Importing IP into SOPC Builder

6.375 Tutorial 13
March 5, 2009

Although SOPC Builder offers many pre-existing cores from which powerful systems may be built, designers often need to add custom IP into a system to probide some custom functionality. In this tutorial, you will learn how to import a custom block into SOPC Builder and connect it to an existing system.

Getting started

Before using the 6.375 toolflow you must add the course locker and run the course setup script with the following two commands.

```
% add 6.375
% source /mit/6.375/setup.csh
```

For this tutorial, we will be using a simple SRAM as our example design. You should create a working directory and checkout the SRAM example project from the course CVS repository using the following commands.

```
% mkdir tut13
% cd tut13
% cvs checkout examples/sram
% cd examples/sram
```

Before starting, take a look at the subdirectories in the `sram` project directory. Figure ?? shows the system diagram which is implemented by the example code. The important feature in the design is the Avalon Bus Slave, which serves as a kind of glue logic to connect our SRAM to the rest of the system. The slave logic will translate requests and responses from our SRAM into signals on the bus.

Examine the directory structure of `sram`. The `src` directory contains the various Bluespec sources needed for the SRAM, and the `test` directory contains a small test harness. The source code will be built in the `build` directory.

We will first compile the example Bluespec code down to Verilog, and then we will import the generated Verilog into SOPC builder. It is also possible to import synthesized gate-level descriptions into SPOC builder. To build the verilog, execute:

```
% pwd
tut13/examples/sram
% cd build
% make avalonmasterslavetester
```

This command will build both the SRAM verilog (`build/vdir/mkSmallAvalonRegisterFile.v`) and the testbench. You can verify that the code built properly by running the testbench:

```
% ./avalonmasterslavetester
PASS
```

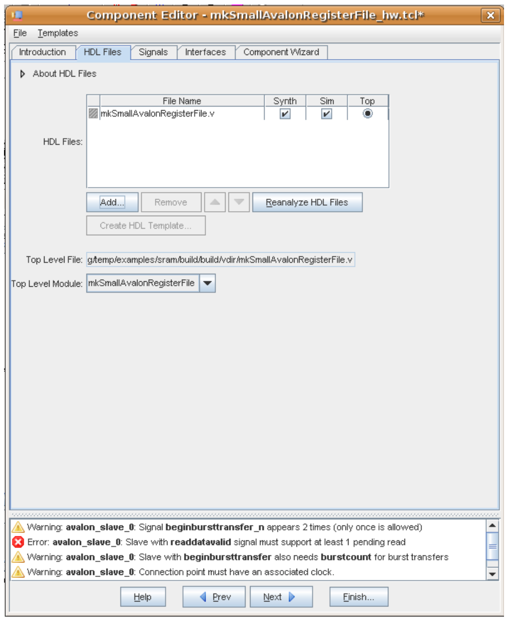
We will now import `mkSmallAvalonRegisterFile` into SOPC builder.

Importing Verilog into SOPC Builder

SOPC builder requires a Quartus II Project in order to run. Using the steps in Tutorial 10, create a new Quartus project. Open SOPC builder using the *Tools* → *SOPC Builder* menu option. This will bring up the SOPC Builder user interface, which will ask you to name the new project. Give a name and click *Continue*.

Before we start using SOPC Builder, we must first point it at the Altera IP repository. Open the *Tools* → *Options* menu. Add `/mit/6.375/tools/altera/altera` to the *I*P Search Path and click *Finish*. As you add IP to SOPC Builder, you will need to extend the search path.

Open the Component Editor using the *File* → *New Component* button. Make note of the introductory screen and then click *Next*, bringing up the HDL import tab. Click on *Add* and select `mkSmallAvalonRegisterFile.v`. The wizard will think for a little bit and then given a few warnings and errors. These are normal and will be addressed on the subsequent tabs. Your screen should look like this:



File Name	Synth	Sim	Top
mkSmallAvalonRegisterFile.v	✓	✓	●

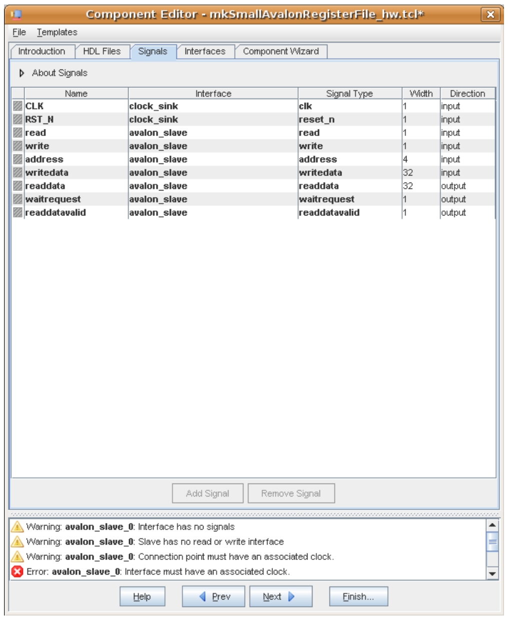
Top Level File: g/temp/examples/sram/build/build/vdir/mkSmallAvalonRegisterFile.v
Top Level Module: mkSmallAvalonRegisterFile

```
Warning: avalon_slave_0: Signal beginbursttransfer_n appears 2 times (only once is allowed)
Error: avalon_slave_0: Slave with readdatavalid signal must support at least 1 pending read
Warning: avalon_slave_0: Slave with beginbursttransfer also needs burstcount for burst transfers
Warning: avalon_slave_0: Connection point must have an associated clock.
```

Figure 1: HDL Screen

Click *Next*, bringing up the *Signals* tab. In this tab we will declare special groups of wires in our design. `mkSmallAvalonRegisterFile` has a clock input and an avalon slave interface. Use the *Interface* column to add a *new Clock Input* interface and a *new Avalon Memory Mapped Slave*. Take a look at the other options in the *Interface* column menu. The most important option, whose usage is not obvious, is *new Conduit*, which exports a wire to the top level of the SOPC design.

This enables you to tie the wire to external pins or other non-SOPC produced hardware. Assign the `mkSmallAvalonRegisterFile` signals to the appropriate *Signal Types*. When you are done, your *Component Editor* should look like this:



Name	Interface	Signal Type	Width	Direction
CLK	clock_sink	clk	1	input
RST_N	clock_sink	reset_n	1	input
read	avalon_slave	read	1	input
write	avalon_slave	write	1	input
address	avalon_slave	address	4	input
writedata	avalon_slave	writedata	32	input
readdata	avalon_slave	readdata	32	output
waitrequest	avalon_slave	waitrequest	1	output
readdatavalid	avalon_slave	readdatavalid	1	output

```
Warning: avalon_slave_0: Interface has no signals
Warning: avalon_slave_0: Slave has no read or write interface
Warning: avalon_slave_0: Connection point must have an associated clock.
Error: avalon_slave_0: Interface must have an associated clock.
```

Figure 2: Signal Screen

Click *Next*, bringing up the *Interfaces* tab. First, click the *Remove Interfaces With No Signals* button to clean up any interfaces that SOPC builder has erroneously inferred. Associate the `avalon_slave` with the `clock_sink`. SOPC uses this information to ensure that communicating interfaces are properly clocked. Set *Max pending read transactions* to 1. In principal, we could set this value to be 2, since we have some buffering capacity in the `AvalonSlave` module, but we will be conservative. Click *Next* again and then click *Finish*. Congradulations, you have just imported a module. You should be able to see the `mkSmallAvalonRegisterFile` module in the *System Contents Menu*, and you can now use it in any SOPC designs. To complete the lab, click the *Generate* button. This should ensure that the path to your ip is set properly