ZeBu Getting Started

August 2007

Applies to ZeBu software version 4.2.0 and later

Foreword

Thank you for considering ZeBu.

This booklet will guide you through the steps to run your first accelerated simulation, as well as give you key information on preparing your design for emulation on ZeBu.

Luc Burgun, CEO

EVE

How Can ZeBu Help My Project?

ZeBu is used in many different ways, from accelerating lengthy simulations to verifying scenarios that require billions of cycles and are totally out of reach of traditional simulation, such as simulating multiple frames of full HDTV content or booting an operating system.

Simulation Acceleration

The most obvious way to benefit from ZeBu is to accelerate an existing simulation that just takes too long. Whether it allows you to run more small tests or add longer tests to a regression suite, ZeBu can accelerate simulation by a factor 10 to 100 with little effort. After optimizing the testbench, acceleration can be in the thousands, opening new opportunities for verification.

"Mobile and Calendar" standard video pattern

Verifying Graphics Applications

Simulating graphics and video is a common challenge for CPUs. Most verification teams resort to tricks such as creating test cases on images the size of a post stamp to make the lengthy simulations bearable. As HDTV pushes video to higher resolutions and graphics moves to always more compression with H.264 or 3D graphics, it is becoming impossible to simulate any realistic scenario.

ZeBu has the raw power to handle the most data intensive graphics and video applications, for instance streaming two HDTV streams to a video chip and displaying the output in seconds, instead of days.

HW/SW Co-Verification

An SOC is really ready to ship when the complete application works, not just when hardware simulations pass regressions. In other words, the ultimate test for a chip is to see it perform its application correctly and completely. That means execute the embedded software together with the RTL, but traditional simulation runs out of steam trying to process such volumes of data. With ZeBu, not only does its speed allow full simulation of the application, but the debug environment allows both hardware and software debugging. Hardware designers have access to waveforms of the chip, while software developers use their standard software debugger, typically connected through some kind of JTAG interface to the chip.

Run JAVA Application on cell phone display

Processor Verification

Processor verification typically involves simulating the processor core(s) with their associated memory, loading those memories with diagnostics and letting the processors execute the code. A series of external checkers verify that the processor is functioning correctly by comparing it to a C++ instruction set reference model.

By mapping the entire processor system with its memory in ZeBu, very high emulation speed is achieved very easily. With ZeBu's direct access to the program memory, it is easy to run non-regression tests in batch mode. External C++ checkers and monitors also plug in easily to the emulated model.

Another need for processor verification is to run applications and operating systems. Successfully booting Linux on a simulated version of a processor core gives more confidence to the design team than hundreds of passing diags.

How Fast Can My Design Run?

Why Do I Hear Different Speed Quotes For The Same Design?

The speed of the design's core is the same whether you run in simulation acceleration mode or prototyping mode. In prototyping mode, however, the external interfaces must run at-speed, while the rest of the design is slower. The trade-off is that there is a mismatch in the relative speeds of the blocks with respect to the original SOC. In acceleration mode, all the blocks of the design are slowed proportionally by the same factor, so that interfaces are perfectly cycle-accurate and can be modeled as precisely as required.

Let’s take the example of an SOC with a processor running at 200 MHz and a USB interface at 25 MHz. Suppose it is partitioned across so many FPGAs that the inter-FPGA communication slows it down by a factor of 20.

In acceleration mode the processor would therefore run at 10 MHz, the USB bus at 1.25 MHz. A testbench could exhaustively test all the different back-to-back conditions on the USB bus. In this case the overall design speed is said to be 10MHz.

In prototyping mode, the USB interface and its controller run at speed (25 MHz). The rest of the SOC, such as the processor, is either scaled down to 10MHz or simply disconnected if you just want to test the controller. The prototype is connected to live traffic and tested in a “real environment,” which is invaluable. However, the USB traffic could now overwhelm the slower processor, forcing you to modify the RTL to keep the prototype functional. The processor may also be unable to keep the interface busy, so corner cases could be missed. It is tempting to say that the prototyping speed is 25 MHz, although much of the design runs slower.

Which Is More Accurate, Prototyping Or Acceleration?

As made clear in the example above, both prototyping and acceleration have benefits and tradeoffs. The DUT in simulation acceleration mode is more accurate, but interacts with software models of the outside world, which may not be so accurate. The DUT in prototyping mode may have been modified from the original SOC, but it interacts with the real world.

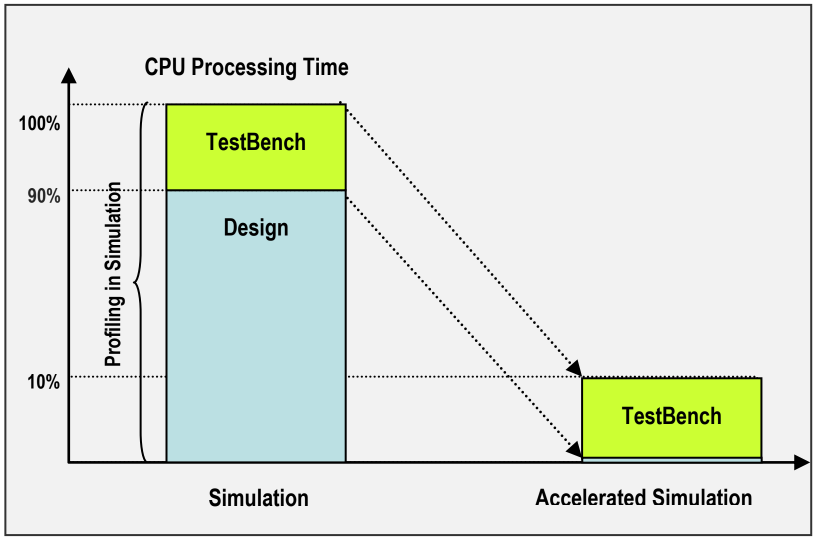
ZeBu supports both modes of operation.

Predicting Simulation Acceleration Speed

You can predict the acceleration factor offered by simulation acceleration very simply, before using ZeBu, by simply profiling[1] your existing RTL simulation as shown below:

```
vcs -f simulation.vf +prof
```

The profiling information will report the time spent by the HDL simulator in various modules of the design and testbench. Adding all the time spent in the testbench and ignoring the time spent in the design itself will give the approximate accelerated simulation time. For instance, as the figure shows, if 90% of the simulation time is spent in the design and 10% in the testbench, then the likely acceleration factor will be ten times (from 100% to 10%).



[1] See XXX for details on how to profile a simulation

What Are The Steps To Map My Design?

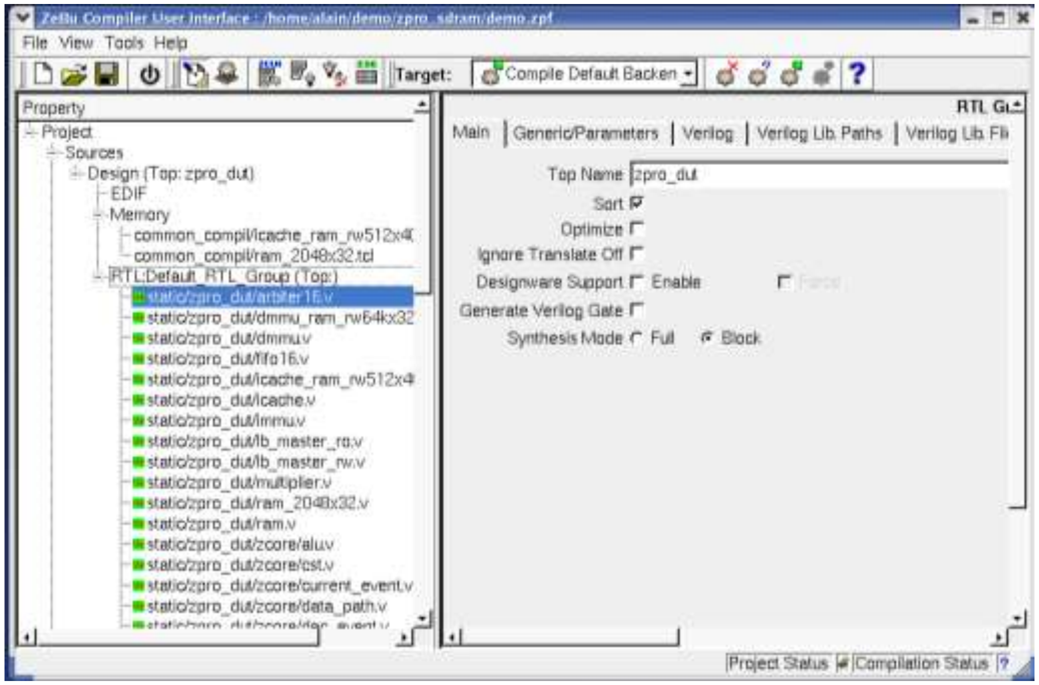
The ZeBu compiler provides very advanced algorithms to make mapping into an array of FPGAs as simple and straightforward as possible. The major issues with such mapping are always the same: clocks, memories and partitioning. Each of those topics is discussed in the next three sections. Below are the general instructions on how to get started with the GUI.

Invoking the ZeBu Compiler

The GUI for the ZeBu compiler is invoked by using the "zCui" command. Make sure the ZeBu software has been correctly installed and that you sourced the script that sets up your environment to point to the ZeBu compiler (`ZEBU_ROOT`, `PATH`, `LD_LIBRARY_PATH`).

Importing the RTL Design

In the "Design" tab of the GUI, add all the RTL source files to the RTL group. Note that you must specify the name of the top-level module: the ZeBu compiler only supports mapping one top-level and all its sub-blocks in the emulator, not multiple top-level modules.



```
Project
  Sources
    Design (Top: zpro_dut)
      EDIF
      Memory
        common_compil/icache_ram_rw512x4(
        common_compil/ram_2048x32.tcl
      RTL:Default_RTL_Group (Top:)
        static/zpro_dut/arbiter16.v
        static/zpro_dut/dmmu_ram_rw64kx32
        static/zpro_dut/dmmu.v
        static/zpro_dut/fifo16.v
        static/zpro_dut/icache_ram_rw512x4
        static/zpro_dut/icache.v
        static/zpro_dut/immu.v
        static/zpro_dut/lb_master_ro.v
        static/zpro_dut/lb_master_rw.v
        static/zpro_dut/multiplier.v
        static/zpro_dut/ram_2048x32.v
        static/zpro_dut/ram.v
        static/zpro_dut/zcore/alu.v
        static/zpro_dut/zcore/cst.v
        static/zpro_dut/zcore/current_event.v
        static/zpro_dut/zcore/data_path.v
```

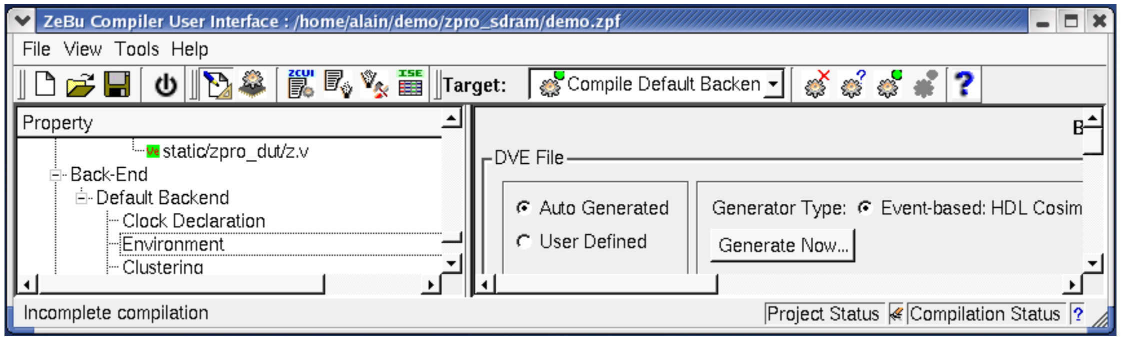
Top Name: zpro_dut
Sort ✓
Optimize ☐
Ignore Translate Off ☐
Designware Support ☐ Enable ☐ Force
Generate Verilog Gate ☐
Synthesis Mode Full [✓ Block]

The ZeBu compiler will automatically synthesize your design for you.

Targeting Simulation Acceleration

In Simulation Acceleration mode, all the top-level ports of the design are exchanged with the HDL simulator at every cycle. From the point of view of the ZeBu compiler, it means that all the top-level ports should be routed to the communication interface, called an HDL driver. The DVE file describes the interconnection of different interfaces with the DUT. In that case, since all the ports go to the same driver, it can be automatically generated from the GUI. All you need to do is list all the top-level input clocks to your design.

In other modes of operation, you need to write your own DVE file that instructs the compiler what verification components interact with the design.


ZeBu Compiler User Interface : /home/alain/demo/zpro_sdram/demo.zpf

Target: Compile Default Backen

Property
- static/zpro_dut/z.v
- Back-End
  - Default Backend
    - Clock Declaration
    - Environment
    - Clustering

DVE File
- [✓ Auto Generated] User Defined
- Generator Type: [✓ Event-based: HDL Cosim]

Incomplete compilation

Specifying an Emulator Configuration

The ZeBu compiler can target different ZeBu architectures. For instance, ZeBu-XXL supports from 8 to 64 FPGAs. ZeBu-UF can have from 1 to 4 FPGAs. You need to tell the ZeBu compiler exactly which configuration you target.

Compiling

Once those options have been set, click on the “Compilation” button and entire process will start, including synthesis of the RTL and place and route of the FPGAs. Once the compilation is done, the design is ready to execute on ZeBu.

Extra options in `zCui` include using remote execution queues such as LSF to share the compilation load on large designs.

How Do I Run Simulation Acceleration?

How Do I Link the Testbench With The Emulated Design

The ZeBu compiler automatically generates a Verilog wrapper in the `zebu.work` directory that has the same module name and top-level pins as the original design. When running your standard simulation, just read that wrapper in replacement of all the design source code. That wrapper makes a PLI call to the ZeBu run-time software, which needs to be included on the Verilog command line as follows. If the RTL simulation is run by:

```
vcs -f testbench.vf -f design.vf
```

Replace it with:

```
vcs -f testbench.vf zebu.work/*.v -P
$ZEBU_ROOT/lib/pli.tab libZebuPLI.so
```

Validating the Test Environment before Emulating

It is strongly recommended to run a gate-level simulation of the design with the original testbench on some simple test sequences before attempting simulation acceleration. As long as that doesn't work, it is unlikely that the simulation acceleration will work. That will flush issues such as subtleties in memory models, reset sequence and hierarchical references inside the DUT.

Performing gate-level simulation with the Xilinx libraries is actually simpler than standard ASIC gate-level simulations, as timing is not required. Synplify Pro for instance generates a gate-level Verilog netlist with the extension "`.vm`" together with the EDIF netlist that the ZeBu compiler requires. To simulate that netlist with the original testbench and the Xilinx library, simply do:

```
vcs -f testbench.vf design.vm
-y $(XILINX)/verilog/src/unisims
-y $(XILINX)/verilog/src/XilinxCoreLib
+incdir+$(XILINX)/verilog/src +libext+.v
$(XILINX)/verilog/src/glbl.v
```

You also want to include in that simulation the ZeBu version of the memory models, so that they also get validated before emulation.

What If the Testbench Makes Hierarchical References to the DUT

If the testbench makes hierarchical references inside the design to read[2] values from the DUT, then you must specify that list of signals in the DVE file before running the ZeBu compiler, so that those signals will be added to the wrapper, at the exact same place where the testbench expects them. In order to do so, you need to modify the automatically generated DVE file and add the hierarchical paths to the signals in as follows:

[2] If the testbench forces internal signals of the DUT, a different approach is necessary, depending on the kind of force and the type of signal.

```
HDL_COSIM design_hdlcosim (
. . .
  .output_bin({
    dut.path.path.signal,
    dut.path2.path.bus2[31:0],
    . . .
```

What If the Testbench Initializes Memories inside the DUT?

Memories inside the DUT can be initialized using the same Verilog file format that the $readmem call uses. If the original testbench makes the following call:

```
$readmem("mem.dat", top.dut.mem)
```

Replace it with:

```
$ZEBU_readmem("mem.dat", "dut.mem");
```

Note that the ZEBU version of the memory name is a string, not a Verilog object, and the path starts from the DUT top-level, not the testbench top-level and uses the name of the instance of the DUT in the testbench.

What Do I Need To Know About Clocks?

Top-Level Clocks

You need to declare the top-level clocks of your design to the ZeBu compiler. Their relative frequencies and aspect ratio can be set and changed at run-time.

Derived and gated clocks are automatically handled by the compiler so that no race ever happens and the design is functional.

How Do I Model A PLL?

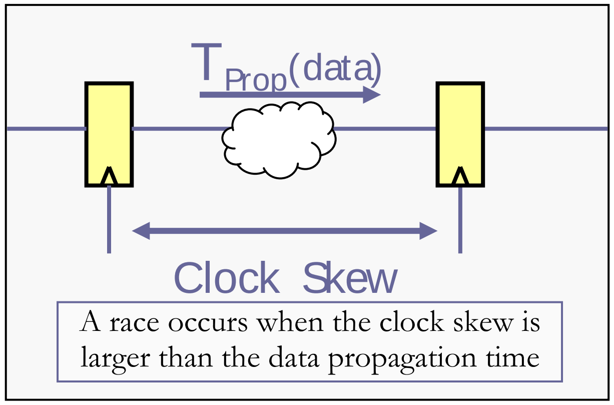
PLLs and DCMs cannot be synthesized. They typically generate derived clocks with fixed ratios compared to a reference clock. In simulation acceleration mode, the easiest way is to let the HDL simulator drive both the reference and the derived clocks (the outputs) of the PLL. This is achieved by blackboxing the PLL module itself and adding the outputs of the PLL to the input_bin of the HDL_COSIM driver in the DVE file.

In transaction-based or in-circuit emulation, you can specify the relationship between primary clocks at run-time, as well as their phase and duty cycle. The sample “designFeatures” file below guarantees that the two primary clocks called “clk66” and “clk33” will be running with a fixed ratio of 2.

```
$B0.clk66.virtualFrequency = 66;
$B0.clk33.virtualFrequency = 33;
```

What About Limitations of Clocks in FPGAs?

A typical FPGA has in the order of 8 or 16 low-skew networks, where design clocks are mapped. That guarantees that the skew between those clocks is less than any skew on the datapaths. If your design has many derived and gated clocks, as is typical in low-power SOCs, then the low-skew network available in FPGAs is not enough. Most clocks will have to go through standard data routes and the design will not be functional because of races between data. As shown on the figure, a race occurs every time the propagation time of the data between two flip-flops is less than the skew between the clocks driving those flip-flops.



A race occurs when the clock skew is larger than the data propagation time

The ZeBu compiler automatically guarantees that the clock skew is never higher than the data propagation time, so the design is always functional. First, the ZeBu compiler will optimize the usage of low-skew networks present in hardware so as to reduce the clock skew as much as possible. If that is not enough to eliminate all races in the FPGAs, then the ZeBu compiler will also automatically modify the data paths to guarantee that they are longer than the clock skew. This approach guarantees both the highest emulation speed possible and first-time design correctness.

What If I Need More Than 8 Primary Clocks

The built-in clock generator is limited to generating 8 primary clocks. Additionally, when the clocks are generated by a software testbench, they can be routed as regular data signals, so their number is not limited.

If need be, you can substitute your own clock generator to replace the built-in clock generator, as long as it's synthesizable. There is then no limit in the number of generated clocks. For instance, the ZeBu-XXL hardware can serve up to 192 low-skew clocks to a design.

How Do I Model Memories?

How Do I Model FIFOs and Small Caches?

Small memories, such as FIFOs and caches typically are mapped into the multiple small memory banks of the Xilinx FPGAs, called BlockRAMs (BRAM). Synthesis will automatically infer BRAMs when the memory model is written in synthesizable RTL and uses an array to reference the memory content.

Where Does The Main Memory Go?

Each ZeBu emulator also has large off-FPGA memory banks called ZRMs with more than 100 MB, used to map large design memories, typically processor memory and so on.

zMem is ZeBu's memory model generator, both on-chip and off-chip. It uses simple commands to describe the main characteristics of any SRAM your design needs, given its width, depth, etc… The "`ZRM`" keyword is how you instruct our compiler to place the memory off-chip vs. inside the FPGAs.

```
memory new mem12Mx32 zrm
memory depth 134217728
memory width 32
memory add_port port0 rw
memory_port port0 addr  addra
memory_port port0 do    doa
memory_port port0 di    dia
memory_port port0 clk   clka POSEDGE
memory_port port0 we    wena LOW
memory generate
```

You can use more banks than are physically present in hardware, as the ZeBu compiler automatically figures out a way to multiplex the design memories transparently using the existing hardware.

How Do I Convert an ASIC Memory Library?

zMem is entirely scriptable using TCL, so that you can generate hundreds of variations of the same memory models using one script. It is very convenient to quickly convert an entire ASIC memory library.

```
foreach mem [info vars Memories::*] {
  memory new $name
  memory depth $depth
  memory width $width
  foreach port $portList {
    memory_port $port di D$pTag
    memory_port $port do Q$pTag
  }
  memory generate $name
}
```

zMem can generate memories with an arbitrary number of ports and supports both synchronous and asynchronous memories.

Does ZeBu Have DDR2 Memories?

If your design has a DDR2 interface[3] (or DDR3, SDRAM or DDR, etc…), you can connect it to one of our memory wrappers. The wrapper behaves functionally like a DDR2 memory. Internally, we use the large SRAM banks to mimic the DDR2 memory. This approach is the most flexible, for instance there is no issue with refreshing the memory despite the fact that the design in emulation is running slower than real-time. The ZeBu model will respond correctly to valid accesses. On the other hand, to verify the detailed timing and compliance of your DDR2 controller, we recommend using a gate-level simulation with timing.

[3] See VSAN001 Application Note for information on how to model the delay line of a DDR controller

Accessing Memory Content

All memories, both inside the FPGAs and on the external memory banks, including ROMs, can be loaded and dumped at run-time. There is no need to specify their content during compilation.

Note that if the memory was described using zMem, then it will appear as one logical memory at run-time, no matter how many BRAMs were needed to implement it.

How Do I Partition My Design Across Multiple FPGAs?

The ZeBu compiler can automatically split the design into the FPGAs available on the system. If a block requires more IOs than what is available in the FPGAs, multiplexing will be automatically inserted transparently.

Do I Need To Synthesize Each FPGA Separately?

There is no need to synthesize each FPGA separately. The ZeBu compiler reads one logical netlist, describing a complete multi-million gate design, and automatically splits it into blocks to fit on each FPGA of the emulator.

Does Multiplexing Slow Down The Design?

When mapping a design to an array of FPGAs, the ultimate speed of emulation is mostly a function of the interconnection between the FPGAs, not the speed of the logic inside the FPGAs. Therefore, the quality of the multiplexing used to connect the FPGAs is critical to achieve the best possible emulation speed. ZeBu uses links of 300 Mbit/s per pin, which guarantees optimal emulation speed. Eventually, too much multiplexing will slow down the emulation, but that effect can be postponed as much as possible with the high inter-FPGA bandwidth available on the ZeBu emulator.

The emulation speed is displayed during the system routing step:

```
# step PATH ANALYSIS : The advised
theoretical frequency using default
settings would be 24242 kHz
```

Note that the frequency displayed (24 MHz) corresponds to the speed of the design if all the design logic only uses one edge of the design clock (one communication per cycle). If the design is sensitive to both edges of the clock and the longest path is a half-cycle path, then that frequency must be divided by two (12 MHz), to obtain the true design frequency.

How Do I Fine-Tune The Placement?

It is possible to override the default placement chosen by the ZeBu compiler and replace it with a list of assignments, based on design hierarchy. There can be two reasons for fine-tuning: when the emulator available doesn't have enough FPGAs for the solution found by automatic clustering; and to achieve higher frequency of emulation when running in-circuit or with transactors. In simulation acceleration mode, the speed bottleneck is the testbench so the design mapped in ZeBu will always run faster than the testbench, no matter how suboptimal the FPGA placement may be.

Each cell of the design is placed in the FPGA that matches the longest mapping directive. For instance, the following lines specify that by default, all the logic should go in FPGA F0_0_0[4], except the ALU block which will go in FPGA F0_0_1, except the FPU part of the ALU which will be placed in FPGA F0_1_0.

```
defmapping core = F0_0_0
defmapping core.alu = F0_0_1;
defmapping core.alu.fpu = F0_1_0;
```

The ZeBu compiler includes a standalone tool called `zNetgen` that lets you explore your netlist after synthesis and analyze possible partition choices.

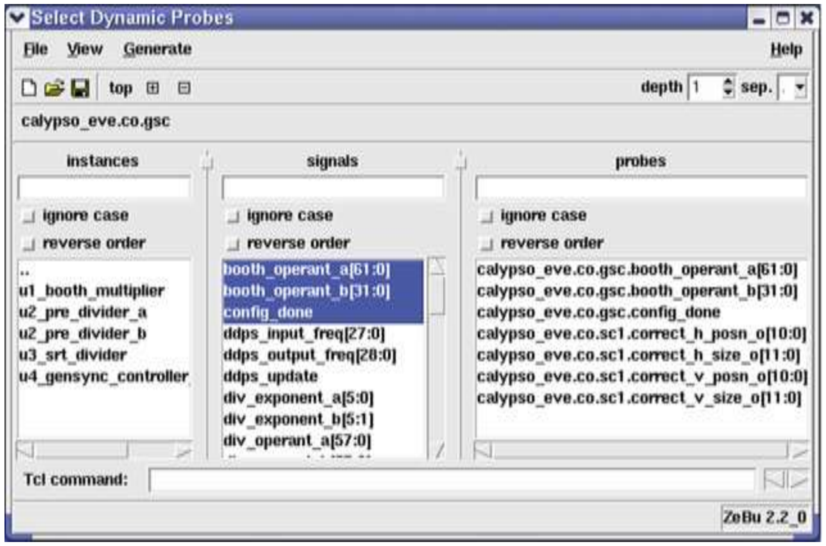
[4] See the ZeBu Reference Manual for the FPGA naming convention depending on the configuration of the emulator

How Do I Debug My Design?

A simulation of a billion cycles requires a different approach to debugging than traditional simulation: locating the source of a problem efficiently is key. ZeBu allows you to view your design at different levels of abstraction, from embedded software all the way down to hardware waveforms, so you can dynamically choose which level is most convenient at any time.

Making Internal Signals Visible To My Simulator

With simulation acceleration, you can make internal signals of the design visible to the simulator, just as if the RTL of the design was being simulated instead of emulated, providing you with the same ease of use. By default, only the top-level ports of the design are visible to the simulator, but you can make any internal register visible (storage element, such as flip-flop or latch) by selecting it in the tool called `zSelectProbes` before simulating. There is no need to recompile your design for that.



..
u1_booth_multiplier
u2_pre_divider_a
u2_pre_divider_b
u3_srt_divider
u4_gensync_controller

booth_operant_a[61:0]
booth_operant_b[31:0]
config_done
ddps_input_freq[27:0]
ddps_output_freq[28:0]
ddps_update
div_exponent_a[5:0]
div_exponent_b[5:1]
div_operant_a[57:0]

calypso_eve.co.gsc.booth_operant_a[61:0]
calypso_eve.co.gsc.booth_operant_b[31:0]
calypso_eve.co.gsc.config_done
calypso_eve.co.sc1.correct_h_posn_o[10:0]
calypso_eve.co.sc1.correct_h_size_o[11:0]
calypso_eve.co.sc1.correct_v_posn_o[10:0]
calypso_eve.co.sc1.correct_v_size_o[11:0]

The extraction of those signals can be turned on and off at run-time, during accelerated simulation, by calling the `zebu_readback()` method. This can be easily mixed with the regular function calls that you use to trace signals in your design:

```
initial begin
  #10000;
  $zebu_readback(1); //
turn on
  $dumpvars(0, top.dut);
  #20000;
  $zebu_readback(0); //
turn off
  $dumpoff();
end
```

Can I Get Traces When Running At Full Speed?

You can trace up to 4,096 signals at-speed by connecting them to the on-board trace memory. The connection is specified in the DVE:

```
SRAM_TRACE(.output_bin( {
  dut.block1.sig1,
  dut.block2.sig2[31:0]) });
```

This trace is called static because you need to recompile your design every time you change that list. Remember that you can always trace all the state of your design at any time without any recompilation using the dynamic tracing capability of ZeBu described in the previous paragraph. It will automatically slow down the emulation speed depending on the amount of data traced.

Why Does My Design Behave Differently In Emulation?

The behavior of the emulated design is actually closer to that of the taped-out chip than the RTL simulation. Incomplete sensitivity lists, propagation of X, initial values at time 0, are all conditions that can cause mismatches in behavior, without either version (RTL or emulation) being incorrect. Emulating with ZeBu is as accurate as running a 0-delay gate-level simulation.

Can I Stop The Emulation?

Except for in-circuit where the outside interfaces would not tolerate clocks to stop, you can always stop the emulation and single-step cycle by cycle. In that regard, ZeBu is very similar to your standard Verilog simulator. You can use this feature for instance to stop after some interesting condition is happening in the chip and then modify the values of some registers or change the content of a memory behind the back of the design, then let it continue executing and observe the differences.

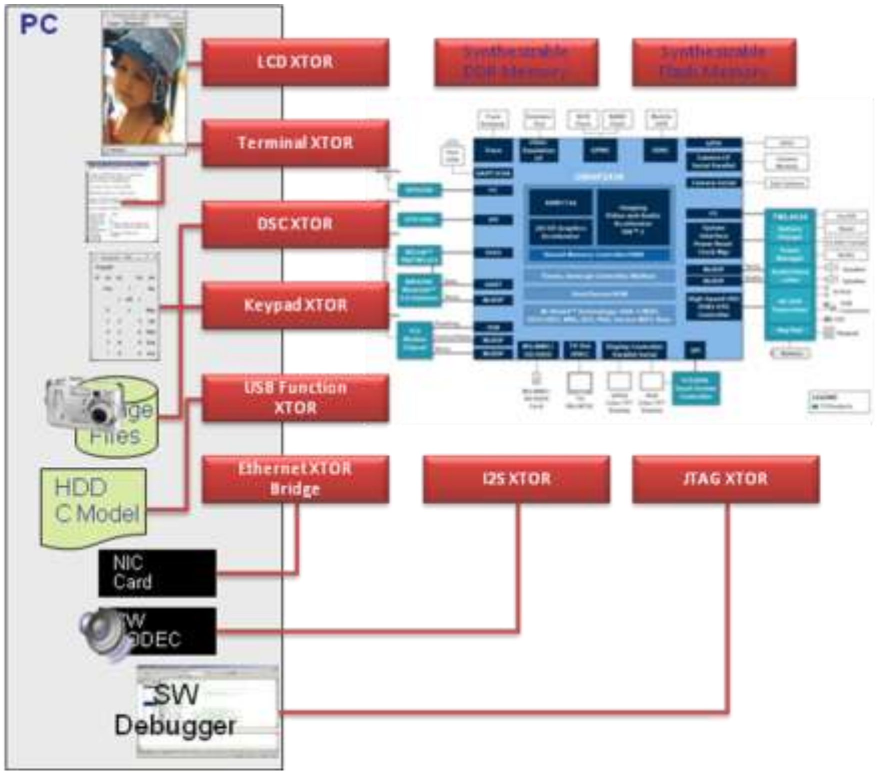
How Do I Easily Build A Testbench Using Virtual Components?

Virtual components allow you to quickly build a complete validation platform, by surrounding your design with transactors that interact with its various interfaces.

And since virtual components use hardware transactors to stream data to and from the device, they offer the same emulation speed as in-circuit.

What Are the Benefits of Virtual Components?

Virtual components offer the flexibility of a software testbench, where setup can be changed literally in seconds. For instance, a PDA virtual platform can change its display settings from a VGA Landscape display to a QCIF Portrait resolution on the fly. Achieving the same results on a real prototype would involve unplugging an LCD display and plugging a different kind, with all the related issues of hardware reliability and availability.



How Does A Testbench Use Virtual Components?

As an example, a UART (RS232 serial port) transactor would typically offer a very simple API that allows the controlling program to **send** and **receive** characters. The code below would display all the characters sent by the device and forward all the keys typed by the user to the device:

```
myUart = new Uart;

while(1) {
  if(myUart->receive(&c) {
    cout << c;
  }
  if(kbhit()) {
    c = readch();
    myUart->send(c);
  }
}
```

How Do I Connect A Virtual Component To The Design?

The DVE file describes how the design connects to all the virtual components that constitute the system environment. You can think of the DVE file as the top-level Verilog module that instantiates your design, with some extra parameters that are emulation specific.

To stay with the UART example, the DVE would simply bind the UART ports of the DUT to the UART transactor component "uart_driver":

```
uart_driver myUartComponent (
.TxD(dut.txd),
.RxD(dut.rxd),
.CTS(dut.cts),
.RTS(dut.rts)
);}
```

Can I Write My Own Components?

Yes. You can build your own component, typically when the interfaces and signals you are interacting with are proprietary. It mostly involves writing a synthesizable bus functional model of the interface, with some glue logic to exchange data between hardware and software.