IP/SOC 2004

Session 2A : Design Platform

SoC IP Qualification & Emulation Environment

Mois Navon, MobilEye

Jerusalem, ISRAEL

Abstract :

Mobileye Vision Technologies has developed a sophisticated vision processor based on two ARM9 processors which off load much of the computation intensive code to application specific IP blocks, known as Vision Computing Engines, implemented in custom logic. This System on a Chip, named "EyeQ", employs four of these Vision Computing Engines (VCEs), each of which is the size and complexity of what would have in the past comprised a full ASIC.

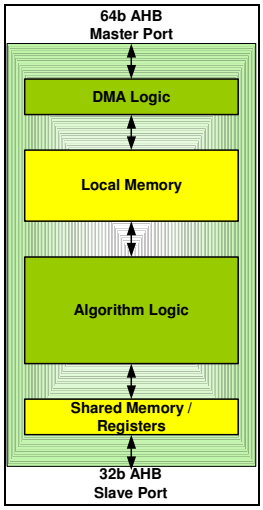
In order to address the myriad qualification requirements of this SoC a number of approaches must be taken. This paper deals in particular with the solutions utilized to service the verification requirements of the VCEs: (1) simulation of a single VCE in the ASIC environment; (2) simulation of a single VCE in an FPGA environment. The FPGA itself is used to provide reduced functionality SoC emulation within a RISC processor development kit.

Due to the nature of the VCEs as image processing units designed to be run in various illumination conditions and analyzing various images (i.e., roads, lane markings, cars, motorcycles, pedestrians, etc.), special consideration is needed to test - as exhaustively as possible - all cases, yet without the ability to run randomized video frames (which would unnecessarily fail the system).

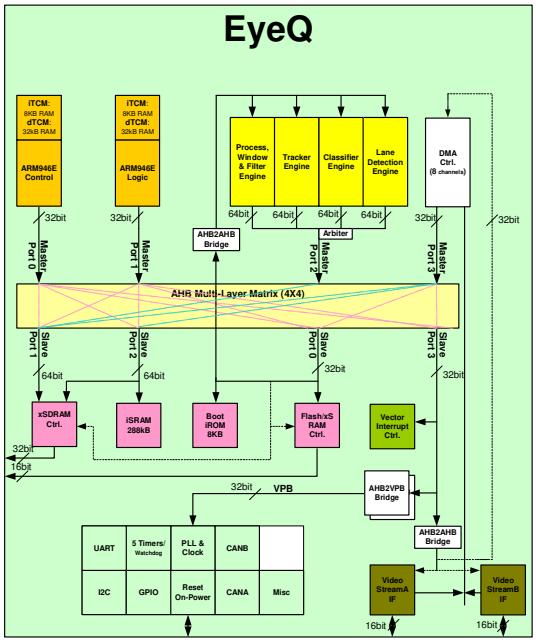
SoC IP Qualification & Emulation Environment

Hardware Description

Mobileye's "EyeQ" SoC is a sophisticated vision processor based on two ARM9 processors which utilize custom IP blocks, known as Vision Computing Engines, to perform computation intensive functions. The "EyeQ" employs four of these Vision Computing Engines (VCEs), each of which implements a unique computer vision algorithm, yet each maintains common external interfaces. The General VCE diagram is as follows:



The VCEs are integrated in the SoC as shown in the general system block diagram (shown in yellow):



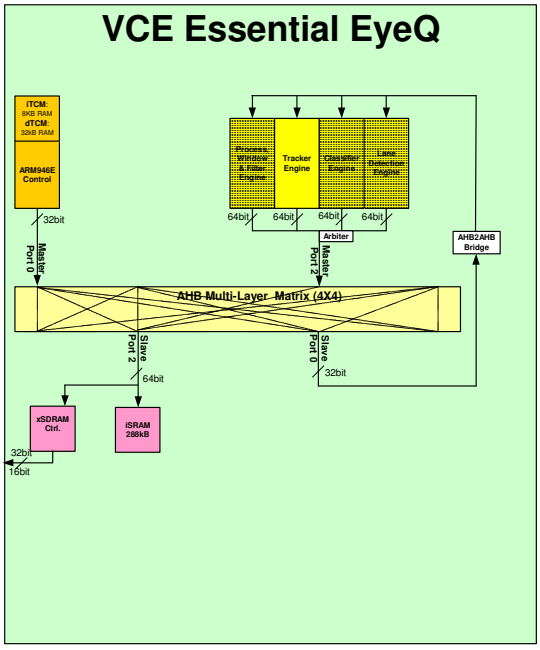
As can be seen from the general block diagram of the chip, all of the various agents in the system are interconnected via an AHB Multi-Layer Matrix. (The Multi-Layer Matrix is a user configurable Phillips supplied IP which provides multi-port interconnections with a transparent arbiter which merely "holds off" busy ports using the READY signal).

Each VCE (i.e., Process Window & Filter; Tracker; Classification; Lane Detection) can be accessed via two separate busses – a peripheral bus and a system bus. The peripheral bus connects all four VCEs via an AHB-to-AHB bridge to the Matrix Slave Port 0. From this port the VCEs act as slaves, responding to memory access commands executed by the ARM processors. The VCEs also have bus mastering capability via a system bus connected to the Matrix Master Port 2. This bus employs a local arbiter which controls bus access requests from the various VCEs. The bus mastering capability is required to allow reading and writing of image data as per the internal processing needs of the particular VCE.

Simulation Environment

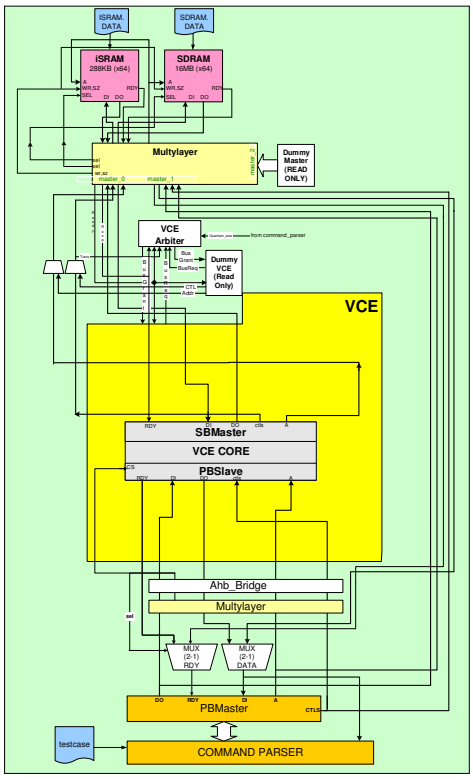
As mentioned, the focus of this paper is on the testing requirements of the VCEs. The testing of the VCEs is critical since they contain the bulk of the custom hardware design, and as such they need to be debugged early on in the system design cycle. The VCEs need to be verified for standalone functionality as well as for operational capability within the system architecture. One way to test the VCEs is to code up the system design and then run software on the CPU which would exercise the functionality of the VCEs. This indeed is how the final verification is handled; however waiting to "ring out" the VCEs till this stage would be detrimental to the project's progress – not to mention that simulation runs on the complete SoC is extremely time consuming.

As such, there is a real need to reduce the system definition to a minimum and then implement an efficient test environment to exercise the full functionality of each individual VCE as a standalone unit. The following diagram shows the reduced system diagram which contains one ARM processor, one active VCE with arbiter and bridge, the Matrix, and the RAM modules:



In order to provide an efficient test environment for this reduced system – again, without having to rely on running software on the CPU, we employ the classic "HDL Testbench" paradigm. Though the "HDL Testbench" is a well known industry tool, it is presented here to demonstrate to what level and complexity it can be successfully utilized.
The HDL Testbench is a non-synthesizable design composed of: the design under test (DUT: in our case a VCE); behavioral models of the DUT's peripherals (e.g., arbiter, multi-layer matrix, SRAM, etc); bus monitors (often built into the behaviorals to detect erroneous activity); a Command Parser to control test flow (here in place of the CPU). The Command Parser reads commands from an ASCII testcase file containing used defined commands and initiates actions within the various behavioral models accordingly. Handshaking between the Command Parser and the peripheral models is accomplished via simple "Start-Done" signaling. Results (e.g., bus errors, data comparisons) are printed to the screen and/or file to be analyzed manually or by scripts. Scripting can be used to automate test runs into batch mode.

Based on the reduced system architecture the following Testbench serves the needs of any VCE, all of which have the same external interfaces:



Here it can be noted that the ARM processor has been replaced by a Command Parser and a PBMaster which emulates the AHB bus protocol. The Command Parser is a non-synthesizable logic unit which executes user defined commands sequentially read out of an ASCII text file "testcase". The testcase files consist of a series of Testbench commands which perform a specific sequence of tasks to verify the DUT. The Testbench commands are a set of design specific commands which exercise the various behavioral models in the Testbench.

The table below describes a number of testbench commands. The simplest method of specifying commands (especially in Verilog HDL) is to simply use hexadecimal signals – though heavy commenting is recommended to make the testcases readable.

Operation	Description	Example
Testcase Name	Assigns a number to the testcase.	00 01 02 08 05
PBUS write	Performs a burst write on the peripheral bus starting at the address specified.	01 00 07 08 00 01 55 55 55 55
Finish	The last command of a testcase.	FF

A sample testcase might look as follows :

```
// TESTCASE
00 10 00 00 01 //TEST NUMBER 10000001
0F 0E  //GRANT ON
0D 01  //RAM WAIT STATES ON
// SEM (Set Semaphore)
01 00 F2 00 08 01 80 00 00 00
// set xCSR
01 00 F2 00 00 01 80 3F 00 04
// set TIQR
01 00 F2 00 04 01 00 00 00 09
// SEM (Reset Semaphore)
01 00 F2 00 08 01 00 00 00 00
// Wait completion of the tasks
04 00 F2 00 00 80 3F 00 0C 00 00 FF FF 00
FF // FINISH
```

The system SRAM and SDRAM are initialized with the contents of ASCII based data files. In order to simulate multiple Masters loading the Multilayer-Matrix a "Dummy" Master performs non-destructive read accesses on the spare Master Port. Furthermore, in order to simulate multiple VCEs accessing the shared system bus, a "Dummy" VCE performs non-destructive read accesses on the bus.

Verification Modes & Methodology

The testbench can be used in a number "modes": Manual, Random, Automated. Manual tests are testcases which are written by the engineer to debug and verify the IP's basic functional integrity.

A far more comprehensive method of qualifying the design is through randomly generated testcases. To generate such testcases, a set of design parameters which can be varied through their range of acceptable values must be defined. A script (e.g., coded in PERL) must then be written which randomly chooses appropriate values and then prints out to a testfile the corresponding testbench commands. This script can then be called by another script which runs "infinitely", generating testfiles and initiating the simulator.

Random tests are ideal for bus verification and other circuitry which operates within a well defined range of rules (e.g., burst length from 0 to 255). However for image processing circuitry which is designed to operate on specific images in specific sequences (e.g., rear-ends of cars, trucks and motorcycles on highways), one can not simply define random parameters for such images. Furthermore applying completely random images will unnecessarily fail the DUT.

In such a case, the best method to qualify the DUT is by automated "exhaustive" testing. Automated testing consists of taking hours of real video footage, running it through the software model which then generates DUT task inputs (i.e., image files and task descriptors) and outputs (e.g., tracking results). The data generated by software model is

then run through a script which adapts the information into testbench command testfiles. A script is then run which initiates the simulator for each "automated" testfile.

By running these automated tests for hours on end, day in, day out for weeks, they serve to qualify the DUT as exhaustively as possible. One method of checking that the code has been nominally exercised is to run Code Coverage software which highlights if there are parts of the code that were never simulated. The results of this tool can then be used to write manual tests to cover the untested circuitry.

FPGA Emulation & Simulation

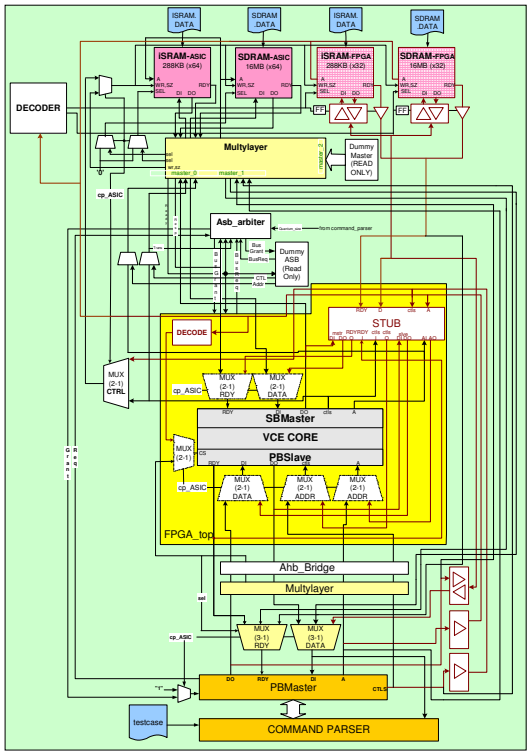
Though the testbench provides a fairly complete and efficient simulation environment for the VCEs, it has a number of drawbacks. The most prominent drawback being that the VCE is not tested in real hardware – and no matter how good the behavioral models are, they always remain true to their names: models. Furthermore, the HDL platform is not an efficient environment to test the SoC's software. To address these issues, the FPGA is an appropriate solution which provides a relatively economical hardware emulation platform.

Though the design going into the FPGA should be a "carbon-copy" of the ASIC design, this is only true in theory. In practice, many system adjustments and design modifications are often necessary to adapt the ASIC design for FPGA implementation. In our case, ARM provides an FPGA based development kit (known as the "Integrator") which has a slightly different architecture than our SoC. Whereas the SoC has two uni-directional busses (a 64-bit system bus and a 32-bit peripheral bus), the FPGA system has but one shared tri-stated 32-bit bi-directional bus. Furthermore, the FPGA system does not have a Multi-Layer Matrix but rather relies on a bus arbiter and address decoder.

Now, just as a complex design going into an ASIC must be well tested before being implemented in hardware, so too should the design going into an FPGA. For though the FPGA has the advantage over the ASIC of being re-programmable, nevertheless, debugging is always simpler in a simulator than in a logic analyzer. In our case, given the serious architectural differences, simulating is imperative.

Though one could develop an FPGA specific testbench, it is desirable to leverage off the existing ASIC testbench for a number of reasons:
(1) Simpler Maintenance & Upgradeability – changes in one environment don't need to be duplicated in the parallel environment (allowing the addition of bus monitors and various behaviorals in stages).
(2) Bugs found in the FPGA due to the physical environment performing differently than the behavioral environment can then be reproduced in the FPGA simulation and subsequently tested on the ASIC design itself.

The unified (ASIC/FPGA) test environment for a VCE, is shown in the following diagram:



The ASIC's VCE is instantiated within the FPGA design (FPGA_top) which includes the FPGA's internal address decoder and an AHB bus "STUB" which bridges between the external bi-directional bus and the two internal uni-directional busses. It also includes several 2-to-1 MUXes which select between STUB bus lines or the straight AHB bus lines. These MUXes are implemented to support the dual test environment (i.e., FPGA and ASIC); however, they are written such that they will NOT be synthesized by the FPGA (as such they are drawn with dotted-lines). Furthermore the Multilayer Matrix is bypassed with tri-state buffers.

Conclusion

Mobileye's EyeQ vision processor provided a case study in the difficulties of efficiently qualifying complicated IPs. It was shown that – complexities notwithstanding – standard verification methodology, applied judiciously, can be used to accomplish the task.