Tutorial 3: Complex Design Example and use Memory Generators

Revision History
March, 2008: Modification from previous versions to incorporate Modelsim and Encounter flow with CACTI based Memory Generator. (Jenkal/Franzon)
January 2012: Revision from previous version (Choi/Franzon)
June 2014: Update power estimation with new version (S. Dey/P. Franzon)
Feb. 2015: Update CACTI version 4.1 to 6.5 (Jong Beom Park/P. Franzon)

Objectives
The purpose of this tutorial is to show you, via a detailed example, how to approach the design, coding and synthesis of a more complex example. You will find this tutorial very useful for your project. The objectives of this tutorial are:
- Illustrate the class method for developing more complex designs, given a specification
- Illustrate how to capture and simulate this design as multiple modules in Verilog.
- Learn to make a model of memory to be used
- Learn to interface design with memory RTL
- Learn to synthesize this design using Design Compiler with memory
- Analyze timing and power of design
- Provide a few hints and leads that might be useful for your project

PLEASE CREATE A SEPARATE DIRECTORY FOR THIS TUTORIAL. Let us assume this is called Lab3. Also, setup the directory structure as stated in tutorial2 within this directory. We will be working exclusively within the Lab3 directory.

Directory Structure Setup (in your Lab3 directory)

> perl ./PAD_Flow.pl -op setup
> mkdir MEMORY

Copy files into the following directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDL/run_s</td>
<td>Engine.v, Controller.v, top.v, top with mem.v</td>
</tr>
<tr>
<td>SIMULATION/run_s</td>
<td>test.v, memdata.txt, modelsim.ini</td>
</tr>
<tr>
<td>SIMULATION/run_f</td>
<td>test.v, memdata.txt, modelsim.ini</td>
</tr>
<tr>
<td>SYNTH/run_s</td>
<td>All 4 tcl files from Lab1, .synopsys_dc.setup</td>
</tr>
<tr>
<td>SYNTH/run_f</td>
<td>designenv.tcl</td>
</tr>
<tr>
<td>MEMORY</td>
<td>CreateModel.pl</td>
</tr>
</tbody>
</table>
**Design Specification**

In this tutorial, we will design a simple search engine where two searches happen in parallel. The two lines of text which are to be searched through will be stored in memory as a series of words in the ASCII format. Words are separated by spaces (8’h20 in ASCII). When the search for a new 5 character (5 byte) zero padded word is started, the engine will search the two lines of text for a match. Note that new word will be presented as a 5 Byte zero-padded complete word. A search is started by inputting it at the start input and asserting the new signal for one cycle. At the same time, the StartAddress1 and StartAddress2 are input to provide the start addresses for the two lines in memory. If a match is found, the output Found will go high. If ASCII ETX (End of Text – 8’h03) is reached and the word is not found in either of the lines, the output flag NotFound will go high. The output flags are reset when new goes high. The words are between one and five characters in length. All words are lower case.

For example, if the following text is stored in memory:

```
the red cat sat on the mat it waited for the match <ETX>
```

If new goes high with “red” at the input search, found will go high. Conversely if new goes high with “white” at the input, eventually NotFound will go high.

The details of the I/O are as follows:
- **Search**: 40-bit word to be searched. 0-padded.
- **New**: (1 bit). Goes high for one clock cycle when a new word is in Search.
- **Found, NotFound**: (1 bit) Goes low for one cycle after new goes high. Goes high/low if the right condition met.
- **StartAddress1, StartAddress2**: (10 bits) Start addresses for strings in memory for the two searches to begin.

**Design Procedure**

1. Understand the intent of the design
2. Determine the required I/O
3. Determine what memories are needed, what the memory interfaces are, and how data will be stored within them.
4. Identify the rough design partition
5. Design the datapath
   a. Determine the algorithm you will use
   b. Identify all registers
   c. Describe the function of combinational logic
6. Design the control path
   a. Determine what approach: FSM, counter, or microcode (or multiples there-of)
   b. Design the FSD, count sequence or codeword
   c. Code the controller
7. Check the partition and wire the modules together
8. Run Memory Generator to create the Verilog, .lib and .db for the memory
9. Write the text fixture and verify the design

We will use this sequence to design this engine.

1. Understand the intent of the design.

If need be, run some more examples. It is often very useful to capture the intent in higher level code (e.g. in C++ or MATLAB). That is not necessary in this small example.

2. Determine the required I/O

We will write this in Verilog:

```verilog
input clock;
input reset;
in input new;       // sent high to commence new search
input [(5*8)-1:0] search;  // search word is up to five characters
                            // long with zero padding
input [9:0] StartAddress1;  // start address for string 1
input [9:0] StartAddress2;  // start address for string 2
output  Found;    // word in search is found
output  Notfound; // EOF reached without word being found
```

3. Design the Memory and Data Structure

The two lines of text to be searched are stored in memory, one ASCII character per 8-bit word. The simplest way to structure the memory is with an 8-bit data bus. With a maximum text length of 1024 characters, necessitating a 10-bit address bus.

This SRAM will be built using a Memory Generator that is provided with this tutorial. Please remember, DO NOT SYNTHESIZE THIS SRAM. The details of this generation and use of the SRAMs will be presented in Sections 8 - 10. At this point continue with the assumption of a 1024 word deep 8 bit wide memory i.e. 1K Byte memory.

This SRAM will be built using a Memory Generator that is provided with this tutorial. Please remember, DO NOT SYNTHESIZE THIS SRAM. The details of this generation and use of the SRAMs will be presented in Sections 8 - 10. At this point continue with the assumption of a 1024 word deep 8 bit wide memory i.e. 1K Byte memory.

Note that most SRAMs are organized as a 1-D array of words. You provide an address and thus get to read or write to one word. If you want a different data structure, for example a 2-D array, or a linked list, then you need to work out how to map that onto the RAM 1-D array. Also note that this is SRAM has three ports – two read ports and one write port. Using this model, two reads and one write can be done at one time. This memory is synchronous and hence all the reads and writes happen at the positive edge of the clock. Moreover, each port comes with an active low enable which allows for low power models by virtue of toggling these enables.

More ports can be added, at the expense of increasing the silicon area and delay of the SRAM. The number of ports allowed in your project, will be listed in that specification.
4. Identify the rough design partition

This is a fairly simple design, so two modules will suffice – one for the datapath (one engine for searching one line of text), and one for the controller. Please remember to always clearly separate control and datapath. We will be instantiating two datapath units to perform 2 searches in parallel. Both of these datapath units would be controlled by one controller.

5. Design the Datapath

5.a. Determine the Algorithm

We will have two parallel searches running. This would be done by two instances of the same basic search engine. The base algorithm would be to perform a linear search on the text in memory for the word being searched for. We will store the search word in a 5-Byte register SearchWord. We will retrieve the text to be searched from the memory one character at a time, starting from the StartAddress into a 5-Byte register. The contents of these two registers would be asynchronously compared to create Match signal. Whenever a space or ETX is encountered in the text retrieved from memory, this is reported and a new comparison is started by retrieving a new word from memory if not match is yet to be found. If there is a match, the search is terminated with Found. If ETX is encountered, and there is no match, the search is terminated with NotFound.

5.b. Sketch the Datapath

A sketch of the datapath is found below. It is fairly rough. The main thing is to clearly identify ALL the registers and have a good idea of what the clouds of logic do. Note that there are two engines as stated in the above section. In this case, we need the following features in the design:

- A register to store the word we are looking for called SearchWord.
- A register to store each word as it is fetched from memory (called Word). Since it is fetched one byte at a time, and will be sent into the correct position within Word register using a demux. The selection of the right location is made using the Charcount value (between 0 and 4).
- Registers to store Found and NotFound. Registers are needed as the data must be preserved until new goes high again.
- Logic to detect a match when sp or ETX. sp signifies an end of word and ETX signifies the end of the entire string of words.

A sketch of the datapath is found below.
6. Design the Control path

The controller is usually some combination of FSMs and counters. In this case, it makes sense to use counters:

- One counter to iterate through the characters stored in memory using Address1 and Address2 (for the two strings) and starting from StartAddress1 and StartAddress2.
- One counter to control the demux feeding the Word register in each engine called Charcount1 and Charcount2.
- Note also that we are going to need two read enables (active low) to enable reads from memory.

These counters start when new goes high. We stop them when a match is found, so as to reduce the electrical power consumed in the design.

A sketch of the controller is found below.
Note that the first two registers service the first engine and the last two service the second engine.

7. Check the Partitioning

Does the partitioning follow the rules specified in class? The most important rule to check is to make sure that every path is “constrained”, i.e. every path from module input to module output goes through a flip-flop. The rule is met here. Next, we need to check that the critical path does not span module boundaries. In this design there is no one critical path – no big arithmetic units for example. Though it is likely that the critical path passes through the match logic and into the Controller, it is not likely to have a significant logic depth. If the design was a little larger we could have assured that the datapath would have the entire critical path between its input and output registers.

7.a Capture the Design in Verilog

The search engines and controller designs are captured in the Engine.v and Controller.v files associated with this tutorial. One thing that needs to be catered for at this point is the creation of the final Found and NotFound signals using the individual Found and NotFound signals from each search engine. We will implement this as

assign  Found =  Found1 |  Found2;
assign  NotFound =  NotFound1 &  NotFound2;

This will be implemented at the top level integration of the design. It is at this point that the detailed design is completed. We note that the ETX, sp, Match from engine come into the Controller and the Controller sends out Charcount to each Engine. Also, the Controller will send out the Addresses and enables out to the Memory. It is important to understand timing of this system. For example, the Charcount input is provided by a synchronous circuit and used used to another synchronous piece of logic in Word. Thus Charcount would change from 0 to 1 at the same instance that you would be writing the first byte into Word corresponding to the Charcount value of 0. You must keep this in mind while determining the actual values that would be assigned to Charcount. Other, minor, design decisions are documented in the files.

7.b. Integrate the Hierarchy

A module “top” connects the Controller and two instances of Engine to be synthesized. This is the file called top.v.

8. CACTI installation

To begin the Memory Model creation, please download the latest version of CACTI from http://www.hpl.hp.com/research/cacti/cacti65.tgz into the directory Lab3. For more detail information, you could read this link : http://www.hpl.hp.com/research/cacti/
Extract the package by doing:
> tar zxvf cacti65.tgz

This would lead to the appearance of a folder called cacti65 in your unpacking directory. Within this directory please do
> make clean (read below for information if there is an error here)
> make

It may so happen that sometimes, “make clean” which is supposed to remove all the .o .aux and certain other files, would return an error saying it cannot find these files. This is because they are already cleaned up. This is NOT an error. You can still go ahead with the “make” command. The successful completion of the above commands would result in the creation of an executable called “cacti” (NOT cacti.exe) within the cacti65 directory.

Overwrite the given cache.cfg file to /cacti65/ directory

8.a. Installing and running Memory generator:
At this point, we would suggest work under the directory, MEMORY, within Lab3. (We suggested you to create it in the beginning)

In MEMORY directory, please make the code executable by performing
> chmod 744 CreateModel.pl

To understand the options available with this script please do
> ./CreateModel.pl -help

AT THIS POINT MAKE SURE YOU HAVE DONE AN “add synopsys2014”

The inputs to this file are:
1. The size of the memory (-depth): [DEFAULT: 4096]. This corresponds to the total number of rows in the memory. For this Tutorial we will use 1024.
2. The width in bits of each entry in memory (-width). This should be multiple of 8. [DEFAULT: 32]: This corresponds to the size of each row entry of the memory IN BITS. Here, we will use 8.
3. The full path to the Cacti Executable (-cacti_path) [DEFAULT: ../cacti65/cacti]: This is the full path to the cacti executable. Here, ../cacti65/cacti

Thus, to create the necessary models, you would run:
./CreateModel.pl -cacti_path ../cacti65/cacti -width 8 -depth 1024

Please run all of the above in one line.

At this point you will see the information for the run spit out on screen. In this case we would see:

Memory Parameters:
Memory Depth (number of words): 1024
Word Size in bits: 8
Implied Number of Address Bits: 10
Implied Memory Size in bytes: 1024

Please determine the correctness of your options to the tool by viewing this message. Also, note that the parameters provided are noted in the info.memgen file. Along with this information, the progress of the script is shown. At the end of the run, the Verilog, .lib and .db in the MemGen_{datawidth}_{addresswidth} format will be created. In this case we have,
a) MemGen_8_10.lib b) MemGen_8_10.v c) MemGen_8_10_RTL.v
d) MemGen_8_10.db

Please note that the top level module name is going to be MemGen_8_10 as well. Also note that we are going to be using the MemGen_8_10.v as the memory model for simulation purposes. Moreover, the top_with_mem.v file integrates the module in the top level integration file (top.v)and the above memory model. Note that the necessary inputs and outputs must be created from this combination. In this case, it appears as seen below:

9. Verification

A test fixture is generated for the purposes of verification. Note that the test Fixture includes a $readmemh command. This command refers to a file “memdata.txt”. This is a text file placed in the same directory as the design. It is simply a list of entries in the memory. E.g. To store ascii “it” followed by ETX, the file would contain

69 // i
The `memdata.txt` file here contains 2 strings, starting from addresses 0 and 100 as signified by the `@0` and `@64` (which is 100 in hex) lines in the file. Therefore, we are going to search for the word “easy” in the two lines

This easy example is keeping me busy
This example is keeping me busy

starting at addresses 0 and 64 in memory. Note that the initial loading in `test.v` is done as

```vhdl
parameter CLKPERIOD = 10;
readmemh("memdata.txt", top_mem.memory.mem);
clock = 0; new = 0;
reset = 0;
#(CLKPERIOD+2)
reset = 1;
search = 40'h6561737900; // The word "easy" padded with zeros
StartAddress1 = 0; // start address for string 1
StartAddress2 = 10'd100; // start address for string 2
#CLKPERIOD
reset = 1'b0; // start the design at a known state
#CLKPERIOD
new = 1'b1; // begin search by asserting new
#CLKPERIOD
new = 1'b0; // assert new for one cycle
```

It is important to notice that the `$readmemh` command points to `top_mem.mem_inst.mem` which is the instance of the `mem` array within the memory instance within `top_with_mem.v`. To run the simulations, move into the `SIMULATION/run_s` folder and do (per Tutorial 1 and Tutorial2)

```
> add modelsim10.0c
> setenv MODELSIM modelsim.ini
> vlib mti_lib
> vlog ../../../HDL/run_s/Engine.v
> vlog ../../../HDL/run_s/Controller.v
> vlog ../../../HDL/run_s/top.v
> vlog ../../../HDL/run_s/top_with_mem.v
> vlog ../../../MEMORY/MemGen_8_10.v
> vlog ./test.v
```

To simulate we would then do:
```
> vsim -novopt test_search &
```

Make sure you have the `memdata.txt` file within the directory that you are simulating in. Note again, that the `top_with_mem.v` contains an instance of the `MemGen_8_10` module.
Assuming that the design is satisfactory, we are now going to synthesize the design without the MemGen_8_10 module i.e. the top.v.

**Action.** Generate three realistic test cases for this design, one where a word is found before a space, another where it is found before ETX, another where there is no match. Run these cases and verify the design. Fix and document any bugs in the design.

10. Synthesis and PAD_flow.pl

Setup the `SYNTH/run_s/` folder for synthesis by copying all the relevant .tcl and .synopsys dc.setup files into the directory from the EDA page. Please note that some minor modifications might have been made to the files there. Please copy them all over again and do not copy the files from the Lab1 folder.

Download PAD_Flow.pl again from EDA Wiki website. Do NOT re-use it from Lab2.

The modifications required in the synthesis scripts are as follows:

1. Modify the setup.tcl file to reflect the correct choices.
   ```tcl
   set clkname clock
   # set variable "modname" to the topmost module name
   set modname top
   # set variable "RTL_DIR"
   set RTL_DIR ../../HDL/run_s/
   # set variable "type" to a distinguishing name
   set type tut3
   # set the number of digits to be used for delay results
   set report_default_significant_digits 4
   set CLK_PER 10
   ```

2. Make sure to read in the Verilog files containing the modules top, datapath and controller. Do NOT read in the SRAM. Thus we change the read.tcl file to reflect:
   ```tcl
   read_verilog $RTL_DIR/Engine.v
   read_verilog $RTL_DIR/Controller.v
   read_verilog $RTL_DIR/top.v
   ```

   We should note there that we are only synthesizing the design without the memory i.e. Top.v and not Top_with_mem.v. At this point we run the synthesis by sourcing setup.tcl, read.tcl, Constraints.tcl and CompileAnalyze.tcl. At the end of this run you will have the top_final.v file in your ./SYNTH/run_s folder and the relevant timing reports for the design. Note that the synthesis area is around 1542.8 um^2. **Please copy the top_final.v to SYNTH/run_f/**.

10.a. Running SPEF Creation

We now need to run the design through the PAD_flow.pl flow (in Lab3 folder). This is going to described just in terms of the commands that were run. Remember to **add**
cadence2010, add synopsys2014” and “add modelsim10.0c” before running the following commands.

In the Lab3 folder type (in a single line):
> perl ./PAD_Flow.pl -op analyze -mod top -
net ./SYNTH/run_f/top_final.v -clkname clock -period 10

Note the modified netlist name and module name. The result of this is the presence of the top.spef file in PR/run_f directory. Note that this step might take a little time.

**10.b. Running SAIF Creation**

We are going to be running the SAIF creation with the routed netlist for top and with the memory instance being unchanged. In **SIMULATION/run_f/** we make the following changes to the test.v file.

```vhd
parameter CLKPERIOD = 10;
...
$dumpfile("Tut2.vcd");
$dumpvars;
clock = 0; new = 0;
reset = 0;
#(CLKPERIOD+2)
reset = 1;
search = 40'h6561737900; // The word "easy" padded with zeros
StartAddress1 = 0; // start address for string 1
StartAddress2 = 10'd100; // start address for string 2
#CLKPERIOD reset =1'b0; // start the design at a known state
#CLKPERIOD new =1'b1; // begin search by asserting new
#CLKPERIOD new =1'b0; // assert new for one cycle
#(50* CLKPERIOD);
```

To setup the mti_lib folder please first do “vlib mti_lib” (this should be common knowledge now). In **SIMULATION/run_f/**, please compile the necessary files by performing:

```bash
> setenv MODELSIM modelsim.ini
> vlib mti_lib
> vlog ../../../PR/run_f/top_routed.v
> vlog ../../../MEMORY/MemGen_8_10.v
> vlog ../..//HDL/run_s/top_with_mem.v
> vlog 
/afs/eos.ncsu.edu/lockers/research/ece/wdavis/tech/nangate/NangateOpenCellLibrary_PDKv1_2_v2008_10/liberty/520/NangateOpenCellLibrary_PDKv1_2_v2008_10_typical_conditional.v
> vlog ./test.v
```

The simulation is run by performing:

`vsim -c -novopt test_search`
On the vsim prompt do a “run –all” to run the simulation to its entirety. Also note the modified top level test name. After it is done, you will note the presence of the Tut2.vcd file.

Now use the vcd2saif tool to capture switching activities. Please use command below:

```bash
> vcd2saif -input Tut2.vcd -instance test_search/top_mem -output top.saif
```

After successfully execution of the above command, it should generate top.saif under ./SIMULATION/run_f. We are now ready to run the power analysis flow using the SPEF and the SAIF files.

10.c. Creating power reports for the Design without memory.

Ensure that the Constraints.tcl file that comes with the second tutorial is copied into the ./SYNTH/run_f/ folder. If so, the necessary command in Lab3 directory to create post-annotation reports would be:

```bash
> perl ./PAD_Flow.pl -saif ./SIMULATION/run_f/top.saif -op power -period 10 -inst test_search/top_mem/top_inst -mod top -clkname clock
```

Note here that we are going to be creating the reports only for the top module within the top_with_mem.v file. Thus the -inst changes to test_search/top_mem/top_inst which is the hierarchical path to the instance top. The new timing and power reports for the design without memory can be found at:

SYNTH/run_f/top_timing_post_annotation.rpt
SYNTH/run_f/top_pwr_post_annotation.rpt

Note the timing from 5.5562ns to 3.8479ns and dynamic power from 17.3154 uW to 33.0089 uW. Note that these values are for the 10ns clock that was used for simulation.

10.d. Creating power reports for the Design WITH memory.

Now, we are going to have to include the memory power values in this simulation. Before we run PAD_Flow, go to SYNTH/run_s directory. Then modify the set_link_library section of .synopsys_dc.setup in the following format (in one line):

```bash
set link_library [concat $synthetic_library $target_library ../../MEMORY/MemGen_8_10.db]
```

We now run the ./PAD_Flow in memory analysis more by doing:

```bash
> perl ./PAD_Flow.pl -saif ./SIMULATION/run_f/top.saif -op memory -period 10 -inst test_search/top_mem -mod top -clkname clock -memdir ./MEMORY/ -memname MemGen_8_10 -topinst top_inst -topfile ./HDL/run_s/top_with_mem.v
```
What you will note here is that we are going to modify the instance name to test_search/top_mem given that we are interested in annotating to the design + memory unit.

The new input options are:

a) `-op: set to memory. This changes the type of analysis to memory based power analysis

b) `-memname : which corresponds to the memory name outputted by the memory generator i.e. MemGen_<datawidth>_<addrwidth>

c) `-memdir: the directory where the memory generator is run and the .db files are created

d) `-topinst: the instance name of the top level design within the memory+design file

e) `-topfile: the full path to the memory+design integration file.

This would result in the following updated set of reports which would contain the memory-annotated numbers within them:

SYNTH/run_f/top_timing_post_annotation_mem.rpt
SYNTH/run_f/top_pwr_post_annotation_mem.rpt

Note the change in timing to 4.0467ns and dynamic power to 4.7641mW. Note that the memory power is strictly based on the produced outputs from CACTI. Please note that the progress of this phase can be tracked by viewing the following file: ./SYNTH/run_f/memanalysis_transcript.out.

ASIDE: It is useful to know that some options like `-hier -cell -net -verbose etc. when used with the report_power (i.e. you can use dc_shell-xg-t> report_power -net -cell -hier -verbose) give you a lot of information on the nets and cell that contribute most to the total power in your design. This can be used to make design decisions.

11. DesignWare

Synopsys has provided a set of pre-designed blocks that you might find useful in your project. These are referred to as “DesignWare” and details can be found in the DesignWare documentation. The weblink to designware is

http://www.synopsys.com/products/designware/buildingblock.html

To simulate with DesignWare, Verilog needs to have a Verilog model of the DesignWare Module. These can be found at /afs/bp/dist/synopsys_syn/dw. You can use a ‘include statement in your Verilog module to include the appropriate file for simulation from the right subdirectory in that space. The list of available DesignWare blocks can be found at

Appendix A. Checklist For Neophyte Verilog Designers

Module Design
1. Did you DESIGN BEFORE CODING?
2. Have you clearly identified the purpose of each register (flip-flop)?

   A common beginner error is to have unnecessary flip-flops. (This is usually a symptom of NOT designing before coding). A flip-flop is only needed when data is stored between clock cycles. Excessive flip-flops makes a design LARGE and SLOWS down synthesis.

3. Do you use non-blocking assignment when assigning to flip-flops?
4. Do you make sure that every variable is only assigned within ONE procedural block or continuous assignment statement?

   This will cause a LINT warning from Synopsys.

5. Do you make sure that you have NO feedback within combinational logic.

   This will cause “timing arcs” in synthesis. An example is...
   
   always@(B or C)
   A=B+C;
   always@(A or E)
   C=A+E

   Instead A or C must be registered.

6. Do you avoid unintentional latches by having no implied memory in combinational logic?
7. Do you avoid having 2D arrays (register arrays) in your modules?

   Put large 2D arrays into your SRAM. Put smaller ones into their own register file module.

8. Did you “right-size” the module?

   i.e. Make it as small as reasonable while containing critical paths, sharable logic, registered paths (see 9) and being “reasonable” to understand.

9. Does every path connecting input and output go through a flip-flop?
10. Are critical paths contained within one module, not split across two of them?
11. Did you have any WHILE or FOR loops?
The ONLY valid construct is to use a for loop to iterate through an array of bits. Most attempts by neophytes to use these constructs are invalid.

12. Did you DESIGN BEFORE CODING?

Design Hierarchy
1. Did you avoid “glue” logic in modules containing other modules?
2. Did you bring all signals connecting to things outside your design up through the ports of your “top” module?
3. Did you instantiate your SRAMs only at the topmost level?

Synthesis
1. Did you set “current_design <top module name>” before running “compile”? 
2. Did you set “current_design <top module name>” before running “report_area”? 
3. If the compile is slow try setting “map_effort low” and try a much slower clock.
4. Make sure you have a correct synopsys setup file and correct references to design ware, if used. (See http://www.synopsys.com/products/designware/buildingblock.html and for the Verilog files, refer to /afs/bp/dist/synopsys_syn/dw/sim_ver )
5. Did you look at the warnings and errors after the “read” statement?
6. Did you look at the timing reports?
7. If reset (global signal) has a high “fan-out” that is OK. This signal can be asynchronous.