a generic IP core that you can integrate into any target platform of your choice. For IP core integration, define and register a custom reference design for your target board by using the `hdlcoder.ReferenceDesign` class. To learn more, see:

- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

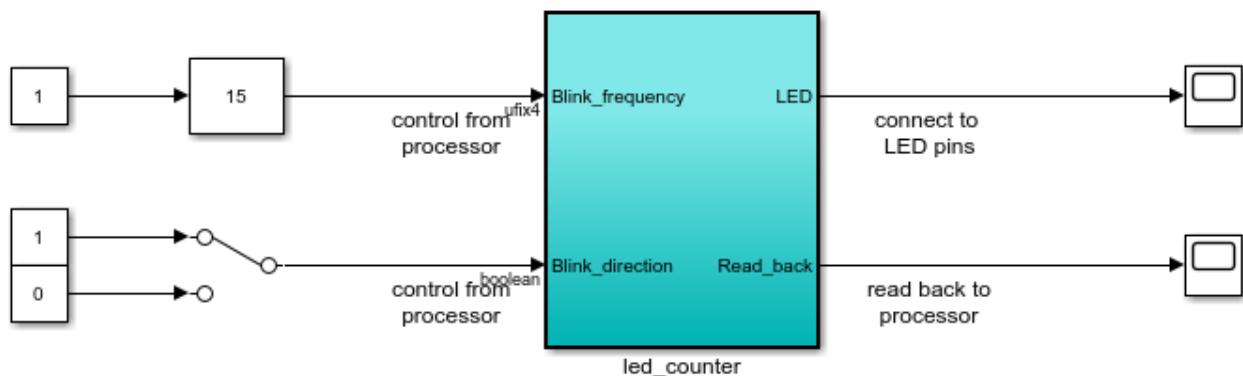
## Generate Board-Independent IP Core

To generate a board-independent custom IP core to use in an embedded system integration environment, such as Intel Qsys, Xilinx EDK, or Xilinx IP Integrator:

- 1 Select your DUT in your Simulink model and open the HDL Workflow Advisor. For example, open the model `hdlcoder_led_blinking`.

```
open_system('hdlcoder_led_blinking')
```

## Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:  
`hdladvisor('hdlcoder_led_blinking/led_counter')`

**Launch HDL Workflow Advisor**

**Run Demo**

Copyright 2012 The MathWorks, Inc.

- 2 Set the path to the installed synthesis tool for the target device by using the `hdlsetuptoolpath` function. For example, if your synthesis tool is Xilinx Vivado, enter this command:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
    'C:\Xilinx\Vivado\2018.2\bin\vivado.bat');
```

See “HDL Language Support and Supported Third-Party Tools and Hardware” for latest supported version of the synthesis tool.

- 3 Open the HDL Workflow Advisor for the DUT Subsystem. For the LED blinking model, the `led_counter` Subsystem is the DUT. In the **Set Target > Set Target Device and Synthesis Tool** task:

- For **Target workflow**, select IP Core Generation.
- For **Target platform**, depending on the synthesis tool and device that you are targeting, select Generic Altera Platform or Generic Xilinx Platform. Click **Run This Task**.

- 4 In the **Set Target > Set Target Interface** task, select a **Target Platform Interface** for each port, then click **Apply**. You can map each DUT port to one of AXI4-Lite, AXI4, AXI4-Stream, AXI4-Stream Video, or External Port interfaces. To learn more about these interfaces, see “Target Platform Interfaces” on page 40-10.

You can also map the ports to multiple target platform interfaces. To learn more, see “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 41-17.

If you do not want to map the DUT ports to AXI4 slave interfaces, you can map them to External Port interfaces.

| Target platform interface table |           |           |                            |                                |
|---------------------------------|-----------|-----------|----------------------------|--------------------------------|
| Port Name                       | Port Type | Data Type | Target Platform Interfaces | Bit Range / Address / FPGA Pin |
| Blink_frequency                 | Import    | ufix4     | External Port              |                                |
| Blink_direction                 | Import    | boolean   | External Port              |                                |
| LED                             | Outport   | uint8     | External Port              |                                |
| Read_back                       | Outport   | uint8     | External Port              |                                |

- 5 Expand the **Set Code Generation Options** task. Right-click the **Set Optimization Options** task and select **Run to Selected Task**.

- 6 In the **HDL Code Generation > Generate RTL Code and IP Core** task, you can specify:

- Whether you want to connect the DUT IP core to multiple AXI Master interfaces. By default, the **AXI4 Slave ID Width** value is 12, which enables you to connect the HDL IP core to one AXI Master interface. To connect the DUT IP core to multiple AXI Master interfaces, you may want to increase the **AXI4 Slave ID Width**. When you run this task, this setting is saved on the DUT as the HDL block property **AXI4SlaveIDWidth**.

To learn more, see “Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface” on page 40-45.

- Whether you want to generate the default AXI4 slave interface. By default, HDL Coder generates AXI4 slave interfaces for signals such as clock, reset, ready, timestamp, and so on.

If you do not want to generate any AXI4 slave interfaces, clear the **Generate default AXI4 slave interface** check box. Click **Run This Task**.

---

**Note** If you mapped any of the DUT ports to AXI4 slave interfaces in the **Set Target Interface** task, even if you clear this check box, the code generator ignores this setting and maps the ports to AXI4 slave interfaces.

When you clear the check box and run the task, the code generator saves this setting on the DUT Subsystem as the HDL block property **GenerateDefaultAXI4Slave**.

After running the task, HDL Coder generates the IP core files in the output folder shown the **IP core folder** field, including the HTML documentation. To view the IP core report, click the link in the message window.

## IP Core without AXI4 Slave Interfaces

When you run the **IP Core Generation** workflow, you can also generate an HDL IP core without any AXI4 slave interfaces in your reference design.

To run this workflow, open the HDL Workflow Advisor, specify **Generic Xilinx Platform** or **Generic Altera Platform** as the target platform, and make sure that you map the DUT ports to only External Port, or AXI4-Stream interface with TLAST mapping. In addition, when you generate the HDL IP core, in the **Generate RTL Code and IP Core** task, clear the **Generate default AXI4 slave interface** check box, and then select **Run This Task**.

Use this capability when:

- You do not want to tune the IP core parameters by using the AXI4 slave interfaces.
- You want to create a custom reference design without AXI4 slave interfaces, such as standalone FPGA boards.

In addition, avoiding generation of the AXI4 slave interfaces in such cases reduces hardware resource usage and design complexity.

---

**Note** External IO and internal IO interfaces connect your HDL IP core to other existing IPs in your custom reference design. To define these interfaces, you use the `addInternalIOInterface` and `addExternalIOInterface` methods of the `hdlcoder.ReferenceDesign` class.

To integrate the HDL IP core, you can create a custom reference design without AXI4 slave interfaces. In the custom reference design, you can only use External IO, Internal IO or AXI4-Stream interface with TLAST mapping. For examples of such reference designs, see:

- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705” on page 40-153
- “IP Core Generation Workflow Without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit” on page 40-162

When you generate an HDL IP core without AXI4 slave interfaces, certain restrictions apply. See “IP Core without AXI4 Slave Interface Restrictions” on page 40-22.

## Requirements and Limitations for IP Core Generation

### Custom IP Core Generation Limitations

- The DUT must be an atomic system.
- The same IP core cannot use both an AXI4 interface and an AXI4-Lite interface.
- The DUT cannot contain Xilinx System Generator blocks or Intel DSP Builder Advanced blocks.
- If your target language is VHDL, and your synthesis tool is Xilinx ISE or Intel Quartus Prime, the DUT cannot contain a model reference.

### AXI4-Lite Interface Restrictions

- The input and output ports must have a bit width less than or equal to 32 bits.
- The input and output ports must be scalar.

### AXI4-Stream Video Interface Restrictions

- Ports must have a 32-bit width.
- Ports must be scalar.
- You can have a maximum of one input video port and one output video port.
- The AXI4-Stream Video interface is not supported in Coprocessing – blocking mode. **Processor/FPGA synchronization** must be set to Free running mode. Coprocessing – blocking mode is not supported.

### IP Core without AXI4 Slave Interface Restrictions

- You can only map the ports to External or Internal IO interfaces, or AXI4-Stream interface with TLAST mapping. Other interfaces that require AXI4 slave interfaces such as AXI4 Master, AXI4-Stream, and AXI4-Stream Video are not supported.
- You must use the Free running mode for **Processor/FPGA synchronization**. Coprocessing – blocking mode is not supported.

## See Also

### Classes

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

## More About

- “Custom IP Core Generation” on page 40-10
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-35
- “Board and Reference Design Registration System” on page 41-39