Alexander Wirthmueller (MPSI Technologies) argues that

It is time to augment the FPGA-SoC developer experience
... and there is Open Source tooling for it

Introduction

As the lineup of affordable FPGA's is evolving towards abundant fabric co-packaged with multiple CPU cores suitable for Embedded Processing and Linux, an update of the tooling used to efficiently leverage all hardware capabilities should be considered.

This article introduces the Open-Source Whiznium framework for the model-based design of robust FPGA-SoC gate- and firmware in the first section and then moves on to give real-life examples of its various benefits using a computer vision demonstrator in the second section.

Sources for the Whiznium tooling and for all examples are publicly available and are listed at the end of the article.

Making the case for a unifying framework

Various Open-Source IP core projects exist; they score with thoroughly verified standard HDL components but are naturally limited to the fabric domain. Countless developers have come up with scripts to keep memory-mapped registers in sync between CPU and FPGA. Commercial tools facilitate the implementation of mathematical algorithms in programmable logic (PL). Finally, FPGA vendors provide some tooling for CPU-FPGA cooperation, for example in the form of Linux device drivers or HLS compilers.

Without questioning good intent or justification of above-mentioned helpers, they leave any FPGA-SoC developer aiming to deliver feature-rich portable solutions with a fragmented tooling landscape. Gaps, ranging from HDL wrappers to device drivers, to scheduling mechanisms and user / M2M interaction concepts, need to be filled manually, and it does take substantial implementation effort to get from initial system design idea to a first working prototype. This effort comes on top of that required for the project's core functionality; it multiplies with the number of project variants and continues along the project lifecycle as multiple development environments need to be kept up to date coherently.

WhizniumSBE/DBE is an Open-Source development framework for Embedded Software that attributes equal importance to the CPU and PL portions of FPGA-SoC's and thus helps implementing well-structured, feature-rich applications with flexible system partitioning. Its methodology is model-based design complemented by automated source code generation. In addition, it retains the flexibility to integrate external artefacts, for example from above-mentioned frameworks and tools. The model description, in the form of text files, serves as single source of truth and thus greatly simplifies project maintainability. The source code generator consistently ensures high code quality across all project aspects. Whiznium spans the Embedded Full Stack, from

fabric level across multi-threaded Embedded Linux daemons, all the way to web UI's and API device access, as shown in Figure 1.

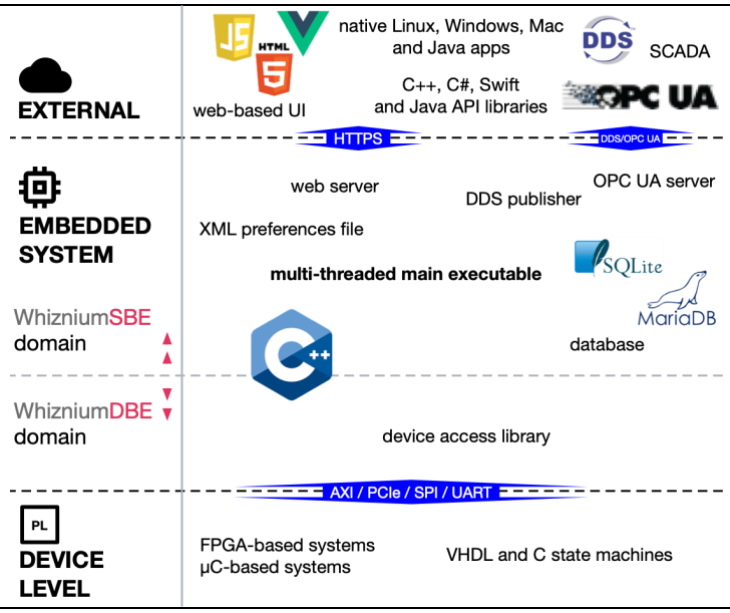

Figure 1: Embedded full-stack coverage of both Whiznium tools

Model entry

Whiznium's model files follow a consistent header / content line approach, establishing a hierarchical structure by indentation, as exemplified in Figure 2. Each model file covers one distinct gate- or firmware aspect, with nomenclature already familiar to FPGA and Embedded engineers.

```
ImelMUnit.sreflxVBasetype  srefSilRefWdbeMUnit  sref          Title                          Easy  srefKToolch  Comment
fpga                       xczu1cg-1sbva484e    zuvsp         video stream processing on ZUBoard  true  vivado
                           ImelMModule.sreflxVBasetype  hsrefSupRefWdbeMModule  srefTplRefWdbeMModule  sref  Comment
                           ...
                           ehostif              vsp_core      hostif_Easy_v1_0               hostif  interface with Cortex-A53's running Linux
                                                ImelAMModulePar.x1SrefKKey  Val
                                                phytype       axi
                                                wA            40
                                                wD            64
                                                ImelAMModulePar.end
                           ...
                           ImelMModule.end
ImelMUnit.end
```

Figure 2: Exemplary *modular structure* model file, excerpt showing the definition of a template-based RTL module *hostif* (host interface) along with three parameters

A full listing of the model files and the system aspects they cover can be found in Table 1.

Initialization data (IexWznmIni.txt)	Initialization data (IexWdbeIni.txt)
Projects and versions (IexWznmPrj.txt)	Projects and versions (IexWdbePrj.txt)
Global features (IexWznmGbl.txt)	Modular structure (IexWdbeMdl.txt) *Module hierarchy with references to extensible module template library*
Database structure (IexWznmDbs.txt)	
Basic user interface structure (IexWznmBui.txt) *Division of the UI into major "cards" aka. browser tabs*	Command set (IexWdbeCsx.txt) *Commands and buffer transfers associated with virtual controllers*
Import/export structure (IexWznmIex.txt)	
Operation pack structure (IexWznmOpk.txt)	Data flows and algorithms (IexWdbeDal.txt)

Custom jobs (IexWznmJob.txt) *Jobs to control hardware aspects in fine-grained fashion*	Fine structure (IexWdbeFin.txt) *Generics, ports, processes, FSM's and FSM steps*
Custom user interface features (IexWznmUix.txt) *Controls*	Deployment information (IexWdbeDpl.txt) *Information on targets to help write make files*
Custom job tree features (IexWznmJtr.txt) *Job-job dependencies, state machines and calls (messaging) between jobs*	
Deployment information (IexWznmDpl.txt) *Information on targets to help write make files*	
App features (IexWznmApp.txt)	
Visualization features (IexWznmVis.txt)	

Table 1: WhizniumSBE (for Embedded Linux domain, left) and WhizniumDBE (for FPGA domain, right) model files

Software design principles

To further smoothen the transition to a Whiznium-backed development workflow, only well-established programming languages are employed: both Whiznium tools are written in C++, backed by SQL databases. They generate generic, standard-conforming, human-readable, VHDL, C++ and XML / JavaScript source code while only relying on few mainstream Open-Source libraries.

Another cornerstone of the development process with Whiznium is the "all code in plain sight" promise. For example, the journey of web UI HTTP requests towards their handling in C++ code up to reply generation is easily traceable in WhizniumSBE project code. A WhizniumDBE analogy is the traceability of CPU-side command invocation through the stages of bytecode encoding, device read / write and FPGA-side decoding and handshake generation. This approach also allows to easily detach from Whiznium at some point in a project's lifecycle, should this ever be desired.

Insertion points and source code tree iteration

On a project's first invocation, Whiznium delivers a comprehensive source code tree – derived from the model description – the contents of which can be compiled and synthesized out-of-the-box without any manual modification. However, the source code tree and its files' content are intended to be complemented manually. Whiznium uses the simple yet scalable concept of *insertion points*, manual sections delineated by tag lines (comments in the respective programming language) in code files, to ensure manual code modifications are carried forward as the code generation process is invoked repeatedly. Examples are given in Figure 3. The concept extends to file system level, using KEEP tags embedded in files / folders.

```
bool PnlWzskLlvRotary::handleCallWzskClaimChg(
        DbsWzsk* dbswzsk
      , const ubigint jrefTrig
    ) {
  bool retval = false;
  // IP handleCallWzskClaimChg --- INSERT
  return retval;
};

bool PnlWzskLlvRotary::handleCallWzskClaimChg(
        DbsWzsk* dbswzsk
      , const ubigint jrefTrig
    ) {
  bool retval = false;
  // IP handleCallWzskClaimChg --- IBEGIN
  set<uint> moditems;

  refresh(dbswzsk, moditems);
  if (!moditems.empty()) xchg->submitDpch(getNewDpchEng(moditems));
  // IP handleCallWzskClaimChg --- IEND
  return retval;
};
```

```
-- IP impl.grayegr.wiring.cust --- BEGIN
--grayToManyAXIS_tdata <= CUST;
-- IP impl.grayegr.wiring.cust --- END

-- IP impl.grayegr.wiring.cust --- RBEGIN
grayToManyAXIS_tdata <= gr0 & gr1 & gr2 & gr3 & gr4 & gr when abcde=0
        else gr1 & gr2 & gr3 & gr4 & gr0 & gr when abcde=1
        else gr2 & gr3 & gr4 & gr0 & gr1 & gr when abcde=2
        else gr3 & gr4 & gr0 & gr1 & gr2 & gr when abcde=3
        else gr4 & gr0 & gr1 & gr2 & gr3 & gr when abcde=4
        else (others => '0');
-- IP impl.grayegr.wiring.cust --- REND
```

Figure 3: Most frequent insertion point types “insert” (INSERT to IBEGIN/IEND or ILINE, left) and “replace” (LINE or BEGIN/END to RLINE or RBEGIN/REND, right)

Whiznium augments – and does not replace – the familiar FPGA-SoC development workflow: code editing and debugging in tools such as VS Code and the FPGA vendor IDE’s is complemented by altering the model description as the project evolves. The process of running Whiznium on a project’s existing source code tree is called *iteration*, as shown in Figure 4. During iteration, Whiznium combines the information from the updated model files with internal derive / generate functionality to arrive at a full project representation in its internal database. From that point, a set of template files, also including a rich library of parametrized IP cores, helps it to write an updated source code tree, while extracting and re-inserting the manual components of the existing source code tree. One helper tool each for WhizniumSBE and WhizniumDBE, cross-platform as written in Java, automates the iteration process.

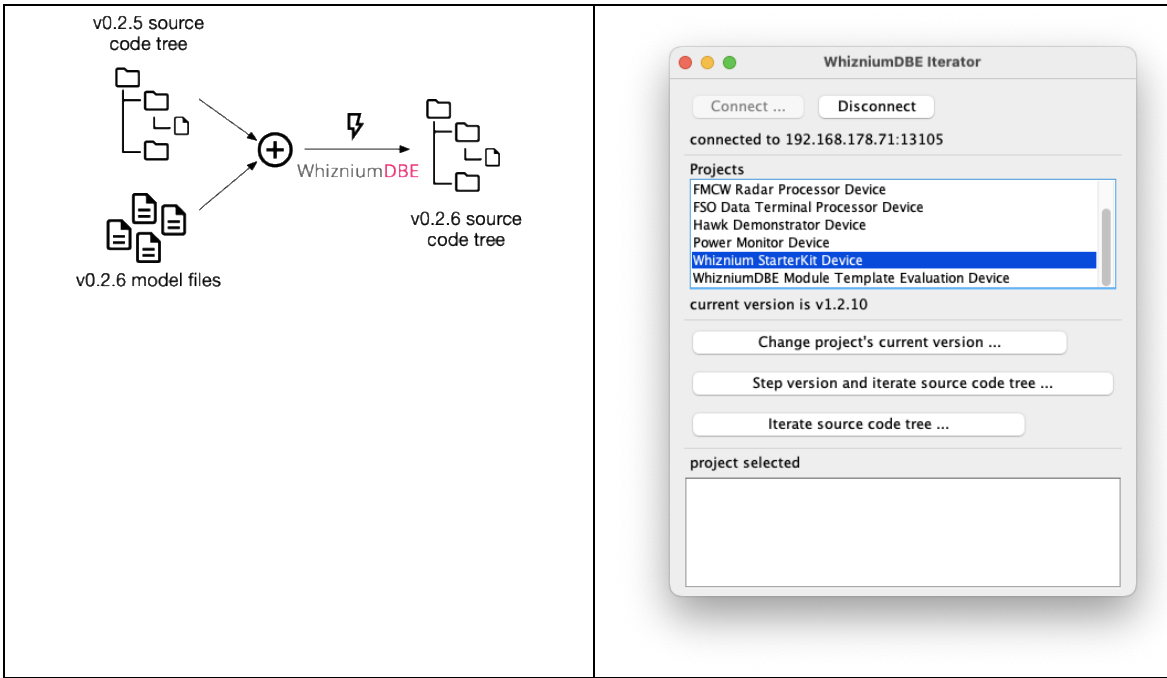

Figure 4: Source code tree iteration (left) and the WhizniumDBE Iterator tool (right)

Whiznium computer vision demonstrator

The Whiznium computer vision (CV) demonstrator, intended to highlight Whiznium's capabilities in a real-life FPGA-SoC project example, is based on the hardware depicted in Figure 5. It comprises a stepper-motor driven turntable with printed checkerboard pattern, along with a cantilever mount holding a camera module and dual red line lasers, plus some alignment / mounting material. This setup can be used to implement numerous CV applications, the culmination of which is a tabletop 3D laser scanner. Currently three FPGA-SoC platforms are supported with systems built around low-cost evaluation boards: AMD MPSoC, Efinix Titanium and Microchip PolarFire SoC.

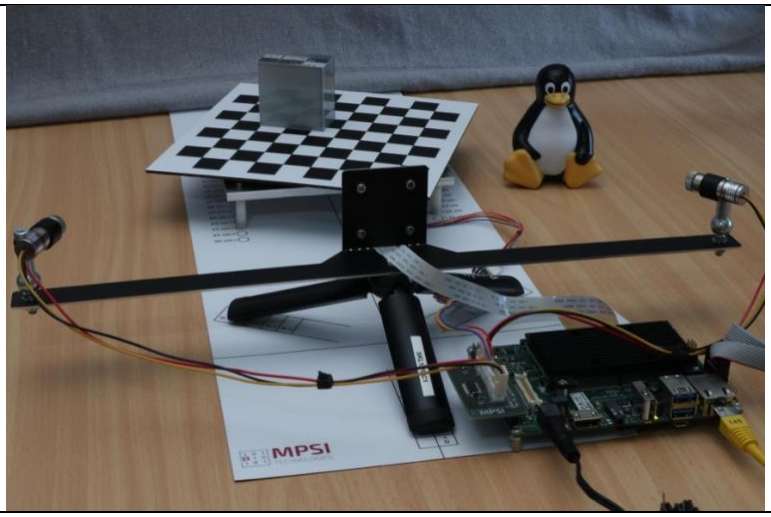

Figure 5: Computer vision demonstrator hardware

Gate- and firmware projects are implemented using WhizniumDBE and WhizniumSBE, respectively. They cover aspects and challenges typical to mid-range FPGA-SoC systems. A block diagram of the full RTL design is shown in Figure 6; it features high-speed interfaces such as MIPI CSI-2 RX and DDR memory access as well as modules implementing classical CV algorithms including binning, corner detection and HDR imaging.

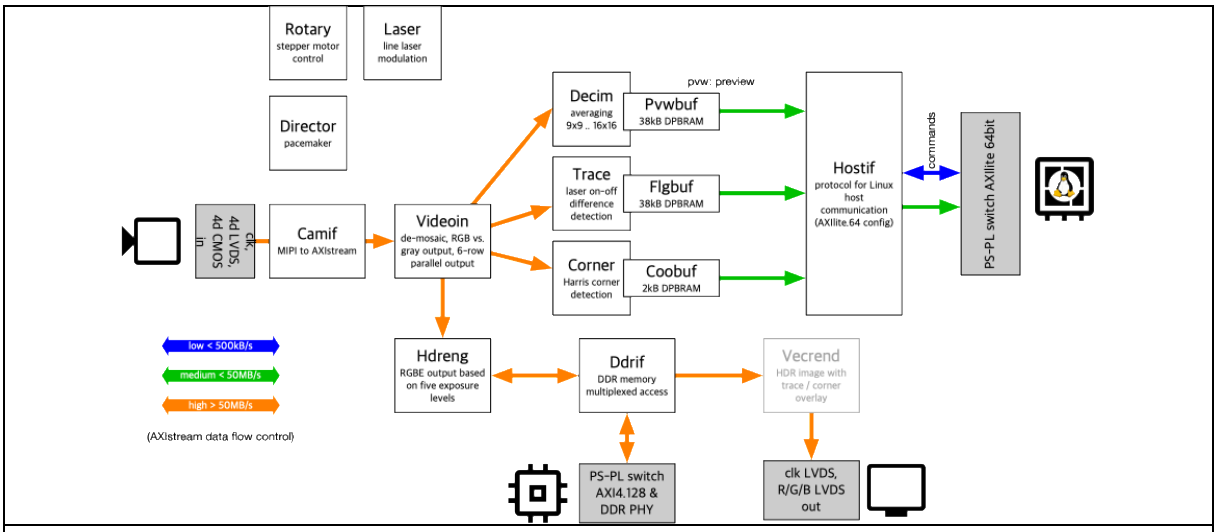

Figure 6: Computer vision demonstrator FPGA block design

Example I: FPGA-based turntable positioning commanded from CPU

Highlights the basic use of FPGA design sub-modules as virtual controllers.

Model and source code file pointers: in [1] _mdl/IexWdbeCsx_wskd.xlsx, ezdevwskd/UntWskdZuvsp/CtrWskdZuvspRotary.h, fpgawskd/zuvsp/{Hostif, Rotary}.vhd

Generating stepper motor pulses of correct period and duty cycle for a turntable positioning task is well-suited for programmable logic, whereas information on the required target angle typically is present in the processing system, either due to algorithmic requirements or from manual UI intervention. Hence, this is a typical example of fabric-based real-time execution with high-level CPU-based control.

The WhizniumDBE implementation starts with declaring the *rotary* RTL module a virtual *controller* and then associating commands with it. On the lowest level, this single definition results in matching CPU- and FPGA-side code artefacts to trigger and receive corresponding command invocations, as can be seen in Figure 7. Furthermore, Whiznium can generate interactive terminals for debug purposes, either as standalone command-line executable or as integrated web UI feature, shown on the bottom left of Figure 10.

ImeIMUnit.sref											
zuvsp											
	ImeIMModule.hsrefSup sref										
	vsp_core	rotary									
		ImeIMController.									
		^									
			ImeIMVector2.srefIxVBasetype	sref	srefsKOption						
			tixlin	VecVWskdZuvspRotaryStepmode	filfed;notit						
				ImeIMVectoritem2.sref	Title	Comment					
				full							
				half							
				quarter							
				ImeIMVectoritem2.end							
			tixlin	VecVWskdZuvspRotaryState	filfed;notit						
				ImeIMVectoritem2.sref	Title	Comment					
				idle							
				move							
				ImeIMVectoritem2.end							
			ImeIMVector2.end								
			ImeIMCommand2.refNum	sref	srefixVRettype	srefivrRefWdbeMSignal	srefRvrRe	srefRerRefV	Comment		
			0	config	void						
				ImeIAMCommandInvpar2.sref	srefixWdbeVPartype	srefRefWdbeMVector	Length	Defval	srefRefWdbeMVectoritem	Comment	
				tixVStepmode	tix	VecVWskdZuvspRotaryStepmode			full		
				TStep	uint8			60		in tkclk clocks: rps = 10000 / (TStep * 800 * 5 * 1,2,4)	
				ImeIAMCommandInvpar2.end							
			0	getInfo	statsng						
				ImeIAMCommandRetpar2.sref	srefixWdbeVPartype	srefRefWdbeMVector	Length	Comment			
				tixVState	tix	VecVWskdZuvspRotaryState					
				angle	uint16			in stepper motor microsteps (16000 per rev.)			
				ImeIAMCommandRetpar2.end							
			0	moveto	void						
				ImeIAMCommandInvpar2.sref	srefixWdbeVPartype	srefRefWdbeMVector	Length	Defval	srefRefWdbeMVectoritem	Comment	
				angle	uint16			0		in stepper motor steps (16000 per rev.)	
				ImeIAMCommandInvpar2.end							
			0	set	void						
				ImeIAMCommandInvpar2.sref	srefixWdbeVPartype	srefRefWdbeMVector	Length	Defval	srefRefWdbeMVectoritem	Comment	
				rng	_bool			fls8			
				ccwNotCw	_bool			fls8			
				ImeIAMCommandInvpar2.end							
			0	zero	void						
			ImeIMCommand2.end								
		ImeIMController.end									
	ImeIMModule.end										
ImeIMUnit.end											

```
/**
 * CtrWskdZuvspRotary
 */
class CtrWskdZuvspRotary : public CtrWskd {

public:
  // ...

  /**
   * VecVStepmode (full: VecVWskdZuvspRotaryStepmode)
   */
  class VecVStepmode {

  public:
    static constexpr uint8_t FULL = 0x00;
    static constexpr uint8_t HALF = 0x01;
    static constexpr uint8_t QUARTER = 0x02;

    // ...
  };

  // ...

public:
  // ...

  void config(const uint8_t tixVStepmode, const uint8_t TStep);
  void getInfo(uint8_t& tixVState, uint16_t& angle);
  void moveto(const uint16_t angle);
  void set(const bool rng, const bool ccwNotCw);
  void zero();
};
```

```
entity Rotary is
  port (
    reset: in std_logic;
    mclk: in std_logic;
    tkclk: in std_logic;

    -- ...

    reqInvConfig: in std_logic;
    ackInvConfig: out std_logic;

    configTixVStepmode: in std_logic_vector(7 downto 0);
    configTStep: in std_logic_vector(7 downto 0);

    getInfoTixVState: out std_logic_vector(7 downto 0);
    getInfoAngle: out std_logic_vector(15 downto 0);

    reqInvMoveto: in std_logic;
    ackInvMoveto: out std_logic;

    movetoAngle: in std_logic_vector(15 downto 0);

    reqInvSet: in std_logic;
    ackInvSet: out std_logic;

    reqInvFromRpuifSet: in std_logic;
    ackInvFromRpuifSet: out std_logic;

    setRng: in std_logic_vector(7 downto 0);
    setCcwNotCw: in std_logic_vector(7 downto 0);
    fromRpuifSetRng: in std_logic_vector(7 downto 0);
    fromRpuifSetCcwNotCw: in std_logic_vector(7 downto 0);

    reqInvZero: in std_logic;
    ackInvZero: out std_logic
  );
end Rotary;
```

Figure 7: IexWdbeCsx model file for rotary controller (top), resulting methods in CPU host library and RTL project (bottom left and right, respectively)

On each command invocation, encoding into byte code, AXIlite communication and decoding on the PL-side are handled in auto-generated code. The byte code format is detailed in Figure 8.

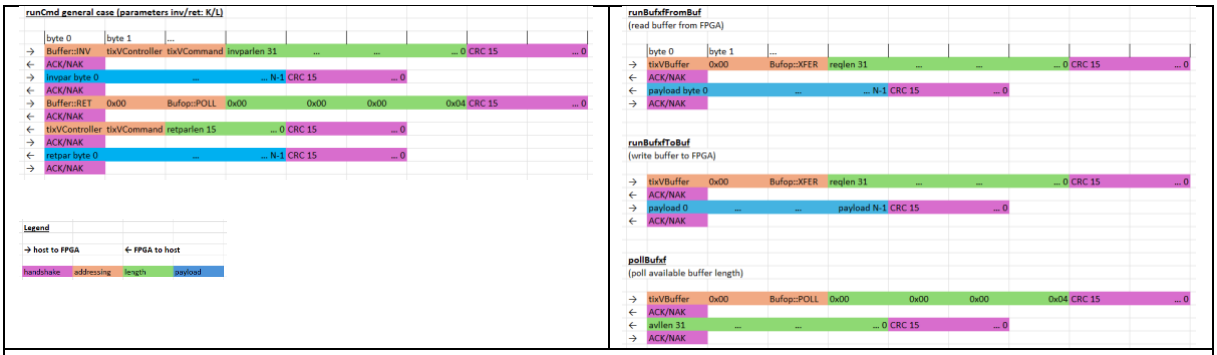

Figure 8: Interface agnostic, CRC-guarded bytecode for CPU-FPGA communication

Excursion: The job tree in WhizniumSBE-backed projects

Model and source code file pointers: in [2] _mdl/IexWznm{Job, Jtr}_wzsk.xlsx, wzskcmbd/Wzskcmbd.{h, cpp}, wzskcmbd/gbl/JobWzsk{SrcZuvsp, ActRotary}.{h, cpp}

At this point, the CPU-side concept of the *job tree* should be introduced. Any Embedded System development mandates clearly defined software responsibilities for control and observation of hardware features. WhizniumSBE encourages splitting functionality into a fine-grained hierarchy of *jobs*, with each job resulting in a dedicated C++ class and source code file, exemplified for the CV demonstrator project in Figure 9. Super jobs *#include* their sub-jobs and can access their functionality directly. In addition, sophisticated thread-safe inter-job call passing is part of every WhizniumSBE-backed design.

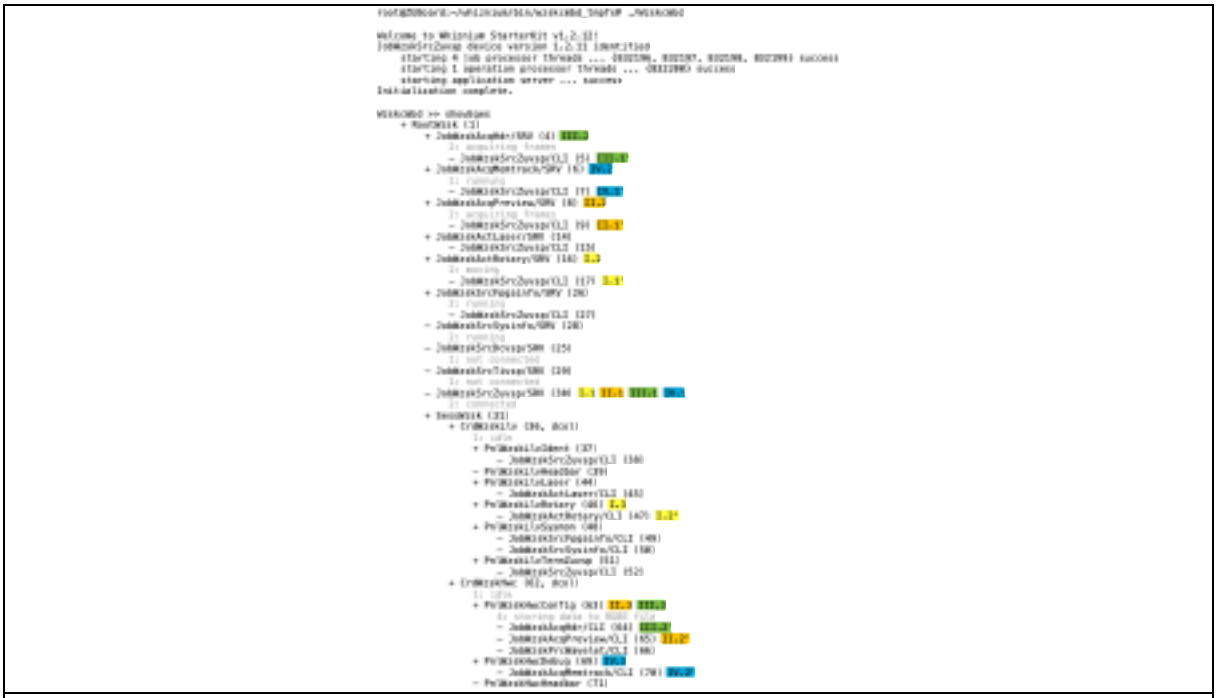

Figure 9: CV demonstrator runtime job tree (slightly shortened) with jobs relevant for examples I-IV highlighted in color; stages in gray for jobs with attached state machines

According to the above, on the one hand each hardware-controlling job should exist only once in controlling fashion, while on the other hand there can be multiple super-jobs requiring their functionality as sub-jobs. This contradiction is resolved by instantiating each hardware-controlling job once in server mode and multiple times in client mode – this is denoted as /SRV and /CLI, respectively, in Figure 9. Only the server instance actively performs operations on hardware, for example by holding file descriptors and allocated memory while relaying requests and results to client instances.

FPGA subsystem access is achieved, for the AMD MPSoC variant of the CV demonstrator, by instantiating the job *JobWzskSrcZuvsp*. It funnels all command invocations and buffer transfers through a single character device, */dev/dbeaxilite0*. To allow mutually independent super-jobs, such as *JobWzskActLaser* for line laser control and *JobWzskActRotary* for turntable control, to gain access to only a portion of the FPGA subsystem, the concept of *claims* is implemented, along with thread-safe handling methods. With it, at any given time, exactly one super-job can claim controlling access of either of the domains *corner/decim/hdreng/laser/step/trace/*track*. In an alternative implementation, multiple WhizniumDBE-backed IP cores, one for each domain, could be attached to the system.

Also visible in Figure 9 are web UI session related jobs, which can be present multiple times in the job tree and which are not subject to the server / client construct of hardware control jobs. Specifically, under the user session *SessWzsk (31)*, handler jobs for *cards* – each card corresponds to one browser tab in the web UI – and *panels* are listed.

Returning to Example I, a user-commanded turntable movement (see top left of Figure 10), is implemented via the job tree path *PnlWzskLlvRotary – JobWzskActRotary – JobWzskSrcZuvsp*, where *JobWzskActRotary* controls the move operation with the help

of an attached state machine: it first issues a *moveto()* command to the FPGA subsystem via *JobWzskSrcZuvsp* and then receives periodical wakeup calls at which it tracks the progress, using polling and *getInfo()*, until done. *JobWzskActRotary* also serves as hardware abstraction layer (HAL): depending on the CV demonstrator hardware configured, it instantiates *JobWzskSrcDcvsp* (Microchip PolarFire SoC) or *JobWzskSrcTivsp* (Efinix Titanium), instead of *JobWzskSrcZuvsp* (AMD MPSoC), without changing the API towards its super-jobs.

Example II: FPGA-based image decimation, CPU readout via AXIlite and display in web UI

Shows vertical integration FPGA -> Linux -> UI.

Model and source code file pointers: in [1] _mdl/IexWdbeCsx_wskd.xlsx, fpgawskd/zuvsp/Decim.vhd; in [2] _mdl/IexWznmUix_wzsk.xlsx, wzskcmbd/gbl/JobWzsk{SrcZuvsp, AcqPreview}.{h, cpp}, wzskcmbd/CrdWzskHwc/PnlWzskHwcConfig.{h, cpp}, webappwzsk/CrdWzskHwc/PnlWzskHwcConfig{.js, _CusImg.xml}

The camera sensor delivers frames of at least 2592x1944 raw, or 1296x972 de-mosaiced RGB resolution, at up to 60 fps with an associated data rate of > 200 MB/s. This is excessive for displaying preview images in the web UI. To slash the data rate, on the RTL side, a decimation module, called *decim*, is implemented which, based on a single configurable *edge* parameter, averages over edge x edge raw pixels to generate preview images. This is achieved using a sophisticated load/store unit and a block RAM row buffer.

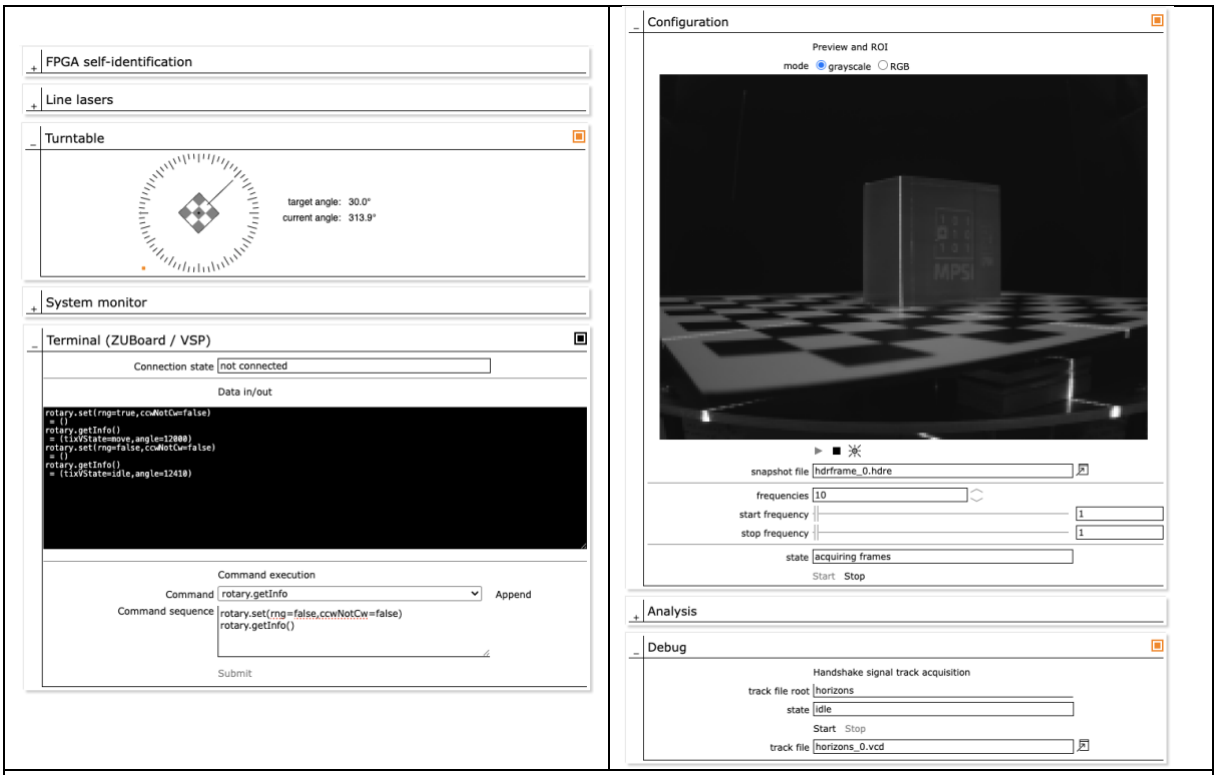

Figure 10: Web UI snippets for the “low-level access” card (left) and the “HDR wavelet classification” card (right)

In analogy to Example I, control from the CPU side is governed by specifying commands for the *decim* RTL module. In WhizniumDBE-backed projects, besides commands, *buffer transfers* can be attributed to controllers: they are not copy-free in the same way as DMA is (cf. Example III) but retain the interface-agnostic characteristic (for example SPI or UART instead of AXIlite) of command passing. This feature is used to read out the 38 kB *pvwbuf* buffer which is implemented in block RAM as a sub-module of *decim*.

CPU-side a separate thread *runPvw()* inside the job *JobWzskAcqPreview* handles configuration, status polling and frame acquisition. The buffer transfer target is a three-item result buffer *resultPvw* initialized once, thus avoiding dynamic memory allocation at run-time. Once a frame acquisition completes, notification of the daemon's *job processors* (aka. worker threads) is handled via the external call *CallWzskResultNew*, which is then passed upwards the job tree to web UI jobs, specifically to *PnlWzskHwcConfig*. As each *result item* is equipped with a mutex lock, the acquisition thread can keep acquiring new preview frames from the FPGA subsystem while older frames are being processed upstream in the job tree without provoking memory access conflicts.

To enable the web UI display captured in the top right of Figure 10, Whiznium auto-generates the infrastructure for message-passing between multi-threaded daemon and web UI using HTTP(S): to this end, *dispatches*, which are C++ objects server-side, are used. They are serialized to XML or JSON (using base64 encoding for binary data) in auto-generated code. Dispatches handling standard UI controls such as text boxes, buttons and sliders are derived automatically. For the special purpose of image display, a custom control *PnlWzskHwcConfig.CusImg* is declared along with a custom dispatch *DpchWzskHwcConfigLive* in *IexWznmUix* and *IexWznmJtr*, respectively. On reception of the *CallWzskResultNew*, one such dispatch is populated via straight copy from the FPGA-delivered data and subsequently scheduled for transfer to the web UI.

Finally, in the web UI, which performs continuous "long-polling" to the server, the dispatch arrives. Its RGB content is extracted into a *Uint8Array* after which the custom JavaScript method *refreshLive()* handles its display in a HTML5 canvas.

Example III: FPGA-based full-frame HDR image calculation in shared DDR memory space

Highest-bandwidth and platform-specific implications.

Model and source code file pointers: in [1] _mdl/IexWdbeMdl_wskd.xlsx, fpgawskd/{dcvsp, tivsp, zuvsp}/{Hdreng, Ddrif}.vhd; in [2] wzskcmbd/gbl/JobWzskAcqHdr.{h, cpp}

The previous examples do not show exchange of image data between CPU and FPGA subsystem at the ultimate bandwidth, which in FPGA-SoC systems is achieved by sharing a section of the system's DDR memory space. HDR image calculation is a perfect use case for this, as already the FPGA-based HDR algorithm must load and store

MB-sized intermediate results that would not fit the internal block memory of mid-range FPGA-SoC's.

The RTL module *hdreng* handles the acquisition of five consecutive RGB frames, with each taken at one eighth the exposure time of the previous one. For 12 bits of dynamic range, this provides sufficient overlap to iteratively determine a 3x8 + 8 bit RGBE value for each pixel, where 'E' represents an exponent shared between the three base colors. A sophisticated row load/store algorithm using A/B pixel buffers mitigates the non-deterministic nature of DDR memory access, such that useful synchronization of stock pixel data from previous frames with live data of the current frame can be achieved. As the clock speed towards the DDR memory controller is higher than the pixel / system clock, clock domain crossings (CDC's) are implemented across the buffers, with WhizniumDBE auto-generating additional CDC logic for handshake signals.

In this context, the flexibility provided by the *ddrmux_Easy_v1_0* (DDR memory access multiplexer) module template, instantiated as *ddrif*, greatly simplifies implementation. While DDR memory access is achieved via AXI4 full in all three CV demonstrator variants, bus widths and allowed clock speeds vary. For instance, for the Efinix Titanium implementation with 512-bit wide data ports, the module template features integrated gearing to the 128-bit wide ports of *hdreng* and its buffers.

Regarding the shared memory space, management of active slots of the 12-item buffer is done CPU-side. For this purpose, the *JobWzskAcqHdr* job periodically invokes the *assign()* command of the RTL module *hdreng* to let it know which slot to write to *next*. As one full HDR acquisition takes at least 5x 33 ms = 167 ms, the timing of this assignment, performed in the *runHdr()* thread is not very critical. In other projects on similar platforms, successful scheduling of this type was achieved with < 100 µs latency using standard Linux. In analogy to Example II, results can be locked by super-jobs of *JobWzskAcqHdr* for CPU-side processing. This feature is used for storing snapshot .RGBE files to SD card via *PnlWzskHwcConfig*. Templated WhizniumSBE *capabilities* assist in attaching a managed file archive to the project, providing auto-generated code for meta-data storage in the project's SQLite database and web UI file download.

Example IV: IP cores for vendor-agnostic design probing

Vendor-neutrality applied for system debugging, from design to .vcd trace display.

Model and source code file pointers: in [1] _mdl/IexWdbeMdl_wskd.xlsx, fpgawskd/zuvsp/Memtrack.vhd; in [2] wzskcmbd/gbl/JobWzskAcqMemtrack.{h/cpp}, wzskcmbd/CrdWzskHwc/PnlWzskHwcDebug.{h/cpp}

The fourth and final example covers the use of Whiznium for a topic that oftentimes implies unnecessary vendor lock-in: live probing of an FPGA design. The module templates *gptrack_Easy_v1_0* (general purpose signals) and *fsmtrack_Easy_v1_0* (FSM states) serve this purpose.

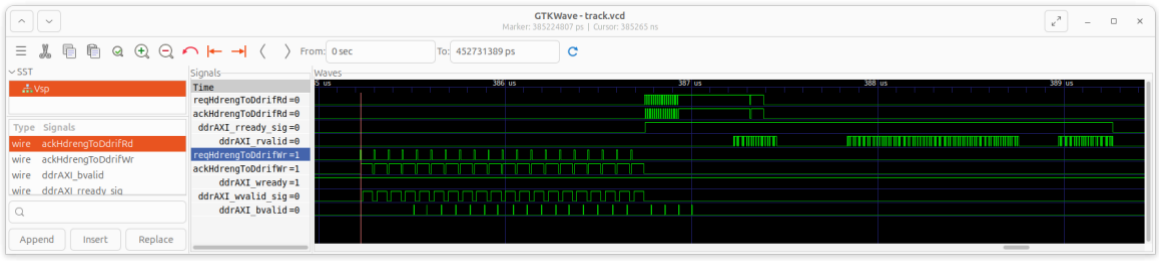

Figure 11: GTKWave displaying AXI4 full handshake signals for DDR memory write-then-read at each end-of-line of Example III

To track the relevant AXI4 full signals during DDR memory operations, *gptrack_Easy_v1_0* is instantiated as *memtrack*. In the module parameter section of the *IexWdbeMdl* model file entry, the capture signal clock, capture / trigger signal identifiers and the sequence buffer size are specified. Based on the vendor- and CPU-host-interface-agnostic IP core, then a customized version is generated which is accessible through commands and buffer transfers from the CPU side. A single instance captures up to 15-bit wide data with registered inputs, and it also features compressed storage for periods during which none of the signals change.

As in previous examples, a pre-defined WhizniumSBE *capability* is used for buffer read-out, .vcd file generation and storage. The corresponding control panel *PnlWzskHwcDebug* is shown at the bottom right of Figure 10. It instantiates *JobWzskAcqMemtrack* as sub-job which in turn interacts with the FPGA subsystem via *JobWzskSrc*vsp* using a trigger / poll loop. Finally, Figure 11 displays a waveform captured on an AMD MPSoC system in an industry-standard tool.

Conclusion

With Whiznium, a comprehensive Open-Source development framework for the Embedded domain has been introduced. Whiznium's methodology of model-based design, complemented by automated source code generation, provides the missing "glue" in the currently fragmented FPGA-SoC developer tool landscape, resulting in faster, more coherent results with simplified maintainability. The Whiznium workflow and an example application have been extensively laid out in the CV demonstrator section.

Owing to the facts that Whiznium maintains complete detailed design representations in a programmatically accessible SQL database, and that Whiznium is highly modular code which is user-extensible, it is easily imaginable to see Whiznium as the host project for other Open-Source efforts in the future. These could be vendor-agnostic IP core libraries or generator frameworks of verification IP, to name two examples.

Trivia

WhizniumSBE and WhizniumDBE, as multi-threaded Linux daemons programmed in C++, with attached SQL database and HTTP API access, by themselves are large WhizniumSBE projects. This is the result of the tooling's decades-long history,

originating in database-backed enterprise software (until taken over by Java) and then high-performance / distributed computing.

SBE stands for “Service Builder’s Edition”, hinting at this heritage. DBE on the other hand stands for “Device Builder’s Edition”.

Call to action

Contributions to the Whiznium Open-Source projects from the community are welcome – some ideas include: open discussions on best coding practices could result in those being incorporated in the automated source-code generation; the current VHDL-only PL-side code generation could be complemented by (System)Verilog output; wrappers around more vendor’s standard IP cores (MIPI DSI, PCIe, …) might prove useful.

Resources

The primary resource for getting started with Whiznium is the repository at https://github.com/mpsitech/The-Whiznium-Documentation. After following through with the first section, “Setting Up A Whiznium Development Environment On Your Workstation”, among other things, the CV demonstrator model files and source code are available for inspection locally. It is recommended to follow the “Setting Up WhizniumSBE and WhizniumDBE On Your Workstation” section to set up a functioning Whiznium environment.

All source code repositories can also be cloned individually from these locations:
[1] CV demonstrator RTL code and C++ access library:
https://github.com/mpsitech/wskd-Whiznium-StarterKit-Device
[2] CV demonstrator Linux daemon: https://github.com/mpsitech/wzsk-Whiznium-StarterKit
WhizniumDBE: https://github.com/mpsitech/wdbe-WhizniumDBE
WhizniumSBE: https://github.com/mpsitech/wznm-WhizniumSBE

For everyday work with WhizniumSBE and WhizniumDBE, their respective reference pages are an indispensable resource:
https://mpsitech.github.io/The-WhizniumSBE-Reference
https://mpsitech.github.io/The-WhizniumDBE-Reference

Finally, in this article, important features – such as user-extensibility for custom IP core generator libraries and many more – have not been touched upon. Based on the CV demonstrator, the following page is constantly updated with examples showing how to get the most out of Whiznium’s capabilities for a wide range of FPGA-SoC system aspects:
https://mpsitech.github.io/Whiznium-Knowledge-Base