

## VHDL Hardware Realisation

### Compiling the Design

At JCU we use the Lattice ispLever software for compiling and fitting the designs. Other manufacturers have similar compilers. The Xilinx foundation software is included in many textbooks.

It is easiest to look at existing examples, to get an understanding of VHDL coding and program structure. If ispLever is installed in the default directory C:\ispLever\_Classic, then the ispLever\_Classic \examples\cpld\vhdl and the corresponding fpga, ispgdx, ispXPLD, spld and Tutorial directories have several complete ispLever projects with VHDL code. The directory C:\ispLever\_Classic \ispcpld\Tutorial contains pdf tutorial files, which give a step by step operation of the software. These tutorial files are also on the subject web page in the resources page under ispLever CPLD tutorials.

To compile the software start the ispLever project navigator software and select *file > new project*. Specify *Schematic/VHDL* as the project type. Specify the project name and select *save* to create the project.

Select the device type and right click to select the actual device to be used. For the labs use device type ispMach 4A5 M4A5-64/32-10JC and select M4A5-64/32-10JC as the device, as this is the device included in the prototyping boards. We also have some of Lattice's prototyping boards for ispLSI2032 devices. These devices are now obsolete and require the obsolete device library to permit them to be programmed. Both those boards also have the slowest (and cheapest) IC's on them, which run at 80 MHz. If the timing is to be investigated, it is important to set the correct speed grade. If changing device types, Click Yes for the warning to change device kits and click Yes at the warning about changing device constraints to return to the project navigator.

Select "Source" and "New" and specify a "VHDL Module" as the source file. Do not confuse this with "VHDL test fixture", which would compile different and give errors. The software then asks for the file name, entity name and architecture name. The entity name and architecture name are included in the VHDL code structure framework that is automatically generated by this process. The text editor is then started up for the complete code to be entered.

The ispLever Classic software CD includes one synthesis tools (compiler): Synplify for Lattice by Synplicity. Previous versions of the ispLever software included two synthesisers: Synplify and Precision for Lattice by Mentor Graphics. Having two compiler makes it easier to locate difficult errors, as each synthesiser will give different error messages, which may help locate the error more precisely.

To compile the VHDL code, right click on *Synplify Synthesise VHDL file* and select *start* to start the compiler. The design is then compiled and any compiler errors are shown. Double clicking on the compiler error message line opens up the editor, if needed, and takes you to the exact line where the error is. Save the edited file and recompile the VHDL code until there are no errors.

An EDIF file can also be produced by selecting *Compile EDIF file*. If the VHDL code in not compiled yet, this will then first invoke the compiler. The EDIF file thus produced can be used as a source file for fitting into other devices from many different manufacturers. Producing this EDIF file also generates a set of compiled Boolean

Algebra equations, corresponding to the VHDL code. These equations are however not the fully reduced equations, which are produced as part of the fitting process.

## **Pin allocation**

Using VHDL, the locking of particular pins to particular signals will depend on the compiler used. In ABEL, the pin configuration is done inside the main ABEL code. This is also possible with VHDL designs.

The way pins are allocated and fitter constraints are handled, vary widely between manufacturers and even widely between different versions of the software. The techniques outlined here may thus not work for designs done using Alterra or Xilinx software and their devices. In addition this information is sometimes very difficult to find and is hidden in the manuals. Most Logic Design text books do not mention pin allocation, however that is a very critical parameter for a circuit designer.

## **Attributes**

With the Lattice software, there are different ways to do the pin allocation. The easiest is use an “attribute” and include that inside the VHDL package declaration. Attributes allow a variety of device dependent parameters to be specified. For more details see Chapter 8 of “Synplicity FPGA Synthesis (for Lattice) Reference Manual, April 2008, *Synthesis Attributes and Directives*”, with an alphabetical listing shown on page 8-8. The manual can be found in `ispLEVER_Classic\synpbase\doc` on the ispLever installation on your hard drive and on the EE4306 web site under [http://eng.jcu.edu.au/subjects/ee4306/resources/VHDL/VHDL-Language/synplify\\_ref.pdf](http://eng.jcu.edu.au/subjects/ee4306/resources/VHDL/VHDL-Language/synplify_ref.pdf)

The attributes can be set without referring to the attributes package as follows:

```
-- design_unit_declarations
attribute attribute_name : data_type ;
attribute attribute_name of object : object_type is value ;
```

The pin allocations for the 8 bit wide adder, the code of which is shown in pages 19 and 20 of the VHDL Language lecture notes is thus as follows:

```
entity AddWide is
  port (A, B : in std_logic_vector (7 downto 0);
        Sum : out std_logic_vector (7 downto 0);
        Cout : out std_logic);
  attribute loc : string;
  attribute loc of Cout : signal is "p21" ;
  attribute loc of A : signal is "p31, p30, p29, p28, p27, p26, p25, p24" ;
  attribute loc of B : signal is "p9, p8, p7, p6, p5, p4, p3, p2" ;
  attribute loc of Sum : signal is "p36, p37, p38, p39, p40, p41, p42, p43" ;
end AddWide;
```

This pin allocation works with both the compilers used with ispLever. In addition one can use the compiler specific library files. For Synplify this is:

```
library synplify;
use synplify.attributes.all;
-- design_unit_declarations
attribute synplify_attribute of object : object_type is value ;
```

*Loc* is the appropriate *synplify\_attribute* and the code is exactly the same as above, with the library declaration being the only added lines. For *ispLever* the “library *synplify*” statement does not need to be used as it is already included when the Synplify synthesis compiler is selected.

Using the *Loc* attribute will only work for the *ispLever* Version 4 and above compilers and other compilers are likely to have different attribute names for locking the pins. The Lattice *ispDesignExpert* and early *ispLever* versions required “lock” instead of “loc” and the pin specification was different. *ispLever* Version 4 required no commas, between the pin numbers. For *ispLever* version 5.0 and above, the format suggested in both the Precision and the Synplify manual includes a comma between the individual pins, and should be used in current designs. The required format in *ispLever Classic* is thus:

```
attribute loc of A : signal is "p31, p30, p29, p28, p27, p26, p25, p24" ;
```

Sometimes other attribute names than “loc” are used. For the Precision compiler the commands can be found on page 2-18 of the Precision Synthesis Reference Manual 2005a as:

```
attribute pin_number : string;
attribute pin_number of clk : signal is "p10";
```

so that if one used the Precision compiler for Altera or Xilinx devices, the word *pin\_number* instead of *loc* needs to be used and this should also be able to be used for Lattice devices. For *ispLever* it is best to use *loc* with the commas between the parts of the array.

Because the pin allocation attribute is not defined in the VHDL standard, each compiler and in many cases the different versions of a compiler, use different attributes for allocating pins. Be aware of this and if the pin allocation is not done as expected, read the manuals to determine the correct format. After compiling the design with a given pin allocation, always check the fitter report produced during the design fitting to verify that the pin allocation is correct.

The relevant lines of the fitter report for the wide adder above, is as follows:

```
Pinout_Listing
~~~~~
| Pin |Blk |Assigned|
Pin No| Type |Pad |Pin   | Signal name
-----
  1 | GND | |   |
  2 | I_O | A7 | * | B_0_
  3 | I_O | A6 | * | B_1_
  4 | I_O | A5 | * | B_2_
  5 | I_O | A4 | * | B_3_
  6 | I_O | A3 | * | B_4_
  7 | I_O | A2 | * | B_5_
  8 | I_O | A1 | * | B_6_
  9 | I_O | A0 | * | B_7_
 10 | JTAG | |   |
 11 | CkIn | |   |
 12 | GND | |   |
 13 | JTAG | |   |
```

|    |      |    |   |        |
|----|------|----|---|--------|
| 14 | I_O  | B0 |   |        |
| 15 | I_O  | B1 |   |        |
| 16 | I_O  | B2 |   |        |
| 17 | I_O  | B3 |   |        |
| 18 | I_O  | B4 |   |        |
| 19 | I_O  | B5 |   |        |
| 20 | I_O  | B6 |   |        |
| 21 | I_O  | B7 | * | Cout   |
| 22 | Vcc  |    |   |        |
| 23 | GND  |    |   |        |
| 24 | I_O  | C7 | * | A_0_   |
| 25 | I_O  | C6 | * | A_1_   |
| 26 | I_O  | C5 | * | A_2_   |
| 27 | I_O  | C4 | * | A_3_   |
| 28 | I_O  | C3 | * | A_4_   |
| 29 | I_O  | C2 | * | A_5_   |
| 30 | I_O  | C1 | * | A_6_   |
| 31 | I_O  | C0 | * | A_7_   |
| 32 | JTAG |    |   |        |
| 33 | CkIn |    |   |        |
| 34 | GND  |    |   |        |
| 35 | JTAG |    |   |        |
| 36 | I_O  | D0 | * | Sum_7_ |
| 37 | I_O  | D1 | * | Sum_6_ |
| 38 | I_O  | D2 | * | Sum_5_ |
| 39 | I_O  | D3 | * | Sum_4_ |
| 40 | I_O  | D4 | * | Sum_3_ |
| 41 | I_O  | D5 | * | Sum_2_ |
| 42 | I_O  | D6 | * | Sum_1_ |
| 43 | I_O  | D7 | * | Sum_0_ |
| 44 | Vcc  |    |   |        |

This shows that all the pin connections have been allocated properly.

### ***Constraint Editor***

The second way for producing the correct pin allocations, using the ispLever Project Navigator, is to firstly produce a project containing the correct device for the design to be fitted to and produce the VHDL code excluding the pin assignments above, which works and compiles properly. Then from the Device double click on the Constraint Manager.

This opens up the compiler constraint manager window as shown below. Double click in the + next to "D.. ADDWIDE", expands that structure repeating this process allows all the allocated pins to be seen. Note that the pin Cout is not shown and clicking on the icon next to Cout brings up the properties for pin Cout and shows that no pin has been reserved. If no pins are locked the lock column in the pin attribute table is empty and the pin number of the desired pin can be entered, using the right mouse click and selecting edit function. Pin properties like slow slew and pull-up can also be altered if permitted for that particular device. Once all the pin specifications are correct, the file is saved and the compiler is run, to include the correct pin routing.

### Constraint File

In addition some other parameters like Pull-up and slew rate can be set for individual pins using this Pin Attributes table. Some global parameters such as default value of Pull-up and placement directives can also be set using the Global Constraints Table. These will have some effect on the resources used in the fitting of the code into the device. Once the pin values have been set, they can be saved as an .lct file. A .lct file with the default name *projectname.lct* file is produced by the compiler when using the loc attributes as outlined above. When saving a .lct file the *projectname.lct* name should not be used since that file will be overwritten as part of the compilation.

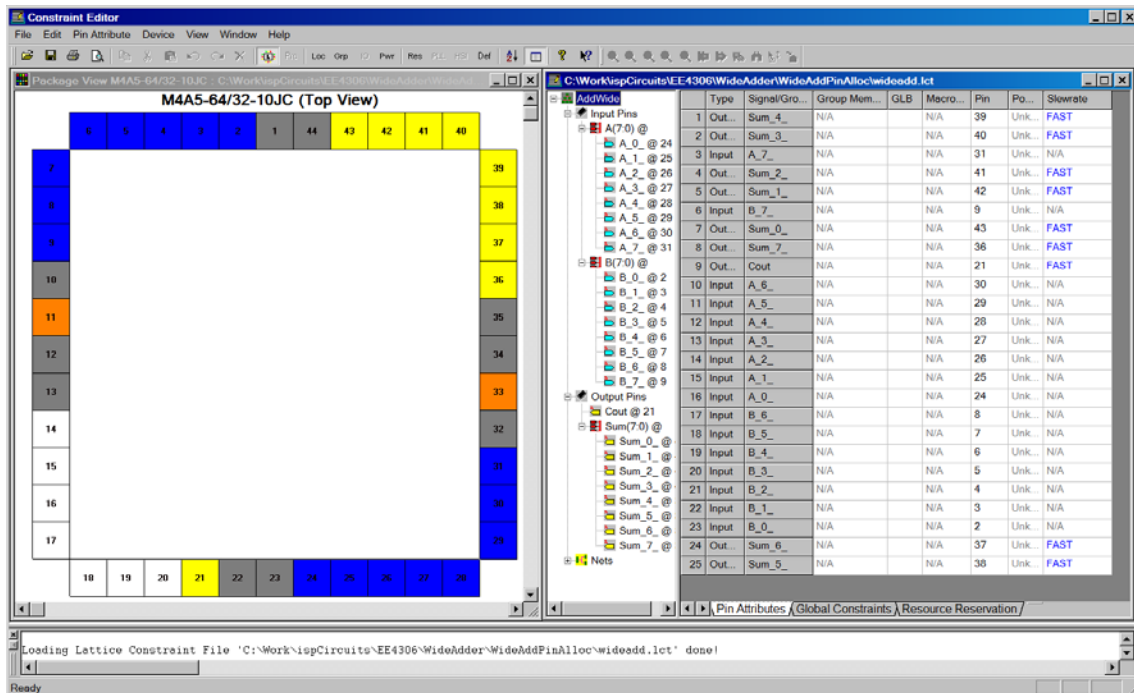


Figure 1. Constraint Editor Window, with pin attributes

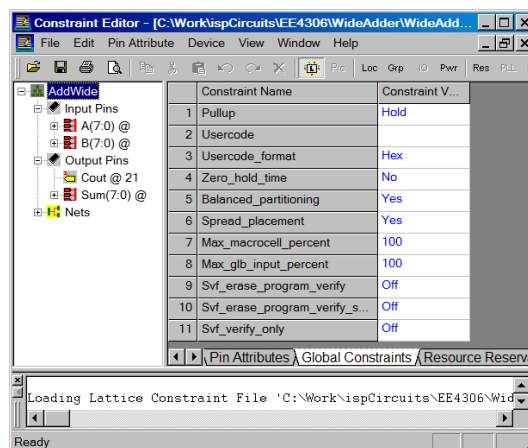


Figure 2. Global Constraint window

When we compile the project again, the saved .lct file can be imported into the constraints editor to reallocate the pin numbers. The .lct file is a text file, so it can be edited to allocate pin numbers that way or make minor changes if desired. The pin attributes, Global Constraints and the Optimisation constraints are all saved in this single .lct file.

For different devices, the Constraint editor and the Optimization editor will look very different. For instance for the ispLSI2032 devices used previously, the Optimization constraint window does not exist and it's function is included in the Constraint editor.

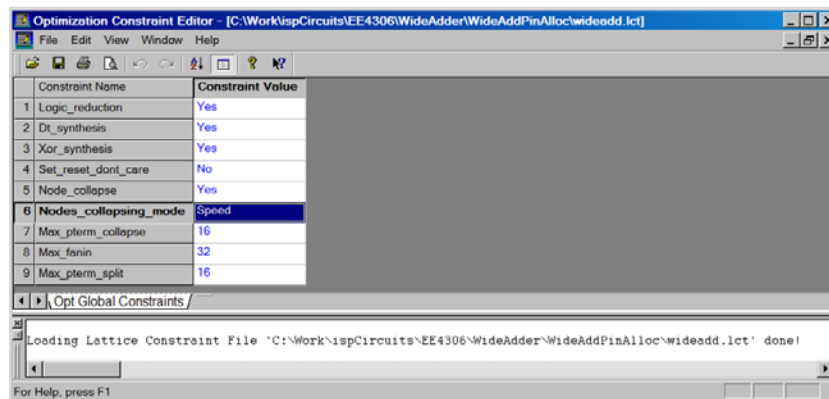


Figure 3. Optimisation Constraint Editor window.

The optimisation constraints can be used to fit a project when it uses most of the resources available. The default Node collapsing mode is *Speed*, which results in the fastest circuit. The alternate node *Area* will generally result in a slower but smaller realisation.

The Global Constraints in the Global Constraints Editor allow the partitioning rules and certain pin parameters like Pullup to be changed. In addition it allows a User Electronic Signatures (UES) or user code to be added to the design. This an 8 hex digit user code which can be used to denote version numbers, manufacturer or designer name, date, device function, etc. This user code can also be set during the programming of the CPLD or FPGA.

Of the above techniques for pin allocation, using the attribute “loc” command to allocate the pins inside VHDL, has a big advantage in that the pin allocation is always kept with the source file. With the other techniques it is easy for the pin allocation to get lost, so that an existing design may not be able to be updated.

## Example

The project window for the 8 bit wide adder, the code of which is shown in pages 22 and 23 of the VHDL Language lecture notes, with the pin assignment shown below is used as VHDL source for a project. The ispLever Project navigator window is shown below. Note that even though the VHDL code is in one file with the Addwide module last, the project navigator recognises the hierarchical structure of the code. The pin assignment code is as follows:

```
entity AddWide is
  port (A, B : in std_logic_vector (7 downto 0);
        Sum : out std_logic_vector (7 downto 0);
        Cout : out std_logic);
  attribute loc : string;
  attribute loc of Cout : signal is "p21" ;
  attribute loc of A : signal is "p31, p30, p29, p28, p27, p26, p25, p24" ;
  attribute loc of B : signal is "p9, p8, p7, p6, p5, p4, p3, p2" ;
  attribute loc of Sum : signal is "p36, p37, p38, p39, p40, p41, p42, p43" ;
end AddWide;
```

The VHDL code can now be compiled, by selecting the top level VHDL module and, selecting Synthesise VHDL file. Right click and select start to start the compiler and after a while, the compiler log will appear and green ticks, yellow warnings or red crosses will appear indicating the success or otherwise of the compilation, as shown below.

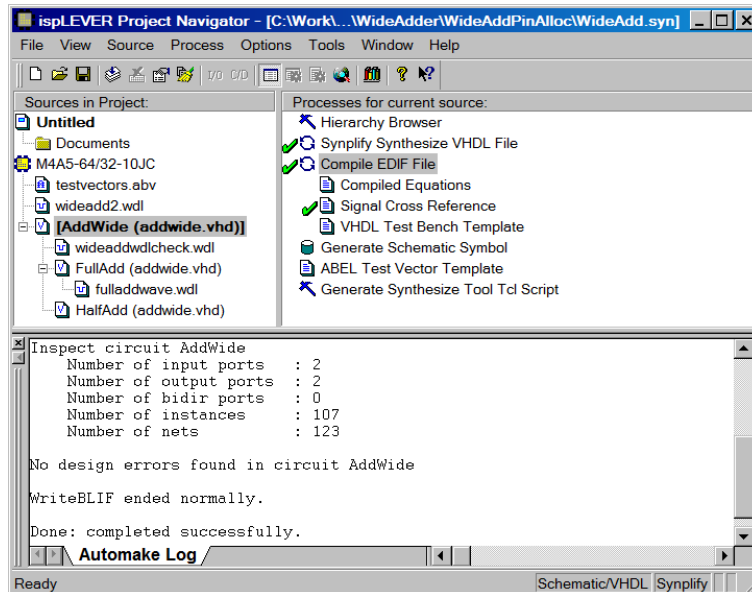


Figure 4. Compiling the Design window, Synplify Synthesis.

The Synplify compiler produces no errors or warnings as shown above. The Automake log produces no warnings or errors as shown in figure 4.

The code can be fitted into the selected device by right clicking and selecting start on the JDEC file option. This will then reduce the Boolean algebra equations and fit the design and implement the appropriate constraints, such as pin numbers, as part of the fitting process.

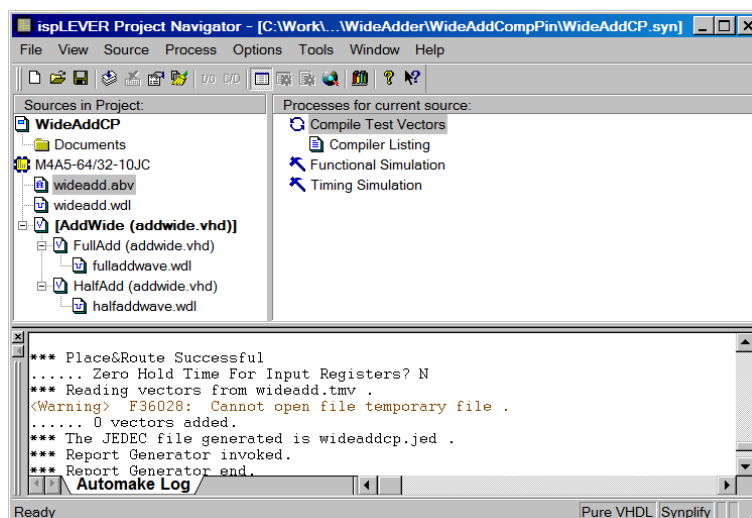


Figure 5. The Design Fitting window

Note that a warning is generated with ispLeverV7.1 running under Windows Vista. This warning does not appear to effect the result of the synthesis. No errors are generated using XP for the same code.

Right clicking on Pre-Fit Equations will display the reduced Boolean algebra equations for the design in ABEL like form. Right clicking on the Fitter report shows the resources used.

Equations:

```
Sum_4_.X1 = (A_3_ & B_4_ & B_3_ # !A_3_ & !B_4_ & !B_3_
# !A_3_ & !B_4_ & N_11 # !B_4_ & !B_3_ & N_11 # A_3_ & B_4_ & !N_11
# B_4_ & B_3_ & !N_11);
```

```
Sum_4_.X2 = (!A_4_);
```

```
Sum_3_.X1 = (A_2_ & B_3_ & B_2_ # !A_2_ & !B_3_ & !B_2_
# !A_2_ & !B_3_ & N_13 # !B_3_ & !B_2_ & N_13 # A_2_ & B_3_ & !N_13
# B_3_ & B_2_ & !N_13);
```

```
Sum_3_.X2 = (!A_3_);
```

```
Sum_2_.X1 = (!A_1_ & !A_0_ & !B_2_ # A_1_ & B_2_ & B_1_
# !A_1_ & !B_2_ & !B_1_ # !A_0_ & !B_2_ & !B_1_ # !A_1_ & !B_2_ & !B_0_
# !B_2_ & !B_1_ & !B_0_ # A_1_ & A_0_ & B_2_ & B_0_
# A_0_ & B_2_ & B_1_ & B_0_);
```

```
Sum_2_.X2 = (!A_2_);
```

```
Sum_1_.X1 = (!A_0_ & !B_1_ # !B_1_ & !B_0_ # A_0_ & B_1_ & B_0_);
```

```
Sum_1_.X2 = (!A_1_);
```

```
Sum_0_ = (!A_0_ & B_0_ # A_0_ & !B_0_);
```

```
Sum_7_.X1 = (B_7_ & A_6_ & B_6_ # !B_7_ & !A_6_ & !B_6_
# !B_7_ & !A_6_ & N_5 # !B_7_ & !B_6_ & N_5 # B_7_ & A_6_ & !N_5
# B_7_ & B_6_ & !N_5);
```

```
Sum_7_.X2 = (!A_7_);
```

```
Cout.X1 = (A_7_ & B_7_ # !A_7_ & !B_7_ # A_6_ & B_6_
# A_6_ & A_5_ & B_5_ # A_5_ & B_6_ & B_5_ # A_6_ & A_5_ & cint_5__n
# A_5_ & B_6_ & cint_5__n # A_6_ & B_5_ & cint_5__n
# B_6_ & B_5_ & cint_5__n);
```

```
Cout.X2 = (!A_7_ & !B_7_);
```

```
Sum_6_.X1 = (A_5_ & B_6_ & B_5_ # !A_5_ & !B_6_ & !B_5_
# A_5_ & B_6_ & cint_5__n # B_6_ & B_5_ & cint_5__n
# !A_5_ & !B_6_ & !cint_5__n # !B_6_ & !B_5_ & !cint_5__n);
```

```
Sum_6_.X2 = (!A_6_);
```

```
Sum_5_ = (A_5_ & B_5_ & cint_5__n # !A_5_ & !B_5_ & cint_5__n
# !A_5_ & B_5_ & !cint_5__n # A_5_ & !B_5_ & !cint_5__n);
```

```
cint_5__.X1 = (A_4_ & B_4_ # !A_4_ & !B_4_ # A_3_ & B_3_
# A_3_ & A_2_ & B_2_ # A_2_ & B_3_ & B_2_ # A_3_ & A_2_ & !N_13
# A_2_ & B_3_ & !N_13 # A_3_ & B_2_ & !N_13 # B_3_ & B_2_ & !N_13);
```

```
cint_5__.X2 = (!A_4_ & !B_4_);
```

Note for most of these outputs, the XOR gates that are part of the hardware are used. For example Sum\_4\_.X1 and Sum\_4\_.X2 = (!A\_4\_); are XORed to produce the Sum\_4 output.

The Optimisation Constraints are very important in order to ensure that the VHDL code fits inside the device with the minimum delay. To show the effect of global constraints,

the wide adder is firstly compiled with the standard “Speed” constraint. This results in the following fitted statistics with the Synplify Synthesiser:

```
Device_Resource_Summary
-----
```

|                                | Total     |      |           |     |             |
|--------------------------------|-----------|------|-----------|-----|-------------|
|                                | Available | Used | Available |     | Utilization |
| Dedicated Pins                 |           |      |           |     |             |
| Input-Only Pins                | ..        | ..   | ..        | --> | ..          |
| Clock/Input Pins               | 2         | 0    | 2         | --> | 0%          |
| I/O Pins                       | 32        | 25   | 7         | --> | 78%         |
| Logic Macrocells               | 64        | 13   | 51        | --> | 20%         |
| Input Registers                | 32        | 0    | 32        | --> | 0%          |
| Unusable Macrocells            | ..        | 0    | ..        |     |             |
| CSM Outputs/Total Block Inputs | 132       | 40   | 92        | --> | 30%         |
| Logical Product Terms          | 320       | 80   | 240       | --> | 25%         |
| Product Term Clusters          | 64        | 21   | 43        | --> | 32%         |

The current ispLever compilers are much better at fitting the functions than the older version 4 compilers. The number of Logic Macrocells used for the wide adder and using the ispLever version 6 or Classic version 7, the number of Macrocells is 13 (20% of IC) for Synplify compiler and 12 (18%) for the Precision synthesiser. For version 4 it was 18 Macrocells (28% of IC). Changing the variables in the optimising constraint window will change the resources required. After the global optimisation is changed all files based on the previous constraints must be removed. This is done by selecting *File > Clean Up All*, in the project navigator window, thus forcing the compilation to be done completely from scratch. Without this, the optimisation strategy will not necessarily produce a different result.

It should be noted that changing the optimisation from speed to area, does not always result in a reduced number of macrocells for the function. For instance in the above design, using the Synplify compiler and changing from Speed to Area in the optimisation constraints, results in an **increase** in the number of macrocells used from 13 to 15 and a **reduction** in the number of Logical Product Terms from 80 to 51. For the Precision compiler, in an **increase** in the number of macrocells used from 12 to 13 and a **reduction** in the number of Logical Product Terms from 73 to 61 results. For the Synplify compiler, the timing analysis shows that for an 10 nS device, the propagation delay between A0, A1, B0 and B1 to Sum(7) or Cout is 24 ns, for the maximum speed optimisation and 52 ns for the minimum area optimisation. For this Wide adder, the Precision compiler which was available in ispLever 7.0 gives slightly better results.

In general the “Area” optimisation will give a reduction in the number of macrocells used. For example the 2005 assignment of a countdown timer, when loaded with a 20 minute maximum time will require 79 macrocells and cannot fit in the IC when “Speed” is used but it will fit comfortably using 55 macrocells when “Area” optimisation is used. For some other designs and other devices, there sometimes is little or no difference between “Speed” and “Area” optimisation.

If the strategy is set to “Speed” and “No XOR” synthesis results in the VHDL code not being able to fit in the device as the gate-width required is greater than 16.

It is at present not possible to maximise both the speed of critical paths and minimise the area by using “Fast” for some parts of the code and “Area” for another part. If the adder has to work faster, but still has to fit inside the M4A5-64/32-10JC, then the VHDL code needs to be modified to produce say a 4 bit wide adder as a component,

using parallel operations and then cascading the carry out from that block into the second 4 bit wide adder component. This makes the design process much more complex.

The constraints window also shows the pin constraints. Right clicking on say the IO types of say A(7) allows the PullUp and Slew Rate to be set for the input and output pins. Note Slew Rate can only be set on output pins. The pin number allocation can also be changed in the constraints window by simply typing in the new values. A view of the pins of the IC and the pin functionality is also available in the constraint window. This is very useful if data books are not available.

For some devices, additional optimisation and constraint function are available. For the ispMach4000 LC4128 device, the Zero hold time, automatic buffering to ease input loading and fanout, and the amount of effort in fitting the code can also be selected. For that device different output voltages or Open drain or Conventional outputs can be selected, so that the device will drive CMOS at 3.3, 2.5 or 1.8V, LVTTTL or the PCI bus. can also be selected. For reliable interfacing, the correct voltages must be set as default.

## Testing the Design

Generally the VHDL code is tested using test vectors. The easiest way to generate the VHDL Test Bench or ABEL Test Vector File for the VHDL module is to use the template generation options in the ispLEVER Project Navigator as shown in the Project Navigator window below.

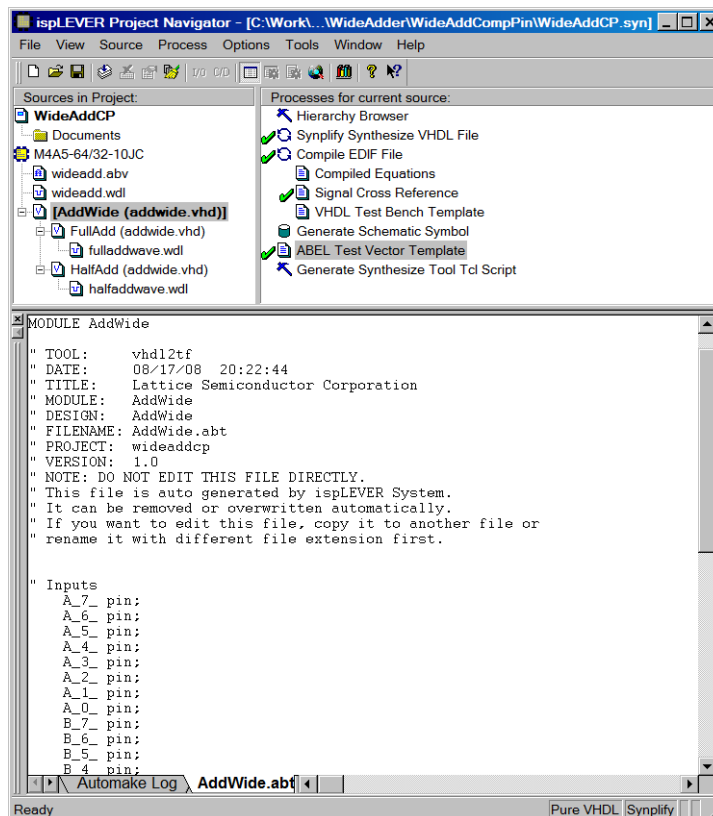


Figure 6. Project Navigator for using an ABEL test vector file

For ispLever-Classic VHDL test benches cannot be used. Test benches are available in the full version of ispLever. In addition the full version of ispLever includes Active-HDL, which allows several other waveform tools and VHDL test benches to be used.

## **ABEL Test Vector**

The process is first illustrated using an ABEL test vector file. Double Click on the ABEL Test Vector Template, generates an ABEL test vector template specially for the particular VHDL code in a separate panel as shown above.

Make a new ABEL test vector file by selecting Source -> New -> Abel test vector and this will then open up a blank text editor screen. Select and copy the text generated in the panel, as shown in figure 6, and paste this in the text editor screen for the test vector file. Add the actual test vectors required and save the resulting ABEL test vector file. The file should have an .abv extension and be part of the project.

The test vector file generated using the template is shown below:

```

MODULE AddWide
" TOOL:   vhd12tf
" DATE:   08/08/05 16:22:05
" TITLE:  Lattice Semiconductor Corporation
" MODULE: AddWide
" DESIGN: AddWide
" FILENAME: AddWide.abt
" PROJECT: wideadd
" VERSION: 1.0
" NOTE: DO NOT EDIT THIS FILE DIRECTLY.
" This file is auto generated by ispLEVER System.
" It can be removed or overwritten automatically.
" If you want to edit this file, copy it to another file or
" rename it with different file extension first.

" Inputs
  A_7_ pin;
  A_6_ pin;
    Other similar lines deleted for brevity
  B_0_ pin;

" Outputs
  Sum_7_ pin;
  Sum_6_ pin;
    Other similar lines deleted for brevity
  Sum_0_ pin;
  Cout pin;

" Bidirs

Test_vectors
([A_7_,A_6_,A_5_,A_4_,A_3_,A_2_,A_1_,A_0_,B_7_,B_6_,B_5_,B_4_,B_3_,B_2_,B_1_,B_0_] -> [Sum_7_,Sum_6_,Sum_5_,Sum_4_,Sum_3_,Sum_2_,Sum_1_,Sum_0_,Cout])
END

```

To provide the appropriate test vectors, it is easiest to edit the above file and remove the section from and including "Bidirs and replace it with the following:

"Declarations

```
A = [A_7_,A_6_,A_5_,A_4_,A_3_,A_2_,A_1_,A_0_];
B = [B_7_,B_6_,B_5_,B_4_,B_3_,B_2_,B_1_,B_0_];
Sum = [Sum_7_,Sum_6_,Sum_5_,Sum_4_,Sum_3_,Sum_2_,Sum_1_,Sum_0_];
```

### Test\_Vectors

```
([A,B] -> [Sum, Cout])
[0,0] -> [0,0];
[0,1] -> [1,0];
[0,2] -> [2,0];
" Many lines removed from listing for sake of brevity.
[6,1] -> [7,0];
[6,2] -> [8,0];
[6,3] -> [9,0];
[6,4] -> [10,0];
[6,5] -> [11,0];
[6,6] -> [12,0];
```

**END**

The use of the vector declaration for A, B and Sum significantly simplifies the test vectors. Also note that in this situation, each vector must be specified. In instances where a clock is used it may be possible to simplify the test vector specification. For example the test vectors for the SeqCount module on page 16 of the VHDL-Language notes can be done using:

### Test\_vectors

```
([Clk,Rset] -> [Count_0_,Count_1_,Count_2_,Count_3_,Count_4_,Count_5_,Count_6_,
LED])
@repeat 3 {[c.,0] -> [.x.,x.,x.,x.,x.,x.,x.,x.];} "3 clock-pulses
@repeat 20 {[c.,1] -> [.x.,x.,x.,x.,x.,x.,x.,x.];} "20 clock-pulses
```

**END**

The **BOLD** text is the code that has been added manually to the automatically generated ABEL test vector file. This test vector file will then generate the waveforms for the VHDL code like those shown in figures 10 and 11.

## Viewing the waveforms

To view the waveforms with an ABEL test vector file, click on the .abv file in the project, that will then show the window below. Right click on the Functional Simulation to start the functional simulation of the circuit. The functional simulation assumes an ideal IC, with no propagation delay. The Timing simulation does the same function, but firstly fits the code to the IC and then does the simulation, which includes the propagation delays for the code fitted in the IC. The maximum speed of operation and any glitches present in the hardware can thus be investigated.

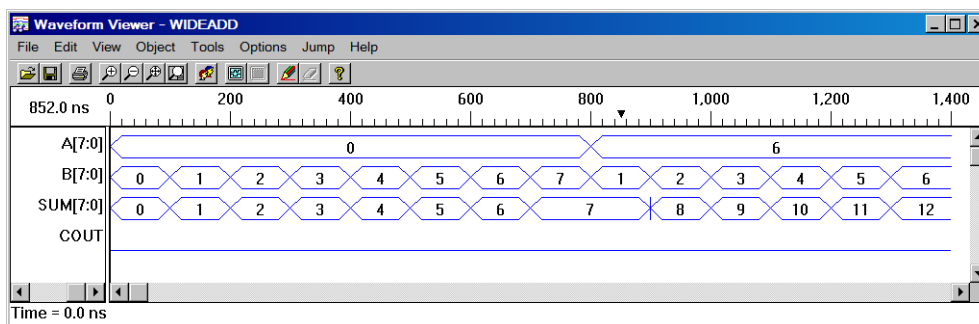


Figure 7. Functional and Timing Simulation from .abv files.

To show what happens if there are any errors between the test vector file and the actual waveforms, the test vector code shown above for the wide adder is changed to include an error by replacing:

```
[0,6] -> [6,0];
```

with:

```
[0,6] -> [5,0];      "error
```

When the timing simulation is now performed the simulator control panel opens and by clicking on the “!” , the waveforms are shown and compared with the expected results. It can be seen that the error that was included in the test vector file, at 600 ns, is detected. Similarly if there is an error in the logic specification, that will be detected. It is desirable to test the circuit with all the possible input conditions. This may require the use of a programming language such as C++ to produce the test vectors. The required test vectors can also be generated using an Excel spreadsheet and then simply copying and pasting the required text into the ABEL test vector file.

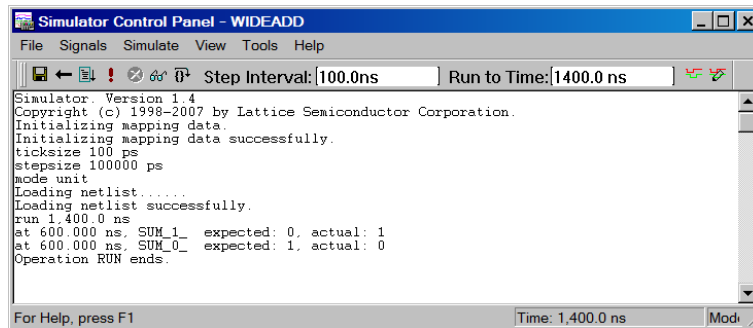


Figure 8. Simulator Control Panel

The resulting waveforms are shown below. The waveforms for A(4) to A(7), B(4) to B(7) and Sum(4) to Sum(7) have been removed for space considerations.

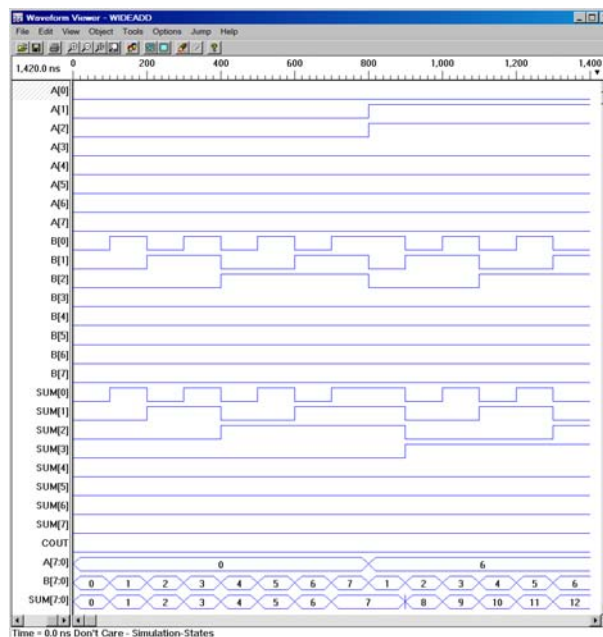


Figure 9. Waveforms produced by .abv file

For a task like the wide adder considered here, the use of a waveform “bus” is more convenient, as shown below. This allows the function of the circuit to easily be verified.

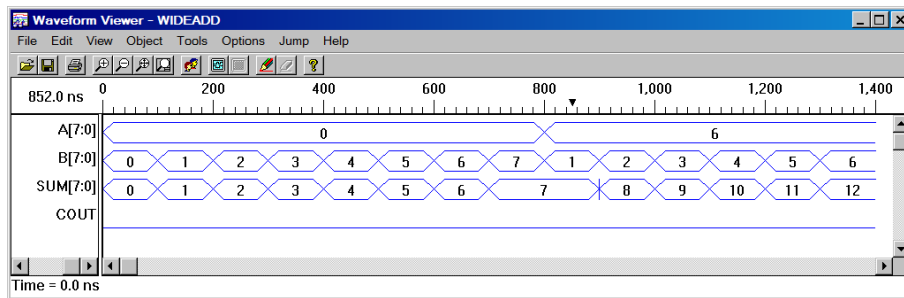


Figure 10. Waveform viewing using Busses.

## Waveform Files

When the functionality of an adder, multiplier or waveform generator is to be tested, then ABEL test vector files, where the input and outputs are numbers, are the most convenient way for testing the device. In some applications, such as counters and phase detectors, the input waveforms must be specified. For example to test a phase detector, two waveforms are required that are slightly different frequency, as shown for waveform R and V in figure 12. Such waveforms cannot be produced easily using ABEL test vector files, they can however be easily produced using .wvl files.

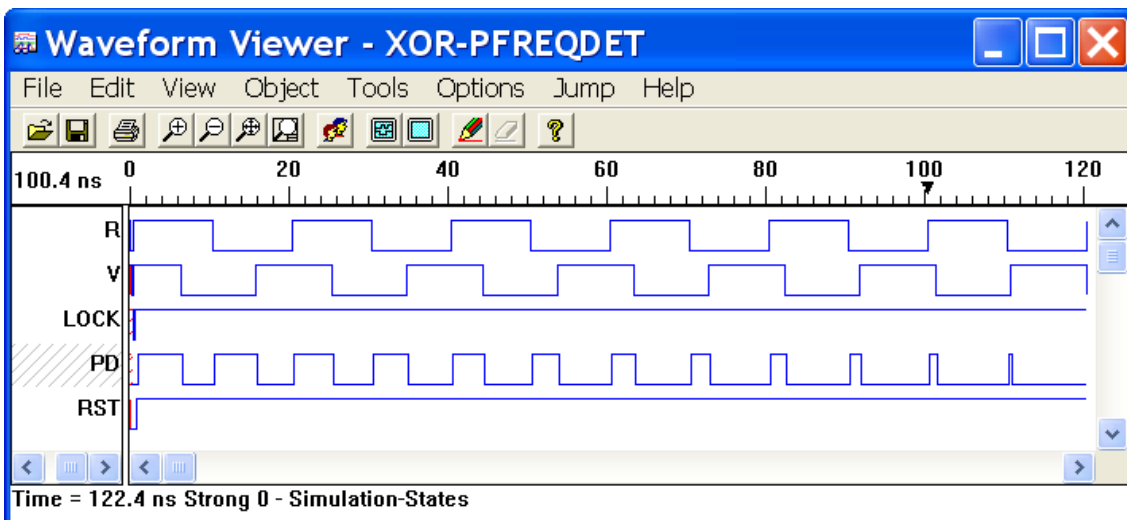


Figure 11. Waveforms of slightly different frequency.

To generate a waveform stimulus file, select *new -> waveform Stimulus* in the project navigator. This will open the waveform editing tool. Select *new wave* to generate a new waveform and type in the name of the signal for which the waveform is to be generated, like Clk. Then clicking in the time section (right hand side of the waveform editing tool window) will start drawing the waveform, the waveform toggles with every mouse click. Repeating the process for all other required waveforms generates the required test vectors. Saving the waveforms will produce a .wvl file. That .wvl file is a text file that contains the waveforms. This file can be edited using the text editor that is part of the ispLever system. A waveform file that will produce waveforms shown in figure 12 is shown below:

```

%MASTERCLOCKMULT = 1;
%SMALLESTUNIT    = 12;
%AUTOASSIGN      = 1;
%DECIMALS        = 0;
%ENDTIME         = 1000000;
R { A In Default None 0 1 50 } = 1 100 0 300 (1 10000 0 10000)#10 ;
V { A In Default None 0 1 50 } = 1 200 0 200 1 6000 (0 9500 1 9500)#10 ;
Rst { A In Default None 0 1 50 } = 0 800 1 100000 ;
PD { A Out Default None 0 1 50 } = ;
Lock { A Out Default None 0 1 50 } = ;

```

The first four lines set the start and end time and the clock interval. The last two lines are the input signals. The next three lines are the waveform files. Waveform R consists of a signal that is a 1 for 100 nS, followed by a 0 for 300 nS and then 10 repeats of a waveform consisting of a 1 for 10000 nS and a 0 for 10000 nS. The V waveform has a corresponding period of 2x9500 nS and will thus slowly slip in phase with respect to waveform R as shown in figure 12. The waveforms PD and Lock are output waveforms, so that the waveform does not need to be specified.

Using the text editor it is very easy to modify these waveforms and add input or output signals. The waveform stimulus (.wdl) file is then saved and imported into the project by selecting import and selecting waveform stimulus and the appropriate file. That will then insert the .wdl file in the project navigator as shown on the project navigator window shown two pages earlier in these notes. It is a lot easier to use a text editor to construct the correct waveform, than using the graphical editor. The easiest way to produce the .wdl file is thus generate the new waveform stimulus file and associate that with the appropriate module. Then insert a simple waveform and save the file to produce the .wdl file. This .wdl file is then edited to set the exact timing and repeat the waveform the required number of times.

ABEL test vector files and .wdl waveform files can be associated with individual modules in a hierarchical structure, so that an individual test vector or waveform file can be used for each of the modules that need to be tested in detail. This allows the individual components to be tested in a simple manner.

## ***VHDL Test Bench***

In order for a VHDL test bench to be used in the ispLever system, other software like Modelsim, which was part of the full ispLever versions up to 7.0, or Aldec Active-HDL, which is included with the full version of ispLever7.1 needs to be installed as part of the compiler system. ispLever Classic V1.1 includes a licence for Active-HDL and ispLever Classic V1.2 includes a special Lattice version. In versions before ispLever Classic V1.1, it was impossible to use test benches. Since the use of ABEL and the subsequent use of ABEL test vectors is reducing, the use of VHDL test benches needs to be included in software. This was done with ispLever Classic V1.1 onwards.

The test bench below was substantially generated automatically, with only a few minor alterations having been made. To generate a VHDL test bench template, simply double click on the VHDL Testbench Template icon in the ispLever Navigator window shown in page 10 of these notes. That generates most of the code below, the part not done is in between the “user defined section”, which contains the actual waveforms to be used.

The user-defined section containing the waveforms can be written from scratch, or they can be generated using the waveform editor, which is used to produce or edit the .wdl

file discussed above. Produce a suitable .wdl waveform for the project. In the waveform editor, load the required waveform Click in File ⇒ Export and select vht to generate a VHDL test-bench file for the waveforms shown in the waveform editor. That .vht file can be loaded into the Lattice Text editor and used to replace the stimulus generation user defined section of the VHDL template generated from within the ispLever Navigator program.

Note: the **LIBRARY** generics; does not exist in Active-HDL and does not exist in ispLever V1.1. The generics library is not needed and the Active-HDL simulator gives an error message unless that line is commented out.

As an example of this process, the sequence generator program on page 15 of the VHDL Language notes is modified to use std\_logic\_vectors instead of integers, resulting in the following program:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity SeqCount is
    port (Clk ,Rset: in std_logic;
          LED : out std_logic;
          Count : inout std_logic_vector(5 downto 0));
end;

architecture SeqCount_Arch of SeqCount is
signal Clast : std_logic_vector(5 downto 0);
begin
    process
    begin
        wait until rising_edge (clk);
        if (Rset = '0') then
            Count <= "000001";
            Clast <= "000000";
            Led <= '0';
        elsif (Count < "110010" ) then           --50 in Binary
            Clast <= Count;
            Count <= Clast + Count;
            LED <= '1';
        else
            LED <= '1';
            Clast <= Count;
            Count <= Clast ;
        end if;
    end process;
end SeqCount_Arch;

```

When using an ABEL or .wdl test file to do a functional or timing simulation, the waveforms shown in figure 12 result. Those waveforms are the same as shown in figure 2 of the VHDL-Language notes,

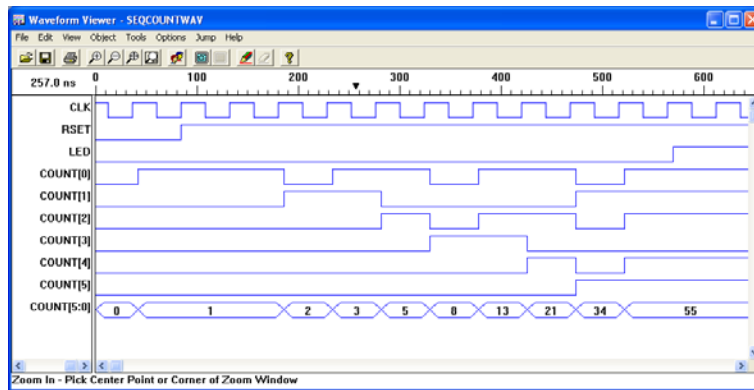


Figure 12. Test Waveforms for the program SeqCount

Generating the VHDL testbench file is using the VHDL Testbench Template generator in the ispLever Project Navigator and inserting the waveform equations as obtained from the Export function of the waveform editor gives the VHDL testbench file as follows:

```
-- VHDL Test Bench Created from source file SeqCount.vhd -- 08/18/08 10:30:54
--
-- Notes:
-- 1) This testbench template has been automatically generated using types
-- std_logic and std_logic_vector for the ports of the unit under test.
-- Lattice recommends that these types always be used for the top-level
-- I/O of a design in order to guarantee that the testbench will bind
-- correctly to the timing (post-route) simulation model.
-- 2) To use this template as your testbench, change the filename to any
-- name of your choice with the extension .vhd, and use the "source->import"
-- menu in the ispLEVER Project Navigator to import the testbench.
-- Then edit the user defined section below, adding code to generate the
-- stimulus for your design.
--
LIBRARY ieee;
--LIBRARY generics;          --not needed, Library not in ispLever Classic
-- comment out the above line for testbench to work
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
--USE generics.components.ALL;      --not needed, not in ispLever Classic
-- comment out the above line for testbench to work
ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS
  COMPONENT SeqCount
  PORT(
    Clk : IN std_logic;
    Rset : IN std_logic;
    Count : INOUT std_logic_vector(5 downto 0);
    LED : OUT std_logic      );
  END COMPONENT;
  SIGNAL Clk : std_logic;
  SIGNAL Rset : std_logic;
  SIGNAL LED : std_logic;
  SIGNAL Count : std_logic_vector(5 downto 0);
BEGIN
  uut: SeqCount PORT MAP(
```

```

Clk => Clk,
Rset => Rset,
LED => LED,
Count => Count );
-- *** Test Bench - User Defined Section ***

tb : PROCESS -- stimulus for signal Clk
BEGIN
    Clk <= '1'; WAIT FOR 12000 PS;
    Clk <= '0'; WAIT FOR 24000 PS;
    FOR i0 IN 1 TO 16 LOOP
        Clk <= '1'; WAIT FOR 24000 PS;
        Clk <= '0'; WAIT FOR 24000 PS;
    END LOOP;
    wait; -- will wait forever
END PROCESS;

p_Rset: PROCESS -- stimulus for signal Rset
BEGIN
    Rset <= '0'; WAIT FOR 12000 PS;
    Rset <= '0'; WAIT FOR 72000 PS;
    Rset <= '1'; WAIT FOR 1100000 PS;
    WAIT; -- forever
END PROCESS;
-- *** End Test Bench - User Defined Section ***
END;
```

The waveforms are generated using two processes, one for the clock, which contains a FOR loop, the other for the Reset, which is a simple set of assignments. The User defined section is fairly simple, so that a VHDL test bench can easily be generated.

Importing the above VHDL test bench into the project (using Source  $\Rightarrow$  Import) and linking it to the FPGA device, and then running a VHDL Functional Simulation, starts the Active-HDL simulator and produces the waveforms as shown below. The operation of the Active-HDL simulator in ispLever is very similar to that for ABEL test vectors.

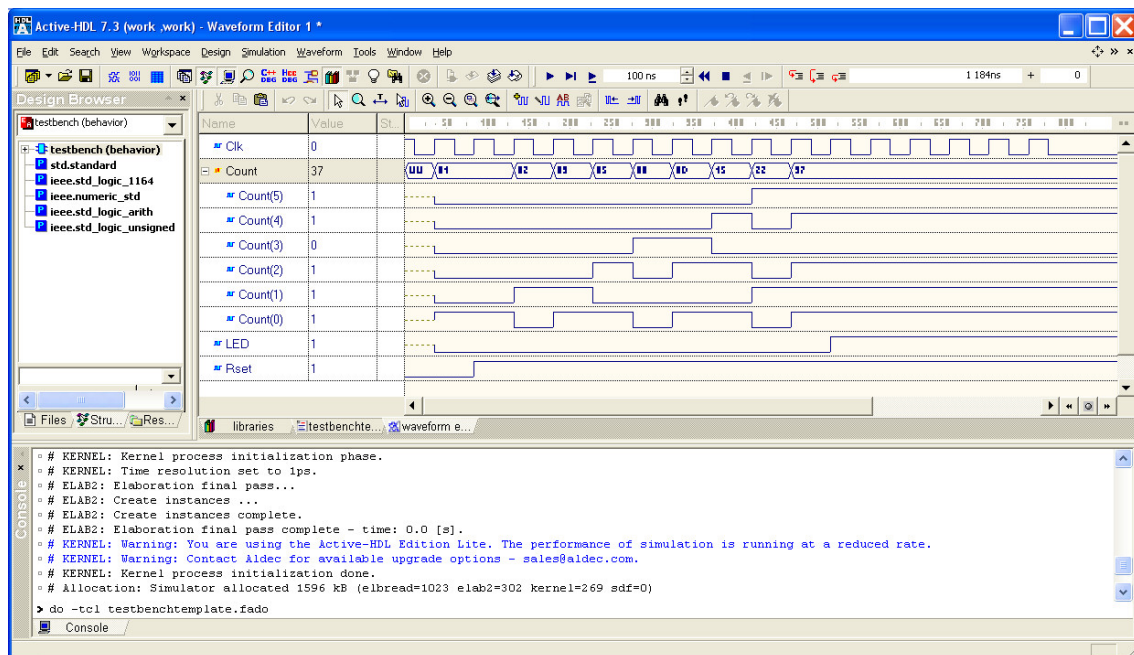


Figure 13. VHDL Test-bench waveforms

The waveforms again are exactly the same as those produced by the ABEL test vectors or the .wdl waveform editor.

The .wdl files and VHDL test benches give much better control of the waveforms and thus allow minimum pulse widths, propagation times etc to be investigated. The easiest way to generate those waveforms is to use a text editor to modify an existing .wdl file.

## Bi-Directional Buffer

Bi Directional buffers are difficult to do correctly in VHDL. A bidirectional latched buffer has two IO ports A and B, a Data Direction input signal and a Clock signal. If the data direction signal is a High, then the input data at A is latched and is available at the output B. The output A is a high impedance under those conditions. If the data direction is Low, then the input data at B is latched and is connected to the output A. The output B is a high impedance under those conditions.

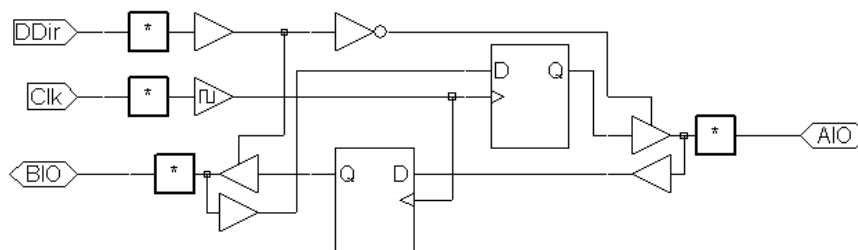


Figure 14. Bidirectional Buffer Schematic.

In ABEL the data direction input of a bidirectional IO pad cannot be accessed separately unlike the AIO.OE (Output enable). In VHDL the setting of the output enable is implied. The simplest architecture code for this is:

```
architecture bireg_a of bireg is
begin
  Buff_P: process
  begin
    wait until rising_edge(Clk);
    if (DDir = '1') then      --data directrion from A to B, A input, B output
      BIO <= AIO;
      AIO <= "ZZZZZZZZ";
    else
      AIO <= BIO;
      BIO <= "ZZZZZZZZ";
    end if;
  end process Buff_P;
end;
```

This code works fine with the Precision compiler, but does not work with the Synplify compiler. By changing the code such that the input is latched on the clock pulse and the flip-flop output is transferred to the output when the flipflop output or the direction status changes, then correct operation is obtained using both compilers. The resulting code is as follows:

```
architecture bireg_a of bireg is
  signal BLatch : std_logic_vector (7 downto 0);
```

```

signal ALatch : std_logic_vector (7 downto 0);
begin
  Buff_P: process
  begin
    wait until rising_edge(Clk);
    if (DDir = '1') then      --data direction from A to B, A input, B output
      BLatch <= AIO;
    else
      ALatch <= BIO;
    end if;
  end process Buff_P;

  DDir_P: process (DDir, ALatch, BLatch)
  begin
    if (DDir = '1') then      --data direction from A to B, A input, B output
      BIO <= BLatch;
      AIO <= "ZZZZZZZZ";
    else
      AIO <= ALatch;
      BIO <= "ZZZZZZZZ";
    end if;
  end process DDir_P;
end;

```

To test the operation of the bidirectional latch, an ABEL test vector file is produced. Much of the file can be generated automatically.

```

MODULE bireg
" TOOL:   vhd12tf
" DATE:   08/13/07 14:35:38
" TITLE:  Lattice Semiconductor Corporation
" MODULE: bireg
" DESIGN: bireg
" FILENAME: bireg.abt
" PROJECT:  bidir
" VERSION: 1.0
" NOTE: DO NOT EDIT THIS FILE DIRECTLY.
" This file is auto generated by ispLEVER System.
" It can be removed or overwritten automatically.
" If you want to edit this file, copy it to another file or
" rename it with different file extension first.

" Inputs
  Clk pin;
  DDir pin;

" Bidirs
  AIO_7_ pin;
  AIO_6_ pin;
  AIO_5_ pin;
  AIO_4_ pin;
  AIO_3_ pin;
  AIO_2_ pin;
  AIO_1_ pin;
  AIO_0_ pin;
  BIO_7_ pin;
  BIO_6_ pin;

```

```

BIO_5_ pin;
BIO_4_ pin;
BIO_3_ pin;
BIO_2_ pin;
BIO_1_ pin;
BIO_0_ pin;

" Variables
AIO = [AIO_7_,AIO_6_,AIO_5_,AIO_4_,AIO_3_,AIO_2_,AIO_1_,AIO_0_];
BIO = [BIO_7_,BIO_6_,BIO_5_,BIO_4_,BIO_3_,BIO_2_,BIO_1_,BIO_0_];

Test_vectors
([Clk,DDir,AIO,BIO] -> [AIO, BIO])
[c., 1, 0, .x.] -> [.x., 0];
[c., 1, 1, .x.] -> [.x., 1];
[c., 1, 2, .x.] -> [.x., 2];
[c., 0, 2, .x.] -> [.x., 2];
[c., 1, 4, .x.] -> [.x., 4];
[c., 1, 8, .x.] -> [.x., 8];
[c., 1, 16, .x.] -> [.x., 16];
[c., 1, 32, .x.] -> [.x., 32];
[c., 1, 64, .x.] -> [.x., 64];
[c., 1, 127, .x.] -> [.x., 127];
[c., 0, .x., 0] -> [0, .x.];
[c., 0, .x., 1] -> [1, .x.];
[c., 0, .x., 2] -> [2, .x.];
[c., 0, .x., 4] -> [4, .x.];
[c., 0, .x., 8] -> [8, .x.];
[c., 1, .x., 8] -> [8, .x.];
[c., 0, .x., 16] -> [16, .x.];
[c., 0, .x., 32] -> [32, .x.];
[c., 0, .x., 64] -> [64, .x.];
[c., 0, .x., 127] -> [127, .x.];

```

**END**

In the above ABEL Test vector file, all the lines above “Variables are generated automatically. To simplify the generation of the test vectors, the individual pins AIO\_7\_ and so on are combined in variables and those variables are then used in the test vector file. Since this is a bidirectional buffer, AIO and BIO are both inputs and outputs. When the direction is from AIO to BIO, AIO is a .x. (don’t care) at the output and BIO is a .x. (don’t care) at the input. The resulting waveforms are shown in figure 16. Some of the individual signals are shown as well as the AIO and BIO busses. Timing simulation is used in order to accentuate the direction of the data flow. On the left side of figure 16, AIO is the input and BIO is delayed with respect to AIO. On the right side, BIO is the input. Two errors are included in the test vector file, where the input is not specified. These clearly show that the output is a high impedance ‘Z’. Since the input is not specified at that time the input is an unknown ‘U’.

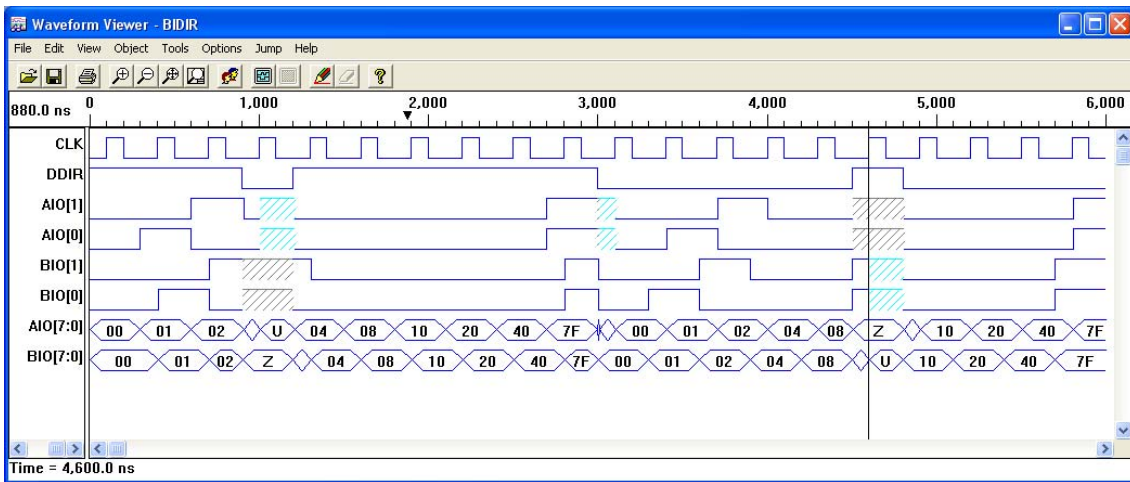


Figure 15. Waveforms of the bidirectional buffer.

## Programming the Device

The Download Cable software is used by running the ispVM System software in the Lattice programming suite. Once the program is loaded, ensure the download cable is connected to the computer and the device to be programmed and select “Scan” to get the software to detect the device. When the “Scan” is executed, a boundary scan of the device is performed. This boundary scan executes a small sequential logic test program to verify that all the pins of the device respond correctly. As part of this process the device is identified. A more detailed description of boundary scans can be found in the CC2510 Lecture notes: <http://eng.jcu.edu.au/subjects/cc2510/notes/Logic/CC2510-LN10-EPLD-FPGA.pdf>. Once the boundary scan is completed, device is identified in the configuration setup window, as shown in figure 16. When the correct device is shown in the Scan Configuration Window, the boundary scan is complete and all the pins function correctly.

To load the JED file double click in the blank space below FileName/IR-Length and use the directory browser to find and select the required .jed file. Then select the “GO” button to program the device. When that is completed, messages relating to the success of the programming are shown on the Scan Configuration Setup window, as shown in figure 17.

It should be noted that all the Lattice programmable devices can all be programmed using this system and that one connection can be made to program several devices in series. It is however more convenient during development of boards, that each of the devices are programmed by having individual programming control lines. Having one programming connection for all the devices would require all the devices to be reprogrammed if a change to one device is to be made. In production, having one programming termination is however an advantage.

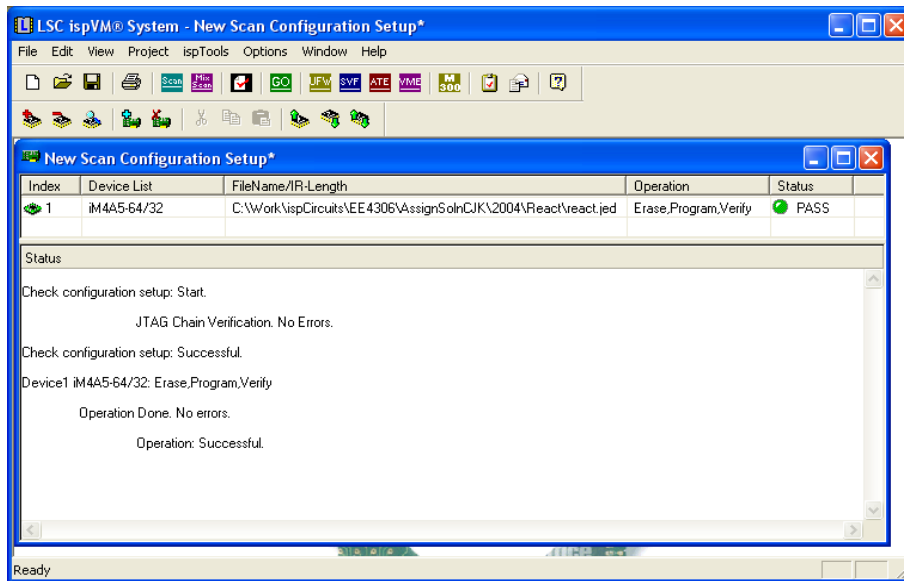


Figure 16. ispVM Device Programming Window

The download program is updated at a different rate from the other software and it is advisable to ensure that the latest version compatible with the download cable hardware is installed.

The schematic diagram for an early version of the download cable, was shown in the 1994 Lattice Databook and can be accessed on the subject web pages.

The boundary scan is a very useful tool when developing Circuit boards. As an example for a Satellite Beacon Receiver we have developed at JCU, a 23 MHz IF signal is sub-sampled at 18 MHz and Digital Down Converted using an Analog Devices AD6620 IC. A Lattice ispMachLC4256V IC is used as glue logic, to perform a bidirectional buffer for the DSP processor, generate the correct waveforms and drive the LCD display, control the DDC, Send serial data from the DDC to the DSP, control (latch and release when read) the overload signal from the ADC and control a microwave noise source for system calibration. The Lattice device also acts as a logic voltage converter, changing 5V input voltages to 3.3V output voltages. The ispMachLC4256V has 100 pins with a pin spacing of 0.5 mm and the AD6620 has 80 pins with 0.65 mm pin spacing. The easiest way to check if any of the pins of these the devices are shorted together, is to do a boundary scan. Such a boundary scan will not show if all pins are correctly soldered to the pads, however the power supply and ground pins must be connected correctly for a scan to succeed.

The bsd or bsdl files used in the boundary scans follow an IEEE standard. As a result the AD6620 can be boundary scanned using the Lattice ispVM software and download cable. The BSDL file for the AD6620, can be downloaded from the Analog Devices web site. This file is part of the design tools for the AD6620. The file is a text file:

[http://www.analog.com/analog\\_root/static/techSupport/designTools/evaluationBoards/downloads/ad6620\\_rev1.txt](http://www.analog.com/analog_root/static/techSupport/designTools/evaluationBoards/downloads/ad6620_rev1.txt)

To use the BDSL file, place it in an appropriate directory, such as the Database subdirectory of the ispvmsystem directory, where ispVM is installed. Rename the file extension from ad6620\_rev1.txt to ad6620\_rev1.bsd. In ispVM, select "ispTools" and Add/Remove Device. Then select Add and in the IEEE 1532 Device information window select browse to fine the required bsd file as shown in figure 18.



Figurer 17. IEEE 1532 Device information from .bsd file.

Selecting OK on the Device Information window and the Add/Remove Device window, updates the database. Connecting the download cable to the appropriate BSD test pins on the AD6620, gives the boundary scan for the AD6620 and identifies the device correctly as shown in figure 18.

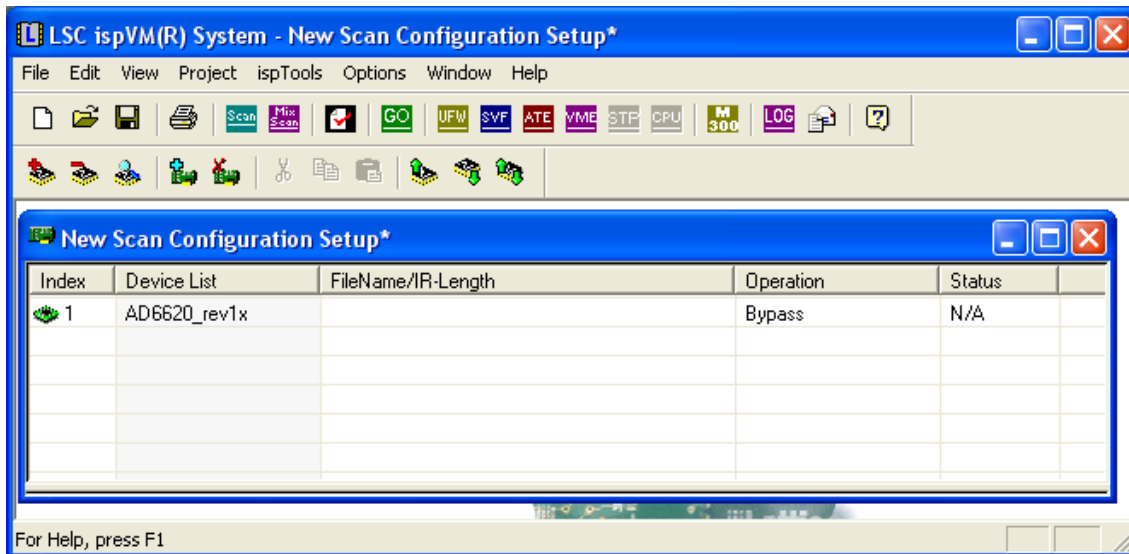


Figure 18. Boundary Scan of AD6620 using the ispVM download cable.

It is actually possible to program the AD6620. However unless one has expert knowledge on how the AD6620 can be reconfigured, that is likely to ruin the device.