Designing zcei transactors

ZeBu-Server Training

THE *FASTEST* VERIFICATION

Agenda

- Introduction
- zcei macros
- C/C++ API
- Linking hardware and software
- Using zTIDE
- Transactor debug
- Tips and hints

Introduction

- ZeBu includes
  - the hardware macros necessary for communication and clock control
  - an API to send and receive messages on the software side
- zTIDE provides a environment for developing transactors using a simulator

Introduction
Transactor design flow

- Transactor development requires:
  - A software part using the ZeBu C/C++ transaction-level API
  - A hardware part between the ZeBu system hardware transaction-level interface and the DUT
- Choosing the architecture of a transactor is very important
  - Some processing is perfectly handled by hardware but not by software, and vice-versa
  - For example:
    - Bit manipulation, like re-ordering, is better handled by hardware than software
  - Also, some processing requires substantial hardware resources and can increase considerably size of transactor
    - Multiplication and (in general) arithmetic operations are better handled by software

Introduction
Transactor design flow

- Hardware transactor development can take from half a day to few months
  - Less than one day
    - video ram, random generators, simple monitors, …
  - Less than one week
    - JTAG, RS232, complex generators and monitors, …
  - More than a month
    - PCI, USB, Ethernet, DRAM, …

Introduction
Recommended transactor design flow

1. Use zTIDE to design, simulate and verify transactor code with a simple test case
2. Synthesize transactor RTL and clean up RTL
   - Optionally simulate with zTIDE and gate-level netlist
3. Emulate synthesized transactor on ZeBu with test case
4. Emulate synthesized transactor with actual DUT

Agenda

- Introduction
- zcei macros
- C/C++ API
- Linking hardware and software
- Using zTIDE
- Transactor debug
- Tips and hints

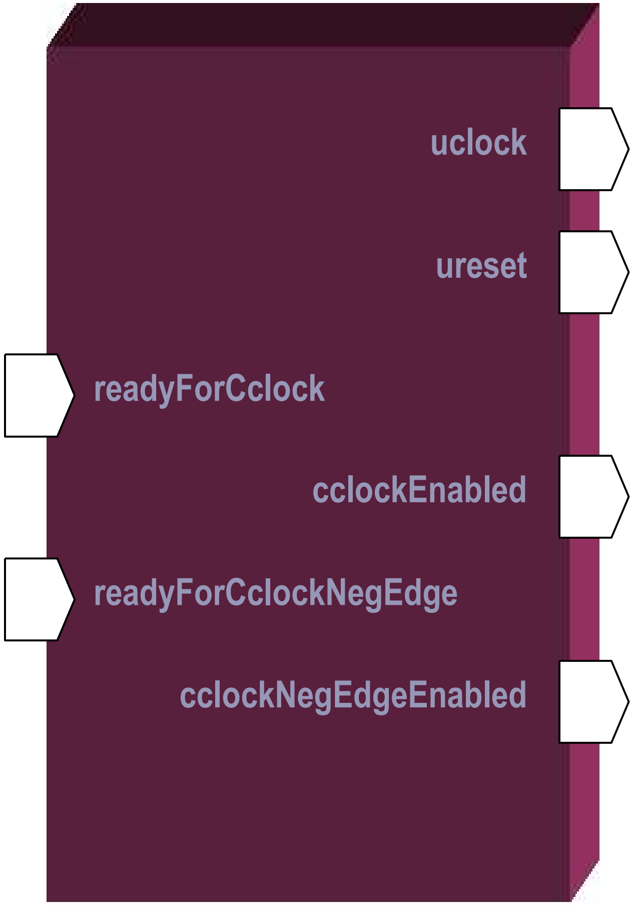
ZeBu hardware macros

- Three macros are available to link transactors to the software:
  - `zceiClockControl`
    Provides a free-running clock to the transactor and the means to control the DUT clocks
  - `zceiMessageInPort`
    Used to receive messages from the software
  - `zceiMessageOutPort`
    Used to send messages to the software
- All these macros are controlled and synchronous with an uncontrolled (free-running) clock: `uclock`

ZeBu hardware macros `zceiClockControl`

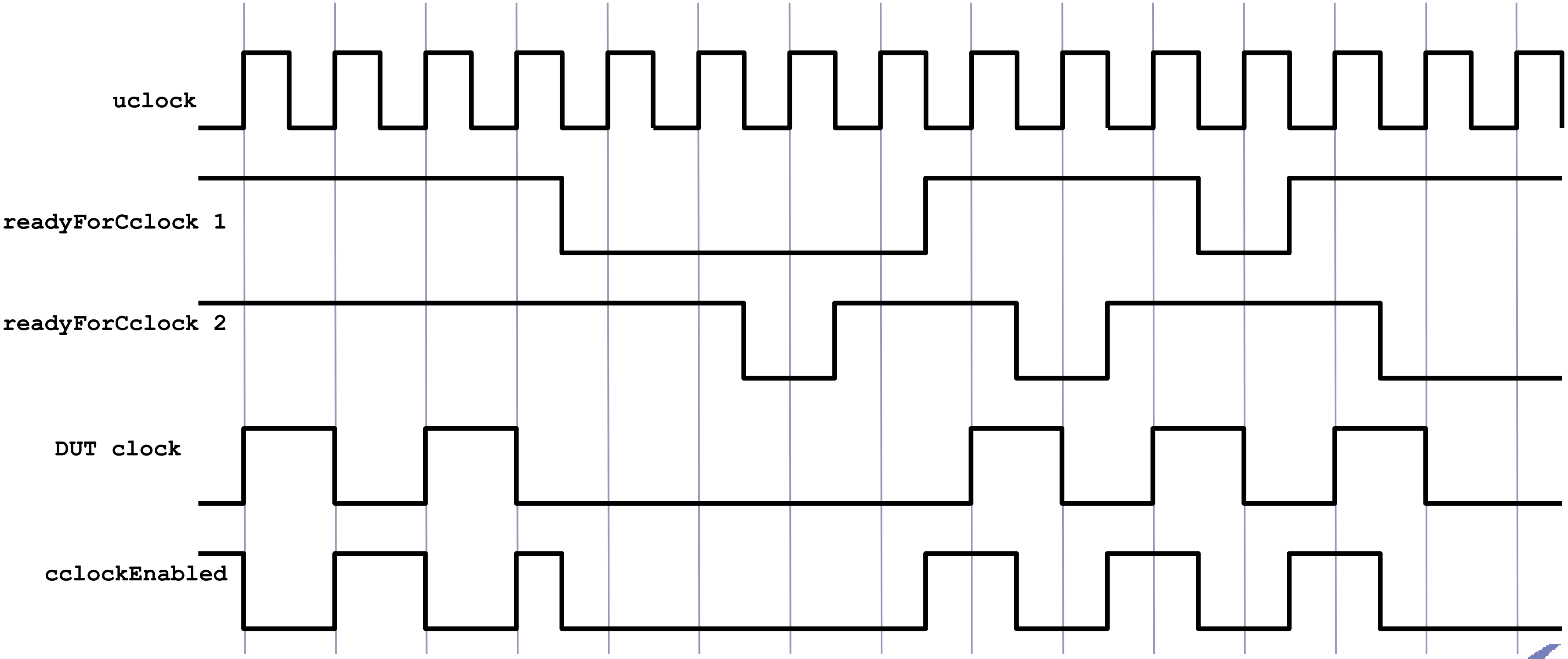
- **Outputs:**
  - `uclock`
    - uncontrolled clock
  - `ureset`
    - driver reset. Active high
  - `cclockEnabled`
    - set by ZeBu when a posedge on the DUT clock occurs with the next `uclock` posedge
  - `cclockNegEdgeEnabled`
    - set by ZeBu when a negedge on the DUT clock happens with the next `uclock` posedge
- **Inputs:**
  - `readyForCclock`
    - allows next posedge on the DUT clock when set
  - `readyForCclockNegEdge`
    - allows next negedge on the DUT clock when set

`zceiClockControl`



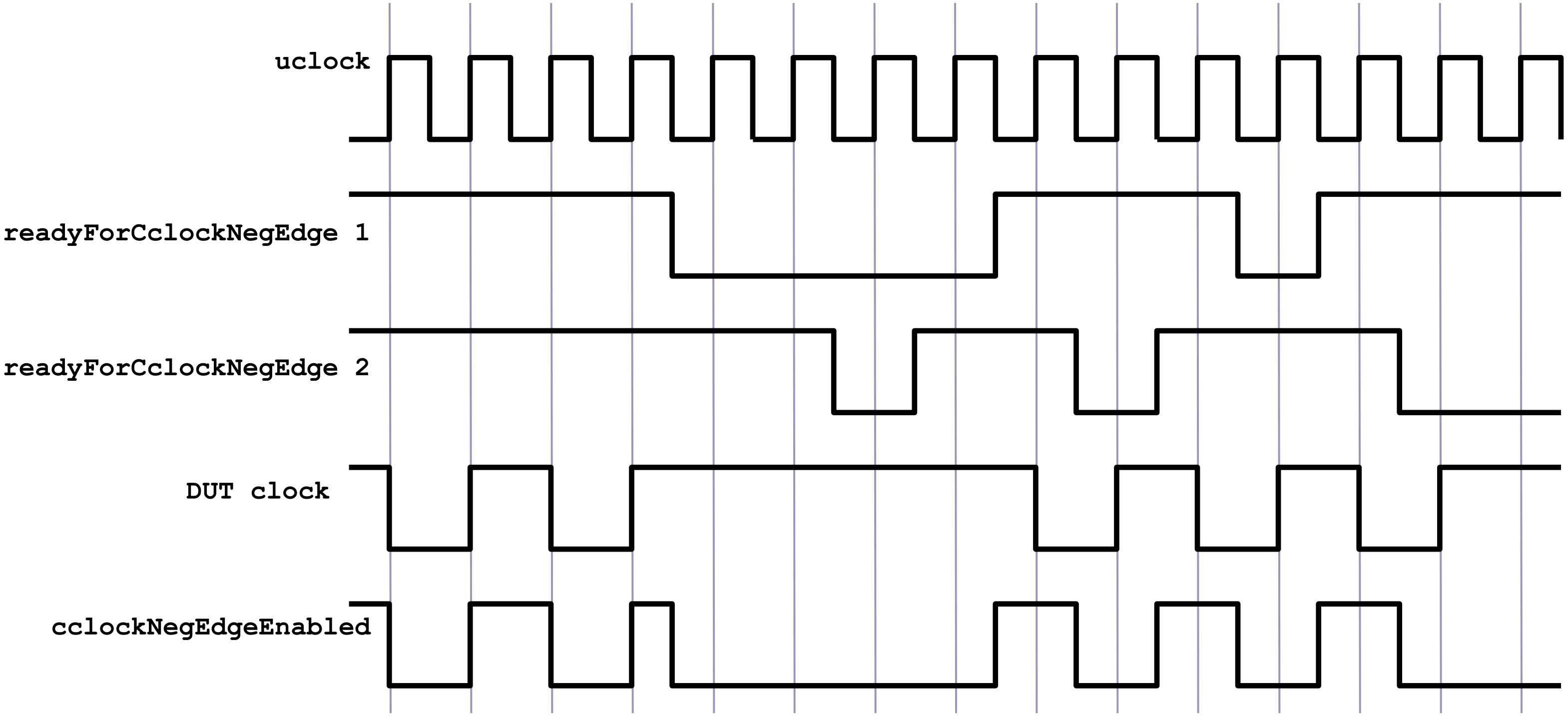
ZeBu hardware macros
`zceiClockControl` behavior

- A rising edge on the DUT clock occurs:
  - On a rising edge of `uclock` when `cclockEnabled` is asserted by ZeBu
- `cclockEnabled` can only be asserted by ZeBu when all the `readyForCclock` signals on all transactors controlling the same clock are asserted



ZeBu hardware macros
`zceiClockControl` behavior

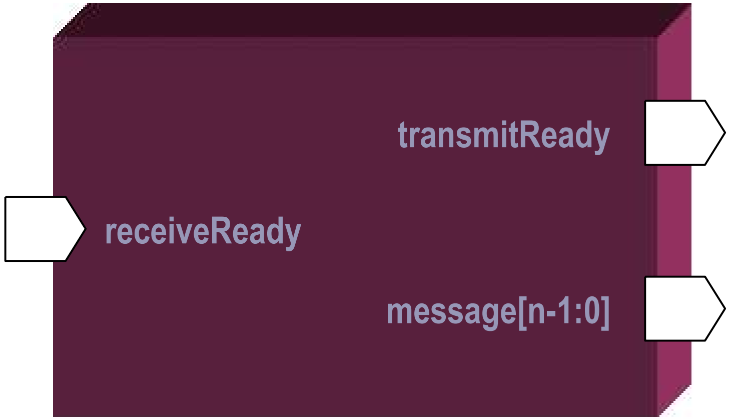
- A falling edge on the DUT clock occurs:
  - On a rising edge of `uclock` when `cclockNegEdgeEnabled` is asserted
- `cclockNegEdgeEnabled` can only be asserted when all the `readyForCclockNegEdge` signals on all transactors controlling the same clock are asserted



ZeBu hardware macros zceiMessageInPort

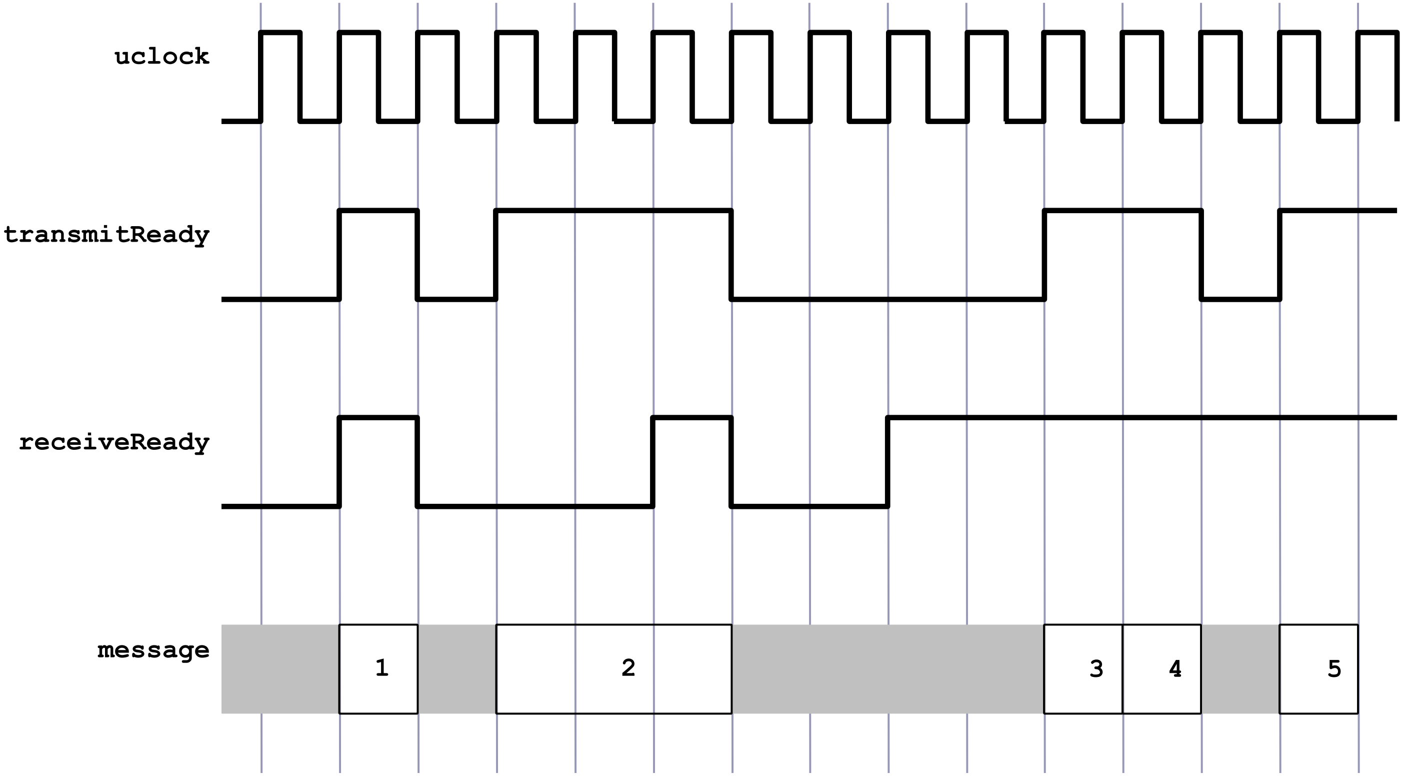
- Outputs:
  - message
    - Message from software. Width range from 1 to 8,160 bits
  - transmitReady
    - Set by ZeBu when a message is available
- Inputs:
  - receiveReady
    - Set by the transactor to acknowledge the current message
- Parameter:
  - Pwidth

zceiMessageInPort



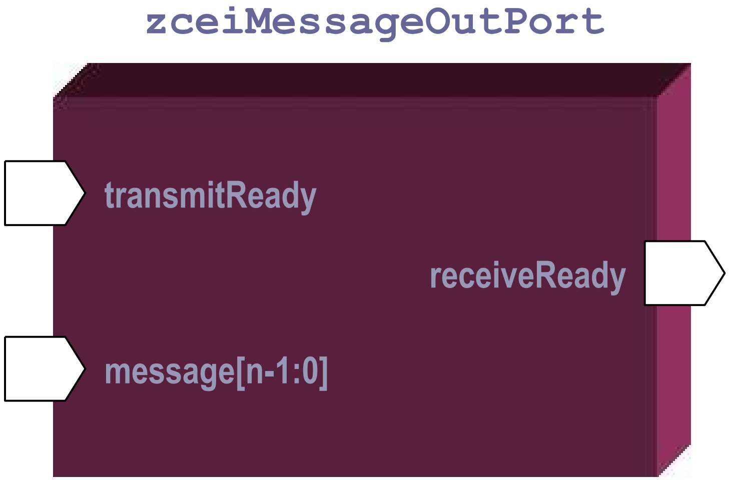
ZeBu hardware macros `zceiMessageInPort` behavior

- The message
  - is available on a `uclock` rising edge when `transmitReady` is asserted by ZeBu
  - remains available as long as `receiveReady` is not asserted by the transactor
- When both `transmitReady` and `receiveReady` are asserted, the message moves



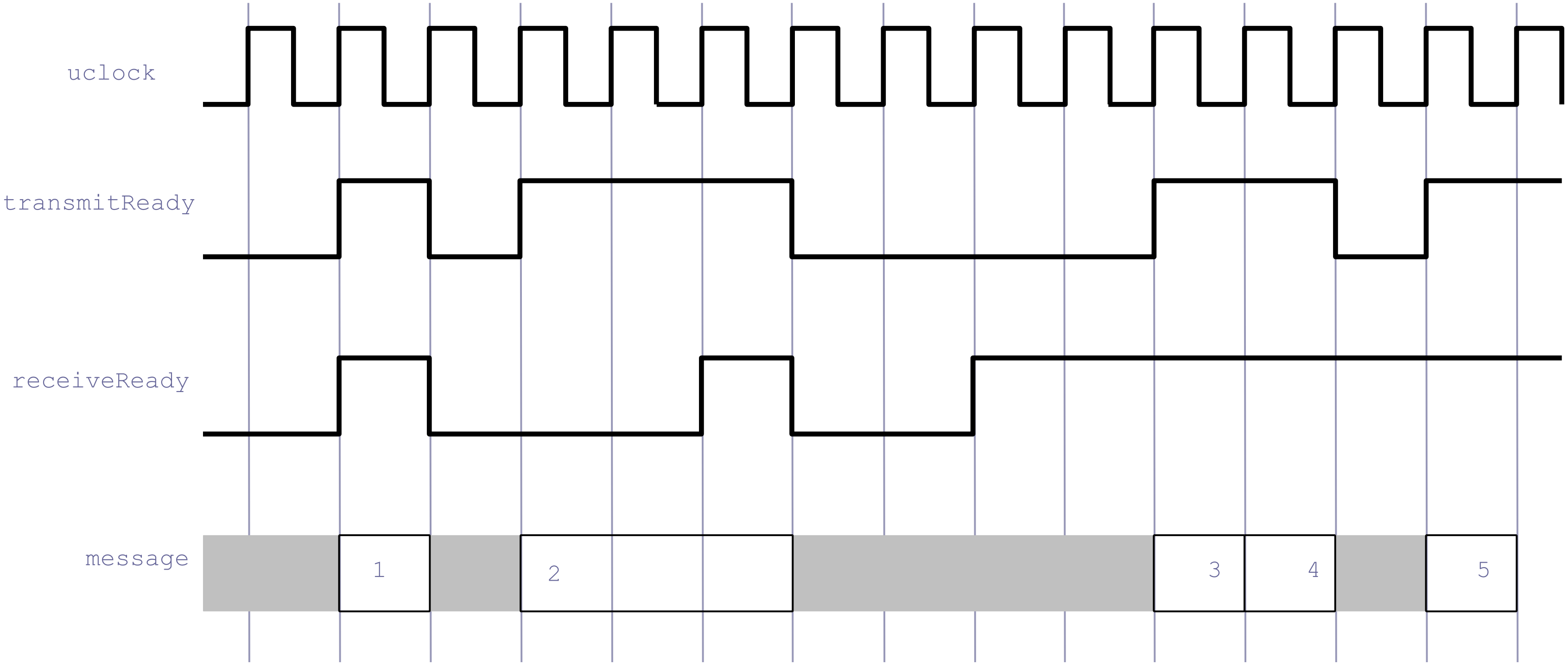
ZeBu hardware macros zceiMessageOutPort

- Outputs
  - `receiveReady`
    - Set by ZeBu whenever the port is ready to receive messages
- Inputs
  - `transmitReady`
    - Must be set by the transactor to send the message
  - `message`
    - Message to software. Width range from 1 to 8,096 bits
- Parameter:
  - `Pwidth`



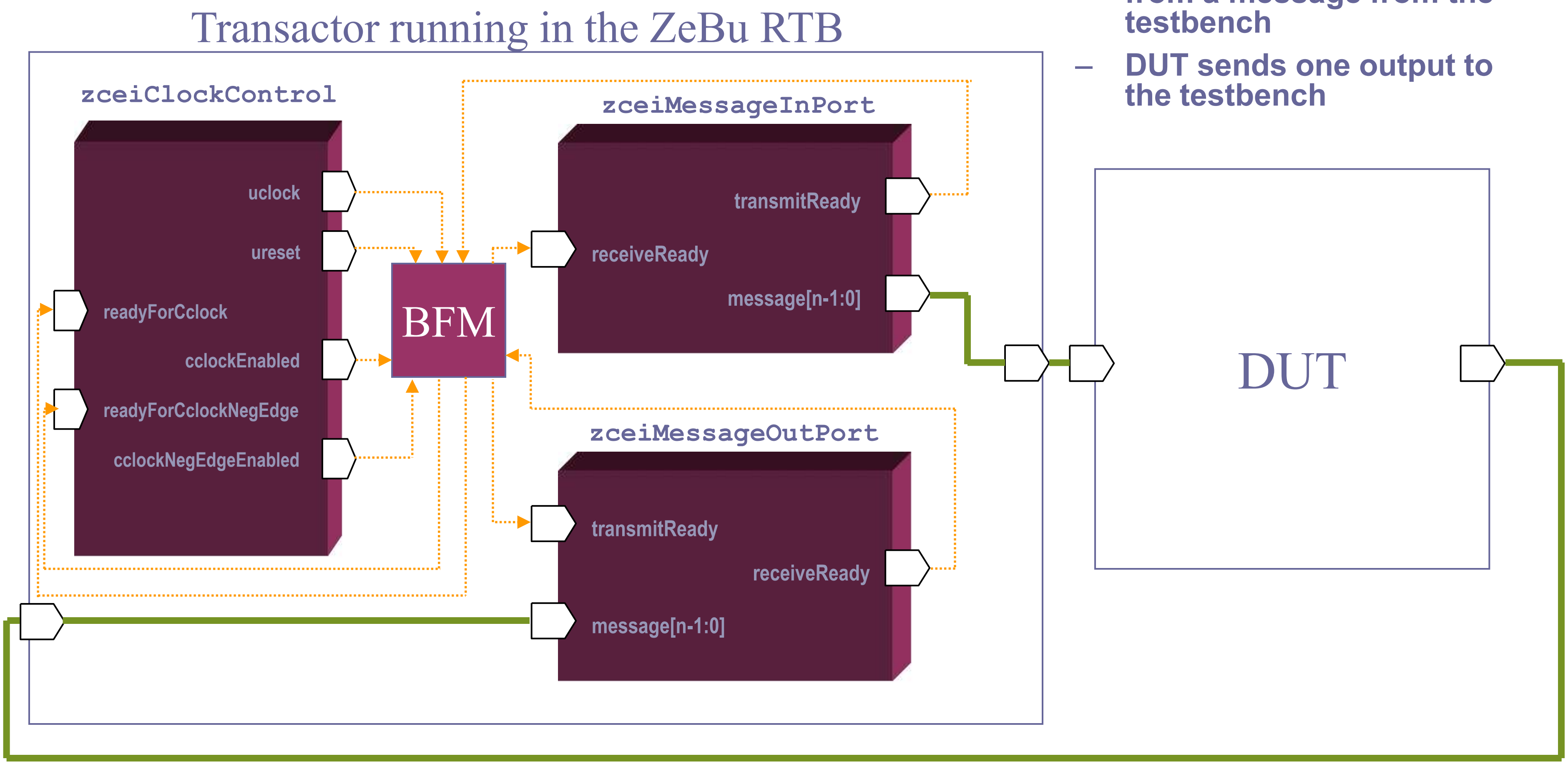
ZeBu hardware macros `zceiMessageOutPort` behavior

- The transactor makes the message available and asserts the `transmitReady` signal
- The message must remain available and `transmitReady` remain asserted until `receiveReady` is asserted by ZeBu on the rising edge of `uclock`
- The message is taken by ZeBu when both `transmitReady` and `receiveReady` are asserted on a rising edge of `uclock`



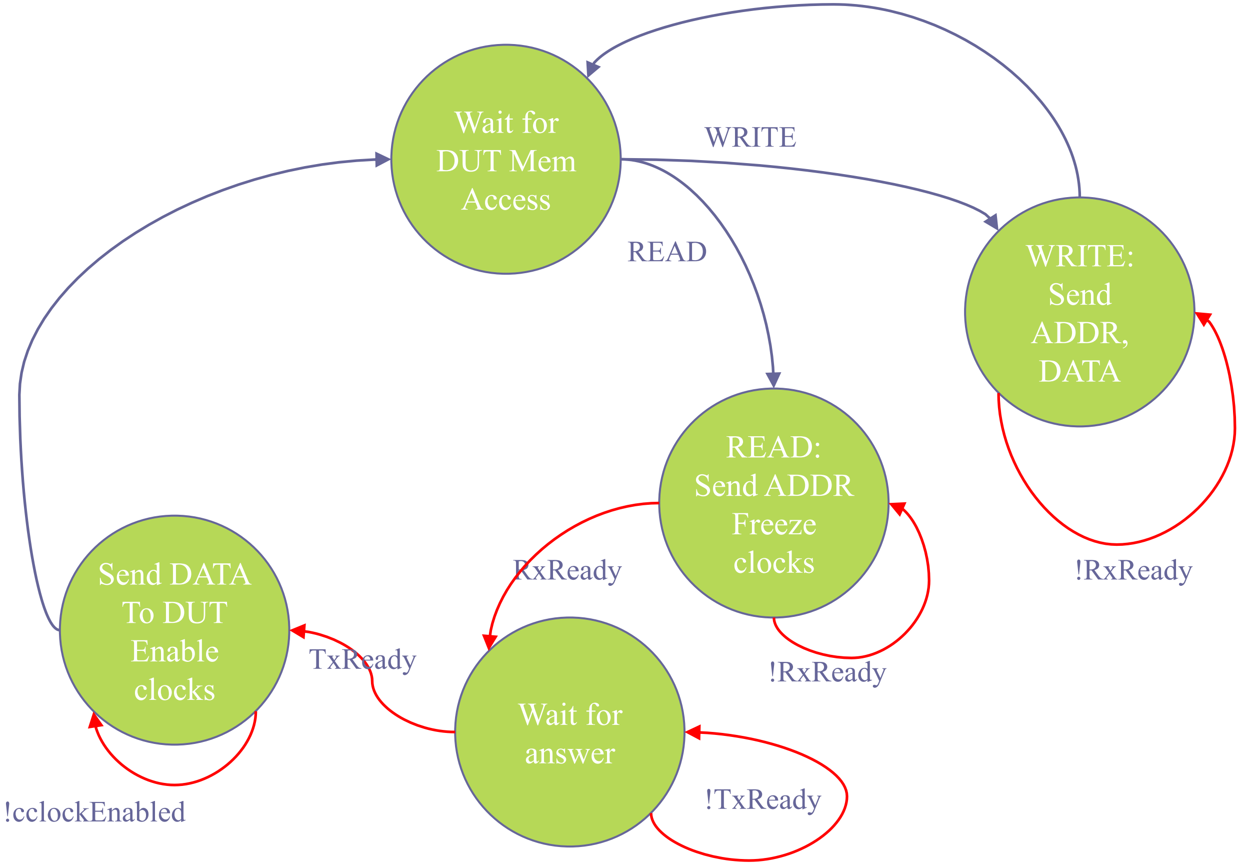
ZeBu Hardware Macros
Simple example

- **Example:**
  - DUT receives one input from a message from the testbench
  - DUT sends one output to the testbench



ZeBu Hardware Macros
Example: Detailed state machine

- Detailed state machine that implements the memory example:
  - References `TxReady/RxReady`, as well as `cClockEnabled`



Agenda

- Introduction
- zcei macros
- C/C++ API
- Linking hardware and software
- Using zTIDE
- Transactor debug
- Tips and hints

ZeBu C++ API
The `TxPort` class

- Allows the software to send messages to the hardware
- One `TxPort` object is connected to each `zceiMessageInPort` in the hardware part of the transactor
- Public Member functions:
  - `connect()`
  - `disconnect()`
  - `isPossibleToSend()`
  - `sendMessage()`
  - `size()`
  - `message()`
  - `write()`

ZeBu C++ API
The `TxPort` class : public methods

- `TxPort( const char *name, unsigned int size=0 )`
  - `TxPort` constructor
  - `name` is the instance name of the port in the netlist of the transactor
  - `size` is the number of 32-bit words used for messages. If given, a check will be done during the connection
- `unsigned int connect( Board *board, const char *DriverName )`
  - Connects the Port with ZeBu
  - `board` is the handler to the ZeBu board
  - `DriverName` is the instance name of the transactor in the DVE file
- `void disconnect()`
  - Disconnects the `TxPort` from the ZeBu system
- `int size()`
  - Returns the message size expressed in number of 32-bit words

ZeBu C++ API
The `TxPort` class : public methods

- `unsigned int *message()`
  - Returns a message handle (pointer to message buffer)
- `void write( unsigned int index, unsigned int value )`
  - Writes a word in a send message
- `bool isPossibleToSend()`
  - Returns true if it is possible to send a message to the port
- `void sendMessage()`
  - Sends the message contained in the buffer to the hardware

ZeBu C++ API
The `RxPort` class

- Allows the software to receive messages from the hardware
- One `RxPort` object is connected to each `zceiMessageOutPort` in the hardware part of the transactor
- Public Member functions:
  - `connect()`
  - `disconnect()`
  - `isPossibleToReceive()`
  - `receiveMessage()`
  - `size()`
  - `message()`
  - `read()`

ZeBu C++ API
The RxPort class : public methods

- `RxPort( const char *name, unsigned int size=0 )`
  - `RxPort` constructor
  - `name` is the instance name of the port in the netlist of the transactor
  - `size` is the number of 32-bit words used for messages.
    If given, a check will be done during the connection
- `unsigned int connect( Board *board, const char *DriverName )`
  - Connects the port to ZeBu
  - `board` is the handle to the ZeBu board
  - `DriverName` is the instance name of the transactor in the DVE file
- `void disconnect()`
  - Disconnects the `RxPort` from ZeBu
- `int size()`
  - Returns the message size expressed in number of 32-bit words

ZeBu C++ API
The `RxPort` class : public methods

- `unsigned int *message()`
  - Get a message handle (pointer to message buffer)
- `bool isPossibleToReceive()`
  - Returns true if a message is available for the port
- `unsigned int *receiveMessage()`
  - Updates buffer and returns a pointer to it
- `unsigned int read( unsigned int index )`
  - Get a word in a read message

ZeBu C++ API
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 driver.cc –I$ZEBU_ROOT/include
g++ -c testbench.cc –I$ZEBU_ROOT/include
```

- Correct linking path must be given to linker:

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

ZeBu C++ API
Code example: Declaration

- **In order to use the ZeBu C++ API, the following header files must be included:**

```
#include <libZebu.hh>
    [...]
using namespace ZEBU;
using namespace std;
```

ZeBu C++ API
Code example : Initialization

- In the testbench, connect to the ZeBu system

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

- In the transactor initialization method, create the ports to communicate with the transactor

```
    Port *tx = new TxPort("tx",size_tx);
    Port *rx = new RxPort("rx",size_rx);
```

- Then connect the ports to ZeBu

```
    tx->connect(zebu,"xtor_name");
    rx->connect(zebu,"xtor_name");
```

  Where `zebu` is the board handler obtained on ZeBu opening and `"xtor_name"` is the instance name specified in the DVE file

ZeBu C++ API
Code example : Initialization

- Get pointer to message buffer

```
unsigned int *tx_msg = tx->message();
unsigned int *rx_msg = rx->message();
```

- From the testbench, initialize the ZeBu system

```
zebu->init();
```

- Check the initialization

```
if ( ! zebu->check() ) {
  // Wrong initialization of the system
  // testbench may exit
}
```

ZeBu C++ API
Code example : Sending a message

- Initialize the message buffer
  - There are two methods to initialize the message to be sent

1. Direct manipulation of the message buffer:
   Get a pointer to the message buffer with the `message()` method . This pointer will not change during runtime, so it is necessary to call this method only once

```
      unsigned int *tx_msg = tx->message();
      tx_msg[0] = 0xbabeface;
```

2. Using the `write()` method:

```
      tx->write(0,0xfacebabe);
```

- Send the message
  - Check if it is possible to send the message

```
while(!tx->isPossibleToSend());
```

  - Send it

```
tx->sendMessage();
```

ZeBu C++ API
Code example : Receiving a message

- Check if it is possible to receive a message

```
while(!rx->isPossibleToReceive());
```

- Receive it

```
rx->receiveMessage();
```

- There are two ways to access the data of a message:

1. Direct access to the message buffer.
   Get a pointer to the message buffer with the method `message()`. This pointer will not change during runtime, so it is necessary to call this method only once

   ```
   unsigned int *rx_msg = rx->message();
   a = rx_msg[0];
   ```

2. Using the `read()` method of the ports

   ```
   a = rx->read(0);
   ```

ZeBu C++ API
Synchronizing multiple ports: using callbacks

- **Easy way to handle messages coming from many ports concurrently**
  - **Need to keep “context” structure so the callback knows its state**
  - **No need to call `isPossibleToSend()`/`isPossibleToReceive()` since the callback is only called once they are true**

```
#include <libZebu.hh>
using namespace ZEBU;
...
void callback(Port *tx) {
        tx->sendMessage();
}

Port *tx = new TxPort("portname",size);
tx->connect(zebu,"drivername");
tx->registerCB((void (*)(void *)) &callback,(void *) tx);
...
while (1) {
        zebu->serviceLoop();
        };
```

ZeBu C++ API

Synchronizing multiple ports: using multi-threading

- Allows true parallelism to handle multiple ports
- Rely on `pthread` C library
- A little more complex than callbacks but potentially leads to better performance
- Use `libZebuThreadSafe.so` library

ZeBu C++ API
Synchronizing multiple ports: using multi-threading

```
#include <libZebu.hh>
using namespace ZEBU;
#include <pthread.h>
...
void* th_func(Port *tx) {
          ...
          while (test) {
                    while (!tx->isPossibleToSend());
                    tx->sendMessage();
          }
          ...
          zebu->closeThread();
          pthread_exit(NULL);
}

Port *tx = new TxPort("portname",size);
tx->connect(zebu,"drivername");
zebu->closeThread();
pthread_t th;
pthread_attr_t th_attribute;
pthread_attr_init(&th_attribute);
pthread_create(&th,&th_attribute, (void *(*)(void *)) th_func, tx);
...
pthread_join(th,NULL);
```

Agenda

- Introduction
- zcei macros
- C/C++ API
- Linking hardware and software
- Using zTIDE
- Transactor debug
- Tips and hints

Linking the hardware and the software The `.install` file

- The `.install` file describes the transactor for the compilation tools
- It defines the location of the transactor netlist
- The syntax of the install file is:

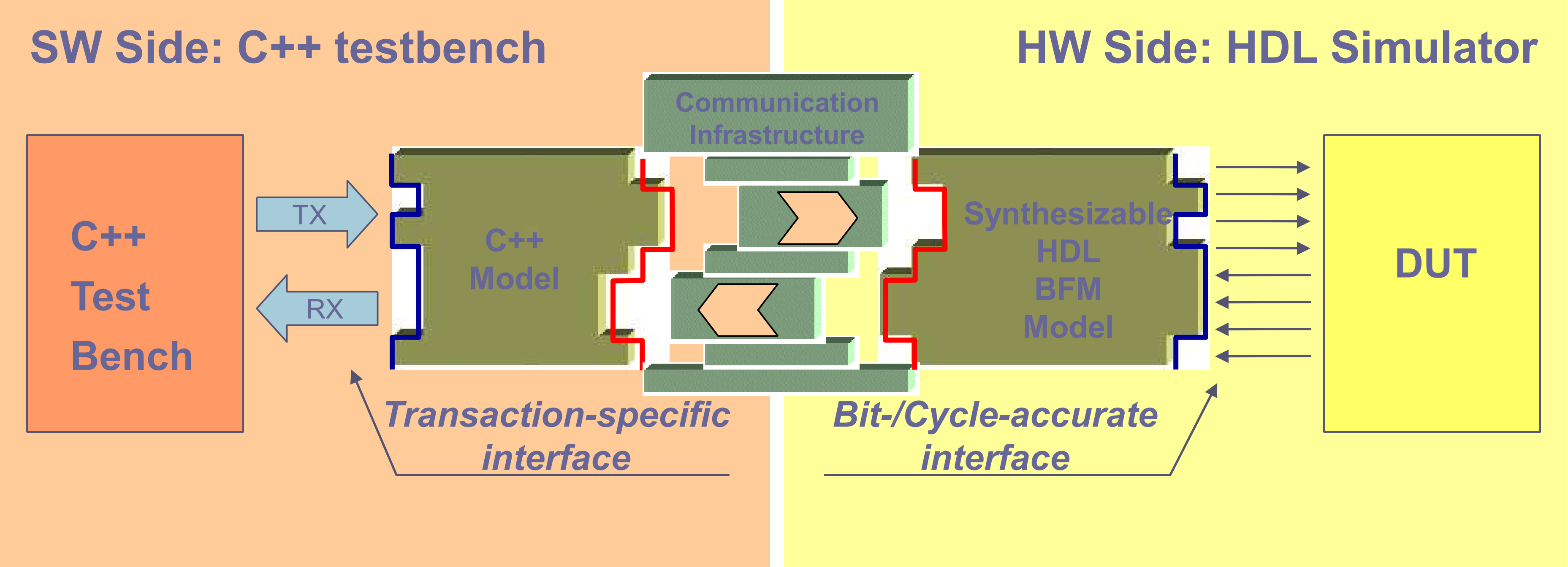
```
<transactor_name> {
       NETLIST = "<Netlist_path>/<Netlist_name>";
       }
```

Agenda

- Introduction
- zcei macros
- C/C++ API
- Linking hardware and software
- Using zTIDE
- Transactor debug
- Tips and hints

Using zTIDE Principle

zTIDE: ZeBu Transaction Interface Development Environment


SW Side: C++ testbench
HW Side: HDL Simulator
C++ Test Bench
TX
RX
C++ Model
Communication Infrastructure
Synthesizable HDL BFM Model
DUT
Transaction-specific interface
Bit-/Cycle-accurate interface

- zTIDE allows to simulate synthesizable transactors with DUT using ModelSim, NC-Sim or VCS

Using zTIDE
zTIDE basics

- Use the same C++ code for the software part of the transactor
  - Link C++ code with zTIDE `libZebu.so` instead of standard ZeBu C++ library
- Use the same HDL code for the hardware part of the transactor
  - Simulate RTL of transactor with actual Verilog/PLI models for `zceiMessageInPort`, `zceiMessageOutPort` and `zceiClockControl`
- Write a zTIDE wrapper to describe the connection between the DUT and the hardware part of the transactor
  - It instantiates and connects
    - the DUT
    - the hardware part of the transactor
    - one or more `zceiClockPort`
    - it play the same role as a DVE file

Using zTIDE
Differences between ZEBU and zTIDE

- Timing
  - zTIDE is fully synchronous
  - zTIDE runs the uncontrolled clock only when a call to the `libZebu.so` is made
    - zTIDE is deterministic. ZeBu is not!!!
- Easier transactor debug
  - Design/debug cycles are usually much faster with zTIDE than with ZeBu
  - zTIDE gives you full accessibility
  - ZeBu features not implemented in zTIDE
    - Some API functions do not work in zTIDE, for example reading and writing a register, loading a memory, …
    - zTIDE supports only one process

Using zTIDE
Setting Up zTIDE

- No need to recompile the testbench between zTIDE and ZeBu
  - Setup `LD_LIBRARY_PATH` with `$ZTIDE_ROOT/lib`
  - This can be done by sourcing the `$ZTIDE_ROOT/ztide_env.bash` file in your Linux shell
- On the simulator command line add:
  - For ModelSim:
    ```
    -pli libZebuSim.so
    ```
  - For NC-Sim:
    ```
    +loadpli1=libZebuSim.so:bootstrap.bootstrap
    ```
  - For VCS:
    ```
    -P $ZTIDE_ROOT/lib/pli.tab $ZTIDE_ROOT/lib/libZebuSim.so
    ```

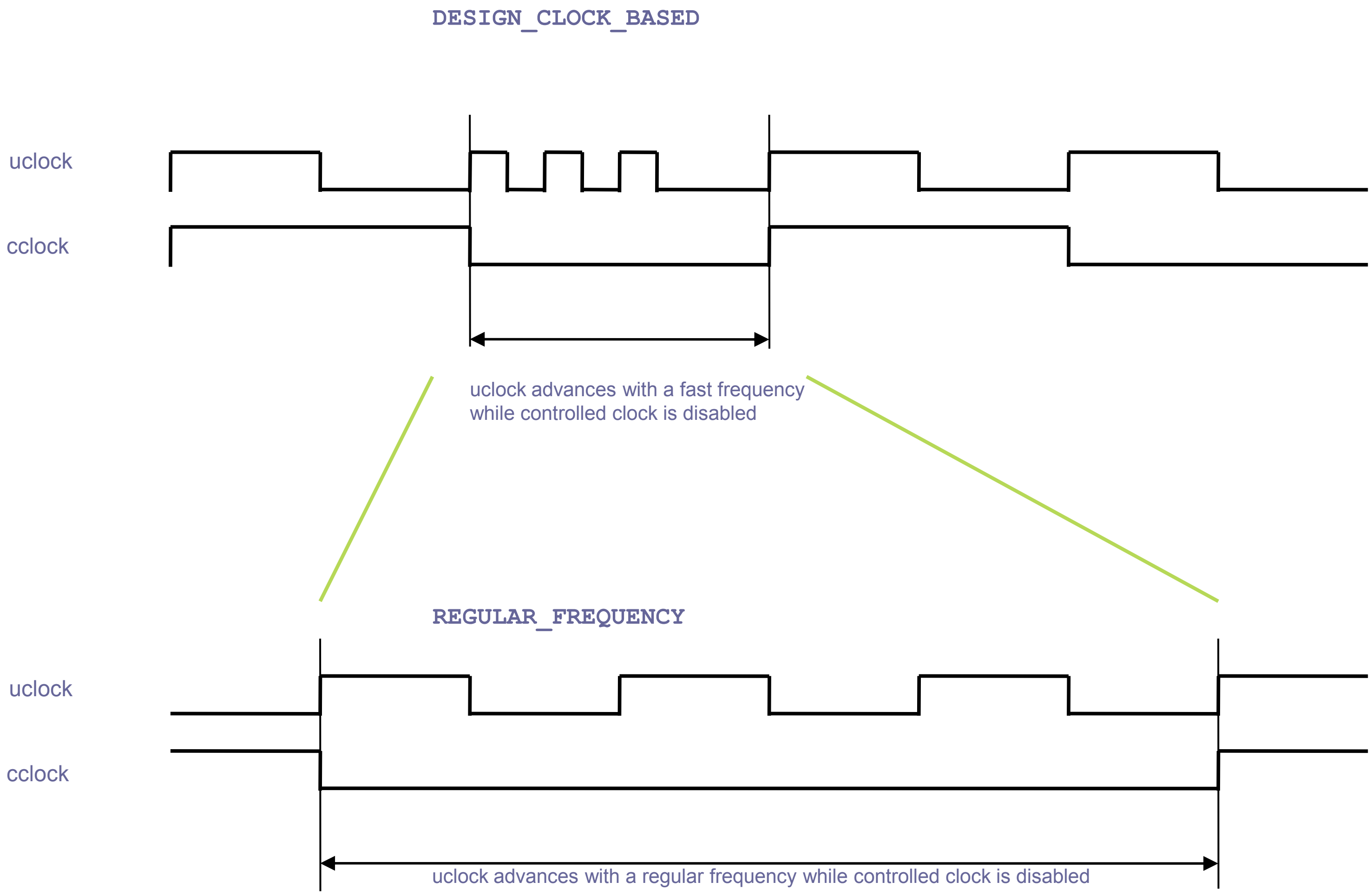
Using zTIDE
Controlling the behavior of zTIDE

- `libZebuSim.cfg`
  - Controls the simulator behavior, found in simulator directory
  - `DESIGN_CLOCK_BASED` vs. `REGULAR_FREQUENCY`
- `libZebu.cfg`
  - Associated with the software, found where you launch the testbench
  - `RUN_LENGTH`
    - Simulates software much faster than hardware or vice-versa

Using zTIDE
Clock control

- zTIDE runs the `uclock` only when a call to the `libZebu.so` is done
- On a ZeBu system, many factors can stop the clock: transactors, STB, ZRM, …
  - The DUT/driver clock ratio is not always 2
- The different zTIDE modes for the driver clock
  - Realistic (`REGULAR_FREQUENCY`).
    - The `uclock` will look regular like it is in reality
    - Useful when debugging a transactor
  - DUT side view (`DESIGN_CLOCK_BASED`)
    - The DUT clock will look regular while in reality it can be stopped
    - Time is "compressed" in simulation somehow
    - Useful view when debugging the DUT

Using zTIDE
Clock control



DESIGN_CLOCK_BASED

uclock advances with a fast frequency while controlled clock is disabled

REGULAR_FREQUENCY

uclock advances with a regular frequency while controlled clock is disabled

Using zTIDE
Good habits

- Use the “random clock stopper”
- Change the `uclock`/DUT clock ratio
  - Make every test at `RUN_LENGTH=1`, `100` and `1000`
- ⚠ Use `REGULAR_FREQUENCY` mode
  Warning: by default zTIDE is in `DESIGN_CLOCK_BASED` mode!!!
- Gated and divided clocks are strictly forbidden in the transactor!!!
- Test every transactor with zTIDE first even “simple” ones

Agenda

- Introduction
- zcei macros
- C/C++ API
- Linking hardware and software
- Using zTIDE
- Transactor debug
- Tips and hints

Transactor debug

- For debugging purposes, you can add accessibility inside the transactor with the following parameter defined in the DVE file:
  `defparam <transactor name>.debug = yes;`
- This will enable dynamic probes in the RTB FPGA; by default they are not visible in the RTB FPGA
- When dynamic probes are enabled, you have access to all transactor registers

Transactor debug

- Additional accessibility is given on transactors:
  - A virtual hierarchy is added to the transactor hierarchy and is visible as dynamic probes
  - You can use `zSelectProbes` to select these probes as you usually do
  - The virtual hierarchy includes:
    - `Zebu_Transactional_Interface.zceiClocks.`*`<clock_name>`*
      Contains all `zceiClockControl` macros interface signals
    - `Zebu_Transactional_Interface.zceiMessageIn`
      Contains all `zceiMessageInPort` macros interface signals
    - `Zebu_Transactional_Interface.zceiMessageOut`
      Contains all `zceiMessageOutPort` macros interface signals

Transactor debug

- When using `zRun`, make the `driverClk` (`uclock`) visible and controllable in `zRun` (by default it is not):

```
zRun –testbench "testbench_command" -debugDriverClk
```

Agenda

- Introduction
- zcei macros
- C/C++ API
- Linking hardware and software
- Using zTIDE
- Transactor debug
- Tips and hints

Tips and hints
Connecting the 2 parts of the Transactor

```
myTransactor.v
          module myTransactor(...);
             [...]
             zceiMessageInPort inport(...);
             zceiMessageOutPort outport(...);
             [...]
          endmodule;

myTransactor.install
          myTransactor {
             NETLIST = "transactorNetlist.edf";}

myTransactor.cc
          myTransactor::myTransactor( Board *board  )
              {
              Port *tx = new TxPort( "inport" );
              Port *rx = new RxPort( "outport" );
              tx->connect( board, "myXtorInstance" );
              rx->connect( board, "myXtorInstance" );
              }
```

Tips and hints
Connecting the Transactor to the DUT

dut.v

```
module dut( dutClock, rst, di, do );
   input  clock;
   input  rst;
   input  di;
   output do;
   [...]
endmodule
```

transactor.dve

```
myTransactor myXtorInstance(
     .dut_in(  di ),
     .dut_out( do ),
     .dut_rst( rst ));
defparam myXtorInstance.cclock = dutClock;
zceiClockPort clockPort( .cclock( dutClock ));
```

myTransactor.v

```
module myTransactor( dut_rst, dut_in, dut_out );
   output dut_rst ;
   output dut_in ;
   input  dut_out ;
   [...]
   zceiClockControl clkCtrl( […] );
   [...]
endmodule
```

Tips and hints

- Message size: The bigger, the faster. Maximum size is about 8K bits
- `zceiMessageInPort` and `zceiMessageOutPort` have a message queue that can contain up to 32 messages
- The following methods do not access the ZeBu hardware (PC memory polling only):
  - `isPossibleToSend`
  - `isPossibleToReceive`
  - `receiveMessage`

Tips and hints

- Always remember that hardware and software are totally asynchronous
  - Most of the time hardware is faster than software, but you must also design for the case where hardware is slower
- Remember that other transactors, ZRM memories, `zRun` and probes may be controlling the same clock as you
  - To check the good behavior of your transactor in this case, use the “random clock noise generator” with zTIDE
- Don’t use the negedge of `uclock` ⚠
- Don’t assume anything about message ports FIFO
  - Sending a message doesn’t mean the other side has read it

Tips and hints

- Make all logic in the transactor synchronous to the `uclock`
  - Avoid using DUT clocks in the transactor
  - Use `cclockEnabled` and `cclockNegEdgeEnabled` signals to check clock state instead

Tips and hints
Transactor Pitfalls

- Transactor randomly hangs
  - Randomness comes from the interaction with the PC
  - Time interval between messages from the PC cannot be deterministic
- Transactor works fine at full speed, hangs when dumping a waveform
  - Bug in the transactor logic of the `zceiClockControl` or zcei message ports
  - When dumping signals, there are several `uclock` cycles between two cycles of user clock
  - Typical bug to assume that setting `readyForCclock` guarantees a user clock edge at the next `uclock` cycle
- Use `zRun –debugDriverClk` as a last recourse
  - Allows to control the `driverClk` (`uclock`) and single step through transactor code
  - Provides full accessibility of zcei message ports interface, transactor state
  - Will not help with issues above, use trace memory to capture debug signals at true speed
  - Use zTIDE first if possible

Tips and hints
Synthesizing Transactors

- Black box models for `zceiMessageInPort` and `zceiMessageOutPort`
  - Remember to set the parameter that specifies the size of your message
    - `zceiMessageInPort #(64) port0 ( …`
  - Make sure the macros survive synthesis
- Transactors with no outputs
  - Some transactors do not send any data to the design, they only send messages back to the software
  - Transactor module has no output
  - Synthesis will optimize away everything by default
- You can use either dummy ports or synthesis directive to preserve macros (these are tools-specific)

Designing zcei transactors
Lab 5

- In this lab you will learn how to design a simple zcei transactor
- We will first design and debug the transactor under the zTIDE environment
- Then we will compile the project for ZeBu
- Finally we will run the emulation

Designing zcei transactors
Lab 5

```
Trainee/lab5
|-- ZeBu
|    |-- zebu.run
|    |    `-- Makefile
|    `-- zebu.src
|         `-- dut.dve
|-- clockNoise
|-- counterDriver
|    |-- c++
|    |    |-- counterDriver.cc
|    |    `-- counterDriver.hh
|    `-- verilog
|         `-- counterDriver.v
|-- dut
|    `-- verilog
|         `-- dut.v
|-- testbench
|    `-- c++
|         `-- testbench.cc
`-- zTIDE
     |-- ztide.run
     |    `-- Makefile
     |-- ztide.simu
     |    `-- Makefile
     `-- ztide.src
          `-- tideWrapper.v
```

Designing zcei transactors
Lab 5

- The DUT is a simple 32-bit counter with a synchronous pre-load and enable signal

```
module dut (
  clk,
  rstn,
  load,
  ena,
  d,
  c
  );

  input         clk;
  input         rstn;
  input         load;
  input         ena;
  input  [31:0] d;
  output [31:0] c;

  reg    [31:0] c;

  always @(posedge clk or negedge rstn)
    if (!rstn)
      c <= 0;
    else if (load)
      c <= d;
    else if (ena)
      c <= c + 1;

endmodule
```

Designing zcei transactors
Lab 5

- We will control this DUT with a simple transactor to which a testbench will send some simple command such as:
  - Run for *n* cycles with `enable=1`
  - Idle for *n* cycles with `enable=0`
  - Load a value in the counter
  - Read the value from the counter

Designing zcei transactors
Lab 5

- Software interface
  - Idle
    - run the DUT clock for *n* cycles with the counter disabled.
  - Run
    - run the DUT clock for *n* cycles with the counter enabled.
  - Load
    - load counter with *n* (run one DUT clock cycle).
  - Read
    - return counter value (DUT clock is stopped).

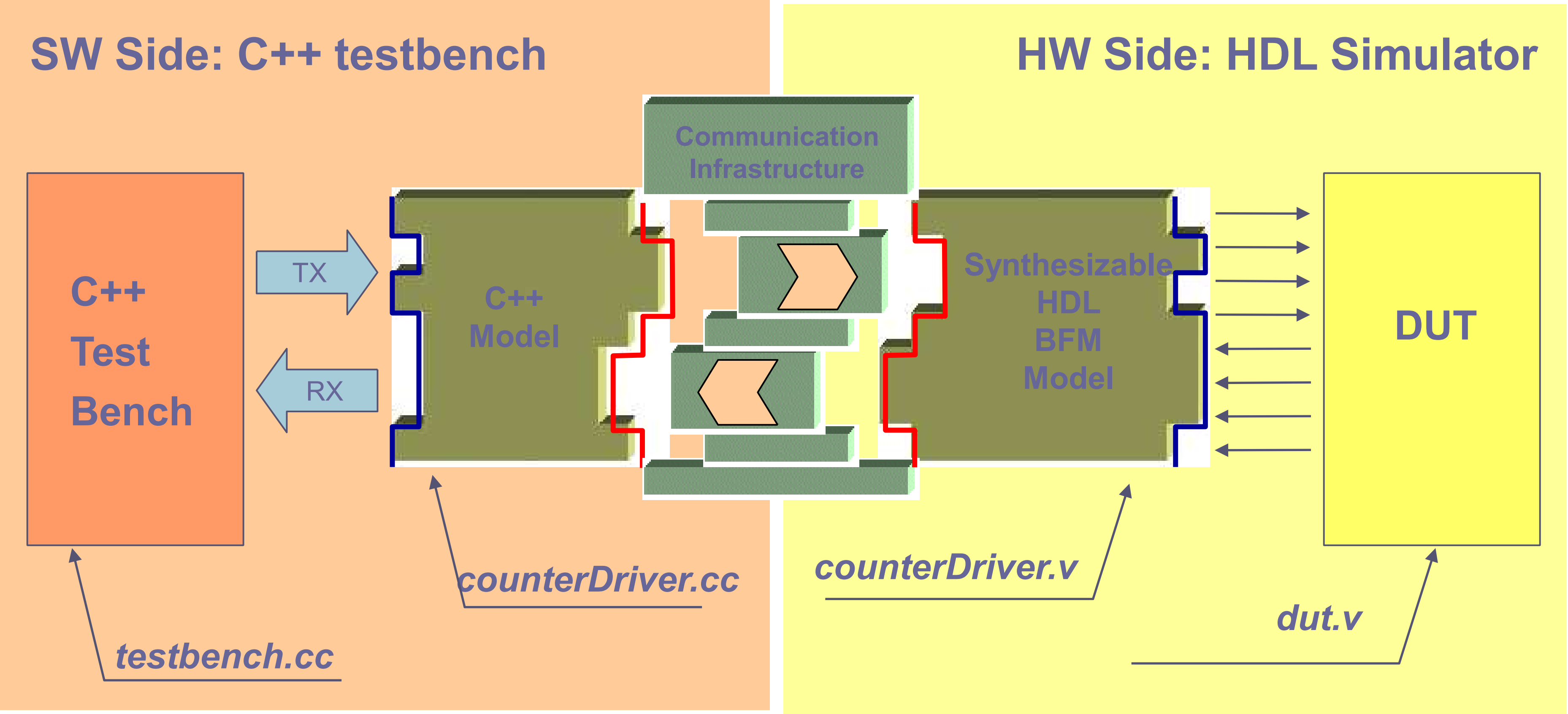
- Hardware interface
  - Outputs
    - `load`
    - `ena`
    - `d[31:0]`
  - Inputs
    - `c[31:0]`

Designing zcei transactors
Lab 5

- The specifications are:
  - Control of the DUT clock:
    - *n* clock cycles for run and idle commands
    - 1 clock cycle for load
    - clock stopped otherwise
    - we will need to implement a DUT clock cycle counter in the transactor
  - Message from software : two 32-bit words
    - a command in the first word
      - 0 : load
      - 1 : run
      - 2 : idle
      - 3 : read
    - a 32 bits value in the second word
      - number of clock cycles for run and idle commands
      - value to load in the counter for the load command
  - Message from hardware : one 32-bit word
    - a 32-bit value (value of the counter)

Designing zcei transactors
Lab 5

- The logical structure



Designing zcei transactors
Lab 5

- We will first implement the run transaction
  - Add the run transaction to the template file `counterDriver/verilog/counterDriver.v`
  - Work from the directory: `zTIDE/ztide.simu`
    This is the directory from which we will run the simulator
  - Use the provided `Makefile` to compile the Verilog files and check the syntax
    - We cannot run until we actually completed the software part
    - The `Makefile` uses the zTIDE PLI library

Designing zcei transactors
Lab 5

- Complete the C++ code
  - Work from the directory:
    `zTIDE/ztide.run`
    This is the directory from which we will run the testbench
  - In `counterDriver/c++/counterDriver.cc` and `.hh` files
    - Complete the `init()` method
      - create a port
      - connect it to ZeBu
      - Remember to initialize the pointer to the data buffer
    - Complete the `run()` method.
  - In the `testbench/c++/testbench.cc` file
    - Add the necessary code to open ZeBu and stimulate the counter
- Have a look at `zTIDE/ztide.run/Makefile` and compile the testbench

Designing zcei transactors
Lab 5

- Run with zTIDE
  - Launch the testbench executable from one terminal
    - From the `ztide.run` directory
  - Launch the HDL simulator GUI from another terminal
    - From the `ztide.simu` directory
    - Use the provided `Makefile` to run the simulation

Designing zcei transactors
Lab 5

- Then go step-by-step
  - Implement the idle method first
  - Then the load method
  - And finally the read method
    - It requires to create a port in the other direction
- Each time simulate in zTIDE until it works
- When everything is working in zTIDE, do the following additional tests
  - Add the random clock stopper
  - Change the `RUN_LENGTH` parameter
- When everything is working fine under these conditions, move to ZeBu

Designing zcei transactors
Lab 5

- Go into the `ZeBu` directory
- Compile the design and the transactor in `zCui`
  - In the default RTL group, add the source file: `../dut/verilog/dut.v`
  - Add a transactor object, name it `counterDriver`
    - The top name is `counterDriver`
    - Add an RTL group to the transactor object, name it `counterDriver`
    - Add `../counterDriver/verilog/counterDriver.v` source file to it
  - Use the provided DVE file in `zebu.src/dut.dve`
  - Select the appropriate target hardware configuration file for your ZeBu system
  - Set the job scheduling preferences

Designing zcei transactors
Lab 5

- Go into the `zebu.run` directory
- Compile the testbench using the provided `Makefile`
- Run
  - Launch the executable and see what happens.
- In case of a problem, debug as follows:
  - Use
    - `zRun -testbench ./testbench` to control the emulation
    - Add dynamic probes and use Monitor and Monitor Dump
  - Use
    - `zRun -testbench ./testbench -debugDriverClk`
    - Control the uncontrolled clock
    - Dump a VCD file using Monitor Dump