WhizniumDBE Code Generation/Iteration Service

The Device Builder's Edition For FPGA-Based Systems

Quick facts

- WhizniumDBE extends the principles of automated code generation and iteration as proven by WhizniumSBE to the world of programmable logic (FPGA) register transfer level (RTL) projects.
- Devices supported by WhizniumDBE include Xilinx's FPGA's and AP SoC's. A number of relevant use cases are available online, ranging from low-end / legacy Spartan3E and Artix7 systems to high-end latest-generation Kintex7 and Zynq boards.
- WhizniumDBE helps modularize FPGA's into virtual *controllers* with functionality-specific command sets. Code generation covers both the host in terms of a C++ access library and the FPGA's VHDL code down to state-machine level. A second focus besides command execution are bulk data transfers.
- The developer can choose between "easy" and "full" implementations, the former being especially lightweight on FPGA resources, the latter permitting non-blocking command execution and cascaded board hierarchies.
- Deployment options for WhizniumDBE include an on-premise container-based solution, or cloud-based / pay-per-use as an alternative.
- Power users can profit from WhizniumDBE's template functionality e.g. for developing products with derivative options. Custom parametrized template RTL modules can be defined and included as add-ons into the standard WhizniumDBE workflow.

Scope

Development with WhizniumDBE covers Register Transfer Level (RTL) source code for Xilinx FPGA-based devices in VHDL, along with C++ connectivity code which links these devices to their Linux host embedded system. Typical applications include high bandwidth hardware with sensitive timing requirements, such as ADC readout, and access to devices with low-level bus protocols, e.g. LVDS cameras. WhizniumDBE provides both source code components in synthesize/compile-ready fashion already from the first iteration the recpective project.

Device architecture

A device can consist of several *units* or µC/FPGA-based PCB's. To form a *system*, they are connected in a hierarchical way, as illustrated in Figure 1. In a *system*, exactly one (root) *unit* has direkt link to the host embedded system. All sub-*units* are reachable via their respective host *units* only.

Depending on the requirements for FPGA-level parallelism and available footprint, each *unit* can be specified to be implemented with either the "easy" or the "full" model.

Both models have in common that there is a communication protocol in place between host and sub-*unit* which enables passing of commands with invocation and return parameters on one hand, and performing bulk data transfers on

Table 1: Xilinx Spartan3E-based *unit* `dcx3`'s *controllers*

Controller	Task
adxl	accelerometer read-out (ADXL345 via SPI)
align	alignment laser control (MAX5383 DAC via SPI)
led	high power LED control (LT3474 PWM)
lwiracq	LWIR camera data interface (FLIR Tau2 via LVDS)
lwirif	LWIR camera control interface (FLIR Tau2 via UART)
phiif/thetaif	phi/theta axis controller interface (axis2 µC PCB via SPI)
pmmu	PMMU with 2MB external SRAM interface (IS61WV20488)
qcdif	QCD module interface (icacam2 FPGA PCB via SPI)
shfbox	SPI/PWM connectior shuffle box
state	state monitor
tkclksrc	10kHz clock source
trigger	trigger source

the other hand. This protocol features CRC-16 checksums and is implemented for the AXI, PCIe, SPI and UART hardware interfaces. The AXI option allows to design for Xilinx's Zynq SoC's.

Any RTL project is defined hierarchically with the top module providing connectivity to physical FPGA pins. This architecture is reflected within WhizniumDBE *units* as well, only that by the specification of various module types, added functionality, e.g. in terms of wiring, is provided automatically.

The *controller* is the most relevant module type, it is user-defined to group related functionality ; Table 1 lists the *controllers* for `dcx3`, a *unit* performing readout of a VGA thermal imager via LVDS along with a number of forwarding/auxiliary functionalities. For each *controller*, a command set can be defined – here is where the fundamental difference lies between "easy" and "full" models.

An example of the "easy" model is given on the right hand side of Figure 1, in the `icm2` *unit*. In this case, the automatically generated SPI host interface module `hostif` includes a finite state machine (FSM) which upon reception of a command from the host *unit* `dcx3` sets registers shared with the respective target *controller*, and uses separate req/ack signals to perform a handshake. Commands with return parameters are also possible, here the *controller* sets the registers to be read back by the host interface to the host. In Figure 1, this is demonstrated with the command `(ntc) = getNtc()` which performs ADC readout on request and passes the ADC reading as return value `ntc`.

While the "easy" model is straightforward and benefits from a low FPGA footprint, it has the obvious limitation of being "one command at a time" which may mitigate the FPGA's advantages of parallelism and master clock cycle accuracy.

The "full" model is dedicated to non-blocking, concurrent command execution. An example is the `dcx3` *unit* shown on the left hand side of Figure 1. In the "full" model, an 8bit command bus `cmdbus` shared between all controllers is inserted. Handshake is ensured using *controller*-specific req/ack/rdy signals. This solution comes with an automatically generated command FSM per controller. The "full" model enables delayed and multiple returns per command invocation.

From the host point of view, command invocation and command returns are buffered through the `cmdinv` and `cmdret` memory modules. Constant polling from the host is required in order to receive updates of the "full"-model unit's state.

To complete the picture on the difference between "easy" and "full" models, in Figure 2 relevant code excerpts on both the RTL and embedded system sides are presented.

A derivate of *controllers* are *forwarding controllers* which serve as the host of their sub-*units* e.g. `dcx3.qcdif` in Figure 1. Their internal wiring is derived automatically from the hardware interface and model chosen for their respective sub-*unit*.

Inter-module buffers are a second important module type which is used to transfer bulk data between *controllers* (e.g. `inbuf0LwiracqToPmmu` in Figure 1), but more importantly between the host and *controllers*. *Inter-module buffers* can be specified by direction and size – they are implemented as dual-port BlockRAM.

WhizniumDBE's remaining module types include *manufacturer primitives* (e.g. clock and I/O buffers), *manufacturer cores* to incorporate pre-existing IP, and *wrappers* – these can be used to generate testbenches replacing physical FPGA pins with debug functionality. Finally, a module category "other" is made available for anything that does not fit into the above types.

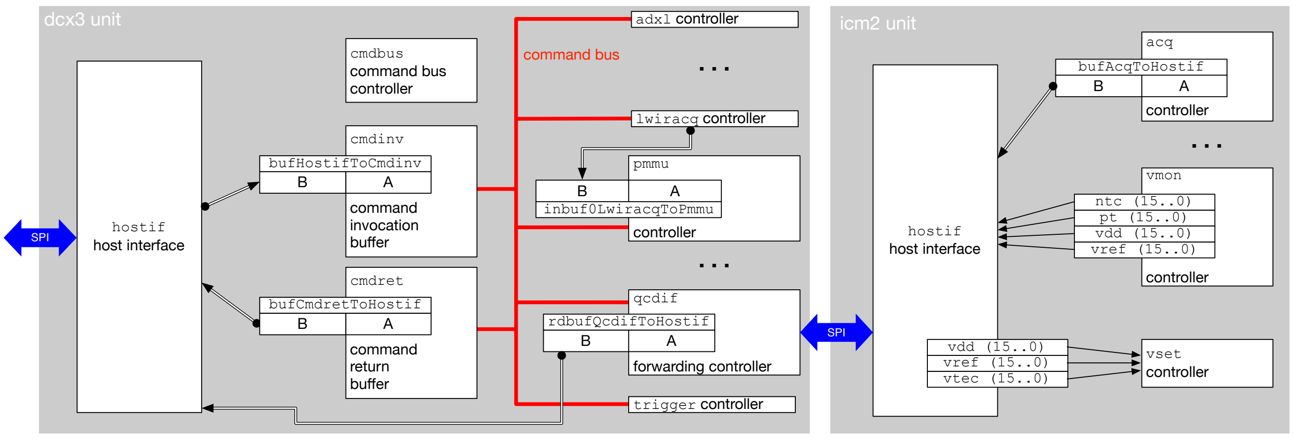


Figure 1: Example device idhw with details of the "full"-model *unit* dcx3 and the "easy"-model *unit* icm2

Detailed RTL design

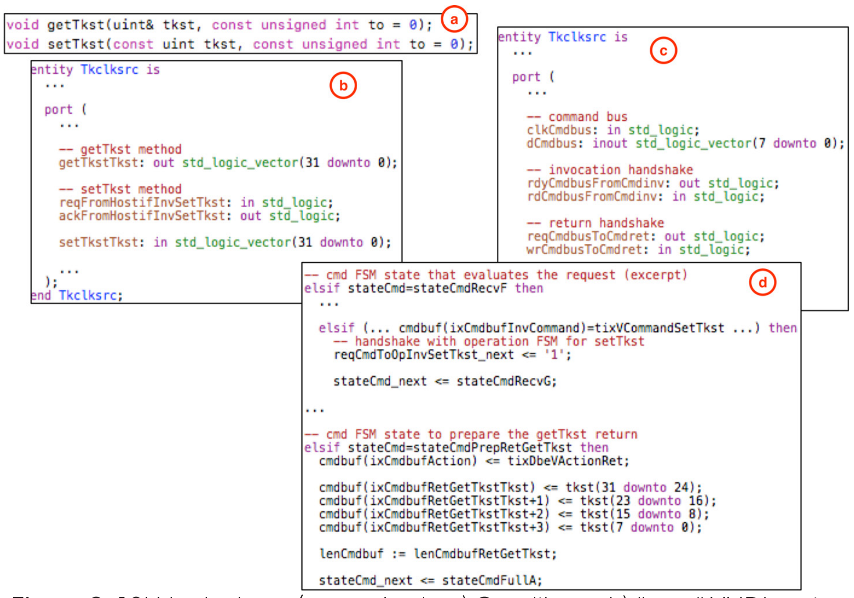
WhizniumDBE's in-detail RTL model reflects the VHDL standard. Accordingly, within modules, *generics*, *ports*, *signals*, *processes* and *variables* can be defined.

Furthermore, WhizniumDBE provides added-value code generation for FSM's (a sub-type of processes). FSM's can be defined along with their states and multi-layered stepping conditions, which are then implemented automatically. Also, handshakes between FSM's based on their states can be specified, and signal values can be made dependent on logical combinations of FSM states.

The physical layer is covered by specifying *pins* grouped into (I/O) *banks*. WhizniumDBE takes care of writing the corresponding constraints files.

Development workflow

WhizniumDBE relies on the proven principles of WhizniumSBE for results which are clearly laid out and easy to maintain. Per project, it uses two text-based model input files, *basic* and *detailed model description*, the contents of which are listed in Table 2. Meaningful naming conventions with high recognition value are implemented throughout: for example, the 10kHz clock source *controller* `tkclksrc` of the *unit* `dcx3` of the device `idhw` is addressed as `CtrIdhwDcx3Tkclksrc` in C++ code. A handshake between the command (`cmd`) and operation (`op`) FSM's within the controller `acq` is represented by the signals `{req/ack}CmdToOpInvGetFrame`.



```
void getTkst(uint& tkst, const unsigned int to = 0);
void setTkst(const uint tkst, const unsigned int to = 0);
```

```
entity Tkclksrc is
  ...

  port (
    ...

    -- getTkst method
    getTkstTkst: out std_logic_vector(31 downto 0);

    -- setTkst method
    reqFromHostifInvSetTkst: in std_logic;
    ackFromHostifInvSetTkst: out std_logic;

    setTkstTkst: in std_logic_vector(31 downto 0);

    ...
  );
end Tkclksrc;
```

```
entity Tkclksrc is
  ...

  port (
    ...

    -- command bus
    clkCmdbus: in std_logic;
    dCmdbus: inout std_logic_vector(7 downto 0);

    -- invocation handshake
    rdyCmdbusFromCmdinv: out std_logic;
    rdCmdbusFromCmdinv: in std_logic;

    -- return handshake
    reqCmdbusToCmdret: out std_logic;
    wrCmdbusToCmdret: in std_logic;
```

```
-- cmd FSM state that evaluates the request (excerpt)
elsif stateCmd=stateCmdRecvF then
  ...

  elsif (... cmdbuf(ixCmdbufInvCommand)=tixVCommandSetTkst ...) then
    -- handshake with operation FSM for setTkst
    reqCmdToOpInvSetTkst_next <= '1';

    stateCmd_next <= stateCmdRecvG;

...

-- cmd FSM state to prepare the getTkst return
elsif stateCmd=stateCmdPrepRetGetTkst then
  cmdbuf(ixCmdbufAction) <= tixDbeVActionRet;

  cmdbuf(ixCmdbufRetGetTkstTkst) <= tkst(31 downto 24);
  cmdbuf(ixCmdbufRetGetTkstTkst+1) <= tkst(23 downto 16);
  cmdbuf(ixCmdbufRetGetTkstTkst+2) <= tkst(15 downto 8);
  cmdbuf(ixCmdbufRetGetTkstTkst+3) <= tkst(7 downto 0);

  lenCmdbuf := lenCmdbufRetGetTkst;

  stateCmd_next <= stateCmdFullA;
```

Figure 2: 10kHz clock get/set methods: a) C++ library, b) "easy" VHDL entity, c) "full" VHDL entity, d) "full" VHDL cmd FSM

Another concept carried over from WhizniumSBE are *insertion points* (IP's), comments in the respective programming language (e.g. VHDL), which mark the locations of manual code in otherwise automatically maintained source code files.

The full project workflow is illustrated in Figure 3. Its main cycle consists of manually editing the source code tree, adapting the model files, and feeding updated versions of both into WhizniumDBE, which in turn establishes the next iteration of the project.

Template modules

RTL modules are predestined for re-use in multiple projects. WhizniumDBE provides a number of simple modules, such as the UART receiver `uartrx` module as ready-to-use VHDL file.

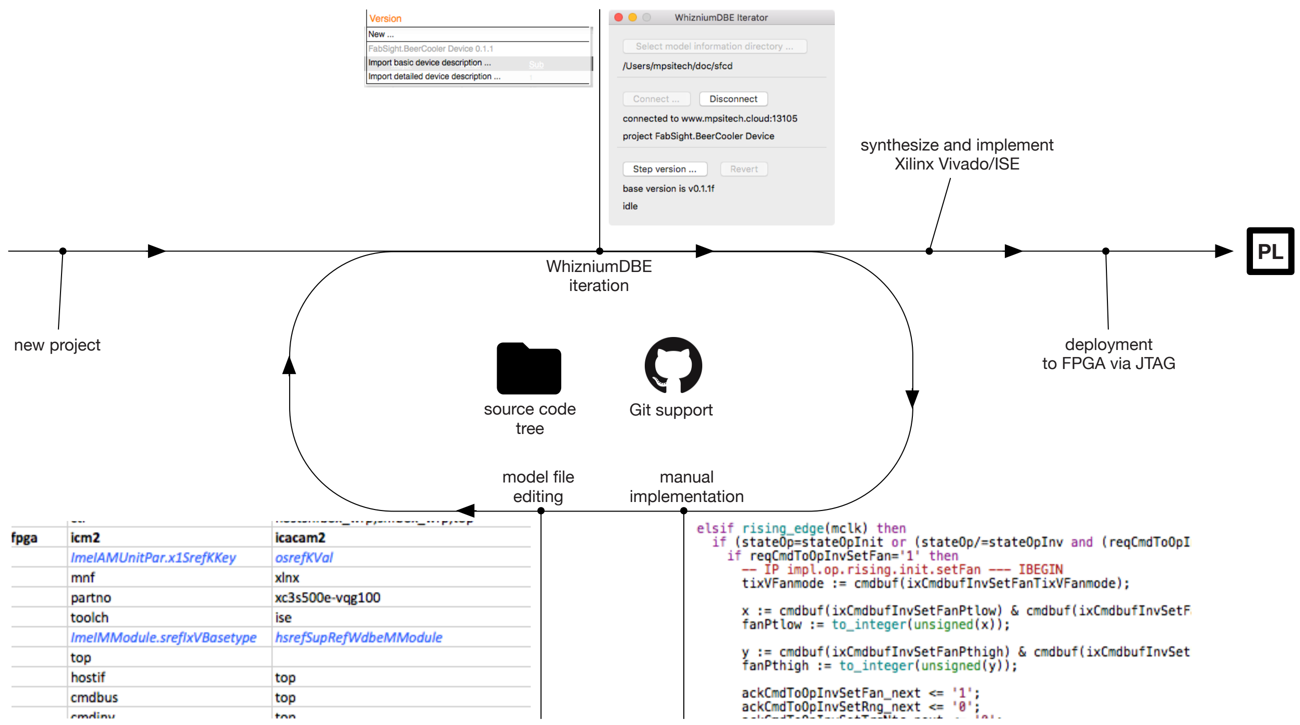
On top of that, virtually any functionality can be "templateified" in WhizniumDBE. This is facilitated by the combination of 1. a VHDL template file store with placeholders and IP's for automated fill-in, 2. the reflection of the RTL design within the WhizniumDBE master database and 3. the possibility of parametrization of templates.

The Page Mapped Memory Unit (PMMU) controller template is a good example: in a project's *basic device description*, the user specifies which data "source" and "drain" modules the PMMU is connected to, along with the parameters of memory, page and TOC size. WhizniumDBE then generates the entire module structure and FSM implementation required, both in terms of master database entries and automatically filled-in IP's.

The option of implementing customer-specific modules for flexible re-use is made available to power users of WhizniumDBE.

Table 2: Model files and their respective content

Model component	Content
Basic device description	systems, targets, units; controllers and inter-module buffers
Detailed device description	banks and pins; commands and errors; generics, ports and signals; processes, variables, FSM's and FSM states



/Users/mpsitech/doc/sfcd

connected to www.mpsitech.cloud:13105

project FabSight.BeerCooler Device

base version is v0.1.1f

idle

fpga	**icm2**	**icacam2**
	ImeIAMUnitPar.x1SrefKKey	*osrefKVal*
	mnf	xlnx
	partno	xc3s500e-vqg100
	toolch	ise
	ImeIMModule.srefIxVBasetype	*hsrefSupRefWdbeMModule*
	top	
	hostif	top
	cmdbus	top

```
elsif rising_edge(mclk) then
  if (stateOp=stateOpInit or (stateOp/=stateOpInv and (reqCmdToOpI
    if reqCmdToOpInvSetFan='1' then
      -- IP impl.op.rising.init.setFan --- IBEGIN
      tixVFanmode := cmdbuf(ixCmdbufInvSetFanTixVFanmode);

      x := cmdbuf(ixCmdbufInvSetFanPtlow) & cmdbuf(ixCmdbufInvSetF
      fanPtlow := to_integer(unsigned(x));

      y := cmdbuf(ixCmdbufInvSetFanPthigh) & cmdbuf(ixCmdbufInvSet
      fanPthigh := to_integer(unsigned(y));

      ackCmdToOpInvSetFan_next <= '1';
      ackCmdToOpInvSetRng_next <= '0';
```

Figure 3: WhizniumDBE development workflow

MPSI TECHNOLOGIES

contact@mpsitechnologies.com
+49 175 918 5480
www.mpsitechnologies.com

Helene-Mayer-Ring 4, 04.19
80809 Munich
Germany