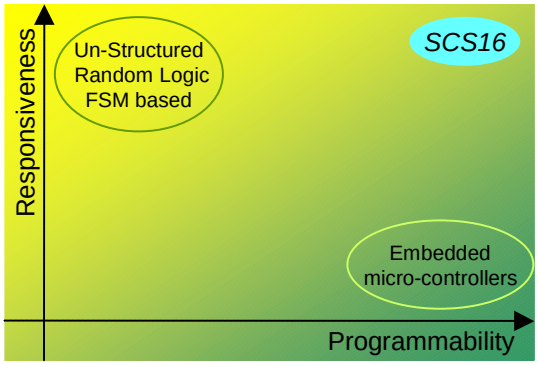
Zorba Technologies

Executive summary of the SCS16 core and development environment

Introduction

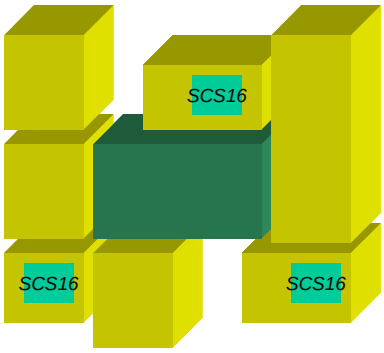
In recent years, SoC designs became very common. The system view of designs is getting a lot of attention at early design stages. Chip architects often introduce some embedded processor for high-level control. It seems that top-level design partitioning is getting the right focus. However, when one moves to sub-chip layers – modules, one probably finds a collection of communicating State Machines (FSM), logic and registers and optionally a few memory blocks. Alternately, a few designers may try to map all, or some of the modules' functionality to software, running on embedded processors (that may already exist), having to deal with the complex task of hardware software partitioning. While the first approach eventually works, the later is very often impossible.

The first implementation approach is hard to maintain and debug, it is often area/power inefficient because logic is not reused. Each minor change requires redoing back-end flow, thus forcing an RTL freeze far before tape out. Postproduction metal ECO's are of limited capability, expensive and require redoing the timing closure.

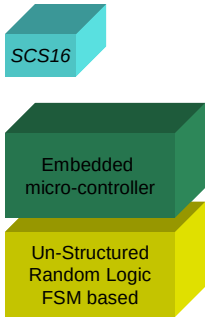


The second implementation approach is clearly preferred by most R&D executives, having in mind that bugs and modifications can be handled in software, thus, eliminating all the limitations and weaknesses of the first approach. Nevertheless, further analysis shows that this solution is not viable for the sub-chip layers. Those processors that are deeply pipelined are not as responsive to external events as are FSM based, single cycle designs. Moreover, since the code for those processors is usually written in C, one has no control over the number of clock cycles (or machine instructions) per code line. These are compiler dependent. Thus, it is not possible to be in sync with external events. Often, due to processor complexity, an RTOS is involved, which further slows down the processor response time. Besides functional limitations, one can expect area penalty and higher cost of ownership due to solution complexity.

Conventional Design



Micro-Code driven Design

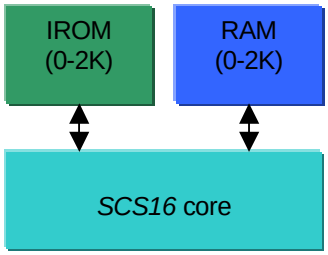


The SCS16 solution

To help improve sub-chip designs, Zorba has introduced the *SCS16* core and Development Environment. The *SCS16* soft core is a single cycle RISC nano controller. It is programmable, like a micro-controller, yet responsive and parallel like an FSM based design. The *SCS16* core, *with its instruction and data memory* is tiny (20K gate). At this size, the *SCS16* fits well into the sub-chips/module space. For instance, The *SCS16* core has been used to implement a functional module sized at 50K gates. This results in a considerable area and routing resources reduction. With the *SCS16* core, designers proceed with code development up to a week before tapeout, effectively shortening project cycle by three months (back-end flow length). Functional bugs could be fixed anywhere during the project. This makes life a lot easier at a minimal cost. In addition to the *SCS16* soft core, Zorba delivers a code development environment. The two solution elements provide all the necessary components for a customer to develop a micro-code driven solution for sub-chip/module level.

SCS16 core

The *SCS16* core is a single cycle RISC nano controller with a 26-bit instruction path and a 16-bit data path. **All *SCS16* commands execute in a single clock cycle**. Most *SCS16* commands are parallel – allowing multiple operations per command. It is designed to connect easily (glue less) to your design. Once connected, the developer has only to decide on the instruction ROM and data RAM sizes and proceed with code development. .



The *SCS16* has separate instruction and data interfaces (Harvard architecture) and fetches a new instruction and optionally data in each cycle. All memory data addressing modes are implemented. It has two mechanisms for context switching; it can react to a maximum of 16 external events within one cycle. It has hardware support for two levels of subroutines and one interrupt source. It has a powerful burst mechanism to streamline I/O data. The *SCS16* core has free instruction codes ready for costumer specific needs.

Feature list

I/O

- 2 x 16 bits input ports. For single/stream data input.
- 16 scalar inputs. For data qualification and/or flow control
- One interrupt signal
- One clock, one reset
- 4 x 4 bits level outputs. For external device control
- 16 scalar pulse outputs. For data qualification and hand shaking.
- Up to 8 x 16bits outputs. For data output.

Architecture

- 26 bits instruction, 16 bits data path
- Up to 2K (11 bits) data and instruction addressing
- 16 bits ALU
- Parallelism - up to 5 operations per command in one cycle
- 2 sets of 8 x 16 bits general propose registers.
- One status register, no configuration registers.

Program Flow

- Single cycle execution for **all** commands
- Zero delay jump/branch, external condition branch and relative jump.
- Zero overhead loops
- Hardware support for two levels of subroutines.
- Powerful wait commands

Special commands

- Burst send/receive with one cycle per transaction from an external device to memory and vice versa
- FIFO read/write to speed up memory mapped FIFO implementations.
- Streamlined memory Linear-Search command with a throughput of one cycle per element
- Optional, on the fly CRC calculation

Miscellaneous

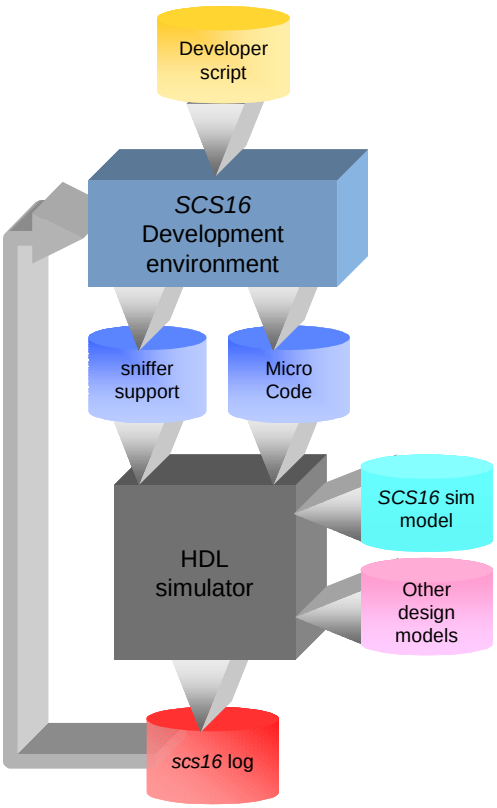
- ASIC/FPGA integration:
- Vendor and technology independent
- Fully synchronous and static design
- 100Mhz at 0.13u process
- Tiny Size - ~7K instance for the core
- Easily extendible, to support customer specific needs

SCS16 Development Environment

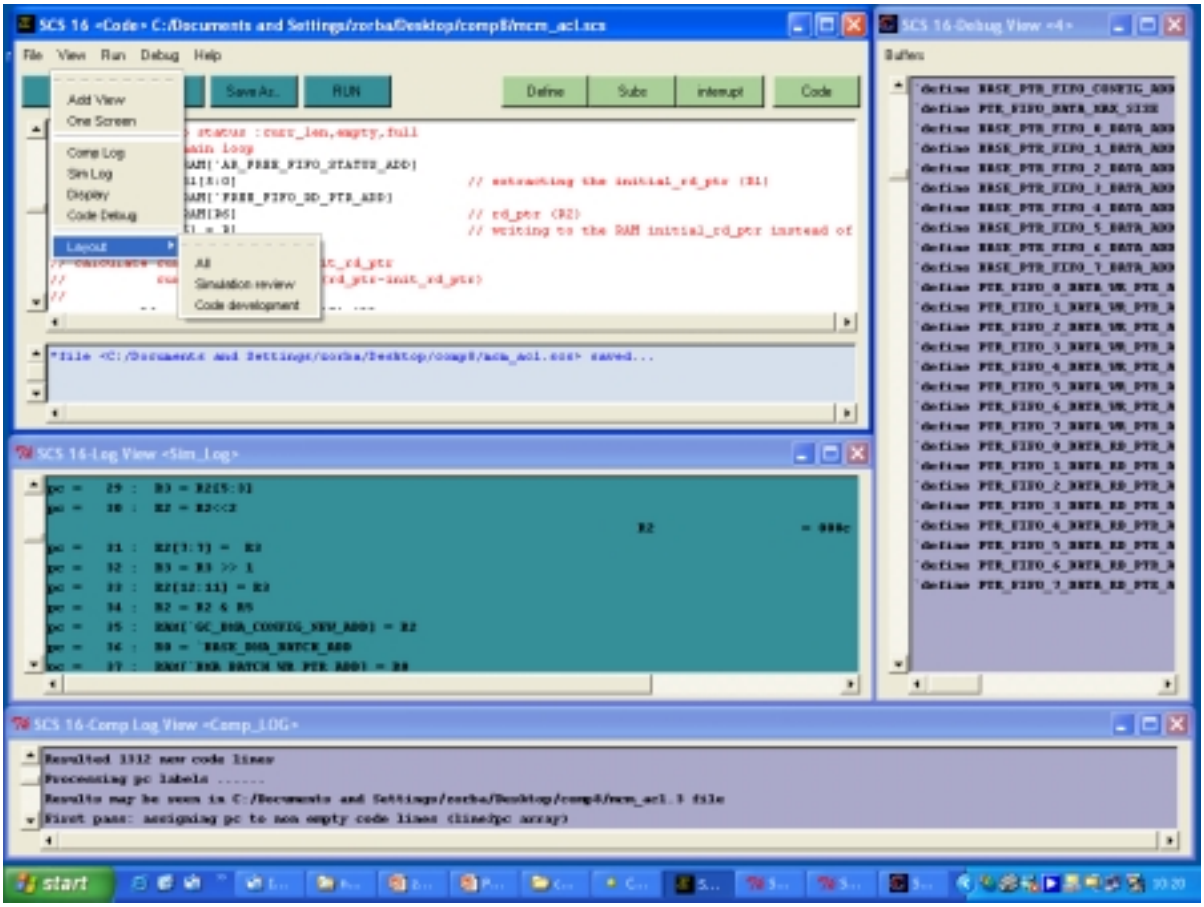
The *SCS16* compiler/translator supports the *SCS16* programming language, a subset of C with additional tokens to support specific commands and core parallelism. It is designed with hardware and clock cycles in mind, thus every non-comment line executes in one clock cycle; it allows the developer to be synchronous with external processes. The environment is very simple and intuitive. Typically, one can write, compile and simulate a significant flow within less than an hour.

The *SCS16* compiler/translator is invoked by the command prompt or from the GUI system. Within the GUI system, the development is structured with features like code coloring and indentation. The Compiler outputs ASCII micro-code files in various formats supporting various simulators and memory generators. It also generates supporting files for simulation.

For simulation review/debug, the *SCS16* simulation log file can be loaded into the GUI system along with the source code. By clicking a line in the log, the engine identifies and highlights the line in the source, and vice versa. All internal registers are monitored continually and each change is displayed immediately following the instruction.



SCS16 Development Environment – screen shot



Deliverables

1. *SCS16* core in low-level verilog language. Used for simulation.
2. *SCS16* core for synthesis. (Format TBD, based on licensing agreement)
3. *SCS16* Development Environment. (For code development and simulation review)
4. *SCS16* simulation sniffer. (Simulation support utility)
5. *SCS16* Synthesis and Static Timing scripts. (For Synopsys Design compiler and Physical compiler)
6. *SCS16* User guide
7. *SCS16* Quick Reference Card

For more information, contact info@zorba-tec.com

Israel
Zorba Technologies Limited
Office Address: 14 Haraby Mibachrah street, Tel Aviv 66849
Mail Address: P.O.B 8126 Tel-Aviv 61081
TelFax: +972 3 6829315