scc-*II* Microsequencer – A New Solution for Platform FPGA Designs

When your project design is too big for a finite state machine, but a microcontroller would be overkill, try Ponderosa Design's scc-*II* microsequencer.

by Aki Niimura
Consultant
Ponderosa Design
ponderosa_design@pacbell.net

As the complexity of FPGA-based systems grows every year, we are asked to implement larger, more complex functionality within tighter schedules. Furthermore, the type of design has changed rapidly in recent years. People used to design an FPGA taking an existing board design, often containing asynchronous clocks. Those days are over. You can not design today's Platform FPGA just by extending yesterday's design practices. New designs often require the implementation of complex sequences or communication protocols. The finite state machine (FSM) is a well-known design methodology to implement such sequences. FSM is very effective when the sequence is not very complex. However, implementing a complex sequence using an FSM is not practical and is often difficult to maintain.

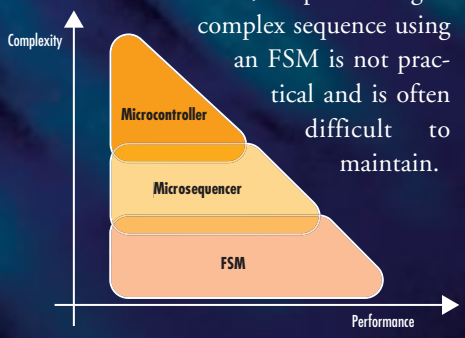

Figure 1 - Performance versus functional complexity

Microcontrollers are commonly used to implement complex protocols. However, they require substantial resources (memory, cost, pins, ...), which can be difficult to justify in real-life situations. On the other hand, software implementations allow designers to cope with mounting logic complexity. They are easy and quick to implement and easier to maintain.

There is a gap, however, between the range of design complexity that FSM methodology can handle and what microcontroller-based methodology is good for, as shown in Figure 1. As microcontrollers become more powerful, the gap is widening.

The scc-*II* is not just another set of microcontrollers. We specifically constructed the microsequencer to fill the gap between low-level FSM solutions and high-level microcontroller designs.

scc-*II* – A Configurable Microsequencer

Sequencers have been used in many LSI projects to implement functions. For example, instructions in a CISC (Complex Instruction Set Computer) CPU were often implemented in this way (called microcode, which is written in a proprietary assembly language). By allowing users to write programs in a high-level language, the scc-*II* can accommodate a wider range of FPGA applications.

The key architectural benefits of the scc-*II* are:

- Small footprint
- High-level language support
- Small code size
- Configurable and customizable
- Capable of handling 16-bit and 32-bit data types
- Timer (integrated into the core architecture)
- Support of interrupt handling
- Developing and debugging tools
- Utilization of Xilinx Spartan™-II and Virtex™-II devices.

A block diagram of the scc-*II* is shown in Figure 2. The core itself requires 400 to 600 LUTs, depending on the configuration and synthesis constraints.

How the scc-*II* Works

The scc-*II* employs a stack-based architecture. Stack computers use data stacks to evaluate given operations (Figure 3). The benefits of stack-based architecture are:

- High-level language ready – can execute syntax tree directly
- Simple hardware – easy to understand, easy to customize
- Small instruction code – most scc-*II* instructions are one byte long.

Another unique aspect of the scc-*II* architecture is the use of register windows. Register windows are used to pass arguments to a function being called. Because the scc-*II* does not use a stack frame in memory to pass arguments, the scc-*II* does not require data memory to run a high-level language program, thus making the scc-*II* more attractive for Platform FPGA applications.

Programs for the scc-*II* are almost entirely written in the high-level language SC.

The Language SC

The scc-*II* assumes the use of a high-level language. However, existing high-level languages are not designed for microsequencer applications. Therefore, we developed a stripped-down version of C language – SC. SC programs do not support "struct" and other complex data types, but SC has several enhancements to describe control applications efficiently. Timer, I/O, and debug features are natively supported in SC.

The following is a code fragment from a project that controls an SDRAM memory.

```
void
init_sdram()
{
   while(!eval_cond(DLL_RDY)) { }    // wait for DLL is ready

   wait(14);       // 286 uS
   outp(SDCMD, SD_PRE);
   repeat(8) {
    outp(SDCMD, SD_AREF);          // looping takes 7 cycles
   }          // tRC = 84nS; 20nS * 5
   outp(SDCMD, SD_MODE);
}
```

In the above code fragment, eval_cond(n), wait(n), and outp(port) are not function calls, but they are natively supported by SC. Note that the loop counter of the repeat statement is placed in the data stack and not in the register file.

scc-*II* Target Applications

The scc-*II* can be used in designing functional blocks to perform procedural control. For example, flow charts or simple arithmetic functions (such as averaging) are good candidates for implementation by the scc-*II*.

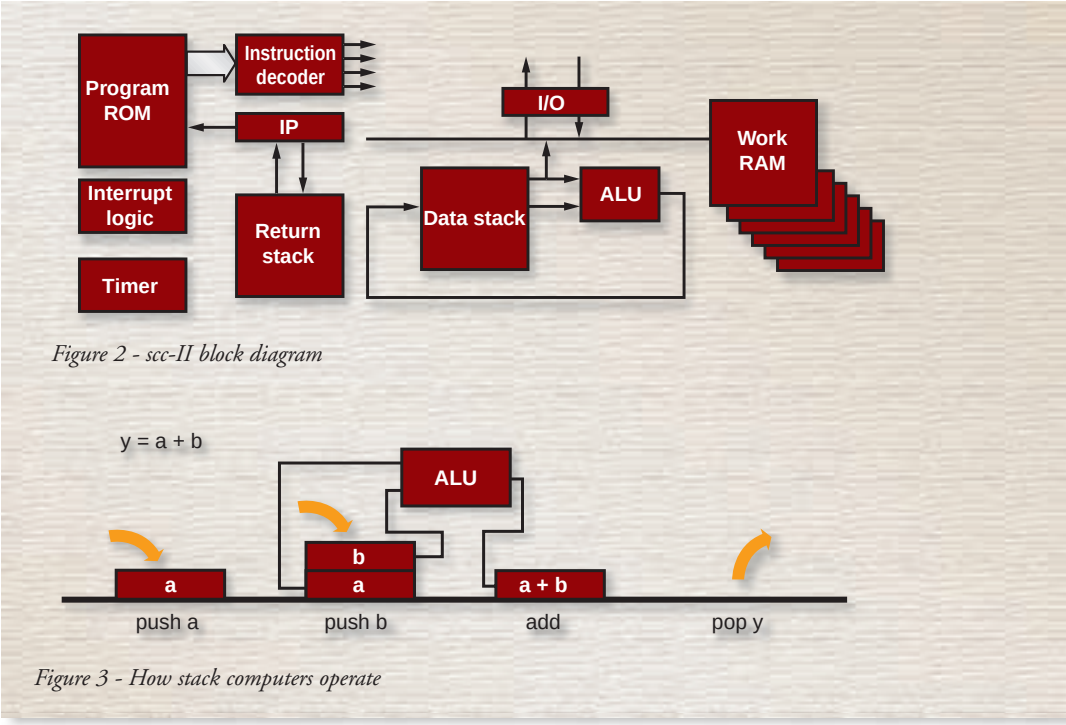

Figure 2 - scc-II block diagram

Figure 3 - How stack computers operate

Other target applications include:

- Read/write flash, EEPROM, 1-Wire™ devices
- Interface to I2C, RS-232, Ethernet, USB1.1, IrDA
- Command interpreter (a block is controlled through commands)
- User interface (such as keypad, LCD, touch panel)
- Servo controller (some arithmetic operations required)
- Design that requires many variants.

The scc-*II* and Virtex FPGAs

The Virtex family of FPGAs are true "system on a chip" platforms. The advanced technology available in Virtex-II devices provides further attractive features to the scc-*II*, including:

- The scc-*II* can run at 70 MHz or faster.
- Larger Block RAM allows larger program sizes (up to 8 KW).
- Native multiply operators are supported.

Case Study – Web on FPGA

To demonstrate the effectiveness of the scc-*II* solution, we have developed a Web server that uses less than 25% of the resources of a Spartan-II FPGA (XC2S150). The only additional hardware required beside the FPGA are an Ethernet PHY device and a signature ROM (optional). We found that the Spartan-II VoIP Development Kit from Insight Electronics *(www.insight-electronics.com)* included all

Figure 4 - WebThermo demo application

the hardware components we needed. Thus, we decided to use this off-the-shelf board to create our Web server design. Figure 4 shows a screenshot from a Web server proj-

ect called WebThermo. The screen displays the current temperature every minute. Table 1 shows utilization statistics from a Synplify analysis of the XC2S150 device.

WebThermo	Usage of XC2S150	Note
LUTs	828 (23%)	Synplify 7.0
Block RAM	8 of 12	3 for TX, RX buffer, 5 for Program
BUFTs	320 (18%)	

Table 1 - WebThermo logic size

The 2 KB WebThermo program implements all Ethernet, TCP/IP, and Web (HTTP) protocols, as well as Celsius-to-Fahrenheit conversion. At start up, the program retrieves a unique 48-bit ID code from a Dallas 1-Wire device (DS18S20), which is used as an Ethernet MAC address. For further details on the WebThermo project, please visit home.pacbell.net/akineko/.

Program Development

One challenge of the scc-*II* solution is in providing reasonable program development and debugging tools. Figure 5 illustrates a typical program development flow. In addition to key software tools, we wrote many scripts and templates to automate the design process. While creating several projects with the scc-*II*, we refined the RTL design, as well as the development software and scripts. As a result, they have become mature and stable.

Currently the development environment is supported under Unix. It is also possible to port some of the tools to Windows platform using Cygwin from Cygnus (RedHat). The tools are developed assuming that the user's RTL design is in Verilog HDL.

Debugging, Then Debugging Again

Debugging is the biggest challenge in developing an scc-*II* based design. We are providing several debugging aids:

1. Debugging starts with simulation:

– Three debug instructions (print, $dump, $stop)

– Self-checking embedded in the code

– Execution trace log generation

– Dis-assembler to display current context (on-the-fly/offline)

2. Ready to try on the board:

– JTAG debugger to download program without backend (synthesis + PAR)

– UART customized for debugging (one can use printf())

– "xdl" script to replace ROM contents without backend.

Lessons learned:

1. Logic simulation is always the best tool for debugging.
2. printf() is a primitive but very powerful means for debugging.
3. Use #ifdef ... #else ... #endif to switch between debug and release.
4. A bigger vehicle is needed for debugging (you may need 2 KB to develop a 1 KB program).

JTAG debugger

The JTAG debugger (jtagdbg) has proved to be a powerful tool to facilitate the debugging process. The JTAG debugger uses the Virtex USER1 JTAG command to communicate with a Virtex FPGA. By substituting the instruction ROM block in the scc-*II* design with a JTAG embedded ROM block, you can perform several debug commands, such as downloading a program without going through the FPGA backend design process. No signal change is required, as JTAG signals are hidden from your RTL code.

Conclusion

We have presented a microsequencer, the scc-*II*, which is new to conventional FPGA design practices. Unlike other IP cores, the potential of the scc-*II* is not limited to its original form. Rather, the scc-*II* can evolve to meet each application challenge. One avenue we plan to explore is adding Galois instructions to the original scc-*II* core. This enhancement can help in error correction or security applications.

Another avenue we plan to pursue is project automation, such as a wizard script that sets up project directories and tools – and then creates a skeleton version of RTL code, as well as skeleton SC program and header files.

The complete scc-*II* design solution is offered by Ponderosa Design in Sunnyvale, California Please write *ponderosa_design@pacbell.net* or visit *http://home.pacbell.net/akineko/.*

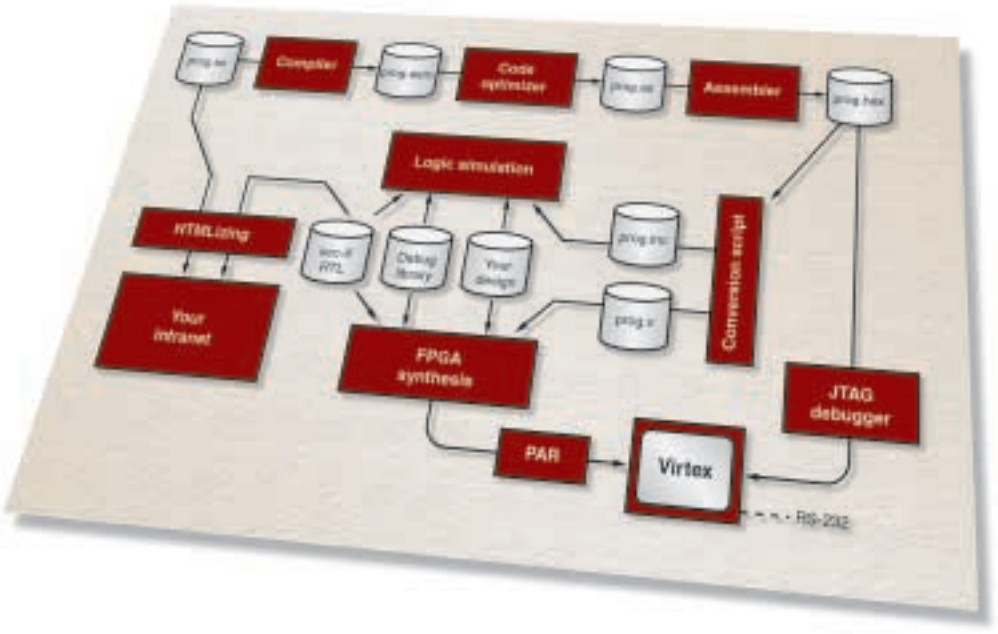

Figure 5 - The scc-II program development flow