C/C++ Co-simulation

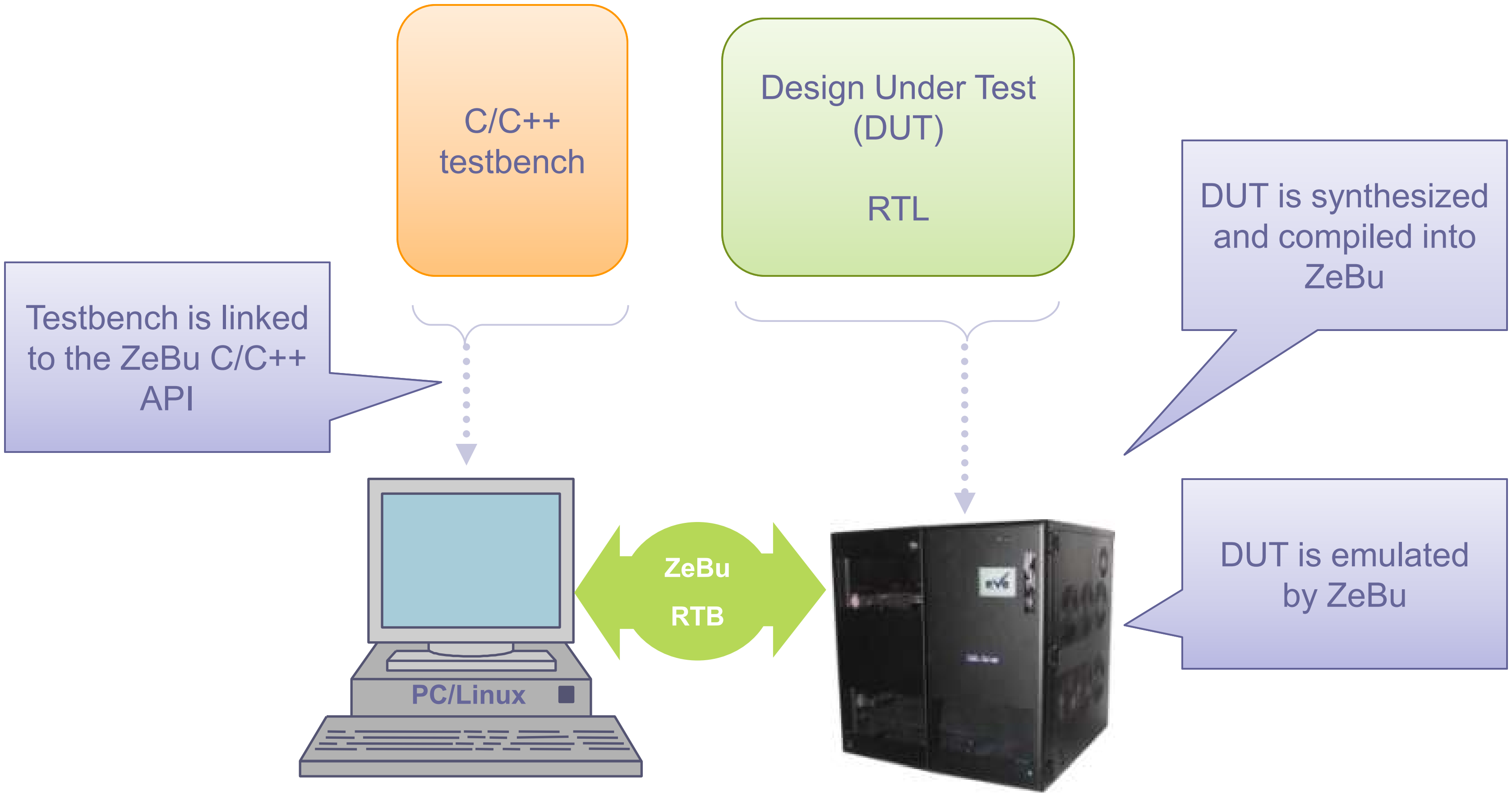
ZeBu-Server Training

THE ***FASTEST*** *VERIFICATION*

Agenda

- Overview
- Compilation
- Writing the testbench
- Debug

Overview


C/C++ testbench
Design Under Test (DUT)
RTL
DUT is synthesized and compiled into ZeBu
Testbench is linked to the ZeBu C/C++ API
ZeBu
RTB
PC/Linux
DUT is emulated by ZeBu

Overview

- The C/C++ testbench is linked to the ZeBu API
- The API is provided for C and C++ languages
  - Include file for C: `#include <libZebu.h>`
  - Include file for C++: `#include <libZebu.hh>`

  - Standard library: `libZebu.so`
  - Debug library: `libZebuDebug.so`
  - Thread-safe library: `libZebuThreadSafe.so`
  - Debug/Thread-safe library: `libZebuThreadSafeDebug.so`

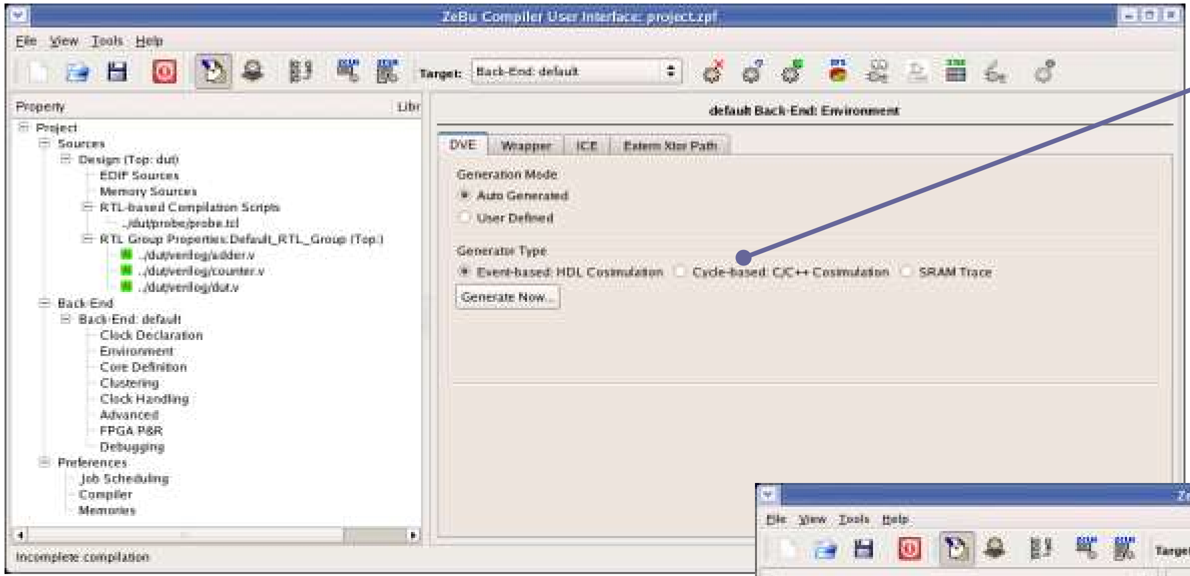
Overview
MCKC versus C_COSIM Drivers

- ZeBu supports three types of drivers with the C/C++ testbench:
  - `MCKC_COSIM`: Event-based Driver
  - `C_COSIM`: Cycle-based Driver
  - Transactors: Application-dependent Drivers
- The three types of drivers can be combined together
- `MCKC_COSIM` and `C_COSIM` drivers are pre-defined and generic drivers serving bit-accurate communication applications
  - `MCKC_COSIM` provides an event-based communication scheme similar to the HDL simulation driver
    - It supports the definition of multiple asynchronous clocks in the testbench
    - It is the ideal interface for integration with 3rd party simulation tools
  - `C_COSIM` provides a cycle-based communication scheme
    - It supports the notion of execution model based on a main clock
    - It is the ideal interface for processor applications

Agenda

- Overview
- Compilation
- Writing the testbench
- Debug

Compilation
zCui



ZeBu Compiler User Interface: project.zpf

File View Tools Help

Target: Back-End: default

Property

- Project
  - Sources
    - Design (Top: dut)
      - EDIF Sources
      - Memory Sources
      - RTL-based Compilation Scripts
        - ../dut/probe/probe.tcl
      - RTL Group Properties:Default_RTL_Group (Top)
        - 🟩 ../dut/verilog/adder.v
        - 🟩 ../dut/verilog/counter.v
        - 🟩 ../dut/verilog/dut.v
  - Back-End
    - Back-End: default
      - Clock Declaration
      - Environment
      - Core Definition
      - Clustering
      - Clock Handling
      - Advanced
      - FPGA P&R
      - Debugging
  - Preferences
    - Job Scheduling
    - Compiler
    - Memories

default Back-End: Environment

DVE | Wrapper | ICE | Extern Xtor Path

Generation Mode

- [✓ Auto Generated]
- User Defined

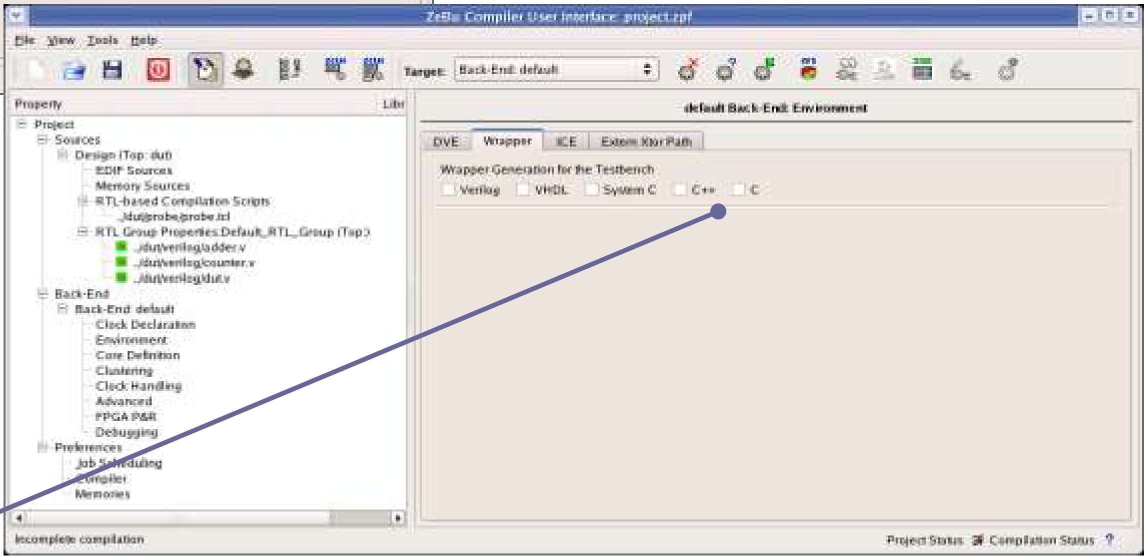
Generator Type

[✓ Event-based: HDL Cosimulation] Cycle-based: C/C++ Cosimulation SRAM Trace

Generate Now...

Incomplete compilation

Select operation mode:
Cycle-based, C/C++ Cosimulation



ZeBu Compiler User Interface: project.zpf

File View Tools Help

Target: Back-End: default

Property

- Project
  - Sources
    - Design (Top: dut)
      - EDIF Sources
      - Memory Sources
      - RTL-based Compilation Scripts
        - ../dut/probe/probe.tcl
      - RTL Group Properties:Default_RTL_Group (Top)
        - 🟩 ../dut/verilog/adder.v
        - 🟩 ../dut/verilog/counter.v
        - 🟩 ../dut/verilog/dut.v
  - Back-End
    - Back-End: default
      - Clock Declaration
      - Environment
      - Core Definition
      - Clustering
      - Clock Handling
      - Advanced
      - FPGA P&R
      - Debugging
  - Preferences
    - Job Scheduling
    - Compiler
    - Memories

default Back-End: Environment

DVE | Wrapper | ICE | Extern Xtor Path

Wrapper Generation for the Testbench

☐ Verilog ☐ VHDL ☐ System C ☐ C++ ☐ C

Incomplete compilation

Project Status Compilation Status ?

Select testbench
language, either C or C++

Compilation
Generated files

- **The compilation flow will generate different wrapper files:**
  - `<driver>.c`, `<driver>.h` files for C co-simulation
  - `<driver>.cc`, `<driver>.hh` files for C++ co-simulation
  - `TOP_<top>.cc`, `TOP_<top>.hh` files for memories and dynamic probes access with C++

Compilation
Generated C++ files

- `<driver>.cc` and `<driver>.hh`
  - These files provide access to the C++ co-simulation driver
  - `<driver>.hh` must be included in the testbench
  - `<driver>.cc` must be compiled and linked with the testbench
- These files declare:
  - `ZEBU::Driver * <driver>`
    - object on the C++ co-simulation driver
  - `ZEBU::Driver * Init_<driver> (ZEBU::Board * <zebu>)`
    - method used to initialize the C++ co-simulation driver object.
  - A structure `<driver>_drv` of `ZEBU::Signal` objects that provides access to the signals of the C++ co-simulation driver
- Including the generated header file in the testbench provides an access to interface signal and static probes

Compilation
Generated C++ files

- `TOP_`*`<top>`*`.cc` and `TOP_`*`<top>`*`.hh`
  - These files provides access to the memories and dynamic probes of the design
  - `TOP_`*`<top>`*`.hh` must be included in the testbench
  - `TOP_`*`<top>`*`.cc` must be compiled and linked with the testbench
- These files declare:
  - `int ZEBU_`*`<top>`*`_Init (ZEBU::Board * `*`<zebu>`*`)`
    - Method used to initialize memory and register access.
  - A structure named *`<top>`* of `ZEBU::Signal` and `ZEBU::Memory` objects that provides access to the dynamic probes and memories of the design

Agenda

- Overview
- Compilation
- Writing the testbench
- Debug

Writing the testbench

- C/C++ testbenches have three main phases:
  1) Initialization
  2) Run
  3) Closing

Writing the testbench Initialization

- **Initialize the ZeBu system with the following steps:**

1) **Board connection**
```
Board *zebu = Board::open();
```

2) **Clock configuration (for each clock)**
```
Clock *clock_obj = zebu->getClock( "clock" );
```

3) **Driver initialization (for each driver)**
```
driver = Init_driver( zebu );
```

4) **Driver connection (for each driver)**
```
driver -> connect();
```

5) **Board initialization**
```
zebu->init();
```

6) **Board initialization check**
```
zebu->check();
```

Writing the testbench Initialization

- For all the methods it is highly advised to check the return code
- In C++, a `try`/`catch` is highly advised
- The last step (6) is optional and verifies that all the initialization have been done correctly. However its usage is highly recommended
- Note that during the "run" phase of the co-simulation, no test is performed on pointers. This may result in segmentation faults caused by incorrect initialization

Writing the testbench
Accessing signals

- **Signals and vectors are declared in the generated header file as `Signal` objects**
- **Including the generated header file in the testbench is enough to access signals and vectors**
- **The name of a vector or a signal includes the complete hierarchy path:**

DVE is:	Names are:
`C_COSIM driver_c (`	`driver_c_drv.signal`
`.input_bin ({`	`driver_c_drv.vector`
`signal,`	
`vector[95:0]`	

- **It is possible to declare aliases of signals and vectors for easier manipulation:**

```
Signal &signal  =  *( driver_c_drv.signal );
Signal &vector  =  *( driver_c_drv.vector );
```

Writing the testbench Accessing signals

- Value of a signal is set using '=':
  ```
  driver_c_drv.input_bin.signal = b0;
  ```
  (Value can be `b0`, `b1` or `bz`)
- Value of a signal is obtained using '*':
  ```
  value = *driver_c_drv.signal;
  ```
- Value of a vector can be set as 32-bit words using the `set` method, or using '=' for the entire value:
  ```
  driver_c_drv.vector->set(0,value);
  ```
  (vector[31:0] = value)
  ```
  driver_c_drv.vector->set(1,value);
  ```
  (vector[63:32] = value)
  ```
  driver_c_drv.vector = "0x1F0000EEEE0000DDDD";
  driver_c_drv.vector = 1524;
  ```
- Value of a vector is obtained, in 32-bit words, using the `get` method :
  ```
  int bits31_0 = driver_c_drv.vector->get(0);
  ```
  (vector[31:0])
  ```
  int bits63_32 = driver_c_drv.vector->get(1);
  ```
  (vector[63:32])
- A vector can also be accessed bit by bit, using square brackets:
  ```
  int bit10 = driver_c_drv.vector[10];
  driver_c_drv.vector[10] = b1;
  ```

Writing the testbench
Running clock cycle

- Use the `run` method:
  `<`*`driver`*`>->run(`*`n`*`);`
- Will run *n* clock cycles of the clock defined in the DVE file as `defparam`

Writing the testbench
Closing

- At the end of the testbench, drivers need to be disconnected, then the ZeBu system closed:
  - Disconnecting the drivers (for each driver)
    ```
    driver -> disconnect()
    ```
  - Closing the ZeBu system
    ```
    zebu -> close()
    ```
- Once a driver has been disconnected, it is not possible to re-connect it
- The closing operations need to be done even in case of error

Writing the testbench
A simple example

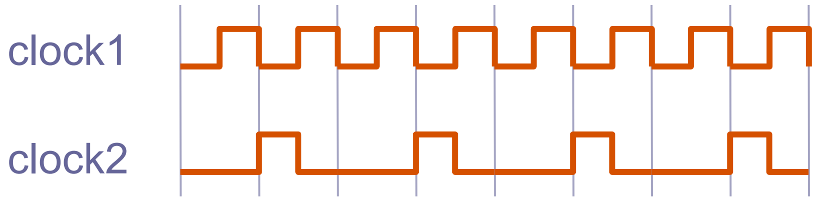
```
#include <libZebu.hh>
#include "topmodule_ccosim.hh "
using namespace ZEBU;
using namespace std;

int main ()
{
  Board *zebu = NULL;
  […]
  try {
    // Open the board connection
    zebu = Board::open(); // No need to test
pointer
    […]
    // Clock parametrization
    Clock *clk_obj
      = zebu->getClock( "clock" );
    […]
    // Initialize the driver and the board


    topmodule_ccosim
      = Init_topmodule_ccosim(zebu);
    if( topmodule_ccosim == NULL) { //error }
    if( topmodule_ccosim -> connect() ) { //error
}

    zebu->init(); // No need to test result

    if ( ! zebu->check() ) { //error }
    […]
```

```
     // Defines signal aliases
    Signal &reset = *(topmodule_ccosim_drv.reset);
    Signal &din   = *(topmodule_ccosim_drv.din);
    Signal &dout  = *(topmodule_ccosim_drv.dout);
    […]

    // Reset the design
    reset = b1;
    din = "0x0"; // all bits set to 0
    topmodule_ccosim -> run(10);

    // Perform the testing
    reset = b0;
    din = "0x52300000FE00271A";

    while ( dout!=0x4266 )
      {
      topmodule_ccosim -> run(1);
      din.set(0, dout.get(0) );
      }
      […]
    // Close the driver and the board
    topmodule_ccosim -> disconnect();
    […]
    }
  catch (exception &excp) {
    cerr  << "Exception trapped : "
          << excp.what() << endl;
    }
  if( zebu ) {
    zebu->close();
    }
```

Writing the testbench
Event-based C/C++ co-simulation

- Event-based C/C++ co-simulation is selected by choosing the `MCKC_COSIM` driver in the DVE file
- It is mostly the same as `C_COSIM`, except that clocks are managed in same way as in HDL co-simulation:
  - Clocks are regular signals whose the value can be forced to 0 or 1
  - An `update` method synchronizes the software with the hardware
  - No clock configuration is required in the testbench

```
topmodule_mckccosim_drv.clock1 = b0;
topmodule_mckccosim_drv.clock2 = b0;
topmodule_mckccosim->update();
topmodule_mckccosim_drv.clock1 = b1;
topmodule_mckccosim->update();
topmodule_mckccosim_drv.clock1 = b0;
topmodule_mckccosim_drv.clock2 = b1;
topmodule_mckccosim->update();
topmodule_mckccosim_drv.clock1 = b1;
topmodule_mckccosim_drv.clock2 = b0;
topmodule_mckccosim->update();
```



Compiling the testbench

- Compilation of C/C++ testbenches is done using GNU Compiler g++
- Use correct g++ version:
  - g++ 3.4, RedHat Enterprise Linux 4
- Correct include path must be given to compiler:

```
g++ -c zcui.work/zebu.work/driver.cc –I$ZEBU_ROOT/include
   –Izcui.work/zebu.work
g++ -c testbench.cc –I$ZEBU_ROOT/include –Izcui.work/zebu.work
```

- Correct linking path must be given to linker:

```
g++ -o testbench driver.o testbench.o –L$ZEBU_ROOT/lib -lZebu
```

C/C++ co-simulation Lab 1

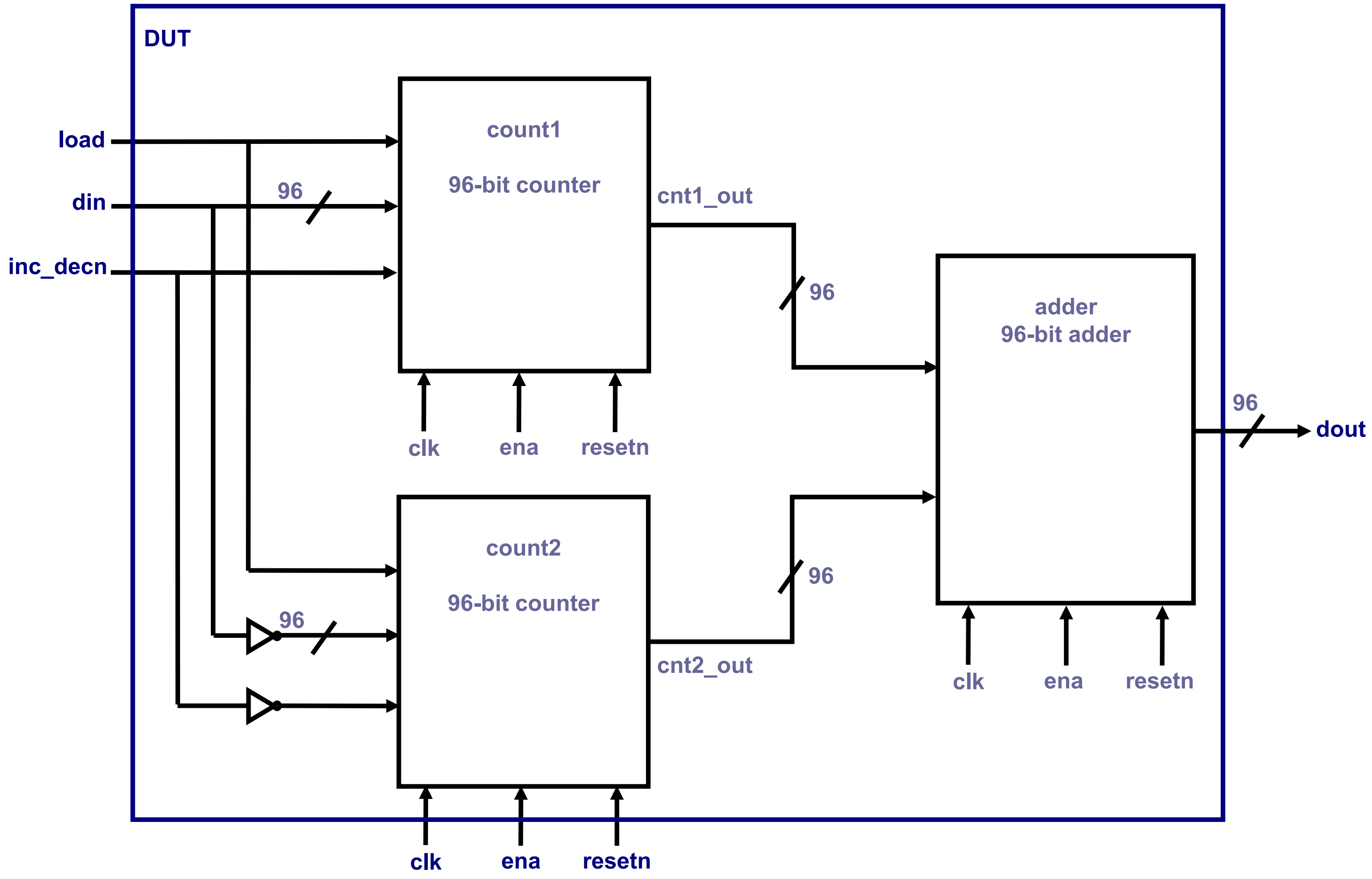
- In this lab you will learn how to compile and emulate a design for C++ co-simulation
- First we will compile the design
- Then we will write the C++ testbench using a template
- Finally we will emulate the design

C/C++ co-simulation Lab 1

```
Trainee/lab1
|-- C++_cosim
|    |-- zebu.run
|    |    `-- Makefile
|    `-- zebu.src
|-- dut
|    |-- probe
|    `-- verilog
|         |-- adder.v
|         |-- counter.v
|         `-- dut.v
`-- testbench
     `-- c++
          |-- testbench1.cc
          |-- testbench2.cc
          |-- testbench3.cc
          `-- testbench4.cc
```

C/C++ co-simulation Lab 1

The DUT



C/C++ co-simulation Lab 1

- Prepare the ZeBu compilation
  - Go into the `C++_cosim` directory
  - Open `zCui`
  - Add `adder.v`, `counter.v` and `dut.v` to the RTL group, define the top module name of the group and select the synthesizer
  - Define the top module name of the design
  - Prepare a probe file in `../dut/probe/probe.tcl`
    - Make sure we declare `dut.count2.cnt[95:0]` as static probe as it will be used in the testbench
    - We also want to see signals `dut.adder.inA` and `dut.adder.inB` as flexible probes
  - Add the probe file to the project

C/C++ co-simulation Lab 1

- Set the target hardware configuration file according to your ZeBu system
- Add the clock in “Clock Declaration” panel
- Save the project
- Generate and modify the DVE file. Static probes need to be manually added to the DVE file but we can generate a template:
  - Left-click on the “Environment” property
  - Click on “Auto Generated”
  - Set “Cycle-based: C/C++ Cosimulation”
  - Click on “Generate Now…”
  - If prompted to launch synthesis, click on “Proceed”
  - When the DVE file browser opens, create the DVE file in `zebu.src` directory, name it `lab.dve`
  - (synthesis and DVE generator run)
  - Click on “Edit…”. This open sa text editor on the DVE file
  - Add the static probe in the `output_bin` section of the DVE file Add `dut.count2.cnt[95:0]` as static probe

HDL co-simulation
Lab 1

  - In the “Environment” panel, click on the “Wrapper” tab and set “Wrapper Generation for the Testbench” to “C++”
  - Set the job scheduling preferences according to your network settings
- Save the project
- Launch the compilation

C/C++ co-simulation Lab 1

- **Prepare the testbench from template**
  - Go into the `zebu.run` directory, it contains a `Makefile` to compile the testbench
  - Edit the `../../testbench/c++/testbench1.cc` file. This is a testbench template which already includes some code as well as some comments. Follow the instructions found in the comments
- **When ready, compile the testbench:**

```
make testbench1
```

- **Then run the emulation by running the testbench binary:**

```
./testbench1
```

Agenda

- Overview
- Compilation
- Writing the testbench
- Debug

Debug

- When debugging a design in C/C++ co-simulation mode you can use the following debug tools:
  - A software debugger such as `gdb`
    - To debug the testbench
  - `zRun`
    - To debug the design
  - The ZeBu C/C++ API to access and control the design with the following features:
    - Static probes
    - Dynamic probes
    - Flexible probes
    - Simulated combinational signals
    - Triggers
    - Trace memory
    - Dump waveforms

Debug
Using `zRun`

- To launch zRun and control the testbench execution type:
  `zRun -testbench "<testbench_run_command>"`
  such as in:
  ```
      zRun -testbench ./tb
  ```
  or in:
  ```
      zRun -testbench "make run"
  ```

Debug
Static probes

- The wrapper file preserves the original hierarchical signal names except that dot characters “.” are replaced by underscore characters “_”:
  - E.g.: `top.u1.u2.signal`
    is renamed to:
    `top_u1_u2_signal`
- Static probes are found in the *`<driver>`*`_drv` structure defined in *`<driver>`*`.hh` wrapper file like any other interface signal
- Static probes are accessible like any other interface signal

Debug
Dynamic probes

- To access dynamic probes proceed as follows:
  1) Create a signal object
     ```
     Signal &net = *zebu->getSignal("<signal name>");
     ```
  2) Access signal `net` as a regular signal

Debug
Dumping waveforms

Action	Interface signals(*)	Dynamic probes
Define the file name	`driver->dumpfile("filename.ext")`	`zebu->dumpfile("filename.ext")`
Define the signals to be added to the waveform file	`driver->dumpvars(signal)` `driver->dumpvars()`	`zebu->dumpvars(signal)` `zebu->dumpvars()`
Turn on the dump	`driver->dumpon()`	`zebu->dumpon()`
Turn off the dump	`driver->dumpoff()`	`zebu->dumpoff()`
Close the dump file	`driver->dumpclosefile()`	`zebu->dumpclosefile()`

(*) Interface signals include static probes

`.ext` waveform file name extension can be any of `.vcd`, `.fsdb`, `.vpd` and `.bin`

Empty parentheses on `dumpvars` method means ALL interface signals (2nd column) or ALL selected dynamic probes (3rd column)

Debug
Flexible probes

- To access flexible probes use with the following steps:
  1) Select the sampling clock
     ```
     FlexibleLocalProbeFile::SelectSamplingClock(zebu, "clock",
            FlexibleLocalProbeFile::POSEDGE);
     ```
  2) Declare the flexible probes waveform file
     ```
     FlexibleLocalProbeFile file;
     ```
  3) Initialize the file
     ```
     file.initialize(zebu);
     ```
  4) Add flexible probes groups
     ```
     file.add("groupname");
     ```
  5) Start dumping to disk
     ```
     file.dumpFile("wave.ztdb");
     ```
  6) Close the file
     ```
     file.closeFile();
     ```

Debug
Flexible probes

- To convert a “ztdb” waveform file to “vcd” or “fsdb” standard formats:

```
ztdb2vcd -i <filename.ztdb> -o <filename.vcd>
ztdb2fsdb -i <filename.ztdb> -o <filename.fsdb>
```

Debug
Simulated combinational signals

- From a C/C++ point of view, simulated combinational signals can be accessed like dynamic probes

Debug
Lab 1

- In this lab, we will add a dynamic probe and dump a waveform from a C++ testbench

HDL co-simulation
Lab 1

- Add a dynamic probe:
  - Type:
    `zSelectProbes -p ../zcui.work/zebu.work`
  - Select the `dut.count1.cnt[95:0]` signal
  - Save the database
  - Click on the “Generate”>”C++” menu item
  - Click on the “Generate”>”Generate” menu item
    - This will generate a new wrapper, have a look at it
  - Exit `zSelectProbes`

C/C++ co-simulation
Lab 1

- Prepare the testbench from a template
  - Go into the `zebu.run` directory, it contains a `Makefile` to compile the testbench
  - Edit the `../../testbench/c++/testbench2.cc` file. This is a testbench template which already includes some code as well as some comments. Follow the instructions found in the comments
- When ready, compile the testbench:

```
make testbench2
```

- Then run the emulation by running the testbench binary:

```
./testbench2
```

C/C++ co-simulation Lab 1

- View the waveform files `interface_wave.vcd` and `internal_wave.vcd` in a waveform viewer

Debug
Trace memory

- **Trace memory must have been instantiated in the DVE file at compilation time**

Instance name given by the user

```
SRAM_TRACE sram (
          .output_bin({
                    dut.count1.cnt[95:0]
                    }));
```

Debug
Trace memory

- Access Trace SRAM with the following steps:
  (include the `sram.hh` header file)

1) Trace SRAM initialization
   `sram = Init_sram(zebu);`
2) Driver connection
   `sram->connect();`
3) Set pre trigger ratio (%)
   `sram->setPreTriggerRatio();`
4) Set dump file name and type
   `sram->dumpfile("filename.ext");`
5) Set dump signal
   `sram->dumpvars();`
   `sram->dumpvars(signal);`
6) Start dump and set dump clock
   `sram->dumpon("clk","posedge");`
7) Stop dump
   `sram->dumpoff();`
8) Write to dump file
   `sram->storeToFile();`
9) Disconnect SRAM
   `sram->disconnect();`

Debug Triggers

- Triggers are declared in the DVE file during compilation
  - Dynamic triggers can be programmed at runtime
- The trigger name defined in the DVE file is used in the C/C++ code

Debug Triggers

- Control trigger with the following steps:
  - Trigger initialization
    ```
    Trigger &trig = *zebu->getTrigger("trigger name");
    ```
  - Dynamic trigger definition
    ```
    trig = "signal == value";
    ```
  - Run until trigger
    ```
    driver->wait(trig);
    ```

Debug
Logic analyzer

- Control trigger with the following steps:

1) Logic analyzer initialization
```
LogicAnalyzer *la = zebu->getLogicAnalyzer();
```
2) Define logic analyzer trigger
```
la->traceOnTrigger(&trig);
la->stopOnTrigger(&trig);
```
3) Start logic analyzer and define clock
```
la->start("clock");
```
4) Stop logic analyzer
```
la->stop();
```

Debug
Lab 1

- In this lab you will add triggers and the trace memory and control them from the testbench
- Modify the DVE file to add:
  - The trace memory which must capture the `dut.count1.cnt[95:0]` signal
  - A static trigger named “`stop_trig`” which check the following condition: `dut.count2.cnt[31:0]==32'h45410000`
  - A dynamic trigger named “`tgram`” which check the following signal: `dut.count1.cnt[95:0]`
- Launch the compilation

Debug
Lab 1

- Prepare the testbench from template
  - Go into the `zebu.run` directory, it contains a `Makefile` to compile the testbench
  - Edit the `../../testbench/c++/testbench3.cc` file. This is a testbench template which already includes some code as well as some comments. Follow the instructions found in the comments
    - The condition you want to check with the dynamic trigger is: `dut.count1.cnt[31:0] == 32'hbabe0100`
- When ready, compile the testbench:

```
make testbench3
```

- Then run the emulation by running the testbench binary:

```
./testbench3
```

Debug
Lab 1

- View the `sramtrace.vcd` file in a waveform viewer

Debug
Lab 1

- We will now add flexible probes control to the testbench
- Prepare the testbench from a template
  - Go into the `zebu.run` directory, it contains a `Makefile` to compile the testbench
  - Edit the `../../testbench/c++/testbench4.cc` file. This is a testbench template which already includes some code as well as some comments. Follow the instructions found in the comments
- When ready, compile the testbench:
  `make testbench4`
- Then run the emulation by running the testbench binary:
  `./testbench4`

Debug
Lab 1

- Convert the flexible waveform file to VCD format:
  - Type:

```
ztdb2vcd -i flex_wave.ztdb -z ../zcui.work/zebu.work
```

- View the `flex_wave.ztdb.vcd` file in a waveform viewer

Debug
Accessing ZeBu memories

- To access a ZeBu memory during a C/C++ co-simulation use the following ZeBu methods:
  (include the `TOP_<top>.hh` header file)
- Initializes memory access
  `ZEBU_<top>_Init(zebu);`
- Dump memory to file
  `<top>.<memory> ->storeTo("filename");`
- Load memory from a file
  `<top>.<memory> ->loadFrom("filename");`