Revision History
March, 2011: Initial version adder/subtractor using DesignWare (Z. Yan / P. Franzon)
December, 2014: 2nd edition: synopsys update, pipelining using DesignWare (Z. Wang)

1. Introduction
The purpose of this tutorial is to provide an example for using DesignWare IP core provided by Synopsys.

2. Learning Objectives
- How to instantiate DesignWare IP core in your design.
- How to process floating point number in ASIC design.
- How to add pipelining to the instantiated DesignWare Ip and how to synthesize it.

PLEASE DO NOT CUT AND PASTE COMMANDS FROM THE TUTORIAL. THIS LEADS TO ERRORS. YOU ARE ADVISED TO TYPE THEM OUT INSTEAD.

3. Example of instantiating DesignWare IP

The following is the code of arith.v, the module provide add and subtract function for 64 bit floating point number. It uses DesignWare IP core DW_fp_addsub.

---

We need to include /afs/eos.ncsu.edu/dist/syn/sim_ver/DW_fp_addsub.v for simulation purpose, but it is not synthesizable, so we have to turn it off during synthesis. The command //synopsys translate_off is not a comment during synthesis, when design compiler sees this, it will ignore the following code. //synopsys translate_on will let design compiler synthesize following code again. This does not apply to all the DesignWare IP cores, you should see the example directory for the IP core you want to use: /afs/eos.ncsu.edu/dist/synopsys2013/syn/dw/examples/

---

// synopsys translate_off
`include '/afs/eos.ncsu.edu/dist/synopsys2013/syn/dw/sim_ver/DW_fp_addsub.v'
  //synopsys translate_on

module arith( clock, inst_a, inst_b, inst_rnd, inst_op, z_inst, status_inst);
Define the parameters needed in DesignWare IP core, in our case DW_fp_addsub have three parameters sig_width, exp_width and ieee_compliance. For detail information of each IP, see the document directory `/afs/eos.ncsu.edu/dist/synopsys2013/syn/dw/doc/`.

```plaintext	parameter sig_width = 52;
parameter exp_width = 11;
parameter ieee_compliance = 0;
```

Declare all the input and output ports, some are declared using the defined parameters to make the design easier to modify.

```plaintext
input clock;
input [sig_width+exp_width : 0] inst_a;
input [sig_width+exp_width : 0] inst_b;
input [2 : 0] inst_rnd;
input inst_op;
output [sig_width+exp_width : 0] z_inst;
output [7 : 0] status_inst;
```

Declare output port z_inst and status_inst as registers, and declare a wire variable to connect to DesignWare IP core. The purpose of doing this is to make the output signal synchronous.

```plaintext
reg [sig_width+exp_width : 0] z_inst;
wire [sig_width+exp_width : 0] z;

reg [7 : 0] status_inst;
wire [7 : 0] status;
```

Synchronize the output ports of DesignWare IP core to the positive edge of clock signal.

```plaintext
always@(posedge clock)
begin
    z_inst<=z;
    status_inst<=status;
end
```

Connect the corresponding signal ports to the DesignWare IP core, and pass the parameters into the IP core, it will overwrite the parameter values in the DesignWare IP core itself.
DW_fp_addsub #(sig_width, exp_width, ieee_compliance)
    U1 (.a(inst_a), .b(inst_b), .rnd(inst_rnd), .op(inst_op), .z(z), .status(status));
endmodule

4. Establishing environment and perform simulation

a. Establish a directory for this tutorial.
   mkdir lab5
   cd lab5

b. Download arith.v, test.v and data.dat from Tutorial 5: DesignWare. Also download modelsim.ini in tutorial 1 into this directory.

c. Perform simulation using modelsim
   ####Set up the environment as tutorial 1####
   add modelsim
   setenv MODELSIM modelsim.ini
   vlib mti_lib
   ####Compile all the source files ####
   vlog /afs/eos.ncsu.edu/dist/synopsys2013/syn/dw/sim_ver/DW_fp_addsub.v
   (OPTIONAL This step is to compile the DesignWare module, in this tutorial, you don’t need to type this, because the instantiated DesignWare module has already been included in arith.v, and will get compiled when compiling arith.v.)
   vlog arith.v
   vlog test.v
   ####Invoke the vsim GUI #####
   vsim –novopt test_fixture&
   run -all

This test bench scans two floating point numbers from data.dat, then performs addition and subtraction between these two float numbers, and output the results. After the simulation, you should see the following results:

# Memory[0]=-2.505220
# Memory[1]=-4.239679
# 77result = -6.744898
# 127result = 1.734459
# ** Note: $finish : test.v(56)

The first result at 62ns is addition result, second result at 102ns is subtraction result.

And get the waveform:
5. Synthesize the DesignWare IP core in your design

a. Download .synopsys_dc.setup and all the .tcl files from tutorial 1 to the current directory.

b. In setup.tcl: change modname to arith(your design name), type to tut5, and CLK_PER to 50(clock cycle)
   In read.tcl: change to read_verilog $RTL_DIR/arith.v

c. Perform the synthesis steps in tutorial 1. Pay special attention to the step: source CompileAnalyze.tcl. Due to the complexity of the DesignWare IP, the compile could take very long time, so be patient. After the synthesis, you should get the timing report timing_max_slow_holdfixed_tut5.rpt:

   | clock clock (rise edge) | 0.0000 | 0.0000 |
   | clock network delay (ideal) | 0.0000 | 0.0000 |
   | input external delay | 0.6580 | 0.6580 |
   | inst_a[0] (in) | 0.0912 | 0.7492 |

   | data arrival time | 39.9875 |

   | clock clock (rise edge) | 50.0000 | 50.0000 |
   | clock network delay (ideal) | 0.0000 | 50.0000 |
   | clock uncertainty | -0.0500 | 49.9500 |
   | z_inst_reg[52]/CK (DFF_X2) | 0.0000 | 49.9500 |
   | library setup time | -0.3068 | 49.6432 |
   | data required time | 49.6432 |

   data required time 49.6432
   data arrival time -39.9875

   slack (MET) 9.6557

Also the critical path in your design:

As we could see this floating point arithmetic module is a huge design, you should consider reuse this module if possible in the design to minimize the area and power consumption.
6. Pipelining the instance from DesignWare IP core in your design

Due to long latency of the designware, as we have seen the 64-bit floating point adder example above, we usually want to add pipelines to the instance. In the following example we will see how we can add pipelines to an instantiated DesignWare, and how we can synthesize it.

Based on the code in arith.v, which has been listed above, we modify it as follows:

```verilog
// synopsys translate_off
`include "/afs/eos.ncsu.edu/dist/synopsys2013/syn/dw/sim_ver/DW_fp_div.v"
//synopsys translate_on

module arith( clock, inst_a, inst_b, inst_rnd, inst_op, z_inst, status_inst);
parameter sig_width = 52;
parameter exp_width = 11;
parameter ieee_compliance = 0;
input clock;
input [sig_width+exp_width : 0] inst_a;
input [sig_width+exp_width : 0] inst_b;
input [2 : 0] inst_rnd;
input inst_op;

output [sig_width+exp_width : 0] z_inst;
output [7 : 0] status_inst;

reg [sig_width+exp_width : 0] inst_a_reg;
reg [sig_width+exp_width : 0] inst_b_reg;
reg [2 : 0] inst_rnd_reg;
reg inst_op_reg;

reg [sig_width+exp_width : 0] z_inst_pipe1, z_inst_pipe2, z_inst_pipe3, z_inst_pipe4;
wire [sig_width+exp_width : 0] z_inst_internal;

reg [7 : 0] status_inst_pipe1, status_inst_pipe2, status_inst_pipe3, status_inst_pipe4;
wire [7 : 0] status_inst_internal;

// Instantiate the DesignWare IP Core
   DW_fp_addsub #(sig_width, exp_width, ieee_compliance)
       U1 ( .a(inst_a_reg), .b(inst_b_reg), .rnd(inst_rnd_reg), .op(inst_op_reg),
           .z(z_inst_internal), .status(status_inst_internal));

always @(posedge clock) begin
   // synchronize the input to clock edge
   inst_a_reg <= inst_a;
end

```
```verilog
inst_b_reg <= inst_b;
inst_rnd_reg <= inst_rnd;
inst_op_reg <= inst_op;
//output to be registered by allowing 4 pipeline stages to be moved
z_inst_pipe1 <= z_inst_internal;
z_inst_pipe2 <= z_inst_pipe1;
z_inst_pipe3 <= z_inst_pipe2;
z_inst_pipe4 <= z_inst_pipe3;
status_inst_pipe1 <= status_inst_internal;
status_inst_pipe2 <= status_inst_pipe1;
status_inst_pipe3 <= status_inst_pipe2;
status_inst_pipe4 <= status_inst_pipe3;
end

assign z_inst = z_inst_pipe4;
assign status_inst = status_inst_pipe4;
endmodule

We name it as `arith_pipe.v`. We implement 4 levels of registers at the end of the instance to be moved.

Then we download the `test_pipe.v` from the tutorial website, and follow the instruction in Section 4 above. In the `test_pipe.v`, we send two sets of input instructions to the module in two consecutive cycles, and read the output four cycles after the first input set sent to the module. You can observe the waveform as you did in Section 4.

However, as we can observe from the code in `arith_pipe.v`, the registers are directly added to the end of the instance, thus the critical path in the instance hasn't been shorted. So we need to rearrange the register positions. We can do it in the synthesis by the retiming command “optimize_registers” or “pipeline_design”. Here is one example how I could modify our synthesis scripts in `CompileAnalysis.tcl`:

```tcl
(after the line “report_timing > timing_max_slow.rpt”)
set CLK_PER 6
create_clock -name $clkname -period $CLK_PER -waveform "0 [expr $CLK_PER / 2]" $clkname
set_clock_uncertainty $CLK_SKEW $clkname
set_max_area 0

optimize_registers
compile -incr -map_effort medium
report_qor > qor_pipe.rpt
report_timing > timing_pipe.rpt
```
The idea is, we firstly generate a mapped gate-level netlist, and then we call \texttt{optimize_register} to determine the location of registers in a design to achieve a target clock period and minimize the number of registers while maintaining that clock period, and then we recompile it. Follow the instruction in \textbf{Section 5}, and use the modified script \texttt{CompileAnalysis_pipe.tcl} to do the re-timing after we source \texttt{Constraints.tcl}. Then we can take a look at the 4 timing reports generated: timing\_max\_slow.rpt, timing\_pipe.rpt, timing\_min\_fast\_holdcheck.rpt, timing\_max\_slow\_holdfixed.rpt. The first report is the one before we optimize the location of the registers, while the second to the last are the ones after we execute the \texttt{“optimize_registers”} command. We could also open \texttt{cell\_report\_final.rpt} to see how the area changes, as we aggressively change the clock cycle for retiming.

\begin{center}
\textbf{APPENDIX : ADDITIONAL INFORMATION ON DesignWare}
\end{center}

\textbf{A. DesignWare Documents and modules}

The location of DesignWare is /afs/eos.ncsu.edu/dist/synopsys2013/syn/dw/, the documentation is in the sub directory /doc, the IP modules are in /sim\_ver and examples for each modules are located in /examples. You could explore all the DesignWare IP modules here, and pick one you need.

\textbf{B. Floating point number in ASIC design}

Verilog provide real type variable to store floating point number, but it could not be synthesized. One way to get around this is translate the real type to bits in test bench and pass it to the design. After process in the design, translate the bits back into real number in your test bench. There are two system tasks to do this, $realtobits and $bitstoreal. $realtobits translate a real type number into a 64bits binary, and $bitstoreal translate 64bits binary back to real type number. In this 64bits binary, it is partitioned as following:

\begin{center}
\begin{tabular}{c|c|c}
\hline
\textbf{exponent} & \textbf{fraction} \\
\hline
\textbf{sign} (11 bit) & \textbf{fraction} (52 bit) \\
\hline
63 & 52 & 0 \\
\hline
\end{tabular}
\end{center}

You should follow this convention during process of the translated floating point number. More information about how the translation works, please follow this link: \url{http://en.wikipedia.org/wiki/IEEE_754-1985}

You could find an example of this in the test.v provided with this tutorial.
C. Ignorable Synthesis Warnings

Besides the warnings mentioned in tutorial 1, the following warnings could be ignored during synthesis.

- Warning: Design '***' inherited license information from design '***'. (DDB-74)
- Warning: Design '***' is being converted to a limited design. (DDB-75)
- Warning: Current design named ** is hidden. (UID-408)
- Warning: In design '***', input pin '***' was left unconnected. Logic 0 assumed. (LINT-0)
- Warning: In design '***', net '***' has no loads. (LINT-2)