Compilation

ZeBu-Server Training

THE *FASTEST* VERIFICATION

Agenda

- Back-end compilation
- `zCui` Preferences
- Memory modeling
- Debug
- `zTime` timing analysis

Compilation Overview

- Back-end compilation
  - Starts from EDIF netlist after synthesis
  - Ends with FPGA bistream file (which can be downloaded in FPGA at runtime), and with a debug database (downloaded in the PC)
- The full back-end compilation flow is integrated in `zCui`
- `zCui` can handle multiple back-end compilations
  - Each back-end can be compiled separately
  - Each back-end can have different parameters
  - All back-ends use the same synthesis results

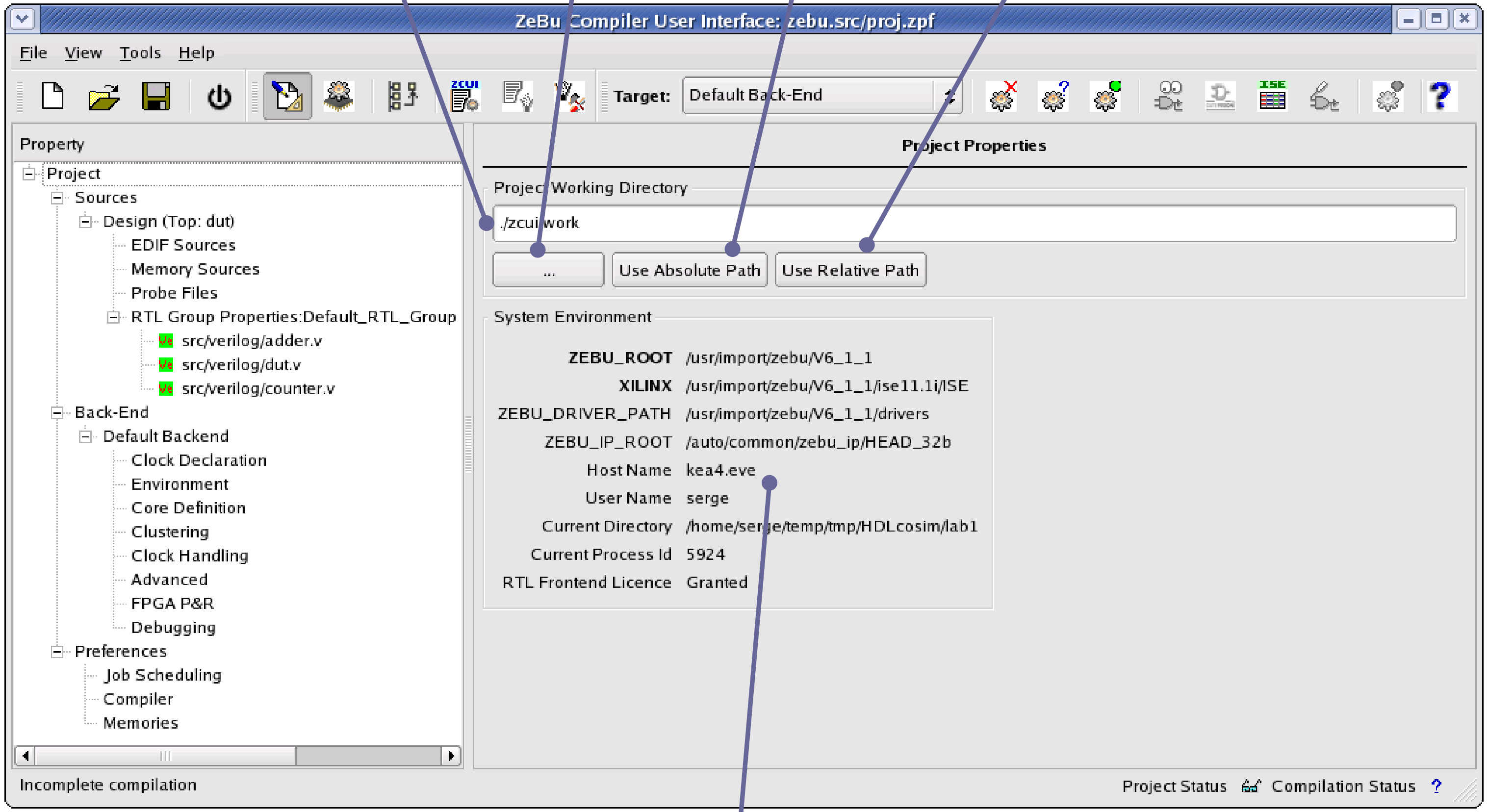
Compilation
Project Properties panel

Project working directory

File browser

Change path to absolute path

Change path to Relative path



- Project
  - Sources
    - Design (Top: dut)
      - EDIF Sources
      - Memory Sources
      - Probe Files
      - RTL Group Properties:Default_RTL_Group
        - src/verilog/adder.v
        - src/verilog/dut.v
        - src/verilog/counter.v
  - Back-End
    - Default Backend
      - Clock Declaration
      - Environment
      - Core Definition
      - Clustering
      - Clock Handling
      - Advanced
      - FPGA P&R
      - Debugging
  - Preferences
    - Job Scheduling
    - Compiler
    - Memories

Project Working Directory: ./zcui.work

System Environment

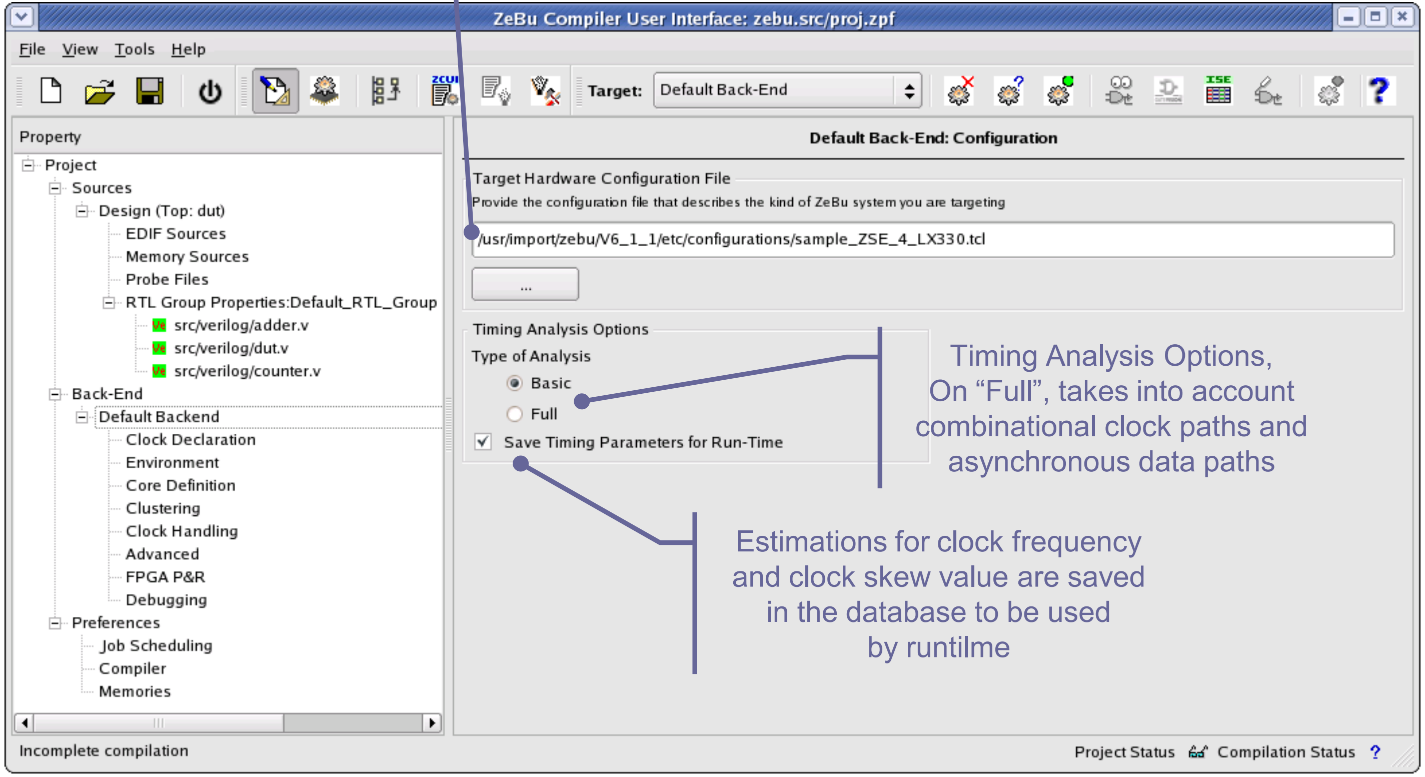
ZEBU_ROOT	/usr/import/zebu/V6_1_1
XILINX	/usr/import/zebu/V6_1_1/ise11.1i/ISE
ZEBU_DRIVER_PATH	/usr/import/zebu/V6_1_1/drivers
ZEBU_IP_ROOT	/auto/common/zebu_ip/HEAD_32b
Host Name	kea4.eve
User Name	serge
Current Directory	/home/serge/temp/tmp/HDLcosim/lab1
Current Process Id	5924
RTL Frontend Licence	Granted

Incomplete compilation

Useful environment variables and settings

Compilation Configuration

Target Hardware Configuration file
Defines the ZeBu platform architecture



Default Back-End: Configuration

Target Hardware Configuration File

Provide the configuration file that describes the kind of ZeBu system you are targeting

/usr/import/zebu/V6_1_1/etc/configurations/sample_ZSE_4_LX330.tcl

Timing Analysis Options

Type of Analysis

[✓ Basic] Full

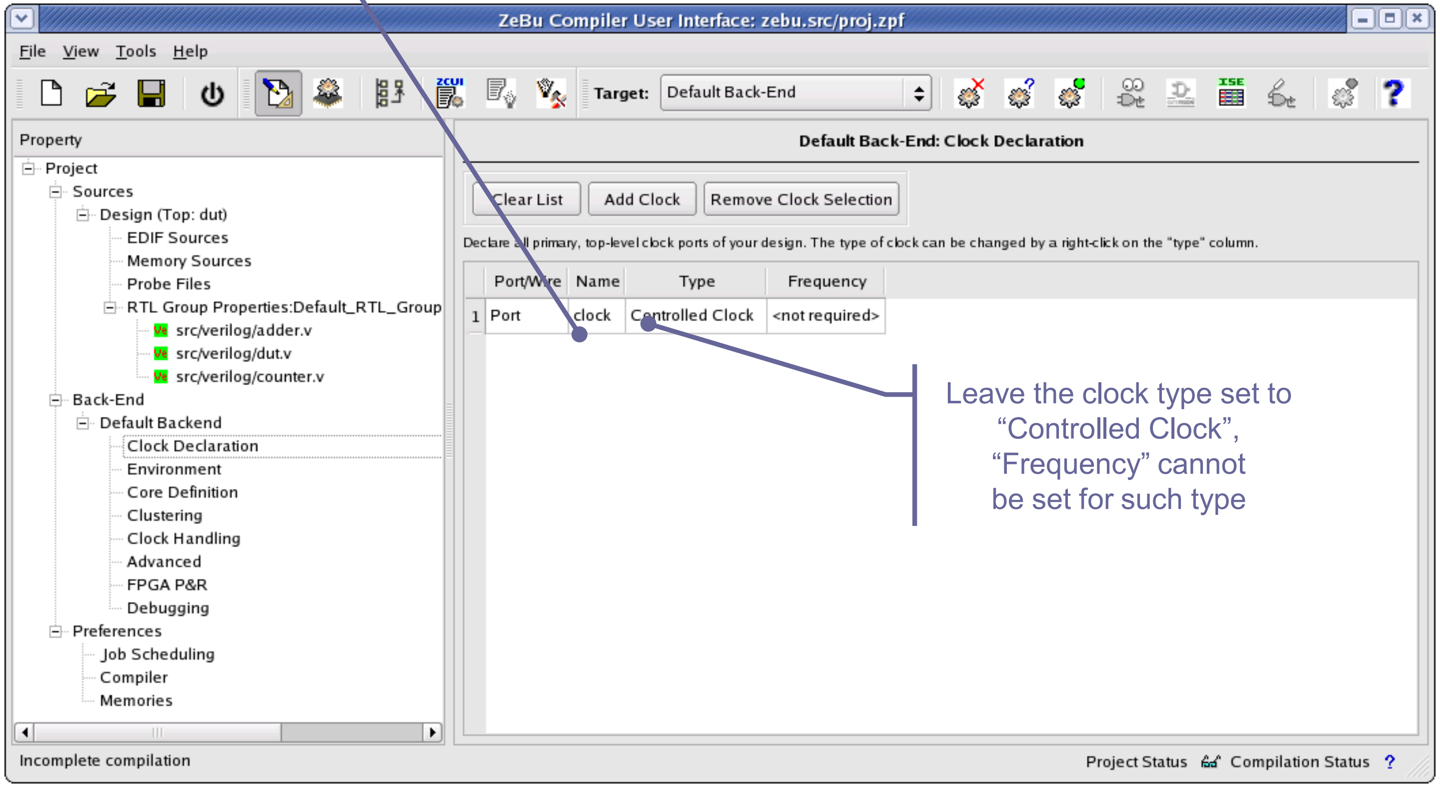
[✓] Save Timing Parameters for Run-Time

Timing Analysis Options,
On “Full”, takes into account
combinational clock paths and
asynchronous data paths

Estimations for clock frequency
and clock skew value are saved
in the database to be used
by runtilme

Compilation
Clock Declaration panel

Add clock names



ZeBu Compiler User Interface: zebu.src/proj.zpf

File View Tools Help

Target: Default Back-End

Property

- Project
  - Sources
    - Design (Top: dut)
      - EDIF Sources
      - Memory Sources
      - Probe Files
      - RTL Group Properties:Default_RTL_Group
        - src/verilog/adder.v
        - src/verilog/dut.v
        - src/verilog/counter.v
  - Back-End
    - Default Backend
      - Clock Declaration
      - Environment
      - Core Definition
      - Clustering
      - Clock Handling
      - Advanced
      - FPGA P&R
      - Debugging
  - Preferences
    - Job Scheduling
    - Compiler
    - Memories

Default Back-End: Clock Declaration

Clear List | Add Clock | Remove Clock Selection

Declare all primary, top-level clock ports of your design. The type of clock can be changed by a right-click on the "type" column.

	Port/Wire	Name	Type	Frequency
1	Port	clock	Controlled Clock	<not required>

Incomplete compilation

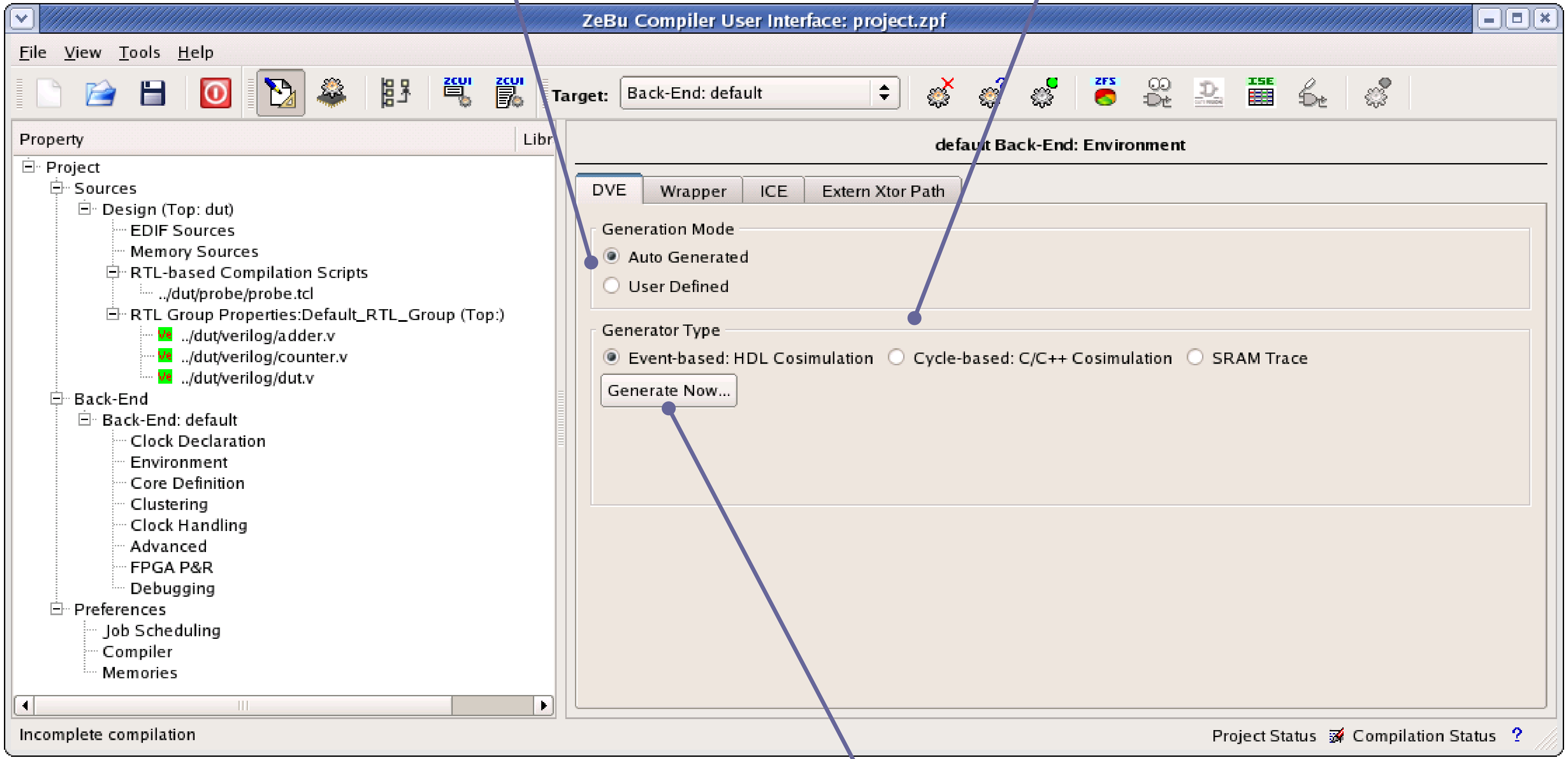
Project Status Compilation Status

Leave the clock type set to "Controlled Clock", "Frequency" cannot be set for such type

Compilation Environment (Auto Generated DVE file)

Choose auto generated or user defined DVE file

Select operation mode:
•Event-Based, HDL co-simulation
•Cycle-based, C/C++ co-simulation
•SRAM Trace


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

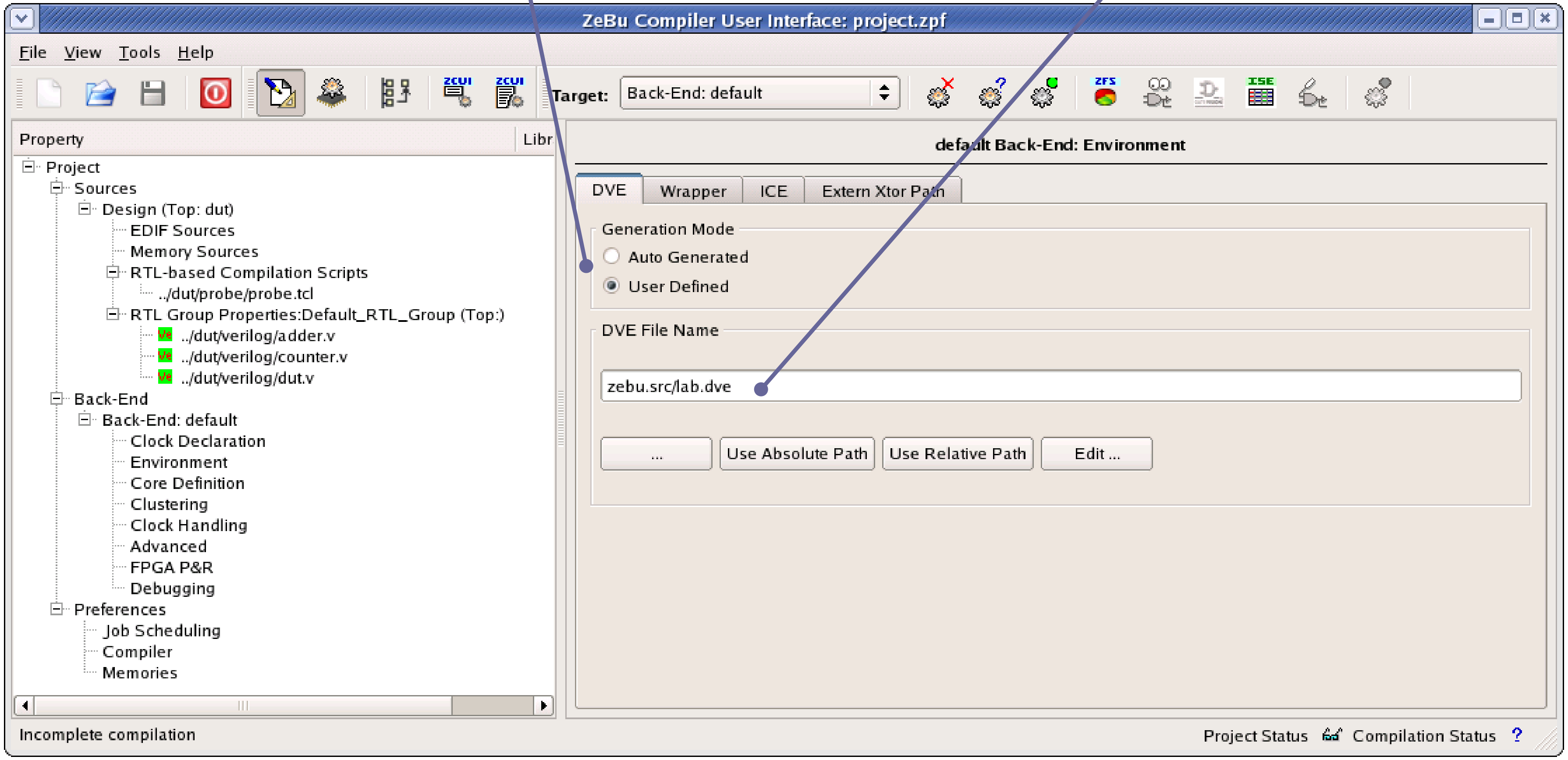
Project Status Compilation Status

Generate DVE file template

Compilation Environment (User Defined DVE file)

Choose auto generated or user defined DVE file

Enter DVE file name



Generation Mode
Auto Generated [✓ User Defined]

DVE File Name: zebu.src/lab.dve

Compilation DVE File example

Type of driver (HDL co-simulation)

Name of driver

Driver parameter definition

Clock generator output

```
HDL_COSIM counter (
        .input_bin ( {
                resetn,
                load,
                ena,
                din[31:0]
        } ),
        .output_bin (
                cnt[31:0]
        ),
        .output_tri (
                cnt_hz[31:0]
        )
);
defparam counter.clock_0 = "clock" ;

zceiClockPort dut_clock (
        .cclock( clock  )
);
```

Binary Inputs of DUT

Binary Outputs of DUT

Tristate Outputs of DUT

DUT clock declaration

Primary clock of DUT

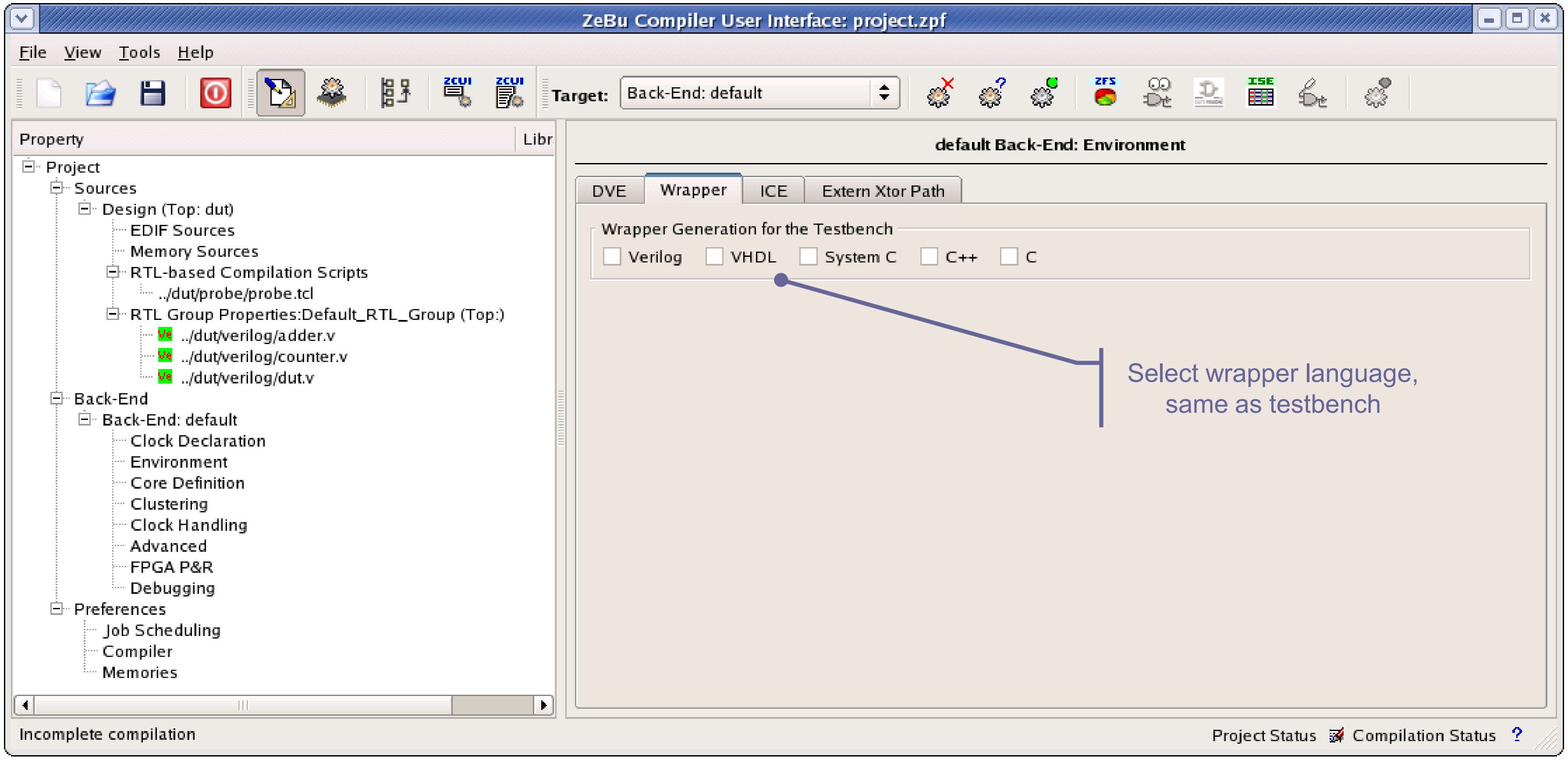
Note: The driver does not drive directly the clocks!

Compilation DVE file

- DVE syntax is very close to Verilog syntax except:
  - DVE (like the EDIF) is case insensitive
  - Vector range must be specified
    E.g.: `.bus(bus[31:0])`
- Drivers provided:
  - `HDL_COSIM`: HDL event-based co-simulation
  - `C_COSIM`: C/C++ cycle-based co-simulation
  - `MKCK_COSIM`(*): C/C++ event-based co-simulation
  - `SRAM_TRACE`: Trace memory instantiation
  - `SMART_ZICE_ZSE`(*): Connection to Smart Z-ICE connector

(*) must be manually instantiated, no auto generation

Compilation Environment (Wrapper)



ZeBu Compiler User Interface: project.zpf

File View Tools Help

Target: Back-End: default

Property | Libr

- 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

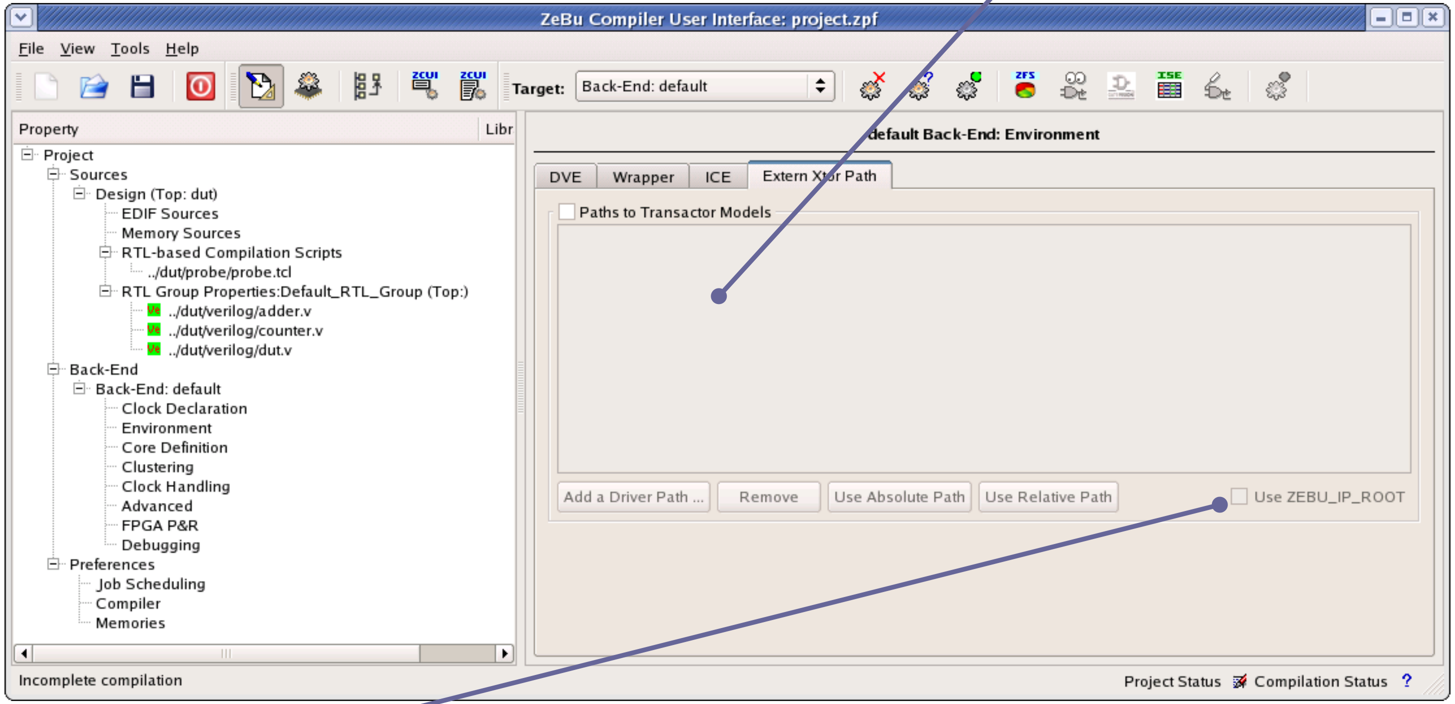
Incomplete compilation

Project Status Compilation Status

Select wrapper language, same as testbench

Compilation Environment (Extern Xtor path)

Add paths to transactor directories



ZeBu Compiler User Interface: project.zpf

File View Tools Help

Target: Back-End: default

Property | Libr

- 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

Paths to Transactor Models

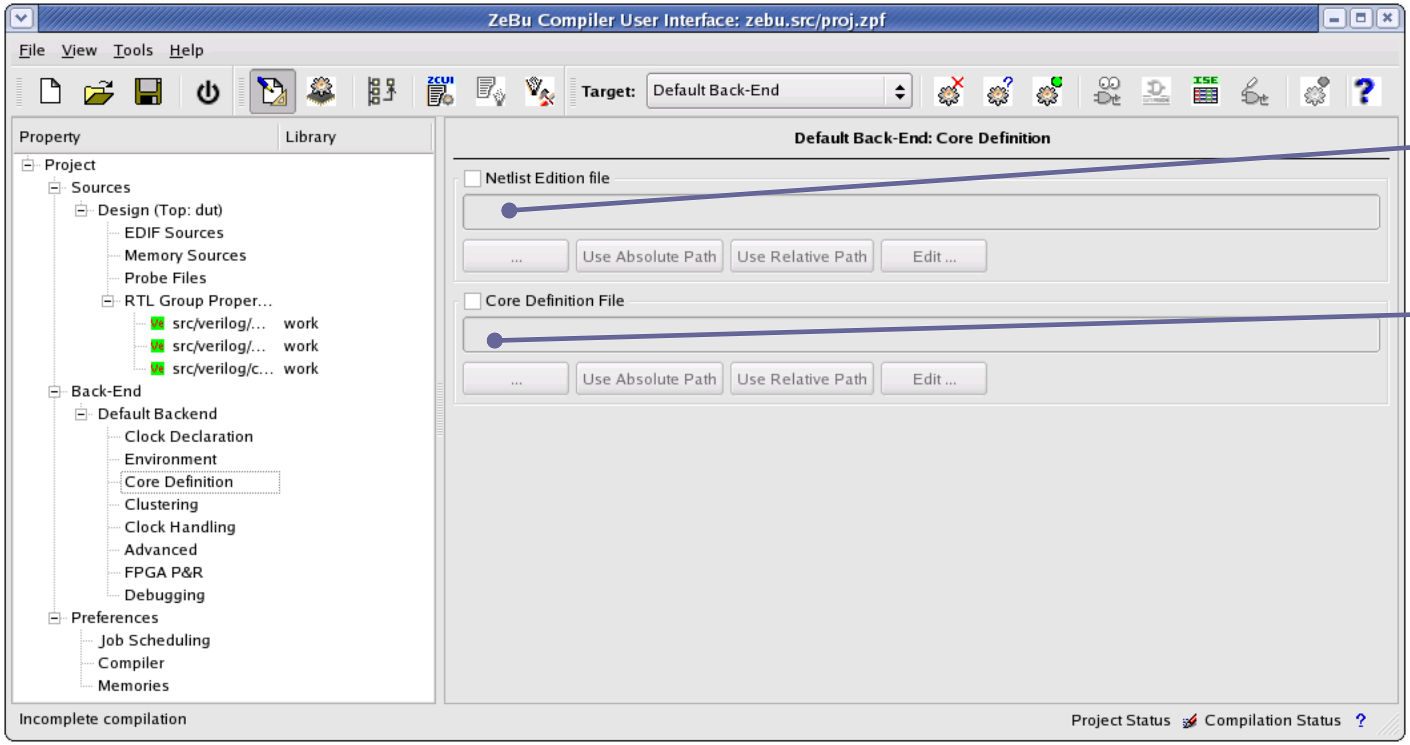
Add a Driver Path ... | Remove | Use Absolute Path | Use Relative Path | Use ZEBU_IP_ROOT

Incomplete compilation

Project Status | Compilation Status

Turn on if using EVE's Vertical Solutions transactors, ZEBU_IP_ROOT variable must be set

Compilation Core Definition panel



Enter name of advanced netlist edition commands file

Enter name of core definition file

Syntax of the core definition file:

```
defcore <core name> -path_list {<instance name>}
```

Compilation
Clock Handling panel

Clock tree glitch removal
- •No Handling Process: No processing
- •Filter Glitches: Insert glitch filter on clock nets
- •Delay Clock Enable: Insert delay on clock-enable path

Control delay insertion for skew offset

Select asynchronous if clock is driven by ICE

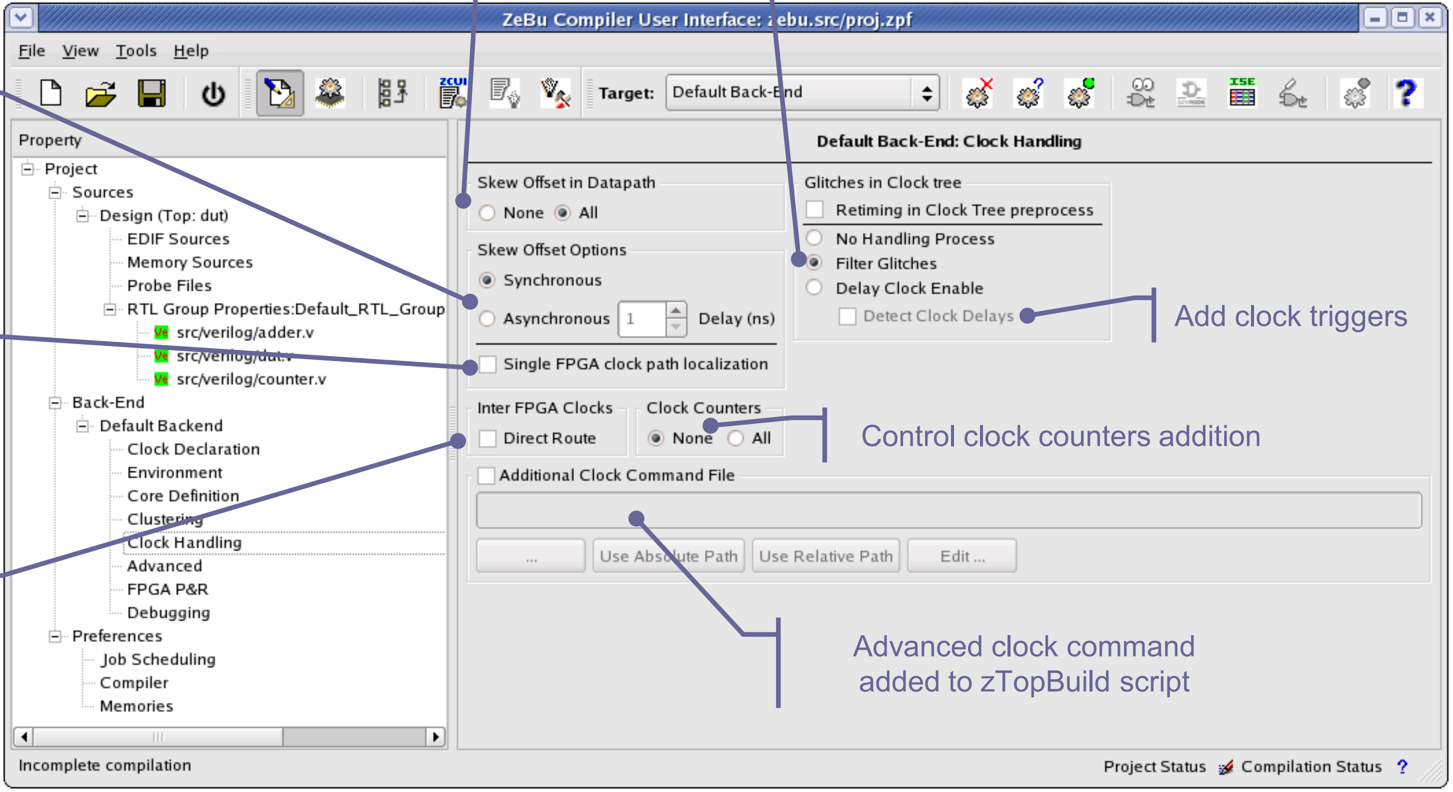
Avoid inter-FPGA clocks by replicating clock trees in each design FPGA

Route clock signals without multiplexing or hops

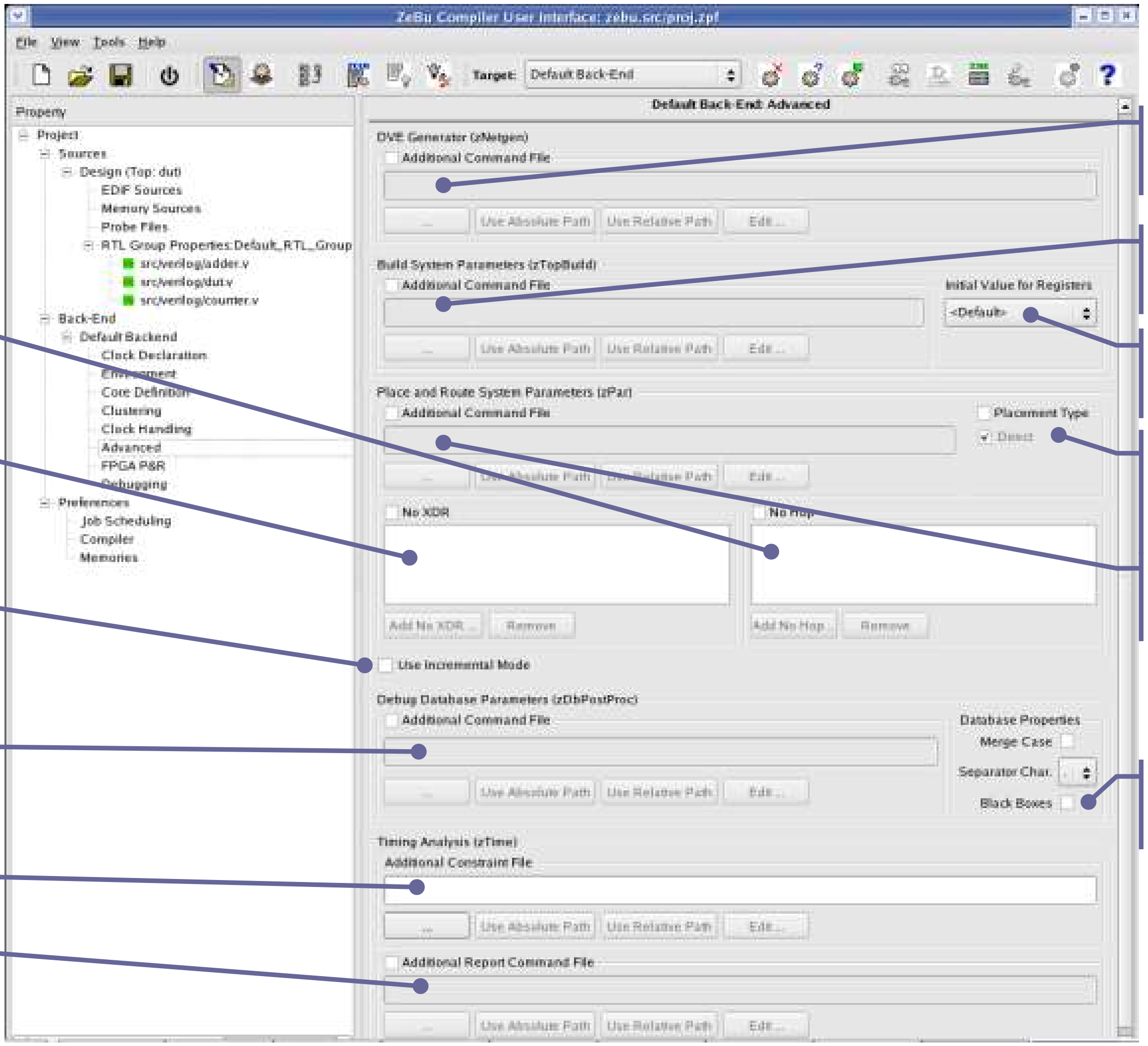
Add clock triggers

Control clock counters addition

Advanced clock command added to zTopBuild script



Compilation
Advanced panel



Property
- Project
  - Sources
    - Design (Top: dut)
      - EDIF Sources
      - Memory Sources
      - Probe Files
      - RTL Group Properties:Default_RTL_Group
        - 🟩 src/verilog/adder.v
        - 🟩 src/verilog/dut.v
        - 🟩 src/verilog/counter.v
  - Back-End
    - Default Backend
      - Clock Declaration
      - Environment
      - Core Definition
      - Clustering
      - Clock Handling
      - Advanced
      - FPGA P&R
      - Debugging
  - Preferences
    - Job Scheduling
    - Compiler
    - Memories

Default Back-End: Advanced

DVE Generator (zNetgen)
- ☐ Additional Command File

Build System Parameters (zTopBuild)
- ☐ Additional Command File
- Initial Value for Registers <Default>

Place and Route System Parameters (zPar)
- ☐ Additional Command File
- ☐ Placement Type
- ✓ Direct

☐ No XDR

☐ No Hop

☐ Use Incremental Mode

Debug Database Parameters (zDbPostProc)
- ☐ Additional Command File
- Database Properties
  - Merge Case ☐
  - Separator Char. .
  - Black Boxes ☐

Timing Analysis (zTime)
- Additional Constraint File
- ☐ Additional Report Command File

Advanced command file for DVE generator

Advanced command file for `zTopBuild`

Set initial value for registers

If placement type is set to "Direct", original placement is kept

Advanced command file for `zPar`

Includes black box in HDL wrapper

Suppress hops on signals

Suppress multiplexing on signals

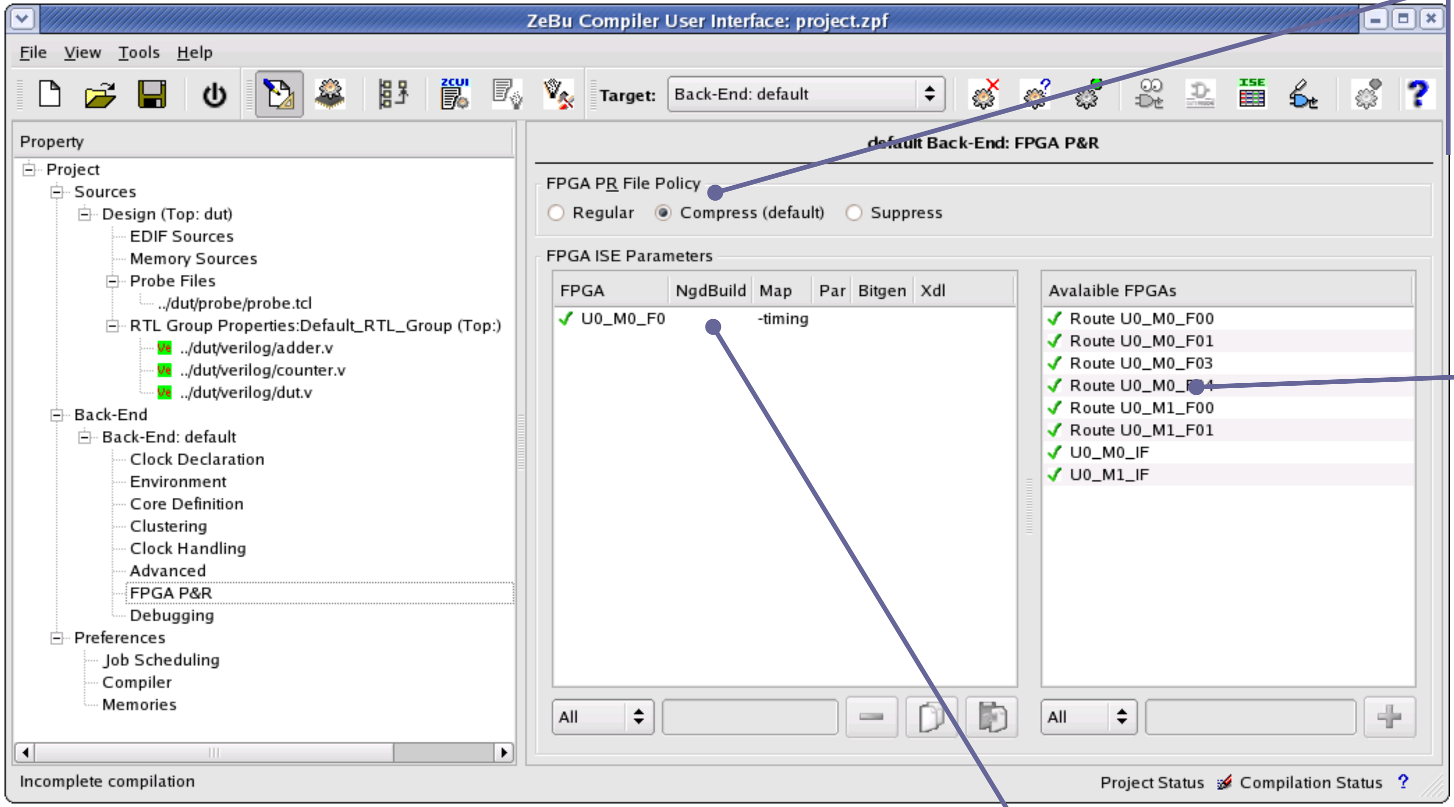
Activate incremental mode

Advanced command file for `zDbPostProc`

`zTime` constraint file

`zTime` additional report command file

Compilation FPGA P&R panel



ZeBu Compiler User Interface: project.zpf

File View Tools Help

Target: Back-End: default

Property

- Project
  - Sources
    - Design (Top: dut)
      - EDIF Sources
      - Memory Sources
      - Probe Files
        - ../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: FPGA P&R

FPGA PR File Policy

Regular [✓ Compress (default)] Suppress

FPGA ISE Parameters

FPGA	NgdBuild	Map	Par	Bitgen	Xdl
✓ U0_M0_F0		-timing			

Avalaible FPGAs
✓ Route U0_M0_F00
✓ Route U0_M0_F01
✓ Route U0_M0_F03
✓ Route U0_M0_F04
✓ Route U0_M1_F00
✓ Route U0_M1_F01
✓ U0_M0_IF
✓ U0_M1_IF

All All

Incomplete compilation

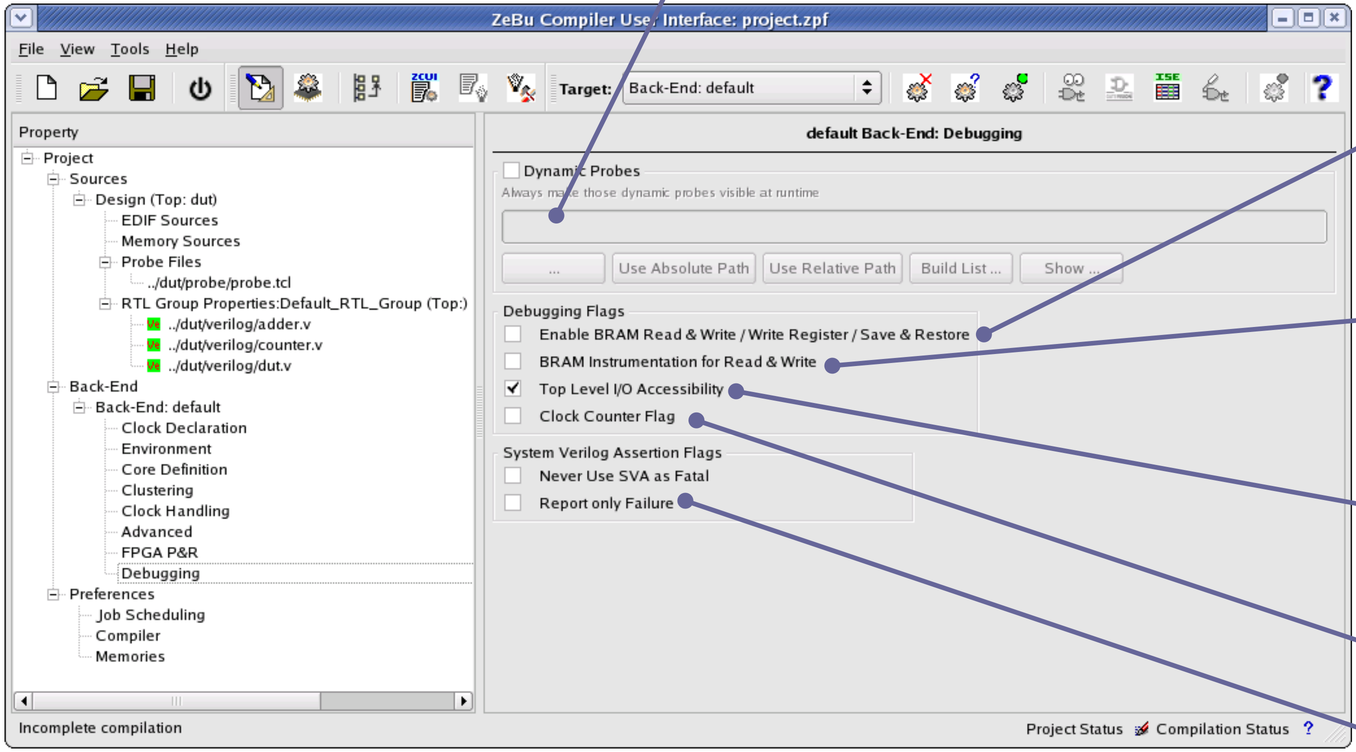
Project Status Compilation Status

FPGA intermediate place & route files are either:
- Regular: kept
- Compress: compressed
- Suppress: suppressed

List of available FPGAs (Only visible after compilation)

Per FPGA and per Xilinx ISE tool additional command line option

Compilation Debugging panel



ZeBu Compiler User Interface: project.zpf

File View Tools Help

Target: Back-End: default

Property

- Project
  - Sources
    - Design (Top: dut)
      - EDIF Sources
      - Memory Sources
      - Probe Files
        - ../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: Debugging

☐ Dynamic Probes

Always make those dynamic probes visible at runtime

... | Use Absolute Path | Use Relative Path | Build List ... | Show ...

Debugging Flags

☐ Enable BRAM Read & Write / Write Register / Save & Restore

☐ BRAM Instrumentation for Read & Write

☑ Top Level I/O Accessibility

☐ Clock Counter Flag

System Verilog Assertion Flags

☐ Never Use SVA as Fatal

☐ Report only Failure

Incomplete compilation

Project Status Compilation Status

List dynamic probes which must be visible

Enable software reading and writing in BRAM, register writing and save and restore

Allow BRAM software read and write when upper switch is not used

Top level I/O accessibility (dynamic probes)

Add clock counters

SVA options

Agenda

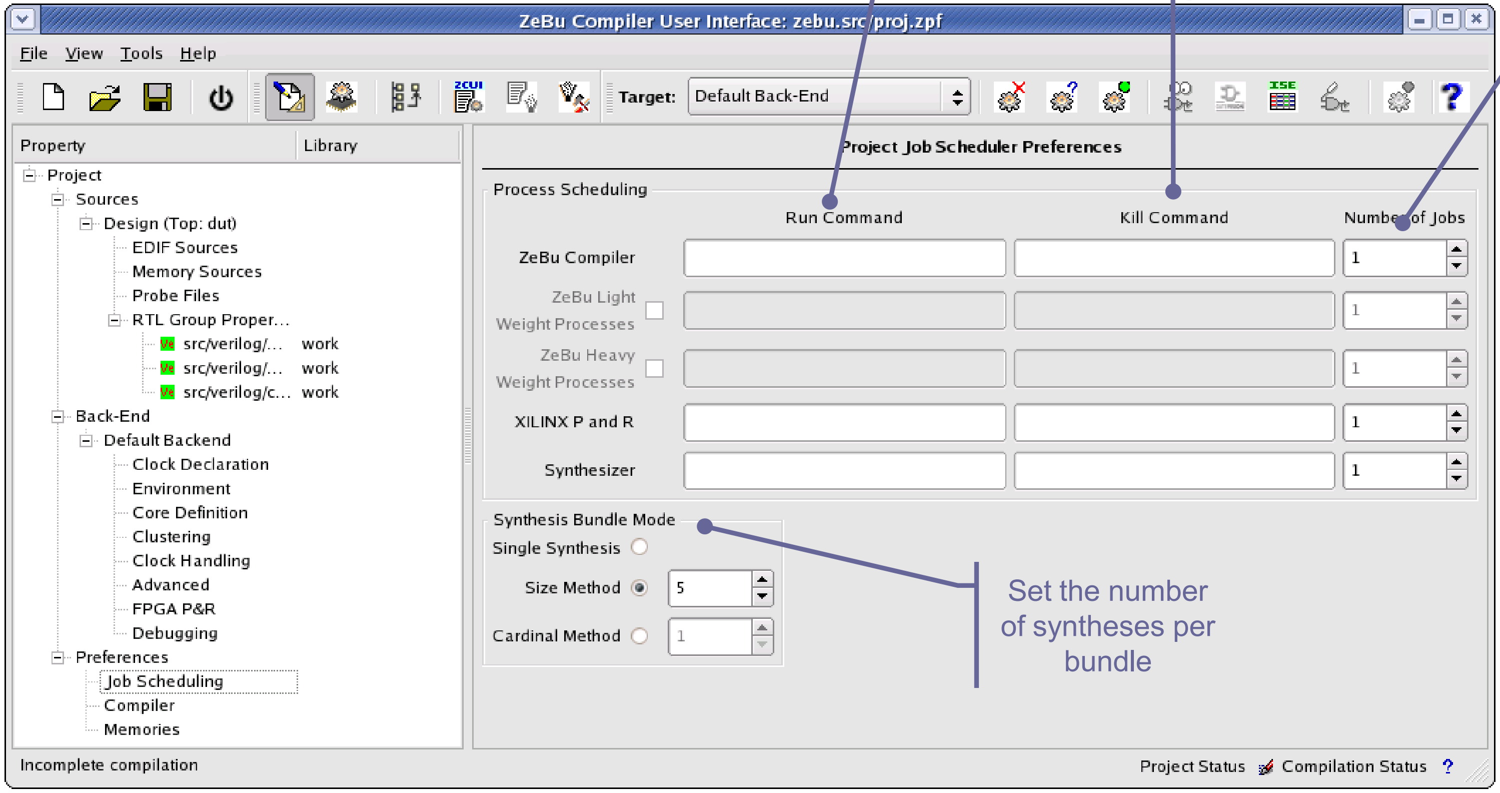
- Back-end compilation
- `zCui` Preferences
- Memory modeling
- Debug
- `zTime` timing analysis

Compilation
Project Job Scheduler Preferences panel

Launch pre-command

Kill command

Number of jobs to be launched in parallel, can be unlimited



ZeBu Compiler User Interface: zebu.src/proj.zpf

File View Tools Help

Target: Default Back-End

Property	Library
Project	
Sources	
Design (Top: dut)	
EDIF Sources	
Memory Sources	
Probe Files	
RTL Group Proper...	
src/verilog/...	work
src/verilog/...	work
src/verilog/c...	work
Back-End	
Default Backend	
Clock Declaration	
Environment	
Core Definition	
Clustering	
Clock Handling	
Advanced	
FPGA P&R	
Debugging	
Preferences	
Job Scheduling	
Compiler	
Memories	

Project Job Scheduler Preferences

Process Scheduling

	Run Command	Kill Command	Number of Jobs
ZeBu Compiler			1
ZeBu Light Weight Processes ☐			1
ZeBu Heavy Weight Processes ☐			1
XILINX P and R			1
Synthesizer			1

Synthesis Bundle Mode

Single Synthesis	○	
Size Method	●	5
Cardinal Method	○	1

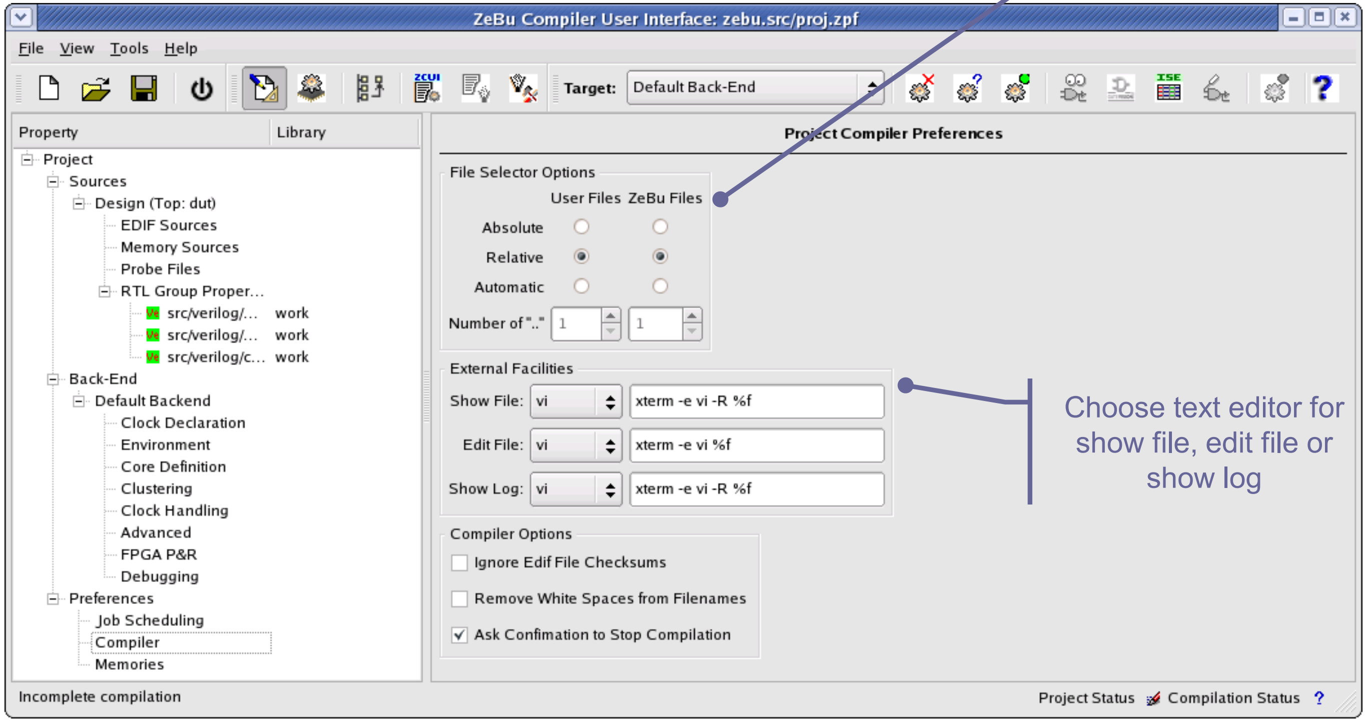
Incomplete compilation

Project Status Compilation Status ?

Set the number of syntheses per bundle

Compilation
Project Compiler Preferences panel

Specify if absolute or relative paths will be used by default



ZeBu Compiler User Interface: zebu.src/proj.zpf

File View Tools Help

Target: Default Back-End

Property	Library
Project	
Sources	
Design (Top: dut)	
EDIF Sources	
Memory Sources	
Probe Files	
RTL Group Proper...	
src/verilog/...	work
src/verilog/...	work
src/verilog/c...	work
Back-End	
Default Backend	
Clock Declaration	
Environment	
Core Definition	
Clustering	
Clock Handling	
Advanced	
FPGA P&R	
Debugging	
Preferences	
Job Scheduling	
Compiler	
Memories	

Project Compiler Preferences

File Selector Options

	User Files	ZeBu Files
Absolute	○	○
Relative	●	●
Automatic	○	○
Number of ".."	1	1

External Facilities

Show File:	vi	xterm -e vi -R %f
Edit File:	vi	xterm -e vi %f
Show Log:	vi	xterm -e vi -R %f

Compiler Options

- ☐ Ignore Edif File Checksums
- ☐ Remove White Spaces from Filenames
- ✓ Ask Confimation to Stop Compilation

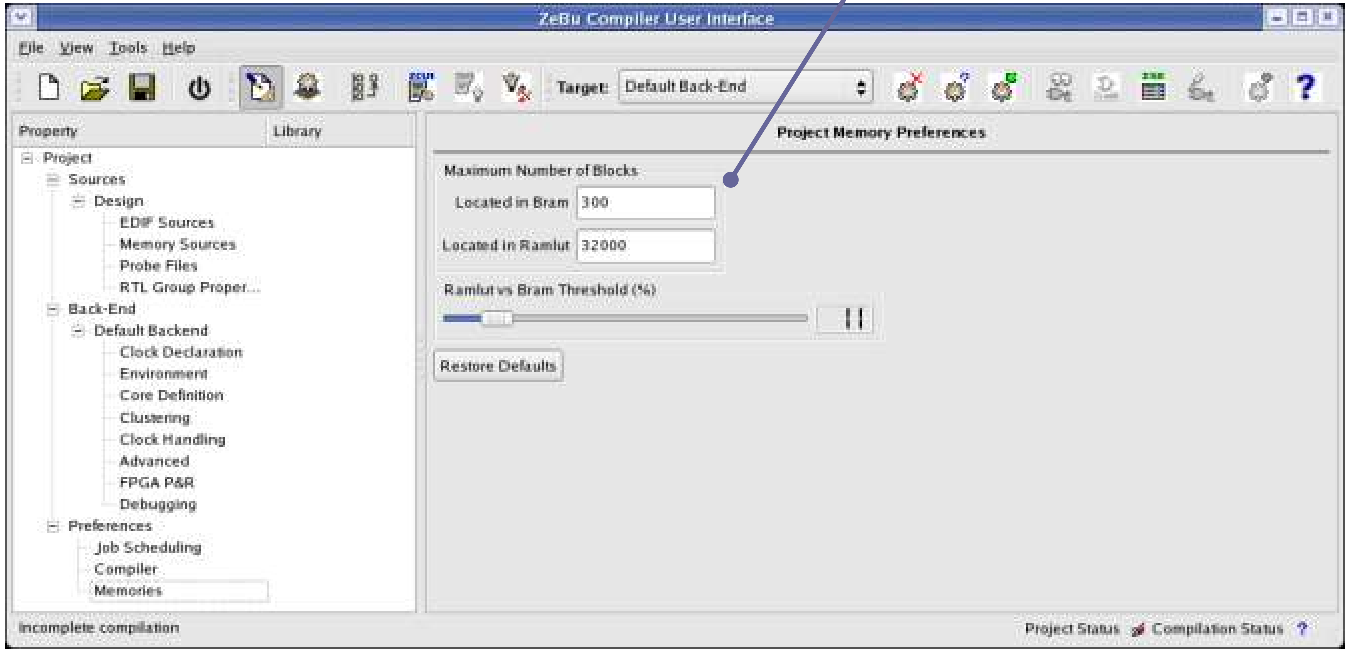
Incomplete compilation

Project Status Compilation Status ?

Choose text editor for show file, edit file or show log

Compilation
Project Memory Preferences panel

Memory compiler settings


ZeBu Compiler User Interface

File View Tools Help

Target: Default Back-End

Property | Library

- Project
  - Sources
    - Design
      - EDIF Sources
      - Memory Sources
      - Probe Files
      - RTL Group Proper...
  - Back-End
    - Default Backend
      - Clock Declaration
      - Environment
      - Core Definition
      - Clustering
      - Clock Handling
      - Advanced
      - FPGA P&R
      - Debugging
  - Preferences
    - Job Scheduling
    - Compiler
    - Memories

Project Memory Preferences

Maximum Number of Blocks

Located in Bram: 300

Located in Ramlut: 32000

Ramlut vs Bram Threshold (%)

Restore Defaults

Incomplete compilation

Project Status · Compilation Status

Compilation
Lab 2

- In this lab you will learn how to synthesize and compile a much complex project
  - You will use RTL Front-End analysis and elaboration commands to build the project
  - You will define cores
- The provided design is a parallel computer made of 4 clusters of 16 processors. It is used to compute a Mandelbrot picture (mathematical picture)
- The design uses transactional emulation to be able to display the picture in real time on the screen
  - Transactional emulation is not covered by this training module
  - We will thus provide all elements necessary for the compilation including a DVE file

Compilation Lab 2

```
Trainee/lab2
|-- file.f
|-- rom
|   `-- rom
|-- xtor
|   |-- designFeatures
|   |-- screen_pilote.v
|   |-- screenIF.c
|   |-- screenIF.h
|   |-- zdisplay.c
|   |-- zdisplay.h
|   |-- zpro_dut.dve
|   `-- zpro_screen.c
|-- zebu.xtor
|   |-- Makefile
|   `-- mem.db
`-- zpro_dut
    |-- *.v
    `-- zcore
        `-- *.v
```

Compilation
Lab 2

- Go into the directory named `Trainee/lab2`
- Prepare a template project based on the file list
  - Type:
    ```
    zRFEvlog -f file.f
    zRFEelab zpro_dut
    ```
  - This will create an elaborated design in the `zfc.work` directory
  - Copy the template project file locally:
    ```
    cp zfc.work/top_elab.zpf project.zpf
    ```
- Open `zCui` on that project file:
  ```
  zCui –p project.zpf
  ```
- Notice that the file list is there and that all top names are set

Compilation
Lab 2

- Select the synthesizer you want to use
  - Left-click on the RTL group, and select the synthesizer in the main tab
- Select the target hardware configuration file
  - Left-click on the “Back-End : default” property
  - Open the file browser and select the appropriate target hardware configuration file for your ZeBu system
- Set the job scheduling preferences
  - Left-click on the “Job Scheduling” property
  - Fill the form in accordance with your network configuration
- Save the project
- Launch the synthesis:
  - Select “Design” as target
  - Click on the “Compile” button
- Let the synthesis run…

Compilation
Lab 2

- On this multi-core design we want to use the core compilation of ZeBu-Server
- We have to prepare a core definition file (TCL file)
  - So you have to provide a list of design instance names which will be used to create ZeBu cores
  - In this design we want to use the processors cluster level as a base for core definition. So we have to provide the instance names of the cluster found in the synthesized netlist
  - As clusters are described using a Verilog `generate` statement, each synthesizer will create its own instance name style for the generated instances
    - You will have to inspect the netlist to find the used instance names

Compilation
Lab 2

- Inspect the design netlist
  - In `zCui`, click on the zBrowser icon
  - Navigate the netlist until you find the 4 clusters
  - Take note of the hierarchical instance path name of the 4 clusters
- Using a text editor, create a file named `zebu.xtor/cores.tcl`
  - Define 4 cores, one for each cluster
- In `zCui`, select the core definition file
  - Click on the “Core Definition” property
  - Turn on the “Core Definition File” checkbox
  - Select the `cores.tcl` file in the file browser

Compilation
Lab 2

- Add the display transactor:
  - Right-click on "Design" property and select "Add Transactor …"
  - Give the name "`screen_pilote`" to the transactor when asked
  - Right-click on the `screen_pilote` transactor property, select "Add RTL Group…". Give it the name "`screen_pilote`" as well.
  - Right-click on the `screen_pilote` RTL group of the `screen_pilote` transactor and select "Add Verilog Sources …"
  - Add the `xtor/screen_pilote.v` file using the file browser
  - Left-click on the RTL group of the transactor and set the RTL group top name in the "main" tab to: `screen_pilote` (again!)
  - In the same "main" tab, select your synthesizer
  - Left-click on the Transactor property and set its top name to `screen_pilote` (this time that's it!)

Compilation
Lab 2

- Select the DVE file:
  - Left-click on the "Environment" property
  - Select "User Defined" in the "Generation Mode" frame
  - In the DVE file browser, choose the file named `xtor/zpro_dut.dve`
- Add the EDIF definition of the SDRAM memory
  - Right-click on "EDIF Sources" property and select "Add EDIF Sources …"
  - In the file browser, select the file located in: `ZSDRAM.3.1/64Mb/x32/edif/zsdram_64Mb_2x32.edf`
- Check if the job scheduling preferences are still ok for the full compilation
- Run the full compilation:
  - Select "Back-End : default" as target
  - Click on the "Compile" button
- Let the compilation finish, then quit `zCui`

Agenda

- Back-end compilation
- `zCui` Preferences
- Memory modeling
- Debug
- `zTime` timing analysis

Memory modeling
Introduction

- ZeBu memory types
  - Inside FPGAs
    - Registers
    - Distributed memory
      - Based on Xilinx RAMLUT
    - Block RAM
      - Based on Xilinx BRAM
  - On the ZeBu board
    - Known as ZRM
      - Based on RLDRAM or DDR2

Memory modeling Introduction

- Memories are generated by either of:
  - the synthesizer
  - the `zMem` ZeBu memory model generator
  - user instantiations of Xilinx primitives (for registers, RAMLUT and BRAM) and ZeBu memory models (for ZRM) in the RTL code
- `zMem` takes a TCL script as memory descriptor and generates ZeBu memory models from it
  - The TCL script is a user input in most cases
    - In this manual flow, the user has to create the TCL script and replace RTL memory models by `zMem` generated models
  - zFAST can do memory inference and generate the TCL script automatically. It also instantiates the generated memory models
- Only models generated by `zMem` (either automatically or manually) can give accessibility at runtime

Memory modeling
Introduction

		Registers	RAMLUT	BRAM	ZRM
zMem			✓	✓	✓
zFAST	zMem inference		✓	✓	✓
	zMem manual		✓	✓	✓
	direct	✓			
3rd party synthesizers	zMem inference				
	zMem manual		✓	✓	✓
	direct	✓	✓	✓	

Note: Runtime accessibility is only provided wherever cell background is orange

Memory modeling Introduction

	Registers	RAMLUT	BRAM	RLDRAM	DDR2
Capacity guidance	<kb	<10kb	<500kb	>500kb	>500kb
# of instances	200k/FPGA	100k/FPGA	286/FPGA	Module dependant	Module dependant
Size	1b	64b	36Kb	256/128MB	4/2GB
# of ports		128	128	8	8
Synchronous	✓	✓	✓	✓	✓
Asynchronous	✓	✓	✓		
Speed	+++	+++	+++	++	+
Runtime accessibility	Will be visible as individual registers	zMem	zMem	✓	✓

Memory modeling
How to write a `zMem` TCL script

- `zMem` is the ZeBu memory model generator
- It is based on TCL scripts and offers extra commands:

Tcl script Example

`memory new my_memory auto`	Define a new memory called my_memory
`memory depth  4096`	Depth of my_memory is 4096
`memory width  22`	Width of my_memory is 22
`memory add_port  port1  rw`	Define a read/write port on my_memory
`memory set_rw_mode  port1  readafterwrite`	Define the read/write port behavior
`memory_port port1 di    data_in`	Input data bus definition
`memory_port port1 we    write   high`	Write enable definition
`memory_port port1 clk   clock   posedge`	Clock definition
`memory_port port1 do    data_out`	Output data bus definition
`memory_port port1 ce    enable  high`	Clock enable definition
`memory_port port1 addr  address`	Address bus definition
`memory generate`	Generates the defined memory

Memory modeling
Running zMem

- To run zMem type:
  `zMem my_script.tcl`
- This will produce the following files:

`zMem.log`	zMem log file
`my_memory.v`	Verilog behavioral model
`my_memory.vhd`	VHDL behavioral model
`my_memory_bbx.v`	Verilog black box for synthesis
`my_mermory_bbx.vhdl`	VHDL black box for synthesis
`my_memory_gates_ins.v`, `my_memory_gates.v`	Verilog gate-level netlist
`my_memory.edf.gz`	ZeBu memory model

- Note: if used with RTL Front-End or zFAST in `zCui`, `zMem` is launched automatically, black boxes are generated and used during synthesis, ZeBu models are later merged during back-end phase, just add TCL script in the "Memory Source" section of the `zCui` project

Memory modeling
Memory mapping

- Each memory instance in the design source code must be modeled. There are two cases:
  - The original model is synthesizable
    - If using zFAST, registers or `zMem` memories will be automatically inferred
    - If not using zFAST, the synthesizer will infer register, RAMLUT or BRAM without runtime accessibility
    - For ZRM or other memory types with accessibility do a manual mapping using `zMem`
  - The original model is not synthesizable
    - Map it manually using `zMem`
- `zMem` generated models will need to be replaced in the original design source code
- Models manually mapped using `zMem` should be verified for correctness
  - Replace the original model by the `zMem` behavioral model and run a simulation if possible

Memory modeling
Memory mapping process

- The remapping process involves 5 main steps
  - Identify memories in the design
  - Select appropriate ZeBu memory type
  - Generate ZeBu memories using zMem
  - Verify remapping in simulation
  - Compile the ZeBu design

Memory Modeling
Identify memories in the design

- This step is manual
  - Look at the design source code
  - Take notes of
    - Memory module/entity name
    - Memory size
    - Memory I/O interface
    - Synchronous/Asynchronous type
    - Read/Write cycles type
- Also, get help from the tools:
  - for non-synthesizable memories, you will get an error message from the synthesizer
  - for synthesizable memories, the RTL Front-End will report memory existence and size when detected

Memory modeling
Select the most appropriate memory type

- For each memory instance in the design select the most appropriate ZeBu hardware memory type, according to:
  - Memory size
  - Memory port structure
  - Memory type
  - Memory speed

Memory modeling
Running zMem

- `zMem my_script.tcl`
- Will produce the necessary models
- Make a subdirectory and works from there to launch `zMem`

Memory modeling
Verify mapping correctness

- Use the behavioral models generated by `zMem`
- Replace the original design memory by the behavioral model
- Simulate the design
- Verify that the design still work after memory remapping
- Do this before proceeding with ZeBu compilation!

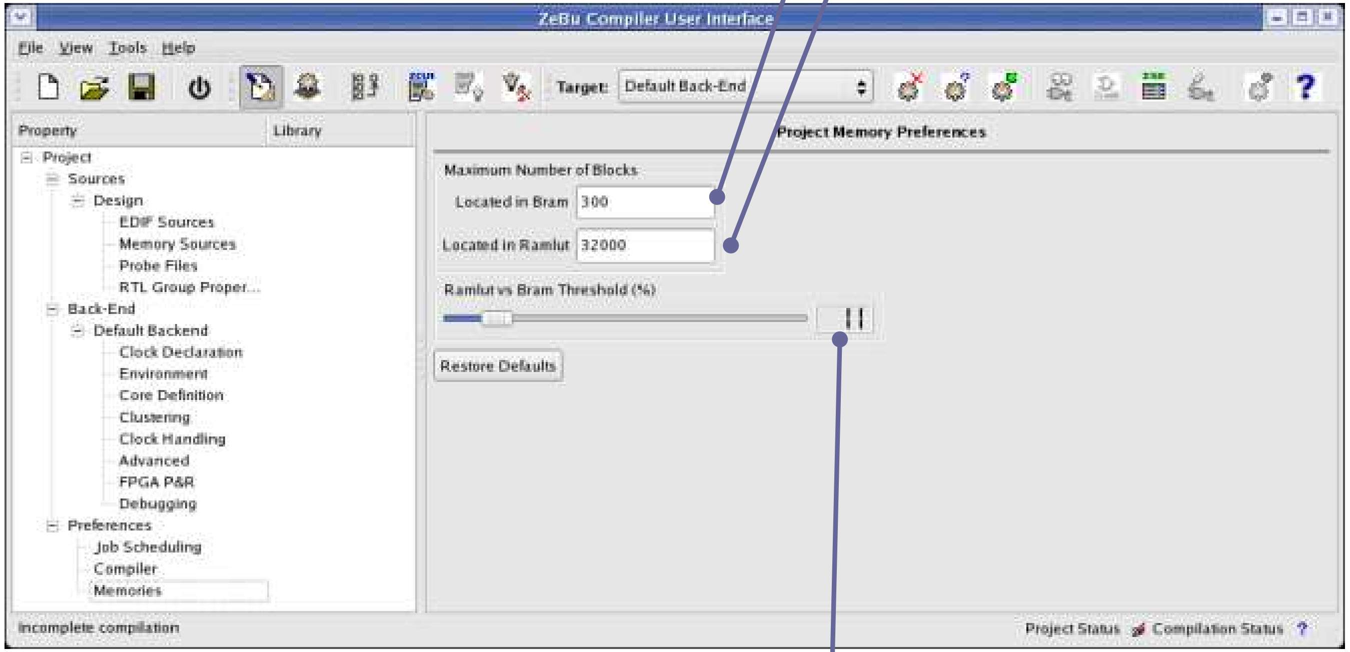
Memory modeling Compilation

- Once the memory mapping is verified
  - Add the TCL script to the `zCui` project
  - Launch `zCui` compilation

Memory modeling Compilation

Threshold for BRAM

Threshold for RAMLUT



ZeBu Compiler User Interface

Project Memory Preferences

Maximum Number of Blocks

Located in Bram: 300

Located in Ramlut: 32000

Ramlut vs Bram Threshold (%)

Restore Defaults

Incomplete compilation

BRAM occupancy threshold

Memory modeling
Compilation

- Threshold for BRAM
  - Memory will not be a BRAM if more than *n* BRAM blocks are needed to implement it
- Threshold for RAMLUT
  - Memory will not be a RAMLUT if more than *n* RAMLUT blocks are needed to implement it
- RAMLUT vs BRAM threshold
  - If the measured BRAM memory occupancy is above the “RAMLUT vs BRAM threshold”, the memory will be implemented as BRAM (RAMLUT otherwise)

Memory modeling
Lab 3

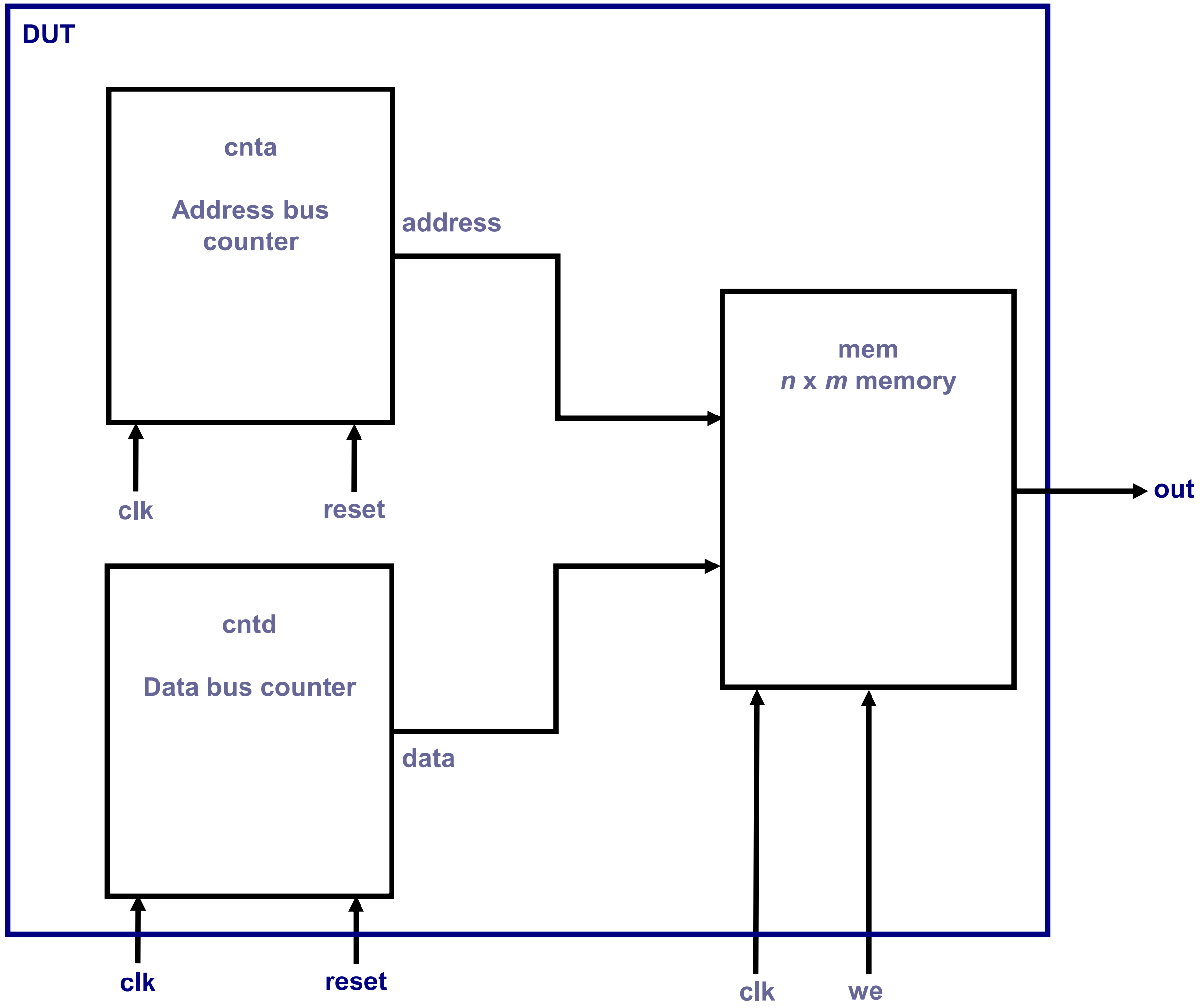
- In this lab you will learn how to map a design memory to a ZeBu memory
- This lab comes in 3 flavors:
  - HDL co-simulation
  - C++ co-simulation
  - SystemC co-simulation
- Select the co-simulation mode you will use the most
  - It does not make much difference for compilation but it implies some differences for runtime

Memory modeling Lab 3

```
Trainee/lab3
|-- C++_cosim
|    `-- zebu.run
|        `-- Makefile
|-- HDL_cosim
|    |-- simu.run
|    |    `-- Makefile
|    `-- zebu.run
|        `-- Makefile
|-- SystemC_cosim
|    `-- zebu.run
|        `-- Makefile
|-- dut
|    |-- TCL
|    `-- verilog
|        |-- counter.v
|        |-- dut.v
|        `-- ram.v
`-- testbench
    |-- c++
    |    `-- testbench.cc
    |-- systemc
    |    `-- testbench.cc
    `-- verilog
        |-- testbench1.v
        `-- testbench2.v
```

Memory modeling
Lab 3

The DUT



Memory modeling
Lab 3

- In this lab, we have a simple design:
- It is made of
  - 2 counters
  - A simple synchronous memory
- One counter controls the address bus of the memory
  - By incrementing the counter, the memory address space is scanned for write or for read
- The other counter controls the data bus of the memory
  - This counter is used to write a different value at each different address of the memory

Memory modeling
Lab 3

- A testbench control the design in order to perform the following operation
  - Reset the two counters
  - Set high the `we` write enable signal of the memory
  - Perform *n* write cycle while incrementing the counters to fill the memory with different values
  - Reset the two counters
  - Set low the `we` write enable signal of the memory
  - Perform *n* read cycle while incrementing the counters and check that the value expected are read
- Notice that a correct execution lead to a normal testbench displayed message at the first read cycle:
  - `# out = xx, expect = ff`

Memory modeling
Lab 3

- Analyze the DUT source file located in `dut/verilog` and find the memory
  - Take note of its key parameters such as:
    - Module name
    - Size
    - Synchronisation
    - I/O port names
- Build a TCL ZeBu memory description file based on previous parameters
  - Put it in the `dut/TCL` directory and name it `ram.tcl`
  - You can use the template on the following slide as a guideline
  - Do not forget to select the appropriate ZeBu memory type

Memory modeling
Lab 3

```
memory new <name> <type>
memory depth <size>
memory width <size>
memory type sync

memory add_port A rw
memory set_rw_mode A readbeforewrite
memory_port A addr <name>
memory_port A di <name>
memory_port A do <name>
memory_port A we <name>
memory_port A clk <name> posedge

memory generate
```

Memory modeling
Lab 3

- Go into the appropriate directory according to your co-simulation mode, either of:
  - HDL_cosim
  - C++_cosim
  - SystemC_cosim
- Try to generate ZeBu memory models using `zMem`:
  - Create an empty directory and go into it:
    ```
    mkdir mem
    cd mem
    ```
  - Run `zMem` on the `ram.tcl` file. This will check the syntax and create ZeBu models
    ```
    zMem ../../dut/TCL/ram.tcl
    ```
  - Look at the created models
  - Go back one directory up
    ```
    cd ..
    ```

Memory modeling
Lab 3

- Skip this slide if you are not doing the HDL co-simulation lab
- Run the design simulation
  - Go into the `simu.run` directory and type
    `make <simulator name>`
- Run the design simulation with the zMem memory
  - Stay in the `simu.run` directory type
    `make <simulator name>_zMem`
  - You should have the same behavior, otherwise fix the `ram.tcl` file
- Go back one directory up
  `cd ..`

Memory modeling
Lab 3

- Prepare a `zCui` project:
  - Select the DUT source files:
    - Right-click on the RTL group and add the TWO files `counter.v` and `dut.v` from the `../dut/verilog` directory. Make sure you omit the `ram.v` file as it will be modeled using the `ram.tcl` file
    - Left-click on the RTL group, in the “Main” tab, set the top DUT module name and select your synthesizer
    - Right-click on the “Memory Sources” property and add `../dut/TCL/ram.tcl` as memory source file
    - Left-click on the “Design” property and set the DUT top module name
  - Select the appropriate target hardware configuration file for your ZeBu system
    - Left-click on the “Back-End : default” property and use the file browser to select the target hardware configuration file

Memory modeling
Lab 3

- Set the clock name:
  - Left-click on the “Clock Declaration” property and add a clock declaration
- Set the environment:
  - Left-click on the “Environment” property
  - Select the “Auto Generated” DVE file mode
  - Select the generator type according to the co-simulation mode you selected for this lab:
    - HDL co-simulation: “Event-Based: HDL Cosimulation”
    - C++ co-simulation: “Cycle-based: C/C++ Cosimulation”
    - SystemC co-simulation : “Event-Based: HDL Cosimulation”
  - Select the wrapper language according to the co-simulation mode you selected for this lab

Memory modeling
Lab 3

- Enable software access:
  - Left-click on the “Debugging” property
  - Turn on the “Enable BRAM Read & Write / Write Register / Save & Restore” checkbox
- Set the job scheduling preferences
  - Left-click on the “Job Scheduling” property
  - Fill the form in accordance with your network configuration
- Save the project
- Launch the compilation

Memory modeling
Lab 3

- Go into the `zebu.run` directory and launch the emulation:
  - For HDL co-simulation
    `make` *`<simulator>`*
  - For C++ co-simulation
    ```
    make
    ./testbench
    ```
  - For SystemC co-simulation
    ```
    make
    ./testbench
    ```
- Verify that you have the correct behavior

Memory modeling Lab 3

- Check for the presence and content of a file named `dumpfile`. It is dumped by the co-simulation testbench at the end of the run. Look in the testbench to find it for reference:
  - For HDL co-simulation:
    `$ZEBU_writemem("dumpfile","dut.m0");`
  - For C++ co-simulation:
    `dut.m0->storeTo("dumpfile");`
  - For SystemC co-simulation:
    `dut.m0->storeTo("dumpfile");`

Agenda

- Back-end compilation
- `zCui` Preferences
- Memory modeling
- Debug
- `zTime` timing analysis

Debug
Trace memory

- **Trace memory must be instantiated in the DVE file**

Instance name given by the user

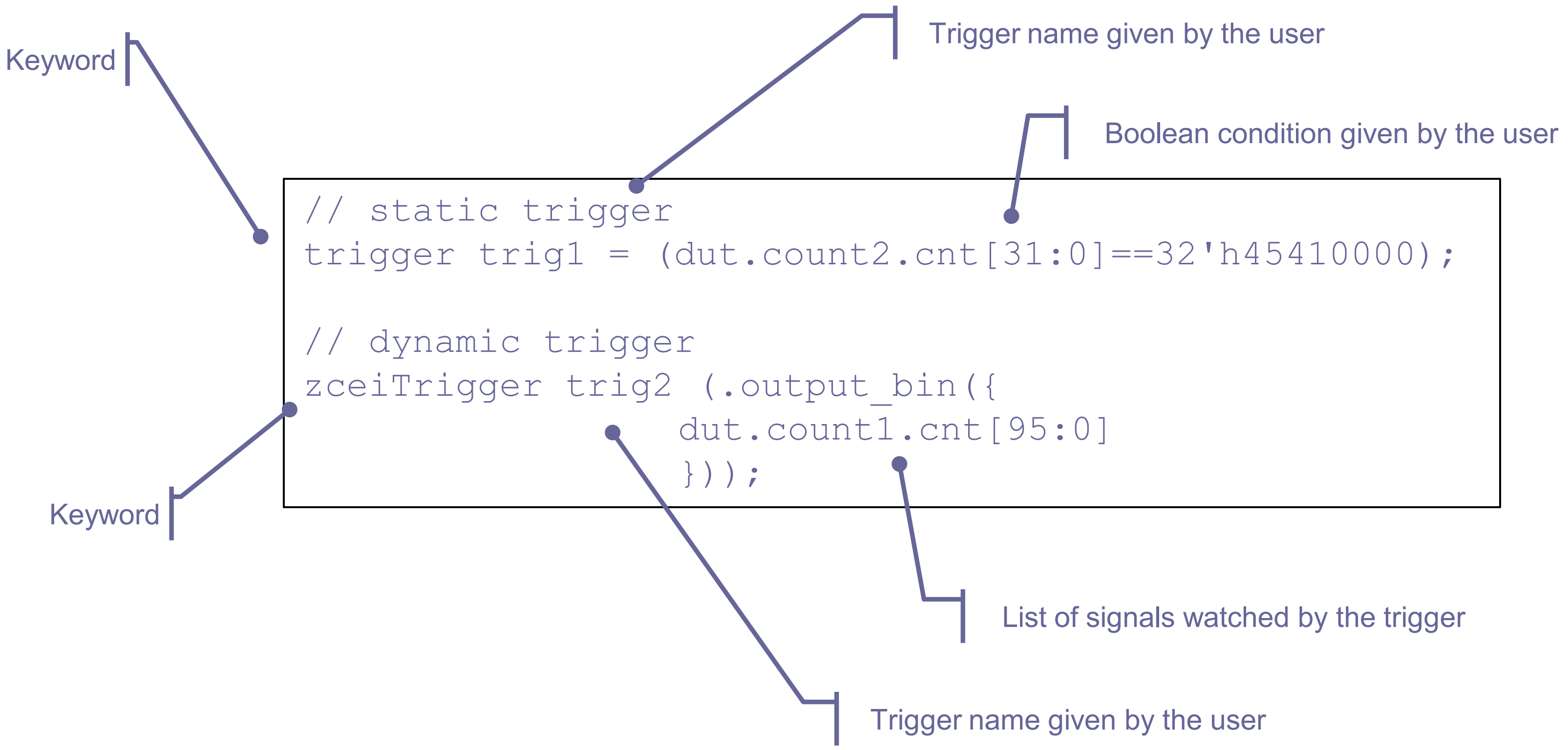
Keyword

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

List of traced signals

Debug Triggers

- **Trigger must be declared in the DVE file**



```
// static trigger
trigger trig1 = (dut.count2.cnt[31:0]==32'h45410000);

// dynamic trigger
zceiTrigger trig2 (.output_bin({
                 dut.count1.cnt[95:0]
                 }));
```

Keyword
Trigger name given by the user
Boolean condition given by the user
Keyword
List of signals watched by the trigger
Trigger name given by the user

Agenda

- Back-end compilation
- `zCui` Preferences
- Memory modeling
- Debug
- `zTime` timing analysis

`zTime` Timing Analysis Introduction

- `zTime` is a ZeBu system-level static timing analysis tool
- It runs after system-level place & route
- It takes into account:
  - Inter-FPGA communication (multiplexing)
  - Intra-FPGA combinational paths, including hops
  - Presence of memories (BRAM, ZRM)
- It does not take into account:
  - Intra-FPGA logic
- It computes:
  - Maximum emulation frequency
  - Different runtime skew and filter delay parameters
- Computed values will be used by default at runtime
  - They can be changed by user

`zTime` Timing Analysis
Declaring false paths

- **Warning**: `zTime` is usually pessimistic
  - Pessimism can be reduced by declaring false paths
  - In the `zCui` "Advanced panel", in the "`zTime` constraint file" field add a file with following type of instructions:

```
set_false_path_from -port=reset

set_false_path_to -port=dummy -ins=F2_0_0
```

`zTime` Timing Analysis
Result files

- Timing analysis summary can be found in: `zcui.work/zebu.work/zTime.log`
- HTML reports for critical paths will be found in: `zcui.work/zebu.work` directory
  - `ztime_out_paths.html`
  - `ztime_clock_out_paths.html`
  - `ztime_filter_out_paths.html`

zTime Timing Analysis zTime.log summary file

```
#    step REPORT : #----------------------------------------------------------------------------------#
#    step REPORT :   Longest inter-fpga filter path delay is : 38 ns
#    step REPORT :   Longest inter-fpga clock path delay is : 58 ns
#    step REPORT :   zClockSkewTime runtime parameter is : 58 ns
#    step REPORT :   Max pessimistic delay (routing and clock skew) is : 106 ns
#    step REPORT :   Critical routing path delay : 48 ns
#    step REPORT :   . Constant part    : 30 ns
#    step REPORT :   . Multiplexed part : 18 ns
#    step REPORT :   Longest memory period is : 150 ns
#    step REPORT :   No flexible probes found
#    step REPORT :   Driver clock frequency is limited by memories
#    step REPORT :   The theoretical frequency using default settings and ignoring clock skew is 6666 Khz
#    step REPORT : #----------------------------------------------------------------------------------#
```

zTime Timing Analysis
Critical filter paths

Delay	Fpga	From	To	Details											
				Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
38 ns	2	rstn	U0_M0_F0.rstn	IF	10 ns	10 ns						rstn		rstn	
					18 ns		4	Y							
				M0/F03	10 ns	38 ns			Y		T	U0_M0_F0/rstn	core3	rstn	zpro_dut.clk_div_0.sqnod239.I1
				Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
38 ns	2	menable	U0_M0_F0.menable	IF	10 ns	10 ns						menable		menable	
					18 ns		4	Y							
				M0/F03	10 ns	38 ns			Y		T	U0_M0_F0/menable	core3	menable	zpro_dut.clk_div_0.sqnod203.I0
				Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
38 ns	2	U0_M0_F0.clk	U0_M1_F0.ztbsplt_...	M0/F03	10 ns	10 ns						U0_M0_F0/clk	core3	ztbsplt_zpro_dut04	zpro_dut.clk_div_0.sqnod239.O
					18 ns		4	Y							
				M1/F00	10 ns	38 ns					T	U0_M1_F0/ztbsplt_zpro_dut037	core1	ztbsplt_zpro_dut04	zpro_dut.zebu_filter_ztbsplt_zpro_dut037.bypass.I0
				Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
20 ns	2	U0_M0_F0.clk	U0_M0_F1.ztbsplt_...	M0/F03	10 ns	10 ns						U0_M0_F0/clk	core3	ztbsplt_zpro_dut04	zpro_dut.clk_div_0.sqnod239.O
					0 ns			Y							
				M0/F02	10 ns	20 ns			Y		T	U0_M0_F1/ztbsplt_zpro_dut03	core0	ztbsplt_zpro_dut04	zpro_dut.zebu_filter_ztbsplt_zpro_dut03.bypass.I0
				Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
20 ns	2	U0_M0_F0.clk	U0_M0_F3.ztbsplt_...	M0/F03	10 ns	10 ns						U0_M0_F0/clk	core3	ztbsplt_zpro_dut04	zpro_dut.clk_div_0.sqnod239.O
					0 ns			Y							
				M0/F01	10 ns	20 ns			Y		T	U0_M0_F3/ztbsplt_zpro_dut037	core2	ztbsplt_zpro_dut04	zpro_dut.zebu_filter_ztbsplt_zpro_dut037.bypass.I0

zTime Timing Analysis Critical clock paths

Delay	Fpga	From	To	Details											
				Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
38 ns	2	rstn	U0_M0_F0.rstn	IF	10 ns	10 ns						rstn		rstn	
					18 ns		4	Y							
				M0/F03	10 ns	38 ns			Y		T	U0_M0_F0/rstn	core3	rstn	zpro_dut.clk_div_0.sqnod239.I1
38 ns	2	menable	U0_M0_F0.menable	IF	10 ns	10 ns						menable		menable	
					18 ns		4	Y							
				M0/F03	10 ns	38 ns			Y		T	U0_M0_F0/menable	core3	menable	zpro_dut.clk_div_0.sqnod203.I0
20 ns	2	U0_M0_F0.clk	U0_M0_F1.ztbsplt_...	M0/F03	10 ns	10 ns						U0_M0_F0/clk	core3	ztbsplt_zpro_dut04	zpro_dut.clk_div_0.sqnod239.O
					0 ns			Y							
				M0/F02	10 ns	20 ns			Y		T	U0_M0_F1/ztbsplt_zpro_dut03	core0	ztbsplt_zpro_dut04	zpro_dut.zebu_filter_ztbsplt_zpro_dut03.bypass.I0
20 ns	2	U0_M0_F0.clk	U0_M0_F3.ztbsplt_...	M0/F03	10 ns	10 ns						U0_M0_F0/clk	core3	ztbsplt_zpro_dut04	zpro_dut.clk_div_0.sqnod239.O
					0 ns			Y							
				M0/F01	10 ns	20 ns			Y		T	U0_M0_F3/ztbsplt_zpro_dut037	core2	ztbsplt_zpro_dut04	zpro_dut.zebu_filter_ztbsplt_zpro_dut037.bypass.I0

zTime Timing Analysis
Critical output paths

Delay	Fpga	From	To	Details											
48 ns	3	U0_M0_F3.ztbsplt_...	U0_M1_F0.ztbsplt_...	Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
				M0/F01	10 ns	10 ns						U0_M0_F3/ztbsplt_lb_dout[13]_2_ztbsplt_lb_dout[13]	core2	ztbsplt_lb_dout[13]_2_ztbsplt_I205	zpro_dut.zpro_0.\\eVe_cluster[2].zcluster.sqnod822.O
					18 ns		4	Y							
				M1/F02	10 ns	28 ns									
					0 ns			Y							
				M1/F00	10 ns	48 ns						U0_M1_F0/ztbsplt_I205_2_ztbsplt_I205	core1	ztbsplt_lb_dout[13]_2_ztbsplt_I205	zpro_dut.zpro_0.sqnod466.I2
48 ns	3	U0_M0_F3.ztbsplt_...	U0_M1_F0.ztbsplt_...	Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
				M0/F01	10 ns	10 ns						U0_M0_F3/ztbsplt_lb_dout[31]_2_ztbsplt_lb_dout[31]	core2	ztbsplt_lb_dout[31]_2_ztbsplt_I2028	zpro_dut.zpro_0.\\eVe_cluster[2].zcluster.sqnod804.O
					18 ns		4	Y							
				M1/F03	10 ns	28 ns									
					0 ns			Y							
				M1/F00	10 ns	48 ns						U0_M1_F0/ztbsplt_I2028_2_ztbsplt_I2028	core1	ztbsplt_lb_dout[31]_2_ztbsplt_I2028	zpro_dut.zpro_0.sqnod448.I2
48 ns	3	U0_M0_F3.ztbsplt_...	U0_M1_F0.ztbsplt_...	Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
				M0/F01	10 ns	10 ns						U0_M0_F3/ztbsplt_lb_dout[15]_2_ztbsplt_lb_dout[15]	core2	ztbsplt_lb_dout[15]_2_ztbsplt_I209	zpro_dut.zpro_0.\\eVe_cluster[2].zcluster.sqnod820.O
					18 ns		4	Y							
				M1/F02	10 ns	28 ns									
					0 ns			Y							
				M1/F00	10 ns	48 ns						U0_M1_F0/ztbsplt_I209_2_ztbsplt_I209	core1	ztbsplt_lb_dout[15]_2_ztbsplt_I209	zpro_dut.zpro_0.sqnod464.I2
48 ns	3	U0_M0_F3.ztbsplt_...	U0_M1_F0.ztbsplt_...	Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
				M0/F01	10 ns	10 ns						U0_M0_F3/ztbsplt_lb_dout[3]_2_ztbsplt_lb_dout[3]	core2	ztbsplt_lb_dout[3]_2_ztbsplt_I3	zpro_dut.zpro_0.\\eVe_cluster[2].zcluster.sqnod832.O
					18 ns		4	Y							
				M1/F03	10 ns	28 ns									
					0 ns			Y							
				M1/F00	10 ns	48 ns						U0_M1_F0/ztbsplt_I3_2_ztbsplt_I3	core1	ztbsplt_lb_dout[3]_2_ztbsplt_I3	zpro_dut.zpro_0.sqnod476.I3
48 ns	3	U0_M0_F3.ztbsplt_...	U0_M1_F0.ztbsplt_...	Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
				M0/F01	10 ns	10 ns						U0_M0_F3/ztbsplt_lb_dout[5]_2_ztbsplt_lb_dout[5]	core2	ztbsplt_lb_dout[5]_2_ztbsplt_I200	zpro_dut.zpro_0.\\eVe_cluster[2].zcluster.sqnod830.O
					18 ns		4	Y							
				M1/F03	10 ns	28 ns									
					0 ns			Y							
				M1/F00	10 ns	48 ns						U0_M1_F0/ztbsplt_I200_2_ztbsplt_I200	core1	ztbsplt_lb_dout[5]_2_ztbsplt_I200	zpro_dut.zpro_0.sqnod474.I2
48 ns	3	U0_M0_F3.ztbsplt_...	U0_M1_F0.ztbsplt_...	Fpga	Delay	Arrival	X	Lvds	Ck	zDly	zFlt	Port	zCore	Wire	Alias
				M0/F01	10 ns	10 ns						U0_M0_F3/ztbsplt_lb_dout[4]_2_ztbsplt_lb_dout[4]	core2	ztbsplt_lb_dout[4]_2_ztbsplt_I2	zpro_dut.zpro_0.\\eVe_cluster[2].zcluster.sqnod831.O
					18 ns		4	Y							
				M1/F03	10 ns	28 ns									
					0 ns			Y							
				M1/F00	10 ns	48 ns						U0_M1_F0/ztbsplt_I2_2_ztbsplt_I2	core1	ztbsplt_lb_dout[4]_2_ztbsplt_I2	zpro_dut.zpro_0.sqnod475.I2