ECE 128 – Synopsys Tutorial: Using BSD Compiler

Created at GWU by William Gibb, Spring 2010

Objectives:

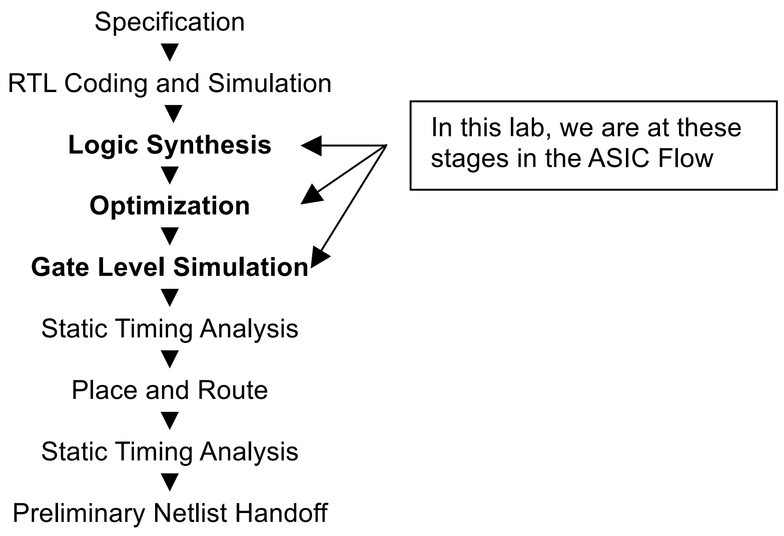
- Use Synopsys BSD Compiler, synthesize ‘boundary-scan’ structures into verilog code
- Use Verilog to test the output of the BSD Compiler’s patterns against the synthesized synopsys code

Assumptions:

- Student has completed lab 5
- Student a basic understanding of boundary-scan and scan-chain theory.

Introduction:

The ASIC design flow is as follows:



In lab 4 we introduced the Synopsys Logic Synthesis tool called the DFT Compiler. In this lab we will use the Design Compiler to insert Boundary-Scan structures into our synthesized verilog code. We will then use the Synopsys BSD Compiler® tool to insert these structures. Prior to using these tools, you should be familiar with the basics of boundary scan and how it is used throughout the industry.

Boundary scan differs from the testing methods that we have seen so far. With ATPG and Scan Chains, we have been interested in testing an Integrated Circuit for manufacturing errors. Boundary scan is capable of running such tests, but it is chiefly used to test a whole-chip once the chip has been packaged and populated onto a PCB. This process is characterized in IEEE standard 1149.x, with 1149.1 being the most common – it is commonly referred to as **JTAG.**

The process of building a boundary-scan design is done in a three step process:

- Add a special set of ports, test access ports.
- Add a special controller, a TAP controller.
- Insert BS logic: Isolate all of the input and output buffers of the chip and turns them into a single scan chain, that data can be shifted through them with the TAP controller.

For a better idea of what Boundary Scan consists of, see figures 1 and 2 on the following page. In Figure 1, a design has been built, with a scan chain added and some additional TAP ports added, but not connected to anything. After we insert the TAP controller and wire up the BSD Chain, we can see what a design looks like with Boundary Scan added to it in Figure 2.

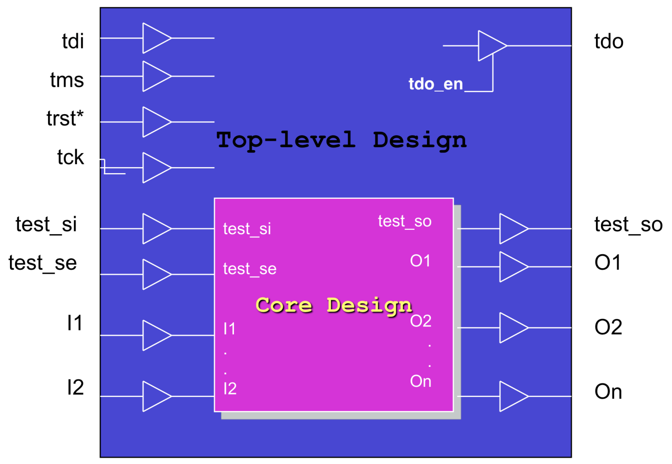


Figure 1 - BSD Ready Design

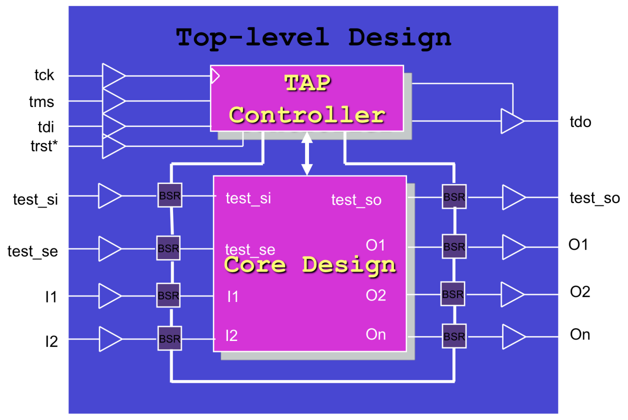


Figure 2 TAP Inserted Design

Multiple devices on a single board can be connected together in a single boundary-scan chain, allowing an engineer to reduce the overhead of testing multiple devices on a single board. Quickly, data can be shifted into the start of the JTAG chain and moved to the appropriate device, which can then be tested and observed. This can be seen in Figure 3 on the following page.

Boundary scan is not limited to testing in-board, but has several other uses as well. These include:

- Device debugging systems in development – A microcontroller may have a JTAG interface designed in order to allow a debugger access to internal signals, such as program registers, in order to facilitate debugging cheaper than it would be to write a whole new test protocol.

- Device configuration – FPGA devices are commonly configured using JTAG chains. The device TAP often has access to the internal configuration scan-chain, and it makes use of feeding the boundary-scan input into the device configuration chain.

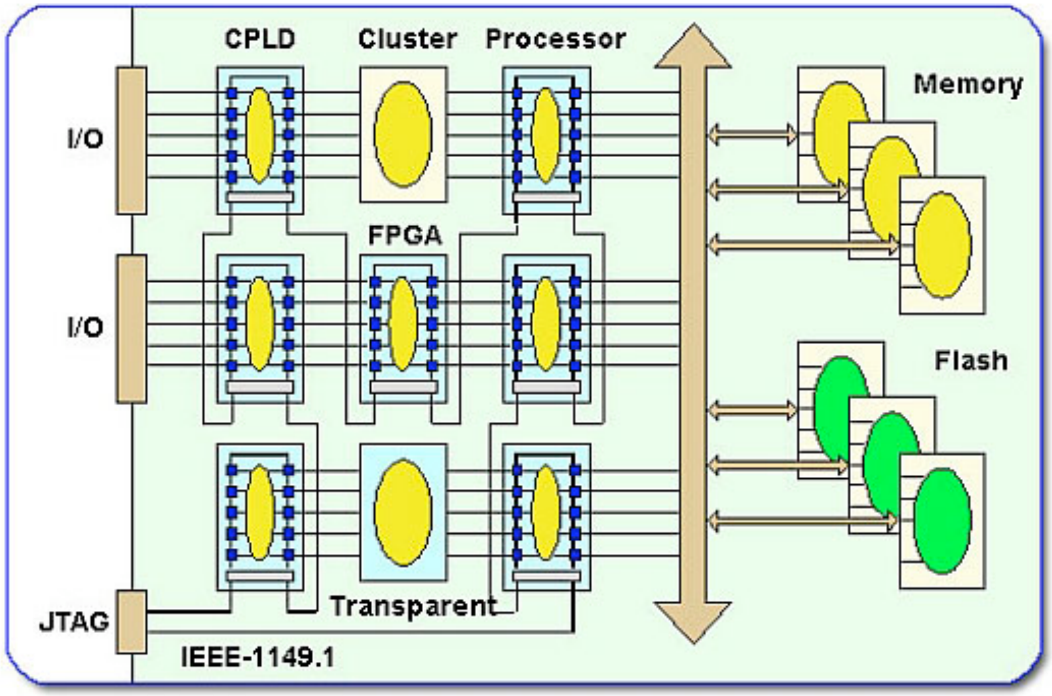


Figure 3 Board Level Boundary Scan Chain - Copyright Corelis.com

In this lab, we will use Synopsys tools to implement the **TAP Controller** with Verilog code. But if you'd like to understand in greater detail about the concepts discussed above, review the BSD Compiler User Guide posted on the VLSI website.

Overview of this Boundary Scan Tutorial:

We will use Synopsys Design Vision to automatically insert the boundary scan controller into our synthesized design and to connect it to IO buffers. We will use a variant of the ripplecarry4_clk design that has been run through the dc_test.tcl script, in order to build it with a scan chain. It has been wrapped around with IO pads from the synopsys gtech library for us already. We'll be driving Design Vision in 3 parts – loading in the design, configuring the boundary scan core & inserting it, then writing out all of the appropriate files to test the TAP controller. We will then test the TAP controller and observe its functionality.

Part I: File Setup for This Lab

Any time you wish to “synthesize” some verilog code, create a directory in your ece128 folder to house all of the files that will be created during the synthesis process. Note: the contents of all the files discussed below are listing in the appendix (for non-GW students viewing this on the web):

1. Login to a workstation, open up a terminal window and type:

```
cd ece128
mkdir lab6
cd lab6

mkdir reports
mkdir work
mkdir src
mkdir db
```

   Always create the “*reports*” “*work*” “*src*” and “*db*” directories in whatever directory you decide to work under. In our case, our 'working' directory will be 'lab4'

   - Copy the Synopsys generic standard cell library's verilog code into your “src” directory:

```
cp ~vlsi/course_ece128/lab_files/lab6/src/class.v  ~/ece128/lab5/src
```

2. Copy the verilog code you wish to synthesize into the “**src**” subdirectory:

   - In this lab, we want to use a modified ripplecarry adder from the ATPG tutorial:
   - We will use a ‘master’ copy of this code, so that the instructions and names for modules in this lab match up. So you will copy the ‘fulladder’ from the VLSI account, instead of using your own for this lab.

```
cp ~vlsi/course_ece128/lab_files/lab6/src/TOP.v ~/ece128/lab6/src
cp ~vlsi/course_ece128/lab_files/lab6/src/pin.txt ~/ece128/lab6/src
cp ~vlsi/course_ece128/lab_files/lab6/src/ripplecarry4_clk_scan.v \
~/ece128/lab6/src
```

   - The above two lines copy a ripplecarry adder with a scan-chain inserted into it, and a instantiation of it connected explicitly to Input and Output buffers.

3. Copy synthesis scripts into your lab 6 directory:

```
cp ~vlsi/course_ece128/lab_files/lab6/bsd_comp.tcl ~/ece128/lab6
```

4. To show that you have completed this section of the lab, fill in the answers below regarding the rippleccarry adder, print this page, and hand it in to your GTA before next lab:

   - How many IO Buffers are there in TOP.v? ____________
   - What IO Buffers are being used from class.v? ____
   - What IO Buffers are not connected to any signals in TOP.v? _______________________________

Part II Synthesizing the Ripple Counter with Boundary Scan using Synopsys Design Vision:

1. Change to your working directory

   ```
   cd ~/ece128/lab6
   ```

2. Start the design compiler's GUI by typing

   `design_vision` *(note: do NOT put an "&" after this command, it needs to run in the foreground)*

3. Copy and paste the following commands into the Design Vision prompt in order to setup variables for boundary scan insertation:

   ```
   set link_library *
   set target_library [list class.db ]
   set synthetic_library \
   [list dw_foundation.sldb ]
   set link_library [concat $link_library \
   $synthetic_library $target_library]

   #Read in top-level design
   read_verilog ./src/ripplecarry4_clk_scan.v
   read_verilog ./src/TOP.v
   set current_design TOP
   link
   #dont_touch CORE and pads
   set_dont_touch [list ripplecarry4_clk BIDI IBUF3 IBUF4 IBUF5 OBUF1 OBUF2]
   ```

4. Copy and paste the following commands into the Design Vision prompt in order to setup variables for the boundary scan controller

   ```
   #read pin map for BSR cell order
   read_pin_map ./src/pin.txt

   #define Tap signals
   set_dft_signal -view spec -type TCK -port TCK
   set_dft_signal -view spec -type TDI -port TDI
   set_dft_signal -view spec -type TDO -port TDO
   set_dft_signal -view spec -type TMS -port TMS
   set_dft_signal -view spec -type TRST -port TRSTN

   #define functional clocks
   create_clock CLK -period 100 -waveform {0 50}

   #Configure tap
   set_bsd_configuration -ir_width 4

   #Define standard instructions
   #opcodes: extest= 0010, sample & preload=0100, bypass=1111
   set_bsd_instruction -view spec [list EXTEST]  -code [list 0001] -reg BOUNDARY
   set_bsd_instruction -view spec [list SAMPLE]  -code [list 0100] -reg BOUNDARY
   set_bsd_instruction -view spec [list PRELOAD] -code [list 0100] -reg BOUNDARY
   set_bsd_instruction -view spec [list BYPASS]  -code [list 1111] -reg BYPASS

   #Define instructions for CLAMP [opcode=0010]
   set_bsd_instruction -view spec [list CLAMP] -code [list 0010] -reg BYPASS
   ```

   #continued on next page

```
#Define compliance enable signals for TEST_MODE=1; RESETN=1
   set_bsd_compliance -name P1 -pattern {TEST_SE 0 RST 1}
#Enable bsd-insertion
         set_dft_configuration -bsd enable -scan disable
```

5. Copy and paste the following commands into the Design Vision prompt in order to preview the TAP controller that will be inserted into your design.

   ```
   Preview_dft –bsd all
   ```

6. To show that you've completed this section of the lab, fill in the answers below regarding the BSD Controller, print this page, and hand it in to your GTA before next lab

   - What may happen to the design if we do not set Don't_touch on the core deisgn or buffers?

     ____

   - How many instructions did we create for the TAP controller? ________________

   - What are the 4 standard instructions used for? *Hint – This will require some research on Boundary Scan / IEEE 1149.1* ____

     ____
     ____
     ____
     ____
     ____
     ____

7. Copy and paste the following commands into the Design Vision prompt in order to insert the TAP controller into your design and check for JTAG compliance.

   ```
   #insert jtag..includes compile
   insert_dft
   #Compliance checking
   check_bsd -verbose
   ```

   You can ignore errors about the following signals floating: TDI, TMS, TRST. These would have pull-up resistors on the PCB, which would prevent them from floating.

8. Copy and paste the following commands into the Design Vision prompt in order to write out the design, boundary scan configuration and TAP testbench.

```
#Generate bsdl file
write_bsdl -out ./src/TOP_bsd.bsdl
#Generate bsd patterns
create_bsd_patterns -type all
write_test -format stil_testbench -output ./src/bsd_patterns
# generate verilog TAP testbench
write_test -format verilog -output ./src/BSD_tb.v
#write out jtag-inserted netlist
write -format ddc -hierarchy -output ./src/TOP_bsd.ddc
change_names -rules verilog -hier
write -format verilog -hierarchy -output ./src/TOP_bsd.v
```

9. To show that you've completed this section of the lab, fill in the answers below regarding the ripplecarry adder design, print this page, and hand it in to your GTA before next lab:

   - Draw a diagram showing the Top level design **BEFORE** and **AFTER** performing boundary scan insertion.

PART III Testing the TAP Structures and Boundary Scan Chain*:*

1. Change to your working src directory:

```
cd ~/ece128/lab6/src
```

2. Viewing the TAP Testbench in Simvision
   - Open up an editor (like gedit) and edit the file: `BSD_tb.v`
   - Go to the very bottom of the code
   - Right before the “end module” statement, add the following lines of verilog:

```
initial begin
   $shm_open ("ripplecarry4_clk_TAP.db") ;
   $shm_probe("AS") ;
end
```

3. Test the generate TAP patterns with our design.

   - Now, on a local workstatio type:

```
sim-nc BSD_tb.v TOP_bsd.v class.v
```

4. To show that you’ve completed this section of the lab, fill in the answers below regarding TAP simulation, print this page, and hand it in to your GTA before next lab:

   - Print out the Waveform view for the TAP simulation.
   - Does this testbench test our Ripple Carry design at all, or just test the TAP controller? What is the reasoning behind your answer?

____

____

____

   - If you had an arbitrary core that you wanted in insert a boundary-scan controller for, how would you prepare that? What is the reasoning behind your answer?

____

____

____

____

____

____

References

- Synopsys BSD Compiler Overview, Synopsys Solvnet ID 027820
- Synopsys BSD Compiler User Guide: Scan, Version D-2010.03

Further Reading on IEEE 1149.1 / JTAG / Boundary Scan

- Corelis is a electronic design automation company which makes testing tools. They have a wonderful JTAG tutorial located here http://www.corelis.com/education/Boundary-Scan_Tutorial.htm
- A slightly dated, but still great overview of boundary scan, from the University of Dayton, can be found here http://www.engr.udayton.edu/faculty/jloomis/ece446/notes/jtag/jtag1.html
- The Wikipedia entry on Boundary Scan is a good jumping-off point for various boundary-scan related topics. http://en.wikipedia.org/wiki/Boundary_scan
- InAccess Networks has a good boundary scan tutorial available which covers the TAP Controller in detail. It can be found here http://www.inaccessnetworks.com/projects/ianjtag/jtag-intro/jtag-intro.html
- A very lengthy and in-depth tutorial on boundary scan, by Asset-Intertech, can be found here http://www.asset-intertech.com/pdfs/Boundary-Scan_Tutorial_2007.pdf
- The TAP controller from Opencores.org is well documented and IEEE 1149.1 compliant core. It can be found here http://www.opencores.org/project,jtag

Appendix:

For students viewing this tutorial who do not have access to our server, this is a listing of the contents of the files reference in Part II of this tutorial:

Bsd_comp.tcl	- Synthesis script for BSD Compiler.
dc_test.tcl	- Synthesis script for DFT Compiler. Omitted from this tutorial. See DFT tutorial for this dc_test.tcl.
Pin.txt	- TCL script to drive Tetramax with tutorial commands. TBA.
TOP.v	- Example design used in tutorial. TBA.

BSD_comp.tcl:

```
set link_library *
 set target_library [list class.db ]
 set synthetic_library \
[list dw_foundation.sldb ]
 set link_library [concat $link_library \
$synthetic_library $target_library]

#Read in top-level design
read_verilog ./src/ripplecarry4_clk_scan.v
read_verilog ./src/TOP.v
set current_design TOP
link

#dont_touch CORE and pads
set_dont_touch [list ripplecarry4_clk BIDI IBUF3 IBUF4 IBUF5 OBUF1 OBUF2]

#read pin map for BSR cell order
read_pin_map ./src/pin.txt

#define Tap signals
set_dft_signal -view spec -type TCK -port TCK
set_dft_signal -view spec -type TDI -port TDI
set_dft_signal -view spec -type TDO -port TDO
set_dft_signal -view spec -type TMS -port TMS
set_dft_signal -view spec -type TRST -port TRSTN

#define functional clocks
create_clock CLK -period 100 -waveform {0 50}

#Configure tap
```

```
set_bsd_configuration -ir_width 4

#Define standard instructions
#opcodes: extest= 0010, sample & preload=0100, bypass=1111
set_bsd_instruction -view spec [list EXTEST]  -code [list 0001] -reg BOUNDARY
set_bsd_instruction -view spec [list SAMPLE]  -code [list 0100] -reg BOUNDARY
set_bsd_instruction -view spec [list PRELOAD] -code [list 0100] -reg BOUNDARY
set_bsd_instruction -view spec [list BYPASS]  -code [list 1111] -reg BYPASS


#Define instructions for CLAMP [opcode=0010]
set_bsd_instruction -view spec [list CLAMP] -code [list 0010] -reg BYPASS


#Define compliance enable signals for TEST_MODE=1; RESETN=1
set_bsd_compliance -name P1 -pattern {TEST_SE 0 RST 1}


#Enable bsd-insertion
set_dft_configuration -bsd enable -scan disable
#Run preview
preview_dft -bsd all

#insert jtag..includes compile
insert_dft

#OPTIONAL: Generate bsd patterns & bsdl before compliance checking
#create_bsd_patterns -type all
#write_test -format stil_testbench -output bsd_patterns
#write_test -format verilog -output ./src/BSD_tb.v
#write_bsdl -out TOP_bsd.bsdl


#Compliance checking
check_bsd -verbose
#Generate bsdl file
write_bsdl -out ./src/TOP_bsd.bsdl

#Generate bsd patterns
create_bsd_patterns -type all
write_test -format stil_testbench -output ./src/bsd_patterns

# generate verilog TAP testbench
write_test -format verilog -output ./src/BSD_tb.v

#write out jtag-inserted netlist
write -format ddc -hierarchy -output ./src/TOP_bsd.ddc

change_names -rules verilog -hier
write -format verilog -hierarchy -output ./src/TOP_bsd.v
```

Pin.txt

```
PACKAGE=my_package;
PORT= CLK, PIN=P1;
PORT= RST, PIN=P2;
PORT= CIN, PIN=P3;
PORT= A[3], PIN=P4;
PORT= A[2], PIN=P5;
PORT= A[1], PIN=P6;
PORT= A[0], PIN=P7;
PORT= B[3], PIN=P8;
PORT= B[2], PIN=P9;
PORT= B[1], PIN=P10;
PORT= B[0], PIN=P11;
PORT= TEST_SI, PIN=P12;
PORT= TEST_SE, PIN=P13;
PORT= SUM[3], PIN=P14;
PORT= SUM[2], PIN=P15;
PORT= SUM[1], PIN=P16;
PORT= SUM[0], PIN=P17;
PORT= COUT, PIN=P18;
PORT= TEST_SO, PIN=P19;
PORT= TRSTN, PIN=P20;
PORT= TDO, PIN=P21;
PORT= TDI, PIN=P22;
PORT= TMS, PIN=P23;
PORT= TCK, PIN=P24;
PORT= VDD, PIN=P25;
PORT= GND, PIN=P26;
```

Top.v

```
module TOP (SUM,
       COUT,
       A,
       B,
       CIN,
       CLK,
       RST,
       TEST_SI,
       TEST_SO,
       TEST_SE,
       TCK,
       TRSTN,
       TDI,
       TMS,
       TDO);

       output [3:0] SUM;
       input [3:0] A;
       input [3:0] B;
       input CIN, CLK, RST, TEST_SI, TEST_SE;
       output COUT, TEST_SO;

       //boundry scan ports
       input TCK, TDI, TRSTN, TMS;
       output TDO;

       wire [3:0] sum;
       wire [3:0] a;
       wire [3:0] b;
       wire cin, clk, rst, test_si, test_se;
       wire cout, test_so;

       IBUF3 U1 ( .A(RST), .Z(rst) );
       IBUF3 U2 ( .A(CLK), .Z(clk) );
       IBUF3 U3 ( .A(CIN), .Z(cin) );
       IBUF3 U4 ( .A(TEST_SI), .Z(test_si) );
       IBUF3 U5 ( .A(TEST_SE), .Z(test_se) );
       IBUF3 U6 ( .A(A[3]), .Z(a[3]) );
       IBUF3 U7 ( .A(A[2]), .Z(a[2]) );
       IBUF3 U8 ( .A(A[1]), .Z(a[1]) );
       IBUF3 U9 ( .A(A[0]), .Z(a[0]) );
       IBUF3 U10 ( .A(B[3]), .Z(b[3]) );
       IBUF3 U11 ( .A(B[2]), .Z(b[2]) );
       IBUF3 U12 ( .A(B[1]), .Z(b[1]) );
       IBUF3 U13 ( .A(B[0]), .Z(b[0]) );

       OBUF2 U14 ( .A(sum[3]), .Z(SUM[3]) );
       OBUF2 U15 ( .A(sum[2]), .Z(SUM[2]) );
       OBUF2 U16 ( .A(sum[1]), .Z(SUM[1]) );
       OBUF2 U17 ( .A(sum[0]), .Z(SUM[0]) );
       OBUF2 U18 ( .A(cout), .Z(COUT) );
       OBUF2 U19 ( .A(test_so), .Z(TEST_SO) );
       // boundry scan pads
       IBUF3 U20 ( .A(TCK) );
       IBUF3 U21 ( .A(TRSTN) );
       IBUF3 U22 ( .A(TDI) );
       IBUF3 U23 ( .A(TMS) );
       BIDI U24 ( .Z(TDO) );

       ripplecarry4_clk core (.sum(sum),
                      .cout(count),
                      .a(a),
                      .b(b),
                      .cin(cin),
                      .clk(clk),
                      .rst(rst),
                      .test_si(test_si),
                      .test_se(test_se),
                      .test_so(test_so));

endmodule
```