Altera JTAG-to-Avalon-MM Tutorial

Version 1.0

D. W. Hawkins (dwh@ovro.caltech.edu)

March 14, 2012

Contents

1 Introduction **3**

2 SOPC Builder and Qsys **5**

3 SOPC Builder Design Flow **6**
3.1 Project Creation 6
3.2 SOPC Builder Component 7
3.3 Top-Level Design 9
3.4 Synthesis 10
3.5 Simulation 12
3.5.1 SOPC Builder `test_bench` 12
3.5.2 Avalon-MM Master BFM 15
3.5.3 JTAG-to-Avalon-MM Master 17
3.6 Synthesis and Simulation Scripts 20

4 Qsys Design Flow **21**
4.1 Project Creation 21
4.2 Qsys Component 22
4.3 Top-Level Design 25
4.4 Synthesis 25
4.5 Simulation 27
4.5.1 Qsys simulation configuration 27
4.5.2 Avalon-MM Master BFM 28
4.5.3 JTAG-to-Avalon-MM Master 29
4.6 Synthesis and Simulation Scripts 31

5 Host-to-FPGA Communications **33**
5.1 System Console 33
5.2 `quartus_stp` 33
5.3 Client/Server 37

A Software Versions **40**

B Tutorial Source **41**

C Altera Tool Improvement Recommendations **42**

D Altera Documentation Web Links **45**

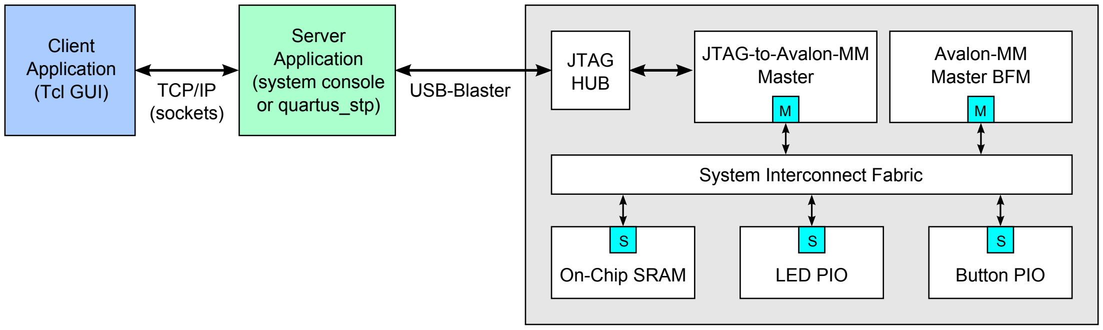


Figure 1: System block diagram for host communication to an Altera Avalon hardware design.

1 Introduction

A common question on the Altera forum is;

How can I use a USB-Blaster to communicate with my system design?

The Altera USB-Blaster interface is used by the Quartus II programmer to configure the FPGA, used by the SignalTap II logic analyzer for trace capture, and used by the NIOS II processor tools for debugging and memory inspection, so it is reasonable for users to assume that the interface can be used to communicate with their designs. This tutorial demonstrates how to implement this communications.

The target audience for this tutorial is developers new to Altera's SOPC Builder and Qsys system design tools. This tutorial improves the user experience with these tools, by providing a step-by-step walk-though of the system design in Figure 1. While many of the concepts covered will be foreign to the new user, having an example of the end-to-end system design sequence, makes reading and comprehension of the *extensive* tool and device documentation a little easier.

The tutorial shows how to create and simulate the hardware design shown on the right of Figure 1, and then how to communicate with the design. The hardware design is based solely on Altera-provided IP components. The software design uses the Altera Tcl-based tools *System Console* and `quartus_stp` for the host-to-JTAG communications, and uses a (generic) Tcl GUI for the client application. Client-to-server communications are performed using ASCII strings transported using TCP/IP (sockets); the client can be easily replaced with one written in your favorite programming language.

The tutorial walks the reader through the creation, synthesis, and simulation of SOPC and Qsys systems. Tcl scripts are provided that automate the regeneration and simulation of the systems. The tutorial points out some of the problems with the Altera-provided IP and software; in part so that the reader can avoid the problems, but also in the hope that Altera will rectify them.

If you liked this tutorial, or have feedback or suggestions on how it can be improved, please post a message to the Altera Forum thread

`http://www.alteraforum.com/forum/showthread.php?t=34787`.

Software Versions

The tutorial was written using Altera Quartus 11.1sp1 and Modelsim-ASE (*Altera Starter Edition*) 10.0c. Appendix A provides details on the operating systems tested, and differences with earlier tool versions.

Table 1: `altera_jtag_to_avalon_mm_tutorial` directory layout.

Path	Description
`doc/`	Documentation
`hdl/`	HDL source code
`tcl/`	Tcl client/server source code
Simulation	
`hdl/sopc_system/`	SOPC system simulation
`hdl/qsys_system/`	Qsys system simulation
Synthesis	
`hdl/boards/`	Hardware targets
`hdl/boards/bemicro_sdk/`	Arrow BeMicro-SDK projects
`hdl/boards/bemicro_sdk/sopc_system/`	SOPC system synthesis
`hdl/boards/bemicro_sdk/qsys_system/`	Qsys system synthesis
`hdl/boards/bemicro_sdk/share/`	Board constraints
`hdl/boards/bemicro/`	Arrow BeMicro projects
`hdl/boards/bemicro/sopc_system/`	SOPC system synthesis
`hdl/boards/bemicro/qsys_system/`	Qsys system synthesis
`hdl/boards/bemicro/share/`	Board constraints
`hdl/boards/de2/`	Terasic DE2 projects
`hdl/boards/de2/sopc_system/`	SOPC system synthesis
`hdl/boards/de2/qsys_system/`	Qsys system synthesis
`hdl/boards/de2/share/`	Board constraints

Tutorial Source Code

The tutorial zip file, `altera_jtag_to_avalon_mm_tutorial.zip` [6], unzips[1] to create the directory layout shown in Table 1. The path to the unzipped directory is referred to in this document via the variable `TUTORIAL`. For example, if you unzip the file into your Windows `c:/temp` directory, then paths in this document are referenced relative to

```
TUTORIAL = c:/temp/altera_jtag_to_avalon_mm_tutorial
```

The example hardware design described in the text targets the Arrow BeMicro-SDK Cyclone IV board, but the procedure works equivalently well on any other development board. The example source contains completed designs for the Arrow BeMicro-SDK Cyclone IV, Arrow BeMicro Cyclone III, and Terasic DE2 Cyclone II boards. The main difference between boards is in their top-level entities, which contain all pins used on each board. The Qsys or SOPC system is instantiated into the top-level entity.

[1]See Appendix B for the recommended unzip methods under Windows and Linux.

2 SOPC Builder and Qsys

Altera provides two tools for graphically building hardware systems; the classic tool *SOPC Builder*, and the new tool *Qsys*. The main difference between tools is the interconnect fabric, and the support for hierarchical designs in Qsys [2].

The *Avalon Interface Specification* consists of bus protocols and an interconnect fabric defined by Altera [1,3]. The bus protocols consist of two variants; *Avalon Memory Mapped (Avalon-MM)* and *Avalon Streaming (Avalon-ST)*. The Avalon-MM protocol is used to create systems like that shown in Figure 1, where multiple masters connect to multiple slaves, and the masters control the slaves by performing read and write accesses to addresses defined by the system *memory map*. The Avalon-ST protocol is used in data streaming applications, such as signal processing, where data sources pass data onto data sinks.

SOPC Builder is used to create systems containing Avalon-MM components and Avalon-ST components. The Avalon fabric is used to connect Avalon-MM masters to Avalon-MM slaves. SOPC Builder automates the systematic and tedious task of creating address decoding, bus arbitration, and multiplexing logic between masters and slaves. The connections between Avalon-ST sources and sinks are point-to-point, so no fabric is required.

Qsys takes a different approach to the implementation of the interconnect fabric; master and slave transactions are converted to packets, and those packets are transported through a *network-on-chip*. This approach abstracts the bus interface protocol of the masters and slaves, allowing different bus protocols to interface to the network. This allows Altera to continue to use the Avalon-MM protocol, and to add support for ARM defined bus protocols such as the Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI). For example, a Qsys system can be created with Avalon-MM masters and AXI slaves, and the interconnect fabric performs the required bus protocol translation. To create this system using SOPC Builder, you would need to create an Avalon-MM-to-AMBA bridge to *adapt* the AMBA slave before connecting it to the Avalon-MM fabric.

The following sections define the system shown in Figure 1 first using SOPC Builder and then using Qsys. The two designs allow you to contrast the two tools.

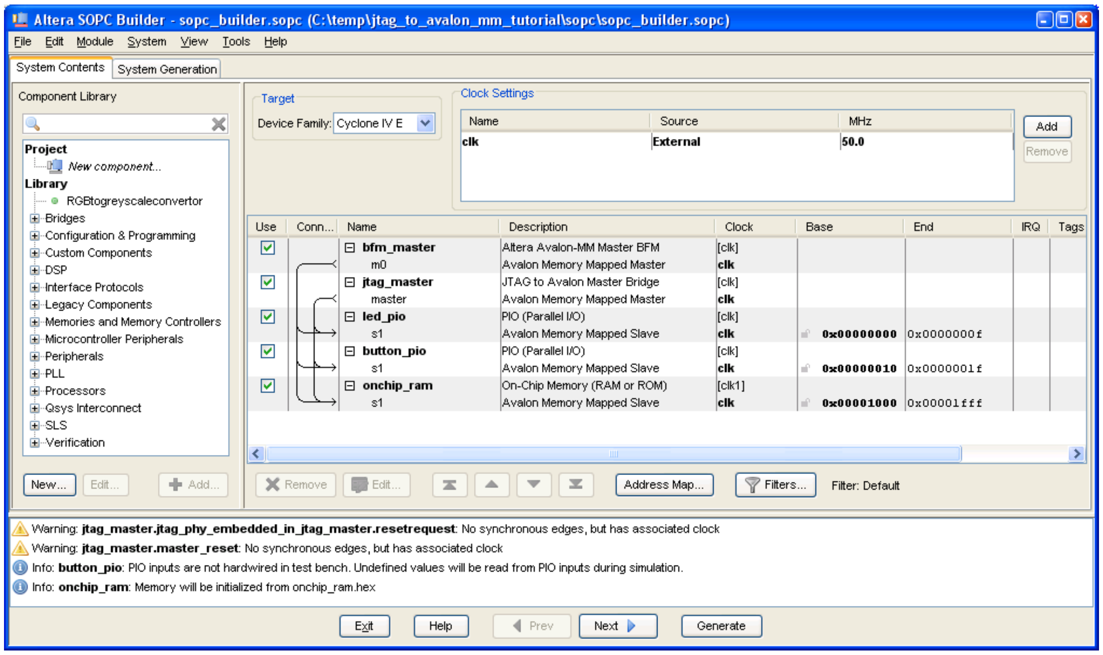
3 SOPC Builder Design Flow

In this section, the system in Figure 1 is created using SOPC Builder, synthesized using Quartus, and simulated using Modelsim. The design is implemented using the Quartus GUI. Once the design is complete, I show how to create Tcl scripts that can regenerate the system from the minimum number of design files.

3.1 Project Creation

Start Quartus 11.1sp1 and create a new project targeting the BeMicro-SDK board;

- File→New Project Wizard
- *Directory, Name, Top-level Entity [page 1 of 5]*
  - Working directory name: `c:\temp\altera_jtag_to_avalon_mm_tutorial\sopc\`
  - Project and top-level entity name: `tutorial`
  - Click *Next*
  - Click *Yes* when prompted to create the working directory
- *Add Files [page 2 of 5]*
  - Click *Next*
- *Family & Device Settings [page 3 of 5]*
  - Device Family: use the pull-down menu to select `Cyclone IV E`
  - In the Available Devices spreadsheet, select `EP4CE22F17C7`
  - Click *Next*
- Click *Finish*, leaving all other settings at their defaults


Altera SOPC Builder - sopc_builder.sopc (C:\temp\jtag_to_avalon_mm_tutorial\sopc\sopc_builder.sopc)
File Edit Module System View Tools Help
System Contents System Generation
Component Library
Project
New component...
Library
RGBtogreyscaleconvertor
Bridges
Configuration & Programming
Custom Components
DSP
Interface Protocols
Legacy Components
Memories and Memory Controllers
Microcontroller Peripherals
Peripherals
PLL
Processors
Qsys Interconnect
SLS
Verification
New... Edit... Add...
Target
Device Family: Cyclone IV E
Clock Settings
Add Remove
Remove Edit... Address Map... Filters... Filter: Default
Exit Help Prev Next Generate

Name	Source	MHz
clk	External	50.0

Use	Conn...	Name	Description	Clock	Base	End	IRQ	Tags
✓		bfm_master	Altera Avalon-MM Master BFM	[clk]				
		m0	Avalon Memory Mapped Master	clk				
✓		jtag_master	JTAG to Avalon Master Bridge	[clk]				
		master	Avalon Memory Mapped Master	clk				
✓		led_pio	PIO (Parallel I/O)	[clk]				
		s1	Avalon Memory Mapped Slave	clk	0x00000000	0x0000000f		
✓		button_pio	PIO (Parallel I/O)	[clk]				
		s1	Avalon Memory Mapped Slave	clk	0x00000010	0x0000001f		
✓		onchip_ram	On-Chip Memory (RAM or ROM)	[clk1]				
		s1	Avalon Memory Mapped Slave	clk	0x00001000	0x00001fff		

```
Warning: jtag_master.jtag_phy_embedded_in_jtag_master.resetrequest: No synchronous edges, but has associated clock
Warning: jtag_master.master_reset: No synchronous edges, but has associated clock
Info: button_pio: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.
Info: onchip_ram: Memory will be initialized from onchip_ram.hex
```

Figure 2: SOPC Builder `sopc_system` design.

3.2 SOPC Builder Component

Start SOPC Builder and create the system shown in Figure 2;

- Tools→SOPC Builder
- In the *Create New System* GUI select `Verilog` as the target language, and name it `sopc_system`.

  Selecting Verilog is a *requirement*, since in the next section I show how to simulate the system using the free version of Modelsim supplied by Altera, i.e., Modelsim-ASE. The Altera Avalon and Avalon Verification IP components are written in Verilog and SystemVerilog, and Modelsim-ASE only supports single language simulation. The full version of Modelsim can be used for mixed language designs.
- Under *Clock Settings*, rename the clock to `clk` (the default clock frequency of 50MHz is correct for the BeMicro-SDK).
- Add the SOPC System components from the *Component Library*;
  - Add the Avalon-MM BFM master;
    - ∗ Verification→Simulation→Altera Avalon-MM Master BFM
      For previous versions of Quartus use;
      Avalon Verification Suite→Altera Avalon-MM Master BFM
    - ∗ Uncheck *Use the burstcount signal* and click finish.
    - ∗ Right click on the component and rename it `bfm_master`.
  - Add the JTAG-to-Avalon-MM master;
    - ∗ Bridges→Memory Mapped→JTAG to Avalon Master Bridge
      For previous versions of Quartus use;
      Bridges and Adapters→Memory Mapped→JTAG to Avalon Master Bridge

    * Accept the defaults, and click finish.
    * Right click on the component and rename it `jtag_master`.
  - Add the LED parallel I/O slave;
    * Peripherals→Microcontroller Peripherals→PIO (Parallel I/O).
    * Accept the defaults, and click finish.
    * Right click on the component and rename it `led_pio`.
    * Connect the slave to the two masters; it will be assigned the base address `0x00000000`.
  - Add the button parallel I/O slave;
    * Peripherals→Microcontroller Peripherals→PIO (Parallel I/O).
    * Change the direction to *input*, and click finish.
    * Right click on the component and rename it `button_pio`.
    * Connect the slave to the two masters; and change its base address to `0x00000010`.
  - Add the on-chip memory slave;
    * Memories and Memory Controllers→On-chip→On-chip Memory (RAM or ROM).
    * Uncheck *Initialize memory content*, and click finish.
    * Right click on the component and rename it `onchip_ram`.
    * Connect the slave to the two masters; and change its base address to `0x00001000`.

- The SOPC System should now look like that in Figure 2. The warnings shown in the SOPC builder GUI can be ignored.
- Click *Generate* to generate the system.

  SOPC Builder pops-up a *Save changes?* dialog window asking if you want to *Save changes to unnamed?* Click the *Save* button to bring up the *Save* dialog window. Enter the system name, `sopc_system.sopc`, and click *Save*[2].
- When the message *Info: System generation was successful* appears, click the *Exit* button (click *Save* again when it prompts you).

So what did this just create? In the project directory, you will see Verilog files for the SOPC system components; `bfm_master.v`, `jtag_master.v`, `led_pio.v`, `button_pio.v`, and `onchip_ram.v`, and a Verilog file for the top-level SOPC system, `sopc_system.v`. Open the files and look at the Verilog code; the BFM master, JTAG master, and on-chip RAM files simply instantiate components, while the LED and button PIO components contain Verilog code. The SOPC system file is the most complicated, it contains automatically generated interconnect code (the main reason for using the system generation tool). The SOPC system file is not particularly readable, however, there are important sections of the file that are discussed in Section 3.5.

The SOPC system description is implemented in the XML file `sopc_system.sopc`. All of the Verilog files generated by SOPC Builder can be considered as *intermediate* files, and they can be deleted and regenerated; much like you would consider object files as intermediate files when compiling and linking programs. Go ahead and delete all the SOPC system generated Verilog files and other related files with `sopc_system` in their names (*except* of course for `sopc_system.sopc`), delete the directories `jtag_master` and `sopc_system_sim` too. Open the SOPC Builder GUI, and you will see the system unchanged. Click *Generate* and all the files you just deleted will be regenerated. The point of this last exercise was to show you that the SOPC System component can be regenerated from the single file `sopc_system.sopc`, so that is the file to preserve when creating a project archive or checking the project into a version control system.

[2]See Appendix C, Note 1.

```
module sopc_system (
    // 1) global signals:
    clk,
    reset_n,

    // the_button_pio
    in_port_to_the_button_pio,

    // the_led_pio
    out_port_from_the_led_pio
);
```

Figure 3: SOPC Builder `sopc_system` top-level Verilog module.

3.3 Top-Level Design

Figure 3 shows the Verilog module generated for the SOPC system component (buried inside the `sopc_system.v` file). So how should this component be used?

SOPC Builder components can be used in one of two ways; the component can be considered *the top-level component*, or the component can be considered just *one component* in a top-level design, i.e., a component that you instantiate in a top-level design. I encourage you to use the latter interpretation. Here is my argument;

- FPGA pin assignments are essentially invariant once the device is placed-and-routed on a PCB. The design may contain multiple general purpose I/Os (GPIO), whose properties can be changed, but the wiring associated with GPIO bit 0 in the schematic will always route to the same pin on the FPGA, regardless of what you would like to name that signal within your specific Verilog project.

- Design constraints for I/O signals are generally applied to the pin names of the signals. To avoid having to copy identical constraints between projects, it is convenient to define a top-level entity for a board with a fixed set of port names, along with a constraints file with the nominal constraints. Design-specific constraints can then either replace or augment the nominal constraints.

- Figure 3 shows the port names of the example SOPC system design. These port names depend on the SOPC system master and slave names. Any changes to the SOPC master and slave names results in port name changes.

  If the top-level SOPC component is used as your top-level design file, then the port names become the *pin names*. Pin constraints must then be applied to pin names that are being generated by SOPC Builder. This is a design maintenance headache.

  By instantiating the SOPC component in a top-level component, you gain the advantage of being able to rename the ports to the standard pin names used for that particular board.

  Consider the case where a SOPC component output or input port is not used on a particular board. If the SOPC component is instantiated in a top-level design, then you can leave unused outputs disconnected, and you can drive unused inputs to static values. You cannot do this if you use the SOPC component as the top-level design, as all ports become pins, and all pins need assignments (otherwise Quartus will assign default values that could be incorrect and result in damage to your board).

You also gain the appreciation that the SOPC system is a reuseable component. For example, the SOPC system developed in this tutorial is instantiated in top-level designs for the BeMicro, BeMicro-SDK, and DE2 boards.

Because the SOPC system can be considered as a component, rather than a top-level design, it can be placed in its own source directory, with a simulation testbench. A board design that instantiates the SOPC system then adds the source path to its project, generates the Verilog source in a work directory, and uses that code for synthesis. The example source provided with this tutorial shows this approach.

Caveat: An SOPC System *should* be able to be treated as a component, however, the `.sopc` file contains a reference to the FPGA device part number used when creating the system (the part number is displayed on the SOPC Builder GUI). The example code provided with this tutorial shows that a single SOPC system file can be used with multiple boards containing different devices. The SOPC System was designed targeting the BeMicro-SDK Cyclone IV E device, but is reused unchanged for the BeMicro Cyclone III and DE2 Cyclone II devices. The synthesis script for each board first copies `sopc_system.sopc` from the common area to the board-specific project work directory. The script needs to do this, as SOPC Builder generates the output files in the same location as the SOPC file. The synthesis script then requests the user to start the SOPC Builder GUI and to *generate* the SOPC system files[3]. The synthesis script request to generate the SOPC system occurs before the device constraint has been setup, so the SOPC Builder GUI will always generate a warning that the device currently selected for the project, the Cyclone IV GX, does not match that in the SOPC file. The warning can be ignored.

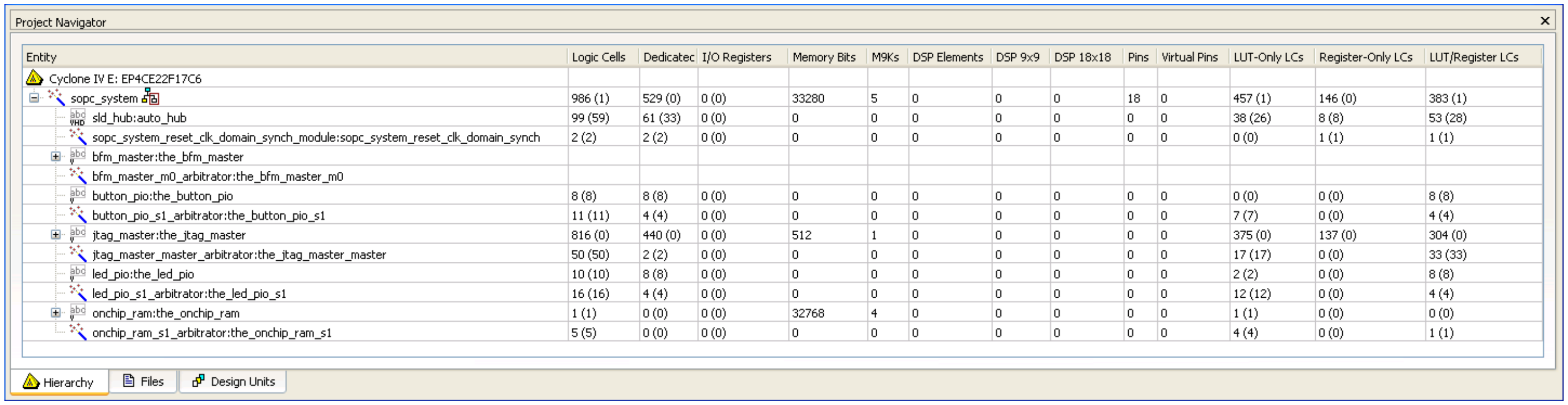
3.4 Synthesis

Now that we have established why you should *not* use the SOPC system as the top-level entity, we will ignore that advice, and do it anyway. In the Quartus GUI, under the Project Navigator window (located on the top-left of the GUI), click on the *Files* tab, and you should see the file `sopc_system.qip` listed. If you do not, add it using the *Project→Add/Remove Files in Project* menu. Right click on the file, and select *Set as Top-Level Entity*, then synthesize the design (press the play button on the GUI).

Figure 4 shows the post-synthesis hierarchy display for the SOPC system design (when synthesized as the top-level design entity). The SOPC design uses a total of 986 LCs, with the majority used by the JTAG hub (`sld_hub`, 99 LCs), and the JTAG master (`jtag_master`, 816 LCs). The Qsys design uses a similar number of LCs.

Before moving on, look at the Quartus II message window (bottom left of the GUI, with the *Processing* tab selected). Scroll up to the top of the messages, and then scroll down until you see the warning text (highlighted in blue). These warnings are generated by the Avalon-MM BFM component. This component is used in simulation only. The warnings appear because the authors of the BFM components have *not* used synthesis directives correctly. The Verilog source for the Avalon-MM BFM should have an Avalon-MM master interface for simulation, and another for synthesis. For synthesis, the Avalon-MM master interface signals should be driven to deasserted levels, allowing the synthesis tool to eliminate the logic (without generating warnings). However, as it is currently implemented, Quartus generates a large number of warnings about missing drivers and dangling pins!

[3] I have not figured out how to automate this from Tcl. The Tcl shell environment variables do not appear to be setup appropriately to `exec` the command-line tool `sopc_builder`.



Project Navigator

Entity	Logic Cells	Dedicatec	I/O Registers	Memory Bits	M9Ks	DSP Elements	DSP 9x9	DSP 18x18	Pins	Virtual Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Cyclone IV E: EP4CE22F17C6													
sopc_system	986 (1)	529 (0)	0 (0)	33280	5	0	0	0	18	0	457 (1)	146 (0)	383 (1)
sld_hub:auto_hub	99 (59)	61 (33)	0 (0)	0	0	0	0	0	0	0	38 (26)	8 (8)	53 (28)
sopc_system_reset_clk_domain_synch_module:sopc_system_reset_clk_domain_synch	2 (2)	2 (2)	0 (0)	0	0	0	0	0	0	0	0 (0)	1 (1)	1 (1)
bfm_master:the_bfm_master													
bfm_master_m0_arbitrator:the_bfm_master_m0													
button_pio:the_button_pio	8 (8)	8 (8)	0 (0)	0	0	0	0	0	0	0	0 (0)	0 (0)	8 (8)
button_pio_s1_arbitrator:the_button_pio_s1	11 (11)	4 (4)	0 (0)	0	0	0	0	0	0	0	7 (7)	0 (0)	4 (4)
jtag_master:the_jtag_master	816 (0)	440 (0)	0 (0)	512	1	0	0	0	0	0	375 (0)	137 (0)	304 (0)
jtag_master_master_arbitrator:the_jtag_master_master	50 (50)	2 (2)	0 (0)	0	0	0	0	0	0	0	17 (17)	0 (0)	33 (33)
led_pio:the_led_pio	10 (10)	8 (8)	0 (0)	0	0	0	0	0	0	0	2 (2)	0 (0)	8 (8)
led_pio_s1_arbitrator:the_led_pio_s1	16 (16)	4 (4)	0 (0)	0	0	0	0	0	0	0	12 (12)	0 (0)	4 (4)
onchip_ram:the_onchip_ram	1 (1)	0 (0)	0 (0)	32768	4	0	0	0	0	0	1 (1)	0 (0)	0 (0)
onchip_ram_s1_arbitrator:the_onchip_ram_s1	5 (5)	0 (0)	0 (0)	0	0	0	0	0	0	0	4 (4)	0 (0)	1 (1)

Figure 4: Quartus II hierarchy display for the `sopc_system` design. The design uses 986 LCs, with the bulk being used by the JTAG hub (`sld_hub`, 99 LCs) and the JTAG-to-Avalon-MM master (`jtag_master`, 816 LCs).

3.5 Simulation

The SOPC System design can be simulated using two possible methods; use the Avalon-MM master BFM documented in the Verification IP Guide, or live-on-the-edge and use the completely undocumented Verilog tasks hidden deep within the JTAG-to-Avalon-MM master implementation. The following sections describe both methods.

3.5.1 SOPC Builder `test_bench`

The observant user would have noticed that SOPC Builder has a simulate option; what does that do? Open up the SOPC Builder GUI, click on the *System Generation* tab, and check the *Simulation. Create project simulator files.* check-box. Check that Modelsim-ASE is setup by using *Tools→Options*, highlighting *HDL Simulator*, and for the simulator *Mentor Graphic's Modelsim-Altera* set the *Application Path*, eg., `c:\software\altera\11.1sp1\modelsim_ase\win32aloem`, and click *Finish*. Now click the *Run Simulator* button. You will get an error about a missing `.mpf` file (this is the Modelsim project file); the *Run Simulator* button should not really be highlighted until the `.mpf` file is present in the project. To create the file, click the SOPC Builder *Generate* button, and then once generation completes, click the *Run Simulator* button.

Run Simulator starts Modelsim-ASE, changes directory to `$TUTORIAL/sopc/sopc_system_sim` directory, and loads the project file. The Modelsim command `where` can be used to display the current directory and project. The SOPC system simulation is controlled by Tcl procedures defined in the script `setup_sim.do`. Open the script and look at the Tcl commands (they are far from easy to read); the Tcl commands determine if the version of Modelsim is the Altera Edition or not, sets up a `vsim` (simulator) command to run on a component called `test_bench`, and creates commands to build library components and the `sopc_system.v` file.

At the Modelsim console type `do setup_sim.do` to source the Tcl procedures defined in the script. Figure 5 shows the Modelsim console output generated by the script. Type the command `s` to build the components and load the simulation. The simulation should load without errors, alas, Modelsim fails with the message `# Error loading design` (along with a message that the design unit for the `altera_avalon_mm_master_bfm` could not be found). The simulation files generated by Quartus 11.1sp1 do not include the Avalon-MM BFM source files (earlier versions of Quartus work fine). The problem can be rectified by editing `setup_sim.do` to add the BFM source files inside the alias for the `s` command, i.e., change the code to

```
alias s "vlib work;
_init_setup

vlog -sv [file join $env(QUARTUS_ROOTDIR)]/../ip/altera/sopc_builder_ip/
verification/lib/verbosity_pkg.sv
vlog -sv [file join $env(QUARTUS_ROOTDIR)]/../ip/altera/sopc_builder_ip/
verification/lib/avalon_mm_pkg.sv
vlog -sv [file join $env(QUARTUS_ROOTDIR)]/../ip/altera/sopc_builder_ip/
verification/altera_avalon_mm_master_bfm/altera_avalon_mm_master_bfm.sv

vlog +incdir+.. ../sopc_system.v;
```

(where the start and end of this script segment already exist in the file, and each `vlog` command is on one line, i.e., 3 new lines are added to the script). After editing `setup_sim.do`, type `do setup_sim.do` at the Modelsim prompt to read the modified script, and then `s` to build the simulation files. This time, the command completes without error (there is a warning about `onchip_ram` not being initialized, but that can be ignored).

```
# @@ setup_sim.do
# @@
# @@ Defined aliases:
# @@
# @@ s -- Load all design (HDL) files.
# @@ re-vlog/re-vcom and re-vsim the design.
# @@
# @@ s_cycloneiv -- For Modelsim SE, compile Cyclone IV models.
# @@ (Ignored in Modelsim AE.)
# @@
# @@ s_stratixiv -- For Modelsim SE, compile Stratix IV models.
# @@ (Ignored in Modelsim AE.)
# @@
# @@ s_stratixv -- For Modelsim SE, compile Stratix V models.
# @@ (Ignored in Modelsim AE.)
# @@
# @@ w -- Sets-up waveforms for this design
# @@ Each SOPC-Builder component may have
# @@ signals 'marked' for display during
# @@ simulation. This command opens a wave-
# @@ window containing all such signals.
# @@
# @@ l -- Sets-up list waveforms for this design
# @@ Each SOPC-Builder component may have
# @@ signals 'marked' for listing during
# @@ simulation. This command opens a list-
# @@ window containing all such signals.
# @@
# @@ h -- print this message
# @@
```

Figure 5: SOPC system Modelsim setup script (`setup_sim.do`) Tcl commands.

Load the Modelsim wave window with the default SOPC system signals by typing the `w` command; this populates the wave window with the `onchip_ram` signals. Not very exiting is it? Type `add wave *` to add the clock, reset, LED output, and button input signals. Type `run 1 us` to run the simulation for 1μs. In the Modelsim console, you will see a message output by the Avalon-MM master BFM. Look at the wave window, and zoom to show the simulation time from 0 to 1us (by clicking on the magnifying glass with the solid blue center). What happened in the wave window? Well, if we had not added the clock and reset, a whole lot of nothing (well, 1μs of nothing really)! By adding the reset and clock, at least we see some activity.

This exercise shows that there is *no free lunch*; just because SOPC Builder has a simulate button, does not mean it will write your simulation testbench for you, all it does is provide the infrastructure for *you* to write your simulation testbench.

To see what infrastructure Altera provides, open up the `sopc_system.v` file and scroll to the bottom of it. There are a couple of things to observe;

- There is a module called `test_bench`. This is the testbench that you just simulated in Modelsim. The testbench contains a clock generator, a reset generator, and the device under test; the SOPC system component. Not much of a testbench really.
- In the Verilog source, above the `test_bench` component, Verilog `include` statements are used to inline a mixture of code from the Quartus install directory, code copied to the project directory, and generated code. Appendix C Notes 5 and 6 have comments on the disadvantages of using this technique to resolve source code dependencies.

  The Tcl simulation script described in Section 3.6 uses the `vsim +incdir+` command line option to compile the source that is `include`'d inline in the Verilog source. For this simulation to work, make sure to check the *Simulation* check box under the *System Generation* tab in the SOPC Builder GUI.

So how then do we simulate this system? The Altera Verification IP suite shows you one option; you create your own testbench and then instantiate `test_bench` as the clock and reset generator. Personally, I do not like that solution, as you lose control of the reset line. Rather, I recommend ignoring `test_bench` entirely. The only useful simulation code that SOPC Builder generates are some of the script commands within `setup_sim.do`; the lines of code telling you the paths to the components included in the project, and the arguments to the `vsim` command to get the simulation to run (without generating lots of warnings).

3.5.2 Avalon-MM Master BFM

The source code for the Avalon-MM master BFM testbench is located at;

`$TUTORIAL/hdl/sopc_system/test/sopc_system_bfm_master_tb.sv`

This testbench uses the Altera Verification IP Suite to generate Avalon-MM master transactions.
The Avalon-MM master BFM testbench can be simulated as follows;

- Start Modelsim.

  This can be performed from the SOPC Builder GUI by clicking on the *Run Simulator* button or you can run the simulator directly.

- Set the tutorial path variable

  ```
  Modelsim> set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
  ```

- If you did not start Modelsim from the SOPC Builder GUI, change directory to the SOPC simulation directory and reset the work library mapping

  ```
  Modelsim> cd $TUTORIAL/sopc/sopc_system_sim
  Modelsim> vmap work work
  ```

  The first argument to `vmap` is the library name, and the second is the path to that library, i.e., the command maps the `work` library to the `work/` directory in the current directory.

- Compile the SOPC Builder source

  ```
  Modelsim> do setup_sim.do
  Modelsim> s
  ```

  This step will only succeed if the script has been edited per the instructions in the previous section.

- Compile the testbench

  ```
  VSIM> vlog -sv $TUTORIAL/hdl/sopc_system/test/sopc_system_bfm_master_tb.sv
  ```

- Run the simulation

  ```
  VSIM> vsim -t ps +nowarnTFMPC sopc_system_bfm_master_tb
  VSIM> do $TUTORIAL/hdl/sopc_system/scripts/sopc_system_bfm_master_tb.do
  VSIM> run -a
  ```

  where the `vsim +nowarnTFMPC` option suppresses warnings about missing connections (this argument was copied from `setup_sim.do`), and the `.do` file populates the wave window.

- Modify the testbench source, recompile, and rerun the simulation via

  ```
  VSIM> vlog -sv $TUTORIAL/hdl/sopc_system/test/sopc_system_bfm_master_tb.sv
  VSIM> restart -f; run -a
  ```

Figure 6 shows the Modelsim console output produced by the testbench. The testbench checks the operation of the LEDs, push buttons, and several locations in RAM. The testbench code contains SystemVerilog assertions that would have generated an error message for any failed test.

```
# ================================================================
# JTAG-to-Avalon-MM SOPC System Testbench (using the BFM master)
# ================================================================
#  * Deassert reset
#
# ----------------------------------------------------------------
# 1: Test the LEDs.
# ----------------------------------------------------------------
#  * Write 0xAA to the LEDs
#    - LED register value = aah
#    - LED port value = aah
#  * Walking 1's test
#    - LED port value = 01h
#    - LED port value = 02h
#    - LED port value = 04h
#    - LED port value = 08h
#    - LED port value = 10h
#    - LED port value = 20h
#    - LED port value = 40h
#    - LED port value = 80h
#
# ----------------------------------------------------------------
# 2: Test the push buttons.
# ----------------------------------------------------------------
#  * Push button value = 55h
#  * Walking 1's test
#    - Push button value = 01h
#    - Push button value = 02h
#    - Push button value = 04h
#    - Push button value = 08h
#    - Push button value = 10h
#    - Push button value = 20h
#    - Push button value = 40h
#    - Push button value = 80h
#
# ----------------------------------------------------------------
# 3: Test the on-chip RAM.
# ----------------------------------------------------------------
#  * Fill 1024 locations of RAM with an incrementing count
#  * Read and check the RAM
#
# ================================================================
# Simulation complete.
# ================================================================
```

Figure 6: Modelsim console output for the SOPC Builder Avalon-MM BFM master testbench, `sopc_system_bfm_master_tb`.

3.5.3 JTAG-to-Avalon-MM Master

Using only the Avalon-MM BFM master to test your design violates the principle

"Test what you fly, and fly what you test"

The main idea of the principle that you should *test what you use*. Using the Avalon-MM BFM to generate Avalon-MM transactions *does not* test the JTAG-to-Avalon-MM master logic, which is what generates the Avalon-MM master transactions in the actual hardware. How then can you be sure that your hardware design will function correctly? Arguably, you could test individual components using the Avalon Verification IP suite, which would then give you higher confidence that the components within the system are functionally correct. The final SOPC system should still have its own testbench, since the system has a *fabric* that is specific to the design. So, lets use the Verification IP suite to test the JTAG-to-Avalon Master ... Oh, but there is no documented way to simulate the JTAG-to-Avalon-MM master interface, bummer. We won't let the lack of documentation stop us though, continue reading!

The Altera wiki entry Avalon-ST_JTAG_Interface_PLI_Simulation_Mode shows how to exercise the JTAG-to-Avalon-MM master in simulation using Verilog PLI (Programming Language Interface). The PLI method uses a socket connection between System Console and Modelsim, and *magically* (the code is hidden in a Java library) generates transactions within the testbench. While the PLI interface could be useful for some applications, eg., developing a software interface, it is not appropriate for use in self-verifying (automated) testbenches. A self-verifying testbench should run completely within the Modelsim simulator, and the easiest way to do that, is to implement the testcase generator, the assertion logic, and the device under test using a hardware description language.

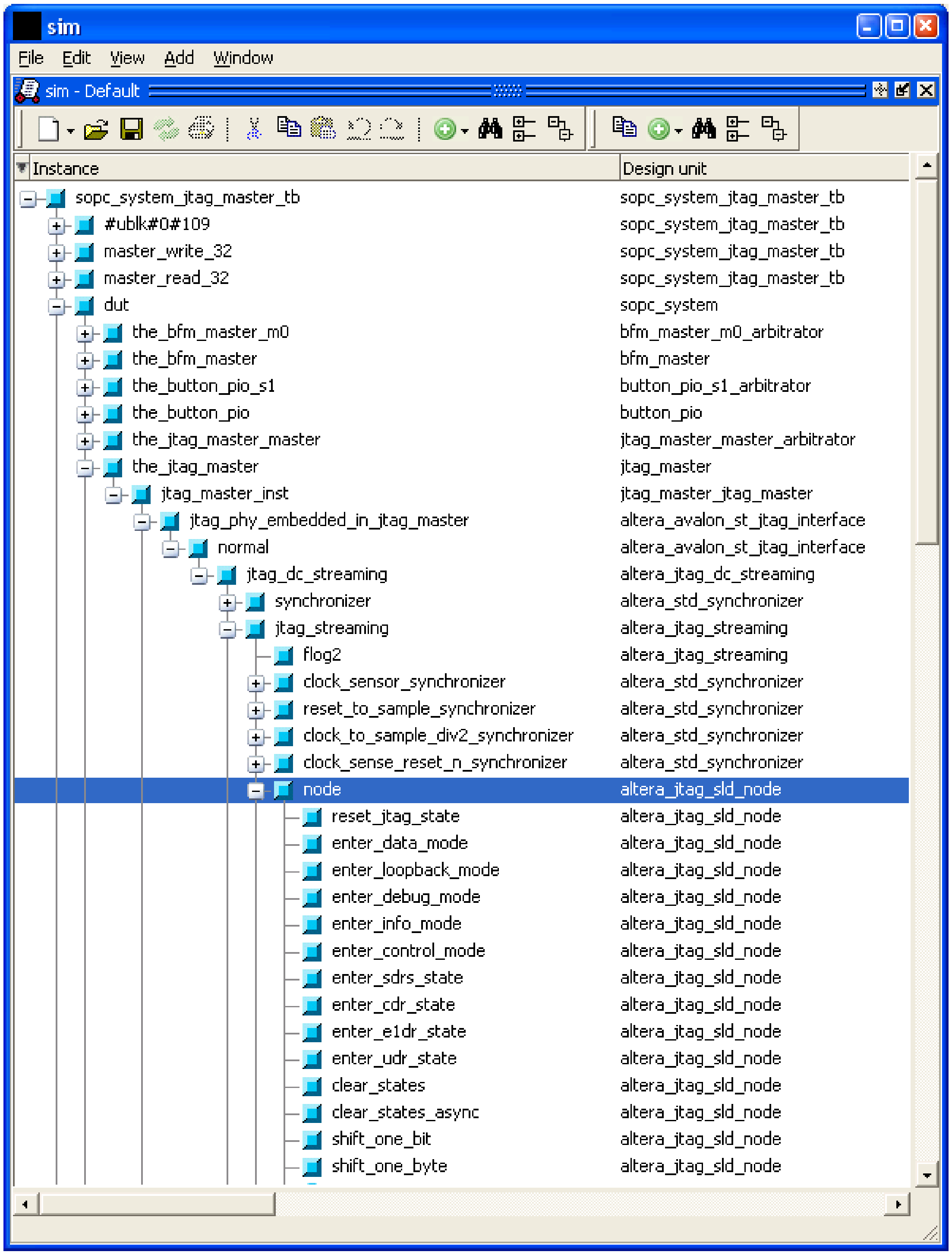
The JTAG-to-Avalon-MM master consists of a JTAG-to-Avalon-ST interface that converts JTAG transactions into byte streams in and out of the design, bytes-to-packets conversion logic that encodes and decodes a binary protocol transported over the byte streams. The binary protocol encodes whether to perform an Avalon-MM read or write transaction, and the response for each transaction type. The packets to transactions component converts the commands into Avalon-MM master commands. The JTAG-to-Avalon-ST, bytes-to-packets, packets-to-transactions, packets-to-bytes, and JTAG-to-Avalon-MM components are only partially documented in the Altera literature. See [5] for a detailed analysis of the JTAG-to-Avalon components.

The analysis document [5] shows that buried deep within the source code for the JTAG-to-Avalon-ST component is the logic for the JTAG node (which connects to the JTAG hub). Hidden within the JTAG node are Verilog tasks for performing low-level JTAG operations. Figure 7 shows the path to the JTAG node in the testbench developed for this section. The JTAG node is highlighted in the figure, and the node's Verilog tasks are listed beneath it. If you know how to generate JTAG transactions, and know how to use them to generate byte streams, then you have the makings of a JTAG-to-Avalon-MM master simulation. I won't bore you with the details, the morbidly curious can read the testbench code and the analysis document.

The source code for the JTAG-to-Avalon-MM master testbench is located at;

```
$TUTORIAL/hdl/sopc_system/test/sopc_system_jtag_master_tb.sv
```

This testbench uses the undocumented Verilog tasks in the JTAG node to generate byte-streams which encode Avalon-MM master transactions for the packets-to-transactions component.



Instance	Design unit
sopc_system_jtag_master_tb	sopc_system_jtag_master_tb
#ublk#0#109	sopc_system_jtag_master_tb
master_write_32	sopc_system_jtag_master_tb
master_read_32	sopc_system_jtag_master_tb
dut	sopc_system
the_bfm_master_m0	bfm_master_m0_arbitrator
the_bfm_master	bfm_master
the_button_pio_s1	button_pio_s1_arbitrator
the_button_pio	button_pio
the_jtag_master_master	jtag_master_master_arbitrator
the_jtag_master	jtag_master
jtag_master_inst	jtag_master_jtag_master
jtag_phy_embedded_in_jtag_master	altera_avalon_st_jtag_interface
normal	altera_avalon_st_jtag_interface
jtag_dc_streaming	altera_jtag_dc_streaming
synchronizer	altera_std_synchronizer
jtag_streaming	altera_jtag_streaming
flog2	altera_jtag_streaming
clock_sensor_synchronizer	altera_std_synchronizer
reset_to_sample_synchronizer	altera_std_synchronizer
clock_to_sample_div2_synchronizer	altera_std_synchronizer
clock_sense_reset_n_synchronizer	altera_std_synchronizer
node	altera_jtag_sld_node
reset_jtag_state	altera_jtag_sld_node
enter_data_mode	altera_jtag_sld_node
enter_loopback_mode	altera_jtag_sld_node
enter_debug_mode	altera_jtag_sld_node
enter_info_mode	altera_jtag_sld_node
enter_control_mode	altera_jtag_sld_node
enter_sdrs_state	altera_jtag_sld_node
enter_cdr_state	altera_jtag_sld_node
enter_e1dr_state	altera_jtag_sld_node
enter_udr_state	altera_jtag_sld_node
clear_states	altera_jtag_sld_node
clear_states_async	altera_jtag_sld_node
shift_one_bit	altera_jtag_sld_node
shift_one_byte	altera_jtag_sld_node

Figure 7: Altera JTAG node location in the SOPC Builder JTAG-to-Avalon-MM master testbench, `sopc_system_jtag_master_tb`. The tasks under the highlighted JTAG `node`, i.e., `reset_jtag_state` down to `shift_one_bit`, are used to simulate the JTAG-to-Avalon-MM bridge.

The JTAG-to-Avalon-MM master testbench can be simulated as follows;

- Start Modelsim.

  This can be performed from the SOPC Builder GUI by clicking on the *Run Simulator* button or you can run the simulator directly.

- Set the tutorial path variable

  ```
  Modelsim> set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
  ```

- If you did not start Modelsim from the SOPC Builder GUI, change directory to the SOPC simulation directory and reset the work library mapping

  ```
  Modelsim> cd $TUTORIAL/sopc/sopc_system_sim
  Modelsim> vmap work work
  ```

  The first argument to `vmap` is the library name, and the second is the path to that library, i.e., the command maps the `work` library to the `work/` directory in the current directory.

- Compile the SOPC Builder source

  ```
  Modelsim> do setup_sim.do
  Modelsim> s
  ```

  This step will only succeed if the script has been edited per the instructions in Section 3.5.1.

- Compile the testbench

  ```
  VSIM> vlog -sv $TUTORIAL/hdl/sopc_system/test/sopc_system_jtag_master_tb.sv
  ```

- Run the simulation

  ```
  VSIM> vsim -t ps +nowarnTFMPC sopc_system_jtag_master_tb
  VSIM> do $TUTORIAL/hdl/sopc_system/scripts/sopc_system_jtag_master_tb.do
  VSIM> run -a
  ```

  where the `vsim +nowarnTFMPC` option suppresses warnings about missing connections (this argument was copied from `setup_sim.do`), and the `.do` file populates the wave window.

- Modify the testbench source, recompile, and rerun the simulation via

  ```
  VSIM> vlog -sv $TUTORIAL/hdl/sopc_system/test/sopc_system_jtag_master_tb.sv
  VSIM> restart -f; run -a
  ```

The JTAG-to-Avalon-MM master testbench reproduces the test sequences performed by the Avalon-MM BFM master testbench shown in Figure 6. The JTAG-to-Avalon-MM master testbench console output is slightly different, as the testbench first performs a JTAG protocol test, and then duplicates the Avalon-MM BFM master testbench sequences. The run-time of the JTAG testbench is much longer than that of the Avalon-MM BFM master testbench, due to the fact that the JTAG clock is slower than the Avalon-MM clock, and byte stream transactions are serialized over JTAG. In practice, you should use the Avalon-MM BFM master for performing exhaustive tests on Avalon-MM slaves, and perform token tests with the JTAG simulation interface to check that devices are connected correctly. Thus we finally reach our goal of *testing what we fly*.

3.6 Synthesis and Simulation Scripts

Up until this point, you have been entering commands in the Quartus or Modelsim console. What happens when you get bored with typing, or forget the commands? Surely there is an easier way? Yes, there is! Tcl scripts.

The BeMicro-SDK SOPC system project directory is located at;

```
$TUTORIAL/hdl/boards/bemicro_sdk/sopc_system/
```

The project directory contains the synthesis script `scripts/synth.tcl`, and the top-level design file `src/bemicro_sdk.sv` (with the SOPC system instantiated as a component). The synthesis script can be run by starting Quartus, selecting the Tcl console (if you cannot see it in the GUI, make it visible using *View→Utility Windows→Tcl Console*), changing to the project directory, and running the script, i.e., at the Quartus Tcl prompt

```
tcl> set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
tcl> cd $TUTORIAL/hdl/boards/bemicro_sdk/sopc_system/
tcl> source scripts/synth.tcl
```

The script will determine that the SOPC System needs to be generated. I have not figured out how to automate that from Tcl, so the script asks you to run SOPC Builder, and *generate* the SOPC system. Generate the system, and then exit the SOPC Builder GUI. In the Tcl console, press the up-arrow to bring back the last command, and re-run the script. Quartus will then synthesize the design. The top-level design connects 7-bits of the LED control register to 7 of the 8 LEDs on the board, and blinks the other LED at about 1Hz (so you can see the board is alive). Thus implementing the design has been reduced to a few Tcl commands and GUI button clicks—much less to remember!

The SOPC System simulations also have a Tcl script that will setup the simulator without having to run the `setup_sim.do` script generated by Quartus. The SOPC system simulation project directory is located at;

```
$TUTORIAL/hdl/sopc_system/
```

That directory contains a `scripts/` directory containing the simulation script `sim.tcl`. The simulation script can be run by starting Modelsim, changing to the project directory, and running the script, i.e.,

```
ModelSim> set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
ModelSim> cd $TUTORIAL/hdl/sopc_system/
ModelSim> source scripts/sim.tcl
```

The script will determine whether the SOPC System needs to be generated (remember, Quartus generates code, so the simulation needs to compile the generated code). The script is configured to look for the SOPC system in the BeMicro-SDK project directory. If it does not find it, it asks you to run Quartus to generate it (which can be done using the `synth.tcl` script following the sequence just described). Once the simulation script finds the Quartus generated source, it compiles the testbenches, and creates two Tcl procedures with the same names as the testbenches. Either testbench can be run by typing the procedure name. Each procedure issues the `vsim` command, populates the wave window, and runs the simulation (see the script for the procedure implementations). Again, much less typing!

The nice thing about the synthesis and simulation scripts is that they show you the *minimum* number of source files needed to reproduce both the Quartus and Modelsim projects. These are the source files that you check into a code versioning system (along with the synthesis and simulation scripts).

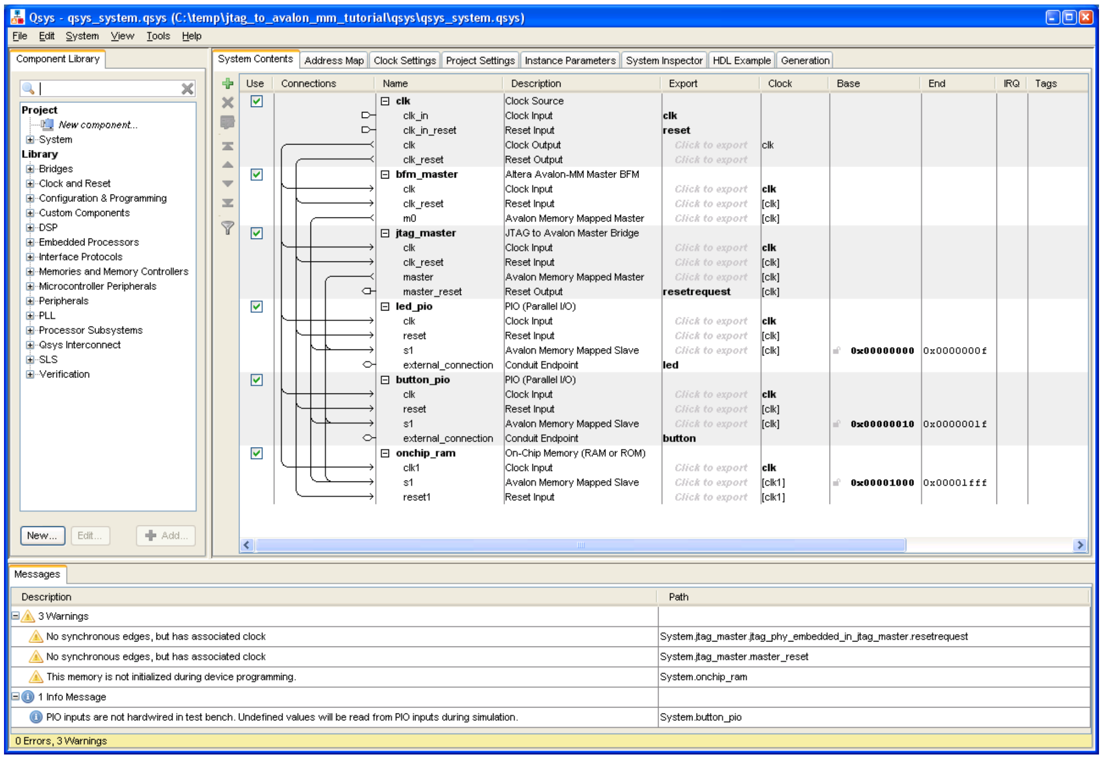
4 Qsys Design Flow

In the following sections, I show how to use the Qsys tool to create a system, and then how to simulate it. Qsys is Altera's the *new-and-improved* replacement for SOPC Builder. Reproducing the SOPC Builder design using Qsys helps contrast the tools, and provides the reader with a reference design when porting their own SOPC system designs to Qsys.

4.1 Project Creation

Start Quartus 11.1sp1 and create a new project targeting the BeMicro-SDK board;

- File→New Project Wizard
- *Directory, Name, Top-level Entity [page 1 of 5]*
  - Working directory name: `c:\temp\altera_jtag_to_avalon_mm_tutorial\qsys\`
  - Project and top-level entity name: `tutorial`
  - Click *Next*
  - Click *Yes* when prompted to create the working directory
- *Add Files [page 2 of 5]*
  - Click *Next*
- *Family & Device Settings [page 3 of 5]*
  - Device Family: use the pull-down menu to select `Cyclone IV E`
  - In the Available Devices spreadsheet, select `EP4CE22F17C7`
  - Click *Next*
- Click *Finish*, leaving all other settings at their defaults



Use	Name	Description	Export	Clock	Base	End	IRQ	Tags
✓	**clk**	Clock Source						
	clk_in	Clock Input	**clk**					
	clk_in_reset	Reset Input	**reset**					
	clk	Clock Output	*Click to export*	clk				
	clk_reset	Reset Output	*Click to export*					
✓	**bfm_master**	Altera Avalon-MM Master BFM						
	clk	Clock Input	*Click to export*	**clk**				
	clk_reset	Reset Input	*Click to export*	[clk]				
	m0	Avalon Memory Mapped Master	*Click to export*	[clk]				
✓	**jtag_master**	JTAG to Avalon Master Bridge						
	clk	Clock Input	*Click to export*	**clk**				
	clk_reset	Reset Input	*Click to export*	[clk]				
	master	Avalon Memory Mapped Master	*Click to export*	[clk]				
	master_reset	Reset Output	**resetrequest**	[clk]				
✓	**led_pio**	PIO (Parallel I/O)						
	clk	Clock Input	*Click to export*	**clk**				
	reset	Reset Input	*Click to export*	[clk]				
	s1	Avalon Memory Mapped Slave	*Click to export*	[clk]	**0x00000000**	0x0000000f		
	external_connection	Conduit Endpoint	**led**					
✓	**button_pio**	PIO (Parallel I/O)						
	clk	Clock Input	*Click to export*	**clk**				
	reset	Reset Input	*Click to export*	[clk]				
	s1	Avalon Memory Mapped Slave	*Click to export*	[clk]	**0x00000010**	0x0000001f		
	external_connection	Conduit Endpoint	**button**					
✓	**onchip_ram**	On-Chip Memory (RAM or ROM)						
	clk1	Clock Input	*Click to export*	**clk**				
	s1	Avalon Memory Mapped Slave	*Click to export*	[clk1]	**0x00001000**	0x00001fff		
	reset1	Reset Input	*Click to export*	[clk1]				

Description	Path
3 Warnings	
No synchronous edges, but has associated clock	System.jtag_master.jtag_phy_embedded_in_jtag_master.resetrequest
No synchronous edges, but has associated clock	System.jtag_master.master_reset
This memory is not initialized during device programming.	System.onchip_ram
1 Info Message	
PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.	System.button_pio

0 Errors, 3 Warnings

Figure 8: Qsys `qsys_system` design.

4.2 Qsys Component

Start Qsys and create the system shown in Figure 8;

- Tools→Qsys
- The Qsys GUI starts up with a *Clock Source* already populated. Right click, and rename it `clk`.
- Add the Qsys components from the *Component Library*;
  - Add the Avalon-MM BFM master;
    - Verification→Simulation→Altera Avalon-MM Master BFM
      For previous versions of Quartus use;
      Avalon Verification Suite→Altera Avalon-MM Master BFM
    - Uncheck *Use the burstcount signal* and click finish.
    - Right click on the component and rename it `bfm_master`.
  - Add the JTAG-to-Avalon-MM master;
    - Bridges→Memory Mapped→JTAG to Avalon Master Bridge
      For previous versions of Quartus use;
      Bridges and Adapters→Memory Mapped→JTAG to Avalon Master Bridge
    - Accept the defaults, and click finish.
    - Right click on the component and rename it `jtag_master`.

    * Export the `master_reset` signal to the top-level, by clicking on the *Click to export* text in the *Export* column, *Reset Output* row, and enter the name `resetrequest`. This creates the top-level port `resetrequest_reset` on the Qsys system (which can be viewed on the *HDL Example* tab). This signal is an output from the JTAG master that is intended for use as a JTAG controlled reset source (the hardware examples instead use this signal to control an LED).

  - Add the LED parallel I/O slave;
    * Peripherals→Microcontroller Peripherals→PIO (Parallel I/O).
    * Accept the defaults, and click finish.
    * Right click on the component and rename it `led_pio`.
    * Export the I/O to the top-level, by clicking on the *Click to export* text in the *Export* column, *Conduit Endpoint* row, and enter the name `led`. This creates the top-level port `led_export` on the Qsys system (which can be viewed on the *HDL Example* tab).
    * Connect the slave to the two masters; it will be assigned the base address `0x00000000`.
  - Add the button parallel I/O slave;
    * Peripherals→Microcontroller Peripherals→PIO (Parallel I/O).
    * Change the direction to *input*, and click finish.
    * Right click on the component and rename it `button_pio`.
    * Export the I/O to the top-level, by clicking on the *Click to export* text in the *Export* column, *Conduit Endpoint* row, and enter the name `button`. This creates the top-level port `button_export` on the top-level Qsys system (which can be viewed on the *HDL Example* tab).
    * Connect the slave to the two masters; and change its base address to `0x00000010`.
  - Add the on-chip memory slave;
    * Memories and Memory Controllers→On-chip→On-chip Memory (RAM or ROM).
    * Uncheck *Initialize memory content*, and click finish.
    * Right click on the component and rename it `onchip_ram`.
    * Connect the slave to the two masters; and change its base address to `0x00001000`.

- Connect the clock and reset signals;
  - Connect the *Clock Source* component (`clk`) *Clock Output* (`clk`) to the clock input on each of the Avalon-MM masters and slaves.
  - Connect the *Clock Source* component (`clk`) *Reset Output* (`clk_reset`) to the reset input on each of the Avalon-MM masters and slaves.
- The Qsys system should now look like that in Figure 8. The warnings shown in the Qsys GUI can be ignored.
- Click on the *Generation* tab. Uncheck *Create block symbol file (.bsf)*, then click the *Generate* button to generate the system.

  Qsys generates a pop-up asking if you want to save changes to `unnamed`. Click *Save* and enter the system name; `qsys_system.qsys`.
- When the message *Generate Complete. 0 Errors, 3 Warnings* appears, click the *Close* button, and then close the GUI using File→Exit.

The Qsys system files are generated in a directory called `qsys_system`. This is an improvement over SOPC Builder, which would generate many of the system files in the top-level of the Quartus work directory. The top-level Qsys system module is located in the file

```
qsys_system/synthesis/qsys_system.v
```

This file is an improvement over that generated by SOPC Builder in that it contains only the **`sopc_system`** component module, making it easier to comprehend. If you check the simulation option in the generate tab, and re-generate the system, another version of the file is created in

```
qsys_system/simulation/qsys_system.v
```

and while the two files appear to be very similar, unfortunately, the code generator generates the Verilog code in a different order, making it impossible to compare the synthesis and simulation versions to determine actual code differences. On a more positive note, Qsys does not use Verilog **`include`** statements, so that is another improvement over SOPC Builder (see Appendix C Note 5 for the SOPC Builder discussion).

The Qsys generated directories

```
qsys_system/synthesis/submodules
qsys_system/simulation/submodules
```

contain *copies* of Quartus II installation library code (eg., the JTAG node, `altera_jtag_sld_node.v`, which is used for JTAG-to-Avalon-MM master simulation). Appendix C Note 6 discusses the disadvantages of copying what is essentially library source code into a project.

The Qsys system description is implemented in the XML file `qsys_system.qsys`. As with SOPC Builder, all of the generated files can be considered as *intermediate* files; these files can be deleted and regenerated. Go ahead and delete all of the Qsys files and directories, except for `qsys_system.qsys`. Open the Qsys GUI, you will be prompted for a system file, select `qsys_system.qsys`, and you will see the Qsys system unchanged. Select the *Generation* tab, and then click the *Generate* button to regenerate the system.

Caveat: When you select the *Generate* tab, you may notice that the *Create block symbol file* check-box is checked again. The settings selected on the *Generate* page are not preserved in the `.qsys` file. See Appendix C Note 7 for a discussion.

```
module qsys_system (
    output wire        resetrequest_reset , // resetrequest.reset
    output wire [7:0]  led_export ,         //           led.export
    input  wire [7:0]  button_export ,      //        button.export
    input  wire        reset_reset_n ,      //         reset.reset_n
    input  wire        clk_clk              //            clk.clk
);
```

Figure 9: Qsys `qsys_system` top-level Verilog module.

4.3 Top-Level Design

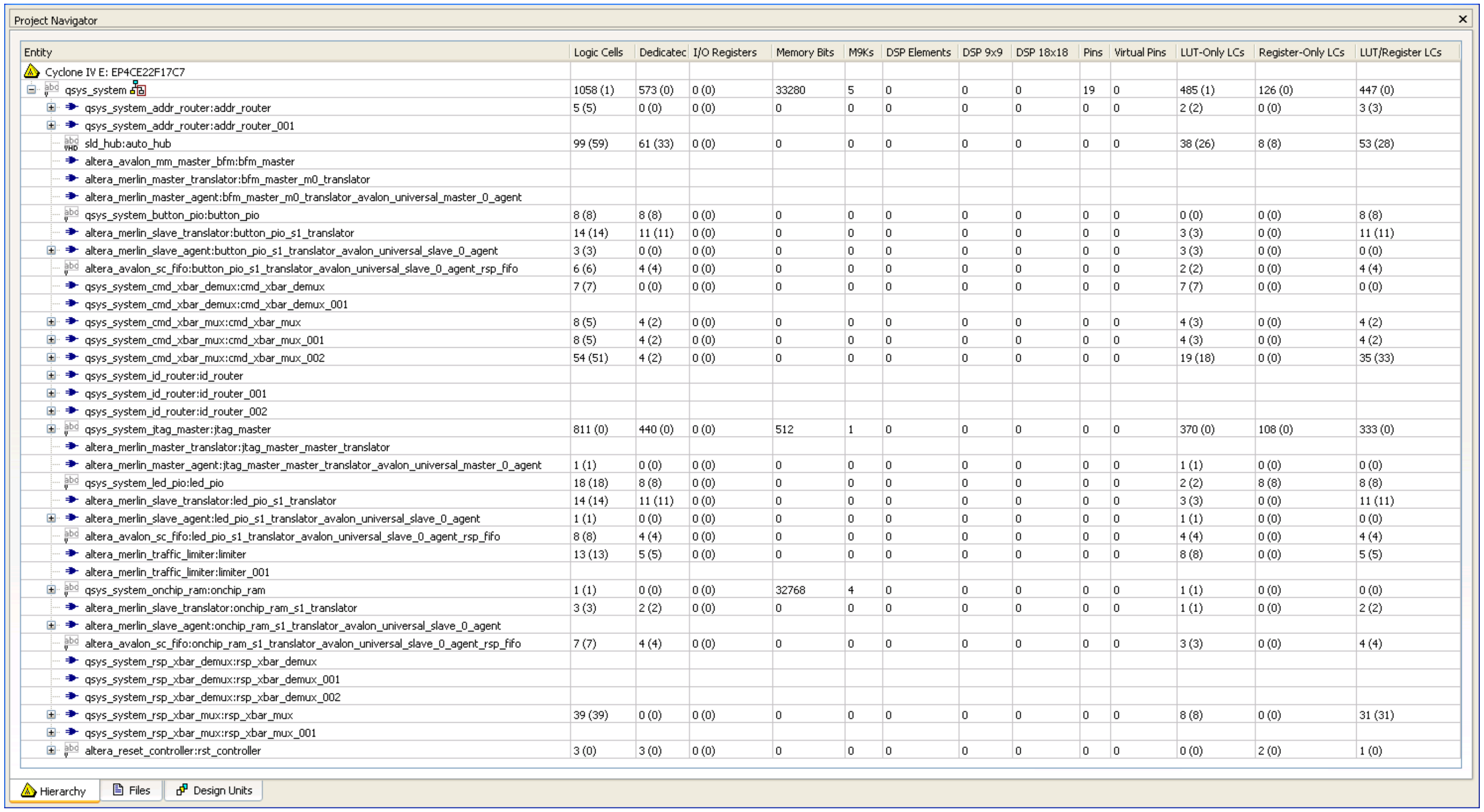
Figure 9 shows the top-level Qsys system module port definitions from `qsys_system/synthesis/qsys_system.v`; the naming convention is a bit redundant isn't it? It seems kind of pointless that Altera allows you to define a port name in the *Export* column in Figure 8, and then they go and munge the names to produce the port names in Figure 9. From the comments after each port, it appears that Altera's port naming convention is a mapping of a SystemVerilog interface definition into a Verilog compatible port name.

4.4 Synthesis

Section 3.3 discusses the reasons why an SOPC System or Qsys System should not be used as a top-level component. Section 3.4 ignores that advice to gauge the logic utilization of the system. We repeat that procedure here, to determine the Qsys system logic utilization.

In the Quartus GUI, under the Project Navigator window (located on the top-left of the GUI), click on the *Files* tab, and then add the Qsys IP file `qsys_system.qip` using the *Project*→*Add/Remove Files in Project* menu. Right click on the file, and select *Set as Top-Level Entity*, then synthesize the design (press the play button on the GUI).

Figure 10 shows the post-synthesis hierarchy display for the Qsys system design (when synthesized as the top-level design entity). The Qsys design uses a total of 1058 LCs, with the majority used by the JTAG hub (`sld_hub`, 99 LCs), and the JTAG master (`jtag_master`, 811 LCs). Figure 4 shows that the SOPC system design uses a similar number of LCs.



Project Navigator

Entity	Logic Cells	Dedicatec	I/O Registers	Memory Bits	M9Ks	DSP Elements	DSP 9x9	DSP 18x18	Pins	Virtual Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Cyclone IV E: EP4CE22F17C7													
qsys_system	1058 (1)	573 (0)	0 (0)	33280	5	0	0	0	19	0	485 (1)	126 (0)	447 (0)
qsys_system_addr_router:addr_router	5 (5)	0 (0)	0 (0)	0	0	0	0	0	0	0	2 (2)	0 (0)	3 (3)
qsys_system_addr_router:addr_router_001													
sld_hub:auto_hub	99 (59)	61 (33)	0 (0)	0	0	0	0	0	0	0	38 (26)	8 (8)	53 (28)
altera_avalon_mm_master_bfm:bfm_master													
altera_merlin_master_translator:bfm_master_m0_translator													
altera_merlin_master_agent:bfm_master_m0_translator_avalon_universal_master_0_agent													
qsys_system_button_pio:button_pio	8 (8)	8 (8)	0 (0)	0	0	0	0	0	0	0	0 (0)	0 (0)	8 (8)
altera_merlin_slave_translator:button_pio_s1_translator	14 (14)	11 (11)	0 (0)	0	0	0	0	0	0	0	3 (3)	0 (0)	11 (11)
altera_merlin_slave_agent:button_pio_s1_translator_avalon_universal_slave_0_agent	3 (3)	0 (0)	0 (0)	0	0	0	0	0	0	0	3 (3)	0 (0)	0 (0)
altera_avalon_sc_fifo:button_pio_s1_translator_avalon_universal_slave_0_agent_rsp_fifo	6 (6)	4 (4)	0 (0)	0	0	0	0	0	0	0	2 (2)	0 (0)	4 (4)
qsys_system_cmd_xbar_demux:cmd_xbar_demux	7 (7)	0 (0)	0 (0)	0	0	0	0	0	0	0	7 (7)	0 (0)	0 (0)
qsys_system_cmd_xbar_demux:cmd_xbar_demux_001													
qsys_system_cmd_xbar_mux:cmd_xbar_mux	8 (5)	4 (2)	0 (0)	0	0	0	0	0	0	0	4 (3)	0 (0)	4 (2)
qsys_system_cmd_xbar_mux:cmd_xbar_mux_001	8 (5)	4 (2)	0 (0)	0	0	0	0	0	0	0	4 (3)	0 (0)	4 (2)
qsys_system_cmd_xbar_mux:cmd_xbar_mux_002	54 (51)	4 (2)	0 (0)	0	0	0	0	0	0	0	19 (18)	0 (0)	35 (33)
qsys_system_id_router:id_router													
qsys_system_id_router:id_router_001													
qsys_system_id_router:id_router_002													
qsys_system_jtag_master:jtag_master	811 (0)	440 (0)	0 (0)	512	1	0	0	0	0	0	370 (0)	108 (0)	333 (0)
altera_merlin_master_translator:jtag_master_master_translator													
altera_merlin_master_agent:jtag_master_master_translator_avalon_universal_master_0_agent	1 (1)	0 (0)	0 (0)	0	0	0	0	0	0	0	1 (1)	0 (0)	0 (0)
qsys_system_led_pio:led_pio	18 (18)	8 (8)	0 (0)	0	0	0	0	0	0	0	2 (2)	8 (8)	8 (8)
altera_merlin_slave_translator:led_pio_s1_translator	14 (14)	11 (11)	0 (0)	0	0	0	0	0	0	0	3 (3)	0 (0)	11 (11)
altera_merlin_slave_agent:led_pio_s1_translator_avalon_universal_slave_0_agent	1 (1)	0 (0)	0 (0)	0	0	0	0	0	0	0	1 (1)	0 (0)	0 (0)
altera_avalon_sc_fifo:led_pio_s1_translator_avalon_universal_slave_0_agent_rsp_fifo	8 (8)	4 (4)	0 (0)	0	0	0	0	0	0	0	4 (4)	0 (0)	4 (4)
altera_merlin_traffic_limiter:limiter	13 (13)	5 (5)	0 (0)	0	0	0	0	0	0	0	8 (8)	0 (0)	5 (5)
altera_merlin_traffic_limiter:limiter_001													
qsys_system_onchip_ram:onchip_ram	1 (1)	0 (0)	0 (0)	32768	4	0	0	0	0	0	1 (1)	0 (0)	0 (0)
altera_merlin_slave_translator:onchip_ram_s1_translator	3 (3)	2 (2)	0 (0)	0	0	0	0	0	0	0	1 (1)	0 (0)	2 (2)
altera_merlin_slave_agent:onchip_ram_s1_translator_avalon_universal_slave_0_agent													
altera_avalon_sc_fifo:onchip_ram_s1_translator_avalon_universal_slave_0_agent_rsp_fifo	7 (7)	4 (4)	0 (0)	0	0	0	0	0	0	0	3 (3)	0 (0)	4 (4)
qsys_system_rsp_xbar_demux:rsp_xbar_demux													
qsys_system_rsp_xbar_demux:rsp_xbar_demux_001													
qsys_system_rsp_xbar_demux:rsp_xbar_demux_002													
qsys_system_rsp_xbar_mux:rsp_xbar_mux	39 (39)	0 (0)	0 (0)	0	0	0	0	0	0	0	8 (8)	0 (0)	31 (31)
qsys_system_rsp_xbar_mux:rsp_xbar_mux_001													
altera_reset_controller:rst_controller	3 (0)	3 (0)	0 (0)	0	0	0	0	0	0	0	0 (0)	2 (0)	1 (0)

Figure 10: Quartus II hierarchy display for the `qsys_system` design. The design uses 1058 LCs, with the bulk being used by the JTAG hub (`sld_hub`, 99 LCs) and the JTAG-to-Avalon-MM master (`jtag_master`, 811 LCs).

```
#
# List Of Command Line Aliases
#
# dev_com              -- Compile device library files
#
# com                  -- Compile the design files in correct order
#
# elab                 -- Elaborate top level design
#
# elab_debug           -- Elaborate the top level design with novopt option
#
# ld                   -- Compile all the design files and elaborate the top
#                           level design
#
# ld_debug             -- Compile all the design files and elaborate the top
#                         level design with -novopt
#
# List Of Variables
#
# TOP_LEVEL_NAME       -- Top level module name.
#
# SYSTEM_INSTANCE_NAME -- Instantiated system module name inside top level
#                         module.
#
# QSYS_SIMDIR          -- Qsys base simulation directory.
#
```

Figure 11: Qsys system Modelsim setup script (`msim_setup.tcl`) Tcl commands.

4.5 Simulation

The Qsys System design can be simulated using two possible methods; use the Avalon-MM master BFM documented in the Verification IP Guide, or live-on-the-edge and use the completely undocumented Verilog tasks hidden deep within the JTAG-to-Avalon-MM master implementation. The following sections describe both methods.

4.5.1 Qsys simulation configuration

The Qsys *Generate* tab has several simulation options that are described in the Quartus II Handbook, Volume 1, Chapter 5, *Creating a System with Qsys* under *Simulating a Qsys System*, on page 5-14 [4]. This tutorial supplies the top-level testbench, so Qsys only needs to generate the simulation model for the Qsys system. The simulation model is created by setting the *Create simulation model* pull-down to *Verilog*, and then clicking the *Generate* button to generate the system. The simulation model files are output in the project directory `qsys_system/simulation`.

The Qsys simulation option creates directories containing copies of code from the Quartus installation, eg., the code in the synthesis directory `qsys_system/synthesis/submodules`, is duplicated in the simulation directory `qsys_system/simulation/submodules` (along with a few extra files copied from the Quartus install). Appendix C Note 6 discusses the disadvantages of copying what is essentially library source code into a project.

4.5.2 Avalon-MM Master BFM

Simulation of the Qsys system in Modelsim is performed with the assistance of the generated simulation script `qsys_system/simulation/mentor/msim_setup.tcl`. The tutorial supplied testbench `qsys_system_bfm_master_tb.sv` can be simulated as follows;

- Start Modelsim
- Change directory to the Mentor simulation directory

  ```
  ModelSim> set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
  ModelSim> cd $TUTORIAL/qsys/qsys_system/simulation/mentor
  ```

- Source the simulation script

  ```
  ModelSim> source msim_setup.tcl
  ```

  Figure 11 shows the script output.

- Compile the device library source (not required for Modelsim-ASE)

  ```
  ModelSim> dev_com
  ```

- Compile the Qsys source

  ```
  ModelSim> com
  ```

- Compile the tutorial testbench

  ```
  ModelSim> vlog -sv $TUTORIAL/hdl/qsys_system/test/qsys_system_bfm_master_tb.sv
           -L qsys_system_bfm_master
  ```

  The library path option, `-L`, is required so that the SystemVerilog verbosity package, compiled by the `msim_setup.tcl` script `com` procedure, is located.

- Set the testbench to the top-level entity (`TOP_LEVEL_NAME` is used by the `msim_setup.tcl` script `elab` procedure)

  ```
  ModelSim> set TOP_LEVEL_NAME qsys_system_bfm_master_tb
  ```

- Elaborate the testbench

  ```
  ModelSim> elab +nowarnTFMPC
  ```

  where the argument to the command gets passed to `vsim` to suppress warnings about missing connections.

- Populate the wave window using

  ```
  VSIM> do $TUTORIAL/hdl/qsys_system/scripts/qsys_system_bfm_master_tb.do
  ```

- Run the simulation

  ```
  VSIM> run -a
  ```

The testbench console output matches that generated by SOPC Builder in Figure 6.

4.5.3 JTAG-to-Avalon-MM Master

The tutorial supplied testbench `qsys_system_jtag_master_tb.sv` can be simulated as follows;

- Start Modelsim
- Change directory to the Mentor simulation directory

  ```
  ModelSim> set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
  ModelSim> cd $TUTORIAL/qsys/qsys_system/simulation/mentor
  ```

- Source the simulation script

  ```
  ModelSim> source msim_setup.tcl
  ```

  Figure 11 shows the script output.

- Compile the device library source (not required for Modelsim-ASE)

  ```
  ModelSim> dev_com
  ```

- Compile the Qsys source

  ```
  ModelSim> com
  ```

- Compile the tutorial testbench

  ```
  ModelSim> vlog -sv $TUTORIAL/hdl/qsys_system/test/qsys_system_jtag_master_tb.sv
           -L qsys_system_bfm_master
  ```

  The library path option, `-L`, is required so that the SystemVerilog verbosity package, compiled by the `msim_setup.tcl` script `com` procedure, is located.

- Set the testbench to the top-level entity (`TOP_LEVEL_NAME` is used by the `msim_setup.tcl` script `elab` procedure)

  ```
  ModelSim> set TOP_LEVEL_NAME qsys_system_jtag_master_tb
  ```

- Elaborate the testbench

  ```
  ModelSim> elab +nowarnTFMPC
  ```

  where the argument to the command gets passed to `vsim` to suppress warnings about missing connections.

- Populate the wave window using

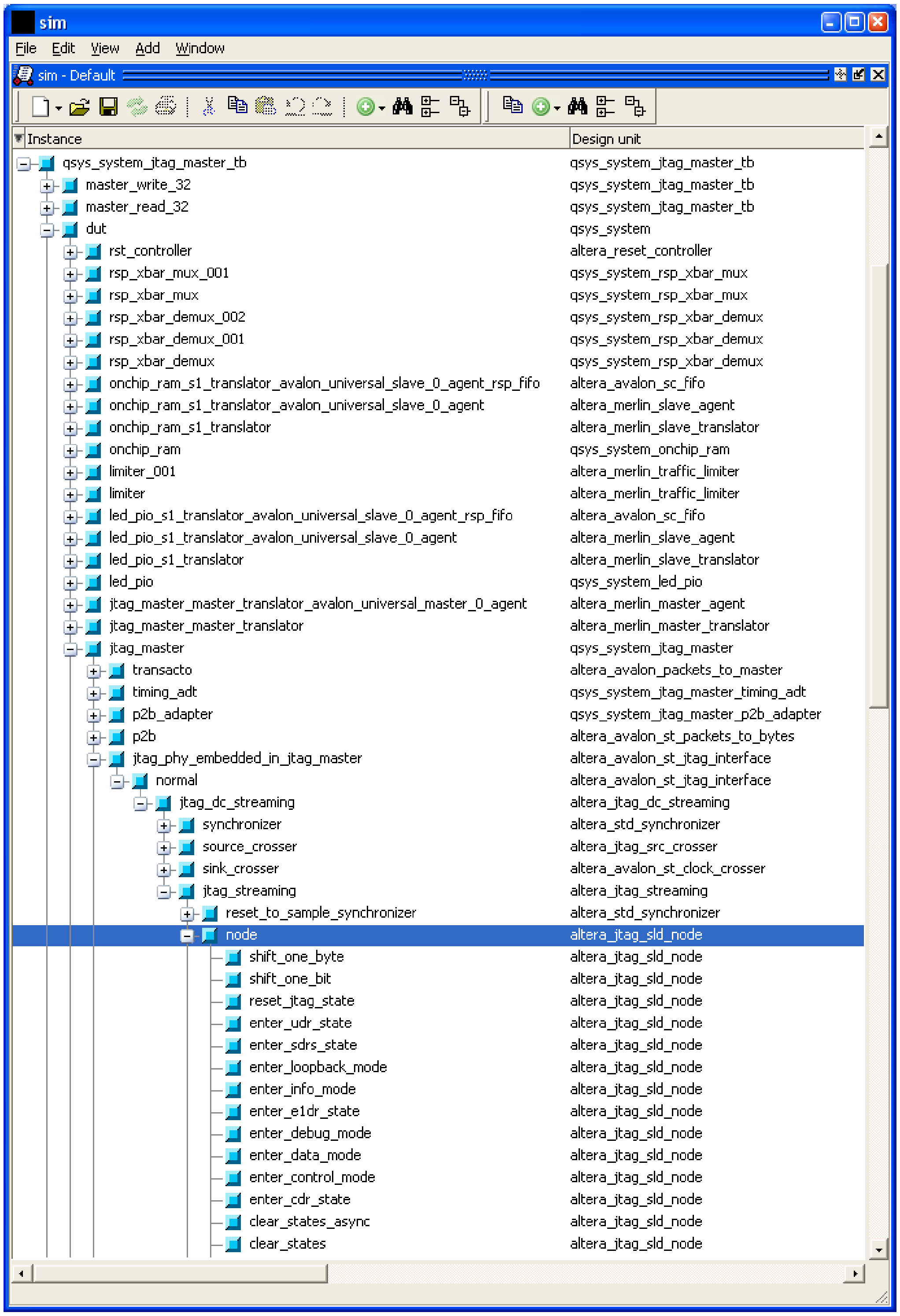
  ```
  VSIM> do $TUTORIAL/hdl/qsys_system/scripts/qsys_system_jtag_master_tb.do
  ```

- Run the simulation

  ```
  VSIM> run -a
  ```

The testbench console output matches that generated by SOPC Builder in Figure 6.

Figure 12 shows the path to the JTAG node within the JTAG-to-Avalon-MM bridge. The path to this node was first determined by elaborating `qsys_system` (rather than the testbench), and then using the Modelsim hierarchy window to determine the path to the node.



Instance	Design unit
qsys_system_jtag_master_tb	qsys_system_jtag_master_tb
master_write_32	qsys_system_jtag_master_tb
master_read_32	qsys_system_jtag_master_tb
dut	qsys_system
rst_controller	altera_reset_controller
rsp_xbar_mux_001	qsys_system_rsp_xbar_mux
rsp_xbar_mux	qsys_system_rsp_xbar_mux
rsp_xbar_demux_002	qsys_system_rsp_xbar_demux
rsp_xbar_demux_001	qsys_system_rsp_xbar_demux
rsp_xbar_demux	qsys_system_rsp_xbar_demux
onchip_ram_s1_translator_avalon_universal_slave_0_agent_rsp_fifo	altera_avalon_sc_fifo
onchip_ram_s1_translator_avalon_universal_slave_0_agent	altera_merlin_slave_agent
onchip_ram_s1_translator	altera_merlin_slave_translator
onchip_ram	qsys_system_onchip_ram
limiter_001	altera_merlin_traffic_limiter
limiter	altera_merlin_traffic_limiter
led_pio_s1_translator_avalon_universal_slave_0_agent_rsp_fifo	altera_avalon_sc_fifo
led_pio_s1_translator_avalon_universal_slave_0_agent	altera_merlin_slave_agent
led_pio_s1_translator	altera_merlin_slave_translator
led_pio	qsys_system_led_pio
jtag_master_master_translator_avalon_universal_master_0_agent	altera_merlin_master_agent
jtag_master_master_translator	altera_merlin_master_translator
jtag_master	qsys_system_jtag_master
transacto	altera_avalon_packets_to_master
timing_adt	qsys_system_jtag_master_timing_adt
p2b_adapter	qsys_system_jtag_master_p2b_adapter
p2b	altera_avalon_st_packets_to_bytes
jtag_phy_embedded_in_jtag_master	altera_avalon_st_jtag_interface
normal	altera_avalon_st_jtag_interface
jtag_dc_streaming	altera_jtag_dc_streaming
synchronizer	altera_std_synchronizer
source_crosser	altera_jtag_src_crosser
sink_crosser	altera_avalon_st_clock_crosser
jtag_streaming	altera_jtag_streaming
reset_to_sample_synchronizer	altera_std_synchronizer
node	altera_jtag_sld_node
shift_one_byte	altera_jtag_sld_node
shift_one_bit	altera_jtag_sld_node
reset_jtag_state	altera_jtag_sld_node
enter_udr_state	altera_jtag_sld_node
enter_sdrs_state	altera_jtag_sld_node
enter_loopback_mode	altera_jtag_sld_node
enter_info_mode	altera_jtag_sld_node
enter_e1dr_state	altera_jtag_sld_node
enter_debug_mode	altera_jtag_sld_node
enter_data_mode	altera_jtag_sld_node
enter_control_mode	altera_jtag_sld_node
enter_cdr_state	altera_jtag_sld_node
clear_states_async	altera_jtag_sld_node
clear_states	altera_jtag_sld_node

Figure 12: Altera JTAG node location in the Qsys system JTAG-to-Avalon-MM master testbench, `qsys_system_jtag_master_tb`. The tasks under the highlighted JTAG `node`, i.e., `shift_one_byte` down to `clear_states`, are used to simulate the JTAG-to-Avalon-MM bridge.

4.6 Synthesis and Simulation Scripts

The Qsys design can be synthesized as follows;

- Start Quartus.
- Change to the Qsys BeMicro-SDK project and source the synthesis script

```
tcl> set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
tcl> cd $TUTORIAL/hdl/boards/bemicro_sdk/qsys_system
tcl> source scripts/synth.tcl
```

- The first time the script is run, it will copy the `qsys_system.qsys` file to a work directory, and request the user to manually run the Qsys GUI.

  The `.qsys` file does not preserve the settings on the *Generate* tab, so uncheck *Create block symbol file* and select *Verilog* for Create Simulation Model. Click *Generate* to generate the system files.

  Source the synthesis script (use the up-arrow in the Quartus Tcl console to bring back the last command issued)

```
tcl> source scripts/synth.tcl
```

- Synthesis of the Qsys design should complete without error.

  The Quartus *Processing* window will generate warning messages (blue text). Warning messages relating to HDL coding style are generated for some of the Altera IP, eg., missing connections for the Avalon-MM BFM master (since it is for simulation only), signals that were assigned and never read, and truncated signals. Ideally, these warnings would be eliminated or suppressed by the Altera IP developers, either by correcting the code or using synthesis constraints.

The Qsys design can be simulated as follows;

- Start Quartus and *Generate* the Qsys system (making sure the *Verilog* simulation option is checked).
- Start Modelsim.
- Change to the Qsys system directory and source the simulation script

```
ModelSim> cd $TUTORIAL/hdl/qsys_system
ModelSim> source scripts/sim.tcl
```

- The simulation script uses the `msim_setup.tcl` script to compile the Qsys source, and creates two Tcl procedures that can be used to run the simulation, i.e.,

```
# JTAG-to-Avalon-MM tutorial testbench procedures
# -----------------------------------------------
#
#   qsys_system_bfm_master_tb  - run the Avalon-MM BFM testbench
#   qsys_system_jtag_master_tb - run the JTAG-to-Avalon-MM testbench
```

  Issue either of these commands to run the respective simulation.

- The simulation script creates a working directory called `mwork`, changes into that directory, sources and then calls procedures in the `msim_setup.tcl` script.

  The `msim_setup.tcl` procedures copy files into the working directory and create Modelsim library mappings in a subdirectory called `libraries`. The library mappings are created with *relative* path names, so the testbenches must be run from within the `mwork` directory, otherwise Modelsim cannot locate the design components.

  To re-run the simulation script, quit the simulation (since Modelsim will not allow you to change directories otherwise), change directory to the top-level directory and source the script again, i.e.,

  ```
  VSIM> quit -sim
  ModelSim> cd $TUTORIAL/hdl/qsys_system
  ModelSim> source scripts/sim.tcl
  ```

  Alternatively, if you simply edited the top-level testbench, recompile it and restart the simulation via

  ```
  VSIM> vlog -sv ../test/qsys_system_bfm_master_tb.sv -L qsys_system_bfm_master
  VSIM> restart -f; run -a
  ```

The tutorial source contains Qsys projects for Quartus synthesis on the BeMicro-SDK, BeMicro, and DE2 boards. The Qsys Modelsim simulation script would ideally be board agnostic, however, it needs to use Quartus generated Qsys source. By default the Qsys Modelsim simulation script checks for the existence of the BeMicro-SDK Quartus work directory (which is setup by the synthesis script). If you synthesize the Qsys design for the BeMicro or DE2 board, and want to simulate that source, you need to edit the the `board` variable in the `sim.tcl` script to reflect the path to the board you are targeting. However, since Quartus generates the same Qsys simulation files regardless of the board type, just follow the synthesis procedure above for the BeMicro-SDK board to create the files needed for Qsys simulation (no editing of tutorial scripts required).

5 Host-to-FPGA Communications

The goal of this tutorial is to demonstrate communications between a client application and an Altera Avalon system such as that shown in Figure 1. Previous sections have shown how to construct the Altera Avalon hardware using both SOPC Builder and Qsys tools. This section shows how to communicate with that hardware via the USB-Blaster interface and Altera-provided applications.

5.1 System Console

The Quartus II Handbook, Volume 3, Chapter 10, *Analyzing and Debugging Designs with the System Console*, describes the *System Console* interactive debugging console. Table 10-3 on pages 10-7 and 10-8 lists the console commands [4].

The BeMicro-SDK board was configured with the Altera Avalon system developed in this tutorial. Figure 13 shows the System Console commands issued to interact with the BeMicro-SDK (the session was repeated for both the SOPC and Qsys hardware configurations).

The System Console procedures were used to create higher-level Tcl procedures for controlling the LEDs, reading the switches and push-button, and for accessing SRAM. The script is located in the BeMicro-SDK shared scripts directory,

```
$TUTORIAL/hdl/boards/bemicro_sdk/share/scripts/jtag_cmds_sc.tcl
```

Similar scripts exist for the BeMicro and DE2 boards. Read the scripts for the slight differences in procedures (due to the slight difference in hardware available on these boards). Figure 14 shows an interactive System Console session with the BeMicro-SDK. Note how the device was never opened; the scripts use a Tcl global variable to track whether the JTAG interface is open, and if it is not, the procedures automatically open the JTAG interface. Read the script source for details.

5.2 `quartus_stp`

The command-line tool `quartus_stp` can also be used for JTAG access. Unfortunately, Altera does not provide Tcl procedures for accessing the JTAG-to-Avalon-MM master component from within `quartus_stp`. The functionality of the JTAG-to-Avalon-MM master communications was reverse-engineered from the source code and SignalTap II logic analyzer traces in [5], and from that analysis a set of Tcl procedures was developed. The tutorial source contains the procedures in the directory

```
$TUTORIAL/tcl/altera_jtag_to_avalon_stp
```

The procedures are written as a Tcl package. Use of the package requires the user to either; configure the environment variable `TCLLIBPATH` to point to the package directory, or the package directory can be copied into the Altera Tcl packages directory, eg.,

```
c:/software/altera/11.1sp1/quartus/common/tcl/packages
```

Figure 15 shows an interactive `quartus_stp` session with the BeMicro-SDK. The session starts by loading the JTAG-to-Avalon-MM master package, and then importing the Tcl procedure names (saving you typing the namespace of the package as a prefix to every command). For details on Tcl packages and namespaces see [7].

```
# Get the list of master services
% set masters [get_service_paths master]
{/devices/EP3C25|EP4CE22@1#USB-0/(link)/JTAG/(110:132 v1 #0)/phy_0/master}

# Select the first master
% set master [lindex $masters 0]

# Open the master service
% open_service master $master

# Write to the LEDs
% master_write_32 $master 0 0x55

# Read the push-button and switches SW[2:1]
% master_read_32 $master 0x10 1
0x00000000

# Change SW[1] and re-read
% master_read_32 $master 0x10 1
0x00000001

# Change SW[2] and re-read
% master_read_32 $master 0x10 1
0x00000003

# Hold down the push-button and re-read
% master_read_32 $master 0x10 1
0x00000007

# Write four 32-bit locations in SRAM
% master_write_32 $master 0x1000 [list 0x33221100 0x77665544 0xbbaa9988
0xffeeddcc]

# Read four 32-bit locations from SRAM
% master_read_32 $master 0x1000 4
0x33221100 0x77665544 0xbbaa9988 0xffeeddcc

# Read sixteen 8-bit locations from SRAM
% master_read_memory $master 0x1000 16
0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xaa 0xbb 0xcc 0xdd 0xee
0xff

# Close the master service
% close_service master $master
```

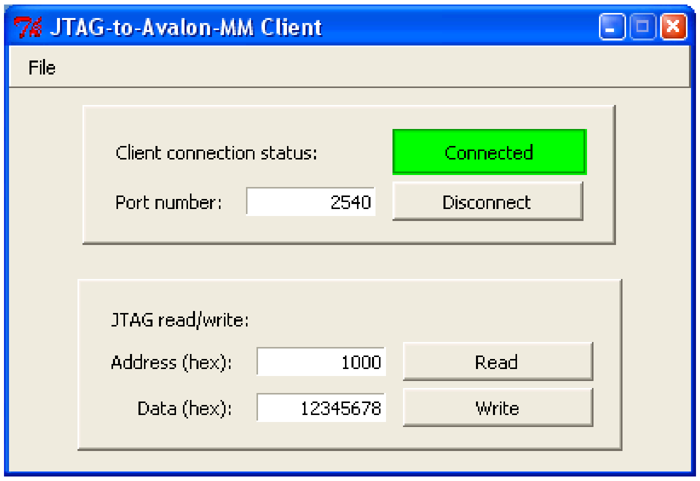
Figure 13: System Console interactive session controlling the BeMicro-SDK board configured with the Altera Avalon system shown in Figure 1. The lines starting with # are comments and were not entered at the console.

```
# Source the System Console JTAG Tcl procedures
% set TUTORIAL c:/temp/altera_jtag_to_avalon_mm_tutorial
% source $TUTORIAL/hdl/boards/bemicro_sdk/share/scripts/jtag_cmds_sc.tcl

# Write/read the LEDs
% led_write 0x23
% led_read
0x23

# Read the switches
% sw
2

# Read the push-button (not pressed)
% pb
0

# Read the push-button (pressed)
% pb
1

# Write/read the SRAM
% sram_write 0 0x12345678
% sram_read 0
0x12345678
```

Figure 14: System Console interactive session controlling the BeMicro-SDK board using the Tcl procedures implemented in `jtag_cmds_sc.tcl`. The lines starting with `#` are comments and were not entered at the console.

```
# Print TCLLIBPATH (needed to load the JTAG-to-Avalon-MM Tcl package)
tcl> puts $env(TCLLIBPATH)
c:/temp/altera_jtag_to_avalon_mm_tutorial/tcl/altera_jtag_to_avalon_stp

# Load the JTAG-to-Avalon-MM Tcl package and import the package commands
tcl> package require altera_jtag_to_avalon_stp
1.0
tcl> namespace import altera_jtag_to_avalon_stp::*

# Print the list of JTAG commands (commands prefixed with jtag)
tcl> info commands jtag*
jtag_idcode jtag_pulse_nconfig jtag_read jtag_node_id jtag_node_is_bytestream
jtag_open jtag_resetrequest jtag_write jtag_node_is_master jtag_send
jtag_print_hub_info jtag_close jtag_usercode jtag_print_node_info
jtag_number_of_nodes

# Open the JTAG interface
tcl> jtag_open
JTAG: USB-Blaster [USB-0], FPGA: @1:  EP3C25/EP4CE22 (0x020F30DD)

# Print the JTAG hub info
tcl> jtag_print_hub_info
Hub info:  0x8086E04
VIR m-width:  4
VIR n-width:  1
Manufacturer ID: 0x6E
Number of nodes:  1
IP Version:  1

# Print the JTAG node info
tcl> jtag_print_node_info
Node index:  0
Node instance:  0 (0x0)
Node manufacturer:  110 (0x6E)
Node ID: 132 (0x84)
Node purpose:  1 (0x1)
Node version:  1 (0x1)

# Write/read the LEDs (the first argument is the JTAG node index)
% jtag_write 0 0 0x23
% jtag_read 0 0
0x23

# Write/read the SRAM
% jtag_write 0 0x1000 0x12345678
% jtag_read 0 0x1000
0x12345678
```

Figure 15: `quartus_stp` interactive session controlling the BeMicro-SDK board.



Port number: 2540
Address (hex): 1000
Data (hex): 12345678

Figure 16: `quartus_stp` JTAG client Tcl/Tk GUI.

5.3 Client/Server

Altera does not provide a shared library or DLL for accessing the JTAG interface from custom code, so how then do you write code in your favorite programming language, eg., C/C++, MATLAB, or LabView? One solution is to use System Console or `quartus_stp` to implement a TCP/IP server that provides hardware access, and implement your custom code as a TCP/IP client application. The tutorial source contains Tcl-based server and client applications in the directory

```
$TUTORIAL/tcl/jtag_client_server
```

Figure 16 shows a Tcl/Tk based client graphical user interface (GUI). The client can issue read and write requests to the server, and the server then performs those accesses on the hardware. The figure shows the result of reading the first address in SRAM (which was previously written with the value 0x12345678).

Figure 17 shows the Tcl/Tk based server console output for `quartus_stp`. Note the comment `Handle the client via a fileevent callback`. The Tcl `fileevent` command is critical to the implementation of a Tcl TCP/IP server [7]. Versions of System Console earlier than Quartus II version 11.1sp1 did not support the Tcl `fileevent` procedure, so they could not implement a proper server, i.e., a server able to handle multiple clients. Figure 18 shows the server output for System Console (Quartus II version 11.1sp1).

The server application has a debug mode. If the server is started via

```
% set debug 1
% source jtag_server.tcl
```

then the server starts in a debug mode where hardware accesses are not performed, and client read/write accesses are performed on a server variable, eg., the client can write to an address and then read it back. The server debug mode allows you to test the client/server interface without having access to hardware. It also allows the client/server TCP/IP communications path to be tested without hardware interaction.

To test the multiple client handling of the server (in either hardware or debug mode), use the client to write the hexadecimal values (address,data) = (1000, 11111111), (1004, 22222222), (1008, 33333333), and (100C, 44444444). Click on the client disconnect button, then reconnect and read from the four hexadecimal addresses 1000, 1004, 1008, 100C. Start multiple `quartus_stp` clients, or start the client in another Tcl/Tk tool, eg., the ActiveState ActiveTcl Tcl/Tk shell `wish84`, and use the multiple clients to write and read to different server addresses.

The client/server source also contains a command-line C language client, `jtag_client.c`. The client can be built at the NIOS IDE shell using

```
bash-3.1$ gcc -Wall -o jtag_client jtag_client.c
```

The client can write to multiple SRAM locations using

```
./jtag_client -w 0x1000 -d 0x11111111
./jtag_client -w 0x1004 -d 0x22222222
./jtag_client -w 0x1008 -d 0x33333333
./jtag_client -w 0x100C -d 0x44444444
```

and read back using

```
./jtag_client -r 0x1000
./jtag_client -r 0x1004
./jtag_client -r 0x1008
./jtag_client -r 0x100C
```

The NIOS IDE shell generates output showing the address and data. The server console shows how the client opens and closes a socket connection each time the client application runs.

The client/server example performs TCP/IP communications using ASCII strings, and the server performs hardware access using 32-bit read/write commands. The performance bottleneck with this approach is not the use of ASCII over TCP/IP, but the server hardware access. When performing accesses to large numbers of bytes, the JTAG 32-bit read/write commands do not make the most efficient use of the underlying JTAG bytestreams. If your custom server needs a hardware access performance boost, then use the System Console `master_read_memory` and `master_write_memory` commands. The JTAG bytestream performance is analyzed in [5]. For simple hardware accesses, eg., updating LEDs, reading switches and push buttons, and accessing sensors, the 32-bit read/write routines used in the client/server example are more than sufficient.

```
tcl> source jtag_server.tcl
JTAG server running under quartus_stp

Open JTAG to access the JTAG-to-Avalon-MM master
JTAG: USB-Blaster [USB-0], FPGA: @1:  EP3C25/EP4CE22 (0x020F30DD)

Start the server on port 2540

Wait for clients

Accept sock1284 from 127.0.0.1 port 1427
Handle the client via a fileevent callback
SERVER (sock1284):  jtag_read 0x00000000
SERVER: jtag_read 0x00000000
SERVER (sock1284):  0x00000045
SERVER (sock1284):  jtag_read 0x00001000
SERVER: jtag_read 0x00001000
SERVER (sock1284):  0x12345678
```

Figure 17: `quartus_stp` JTAG server console output.

```
% source jtag_server.tcl
JTAG server running under system console

This version of SystemConsole (11.1sp1 216) supports fileevent.
The server can support multiple clients.

Open JTAG to access the JTAG-to-Avalon-MM master

Start the server on port 2540

Wait for clients

Accept sock2024 from 127.0.0.1 port 1466
Handle the client via a fileevent callback
SERVER (sock2024):  jtag_read 0x00000000
SERVER: jtag_read 0x00000000
SERVER (sock2024):  0x00000045
SERVER (sock2024):  jtag_read 0x00001000
SERVER: jtag_read 0x00001000
SERVER (sock2024):  0x12345678
```

Figure 18: System Console JTAG server console output.

A Software Versions

Current version of Quartus

This tutorial was tested using the following Quartus II 11.1sp1 editions, and operating systems;

- The tutorial was written to be implemented using Quartus II 11.1sp1.
- The tutorial was developed using Quartus II 11.1sp1 (full edition) under Windows XP Professional 32-bit. The synthesis scripts for all boards and both SOPC and Qsys systems were fully tested. The simulation script was tested targeting the hardware generated for the BeMicro-SDK. Each board was configured with the SOPC or Qsys designs, and system console was used to interactively check each design.
- Linux Centos 6.2 32-bit running Quartus II 11.1sp1 full-edition; tested the synthesis scripts, the simulation script with the BeMicro-SDK generated files, and hardware tested using system console.
- Windows 7 Professional 64-bit running Quartus II 11.1sp1 full-edition (32-bit and 64-bit); tested the BeMicro-SDK synthesis and simulation scripts, and hardware tested using system console.
- Windows 7 Professional 64-bit running Modelsim SE 10.0c 64-bit; tested the simulation scripts with the BeMicro-SDK generated files.

  The Qsys BFM testbench fails to load due to the DLL `bytestream_pli.dll` being incompatible with 64-bit Modelsim. This DLL is not used by the testbench, so the BeMicro-SDK Qsys project `msim_setup.tcl` script `elab` procedure was edited to remove the `vsim` PLI argument. The Qsys BFM testbench then loads and runs correctly.
- Quartus II v11.1sp1 Web Edition for;
  - Windows XP (32-bit)
  - Linux Centos 6.2 (32-bit)
  - Linux Ubuntu 11.10 (32-bit)

  Each installation was used to test the BeMicro-SDK synthesis and simulation scripts, and hardware tested using system console.

  Each Web Edition installation was tested running in a VirtualBox (version 4.1.8) virtual machine (VM). The VMs were tested from hosts running Windows XP, Centos 6.2, and Windows 7. The USB-Blaster can be captured by the VM under Windows XP and Centos 6.2, but not under Windows 7 (so the issue is likely with the host VM support, rather than with the client).

  The tutorial scripts use the environment variable `QUARTUS_ROOTDIR`. This variable is created automatically by Quartus II under Windows, but under Linux it needs to be defined by the user. For example, under Linux, after installing Quartus II Web Edition and Modelsim-ASE into the directory `/opt/altera/11.sp1_free`, the user `.bashrc` should be edited to add

  ```
  export QUARTUS_ROOTDIR=/opt/altera/11.sp1_free/quartus
  export PATH=$PATH:$QUARTUS_ROOTDIR/bin
  export PATH=$PATH:/opt/altera/11.sp1_free/modelsim_ase/linuxaloem
  ```

  The commands `quartus` and `vsim` can then be used from the bash shell.

Older versions of Quartus

- **SOPC Builder**

  The SOPC Builder GUI and generated components have changed slightly between Quartus version 10.1 and 11.1sp1. The slight differences in SOPC Builder GUI paths to components is noted in the tutorial. The source generated by SOPC Builder has changed with Quartus 11.1sp1, eg., the JTAG-to-Avalon-MM bridge is now described solely by a `_hw.tcl` file, with the component being dynamically created by SOPC Builder. Previous versions of Quartus had an explicit JTAG master component (which looked like it was SOPC Builder generated source that was copied to the Quartus IP directory).

- **Qsys**

  The Qsys GUI, generated scripts, and source have changed significantly since 10.1, where Qsys was in beta format. No attempt was made to try and support earlier versions of Qsys (it looked too painful).

B Tutorial Source

The tutorial zip file, `altera_jtag_to_avalon_mm_tutorial.zip`, unzips to create the directory layout shown in Table 1. There are slight differences in how zip extraction tools work under Windows and the bash shell (Cygwin or Linux). To unzip the tutorial zip file into a directory named `altera_jtag_to_avalon_mm_tutorial`, perform the following;

- **Windows extraction**

  Under Windows Explorer, select the zip file, and then right click and select *Extract all* ...

- **Bash command line extraction**

  Unzip using

  ```
  unzip -d altera_jtag_to_avalon_mm_tutorial altera_jtag_to_avalon_mm_tutorial.zip
  ```

  If the `-d` option is not used, then the zip file creates the directories `doc`, `hdl`, and `tcl` in the current directory. Alternatively, you can first create the directory and then unzip, eg.,

  ```
  mkdir altera_jtag_to_avalon_mm_tutorial
  cp altera_jtag_to_avalon_mm_tutorial.zip altera_jtag_to_avalon_mm_tutorial/
  cd altera_jtag_to_avalon_mm_tutorial/
  unzip altera_jtag_to_avalon_mm_tutorial.zip
  ```

  Using the `-d` option is recommended (as it involves less typing).

  The contents of the zip file can be listed using

  ```
  unzip -l altera_jtag_to_avalon_mm_tutorial.zip
  ```

C Altera Tool Improvement Recommendations

During the development of the tutorial, problems experienced with the user interface, the tool IP, or the tool design philosophy were cross-referenced to this appendix. The text references the following note numbers;

1. Page 8: Why does SOPC Builder forget the SOPC System name?

   When the *Generate* button is pressed to generate the SOPC System, a pop-up dialog asks if the `unnamed` system should be saved. The system was however named when the SOPC Builder GUI was started, so this dialog should not be necessary.

2. Page 10: Synthesis issues with the Avalon-MM Master BFM.

   The Avalon-MM Master BFM component is intended for *simulation*-only. However, it can be incorporated into a system that is both simulated and synthesized. The source code for the component should include synthesis directives that disable the component during synthesis (by tying signals to deasserted levels), so that the synthesis tool can eliminate the logic without generated warnings about missing drivers and dangling pins.

3. Page 12: SOPC Builder *Run Simulator* Button.

   The button should not be activated until the `.mpf` file has been generated.

4. Page 12: Missing source files for Avalon-MM Master BFM simulation.

   The `sopc_system.v` verilog file does not include the packages required to simulate the Avalon-MM Master BFM.

5. Page 24: Using Verilog `include` statements to resolve source dependencies.

   In the SOPC Builder example,

   - In the `sopc_system.v` Verilog source, above the `test_bench` component, Verilog `include` statements have been used to include a mixture of code from the Quartus install directory, code copied to the project directory, and generated code (the list of includes changes depending on whether the *Simulation* check-box is checked or not).

     This support or library code should really be included into the project using the scripting features of the tool, eg., using Quartus or Modelsim Tcl commands, it should not use a Verilog-specific language feature embedded within a generated source file.

     For more proof for this argument, consider that VHDL does not even support this type of include construct.
   - Because the SOPC System file `sopc_system.v` includes library source directly, this source is compiled *every* time the SOPC System is changed. This unnecessarily compiles source that has not changed. Under Modelsim, the only source that would need to be recompiled would be the interconnect and any new components added to the system.
   - The use of absolute paths in the `include` statements means that this code is not portable between machines, so the code should not be checked into a code versioning system. Admittedly, this is generated code, so the use of the absolute paths could be tolerated.
   - The Verilog `include` statements are not used consistently. In SOPC Builder, if you *do not* check the *Simulation* checkbox, then the JTAG master components are not listed in the `include` statements, whereas, if you do check the check box, the components are listed. The JTAG master is a *synthesizable* component, it should always be included! If there are differences between synthesis and simulation, then those differences should be hidden in the source code using synthesis directives (which both Verilog and VHDL support).

In this particular example, the *synthesis* tool is provided the path to the JTAG component in the `sopc_builder.qip` file. This is a better solution, however, it is not reuseable by Modelsim.

- Verilog and VHDL source files can be described via two strings; the source file name and the library in which that source should be compiled. The HDL source often has a compilation order requirement, eg., VHDL packages need to be compiled before use, and Verilog modules need to be compiled before they are instantiated in other modules.

  A general-purpose approach to including source would be for SOPC Builder to generate a list of source files per library, or a list of source file and library name pairs, with the list in the appropriate compilation order. Tcl synthesis and simulation scripts can then parse that list and issue the appropriate Quartus or Modelsim commands to include that source for compilation. This would allow source to be included into projects using *scripting* features, not HDL language features.

6. Pages 14 and 27: The copying of library component source to multiple project directories complicates verification.

   The Modelsim simulator can be used to compile Verilog and VHDL source code into libraries. The intention for those libraries is that they contain the components that are reusable across multiple projects. For example, Modelsim-ASE ships with the Altera LPM `lpm` and Megafunction `altera_mf` components pre-compiled for Verilog and VHDL in the directory `c:/software/altera/11.1sp1/modelsim_ase/altera/`. However, this is not how Quartus 11.1sp1 operates for SOPC Builder library components. For example, in the SOPC Builder example on page 14,

   - For synthesis, the JTAG-to-Avalon-MM master source code is copied from the Quartus II installation into the directory `jtag_master`.
   - For simulation, the JTAG-to-Avalon-MM master source code is copied from the Quartus II installation into the directory `jtag_master_sim`, and the `sopc_system.v` Verilog source, `test_bench include` statements refer to this new directory, eg., `‘include "jtag_master_sim/jtag_master.v"`.

     In addition, the directory `jtag_master_sim`, also contains a *copy* of the `jtag_master.v` component instance from the project directory (which is just one directory above the copy).

   Why do all these copies complicate things? Well, if you are verifying all of your designs using Modelsim, then you should use Modelsim to build the library components into libraries, and then reuse those libraries while verifying all of the *project-specific* generated components. If there was a valid argument for copying library components to the project directory, then there should only be a *single* copy, the same copy could be used for both synthesis and simulation, and Modelsim could be used to create a project specific SOPC builder component library. However, SOPC Builder copies the library components a per-*component instance* basis. For example, if you add another JTAG-to-Avalon-MM bridge component called `jtag_master_two`, and regenerate the system with *Simulation* checked, then two new directories are created with additional copies of the library component source.

   Quartus 10.1 would generate Verilog `include` statements that referred to the original source files in the Quartus II installation (the SOPC Builder IP directory). This method at least points to the installed source area, but still uses a Verilog-specific language feature to include source, whereas scripting features should be used, as commented above.

   The Qsys example on page 14 also copies code, but with a few differences relative to SOPC Builder;

- Qsys creates a copy of library source code in synthesis and simulation directories;

  `qsys_system/synthesis/submodules`
  `qsys_system/simulation/submodules`

- The top-level `qsys_system` file no longer contains Verilog `include` statements, the source files are instead included in a tool-specific manner; Quartus is provided the source information in the Tcl script

  `qsys_system/synthesis/qsys_system.qip`

  while Modelsim is provided the files via the Tcl script

  `qsys_system\simulation\mentor\msim_setup.tcl`

  The Modelsim Tcl script hard-codes the library mappings into the relative directory `./libraries` making this script difficult to support in a verification system that uses multiple versions of Modelsim, eg., Modelsim-ASE and Modelsim-SE (the binary library files produced by different versions of Modelsim are not compatible). Page 32 has comments on difficulties experienced with scripting the Qsys design.

  My preference would be to see these separate methods combined into a parseable Tcl list containing the source file name and the target library name. This would ease the writing of custom synthesis and simulation scripts.

7. Page 24: The Qsys system file does not contain all of the system settings.

   The Qsys file `.qsys` does not preserve the state of the generate tab checkboxes. The state of the checkboxes and the simulation pull-down menus is stored in a preferences file. For example, in the Qsys example on page 24, the Qsys *Generation* tab options are stored in the XML file `.qsys_edit/preferences.xml` in an XML entry called `generation`, containing key-value pairs for each of the GUI settings, eg., in the Qsys generation tab

   - If the *Create block symbol file* checkbox is unchecked, that selection is stored in the preferences file as `<generation block_symbol_file="0"/>`.
   - If the *Create simulation model* pull-down menu selects *Verilog*, that selection adds the key-value pair `simulation="VERILOG"` to the existing `generation` entry.

   The use of an *additional* file to store the GUI settings complicates the re-generation of a Qsys system with simulation support (from a minimal set of files), as now both the `.qsys` and preferences files are required. Without the preferences files, the simulation files will *not* get generated.

   This is a change from SOPC Builder, where the system (including simulation settings) could be regenerated using only the `.sopc` file.

8. Page 25: Port names exported from Qsys are *not* what the user entered.

D Altera Documentation Web Links

- Quartus II Software Support
- Quartus II Development Software Documentation
- Qsys System Integration Tool Support
- SOPC Builder Documentation
- SOPC Builder Support
- NIOS II Processor Documentation
- Nios II Embedded Design Suite Support

References

[1] Altera Corporation. Avalon Interface Specifications (version 1.3, for SOPC Systems), August 2010. (mnl_avalon_spec_1_3.pdf).

[2] Altera Corporation. Applying the Benefits of Network on a Chip Architecture to FPGA System Design, April 2011. (wp-01149-noc-qsys.pdf).

[3] Altera Corporation. Avalon Interface Specifications (version 2.0, for Qsys Systems), May 2011. (mnl_avalon_spec.pdf).

[4] Altera Corporation. Quartus II Handbook (version 11.1), November 2011. (quartusii_handbook.pdf).

[5] D. W. Hawkins. Altera JTAG-to-Avalon Analysis, January 2012. (altera_jtag_to_avalon_analysis.pdf, altera_jtag_to_avalon_analysis.zip).

[6] D. W. Hawkins. Altera JTAG-to-Avalon MM Tutorial, March 2012. (altera_jtag_to_avalon_mm_tutorial.pdf, altera_jtag_to_avalon_mm_tutorial.zip).

[7] B. B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 3rd edition, 2000.