Design of a SPDIF Receiver using Protocol Compiler

Ulrich Holtmann
Synopsys, Inc.

700 E. Middlefield Rd., 2.31
Mountain View, CA 94043-4033
+1 (650) 528-4732
ulrich@synopsys.com

Peter Blinzer
Technical University of Braunschweig
Dept. of Integrated Circuit Design (E.I.S.)
P.O. Box 33 29
D-38023 Braunschweig
+49 (531) 391-2380
blinzer@eis.cs.tu-bs.de

1. ABSTRACT

This paper describes the design of a receiver for the digital audio signal SPDIF used by CD-ROM players. The design was done with Protocol Compiler, a high-level synthesis tool for the design of structured data stream processing controllers.

Compared to traditional RTL design, Protocol Compiler makes entry, debugging, and re-use easier. Design time was cut by factor 2 while the results in terms of area and delay are competitive.

1.1 Keywords

High-level synthesis, telecommunication.

2. INTRODUCTION

CD-ROM, DCC or DAT players transmit digital audio signals in the SPDIF protocol [Son83, IEC958]. At the Technical University of Braunschweig, Department for Integrated Circuit Design (E.I.S.), a SPDIF decoder mapped onto a Xilinx FPGA was designed. Design was done at the RTL level in Verilog. Experience showed that the hardest part was to get the controllers of the various modules right. Design of the decoder requires separation of the three SPDIF layers (bit level, block, and frame) and writing the individual controllers for each. Implementing and verifying these controllers as FSMs at the RTL level is an error-prone and time-consuming task. While the design requirements at the protocol level ("*what*" to do) are well structured, the state transition graphs at the implementation level ("*how*" to do it) lack structures. Furthermore, the individual controllers must communicate with each other, so an FSM often contains, besides the functionality required by the protocol, additional states and transitions implementing hand-shakes, stalling, etc.

In the light of these design issues, a joint research project was undertaken to apply the high-level synthesis tool "Protocol Compiler" to the decoder design and determine how the issues are addressed. This paper describes the FPGA design of the SPDIF receiver using Protocol Compiler.

The rest of the paper is structured as follows: Section 3 gives a short description of Protocol Compiler and related languages. The SPDIF protocol and design specification is given in Section 4. Then the design implementation is described in Section 5 and results are presented in Section 6. Finally, we give a summary and conclusions.

3. RELATED WORK

Protocol Compiler is a design environment for the processing of structured data streams such as the ATM, SDH/SONET or MPEG protocol. It is based on previous work by Seawright and Brewer, and later Crews, on logic synthesis from grammatical productions [Sea94, Cre96]. Protocol Compiler was first described in [Sea96] and an application for SDH/SONET is shown in [Mey97]. The design environment consists of entry, debugging support (by means of annotating simulation results back onto the source), protocol synthesis, and HDL generation (Both VHDL and Verilog are supported). The Protocol Compiler language is conceptually higher than RTL and, together with protocol synthesis and intuitive debugging capabilities, is the key element resulting in a productivity gain for design and verification of controllers. Due to limited space please see [Sea96, Mey97] for a detailed description.

Protocol Compiler is designed to model controllers for structured data streams where data processing is done on a continuous basis. Modeling structured data over time is the key aspect. Reactive systems on the other hand focus on external or internal events with different urgency. Each event requires certain actions to be performed and the key aspect is to cope with the complexity of many simultaneous events. The most prominent high-level synthesis systems are Esterel [Ber92] and Statecharts [Har87].

4. DESIGN REQUIREMENTS

Figure 1 gives an overview of the decoder and environment. The decoder receives the SPDIF signal, transmits audio values to the Digital Analog Converter (DAC) over a serial interface, and also drives a few status LEDs which are not shown in the figure.

The target device is a Xilinx XC3042 FPGA clocked at 16 MHz which imposes tough constraints to area (only 144 CLBs available) and timing.

The SPDIF protocol consists of the three layers Bit, Block, and Frame. Figure 2 shows the format of the lower layers Bit and Block. The Block layer carries the audio values and some control bits. Each block consist of a header telling whether the audio value is for the left or right channel, four reserved bits, the 20-bit value with the LSB first, two control bits, a status bit used in the upper frame layer, and a parity bit.

The Bit layer uses a Bi-Phase coding to transport bits. Each data bit is coded with two code bits so that the polarity of the SPDIF signal changes with every data bit (see Figure 2). In case a '1' is coded, then the signal also changes between the code bits, in the case of a

1-58113-049-x-98/0006/$3.50

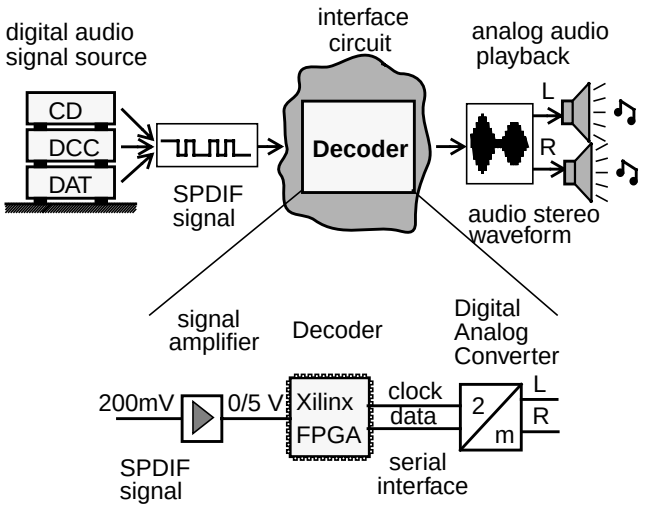


Figure 1: System Overview

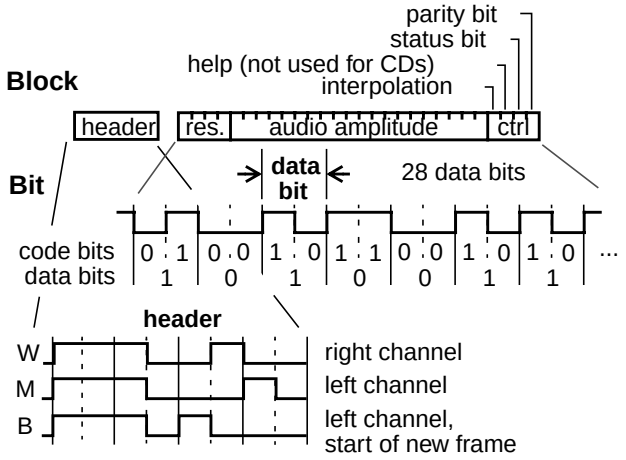


Figure 2: SPDIF Protocol, Bit + Block Layer

'0' it doesn't change. Since the polarity is constantly changed, a '0' may be coded with two high or two low code bits. A '1' is either coded with code bits high+low or opposite polarity. Headers W, M, B are coded differently to allow simple detection of the next block (for sake of simplicity, Figure 2 shows them only in positive polarity). W tells that the value is going to the right channel, M and B indicate the left channel. Header B furthermore indicates the beginning of a new frame. The data bit rate is roughly 2.8 MHz allowing transmissions of audio samples with frequencies up to 44 KHz. The clock speed of the code bits is consequently about 5.6 MHz.

The third, Frame, layer provides status information such as whether it is allowed to copy the CD or not. A frame consists of 192 pairs of left and right blocks, starting with a left block indicated by an B header. Due to limited paper size, we will not explain more details of the Frame layer, however, the receiver must decode it in order to drive three status LEDs (copy permission, copy status, and data is valid).

The Digital Analog Converter (DAC) has two separate converters and a common serial interface shown in Figure 3. The decoder must generate the CLK clock and SDI data signal as well as the two control signals. SDI is sampled on the rising edge of CLK and once all 16 bits (4 control + 12 amplitude value) are shifted in, it needs a falling edge on nLD to apply the values to the converters.

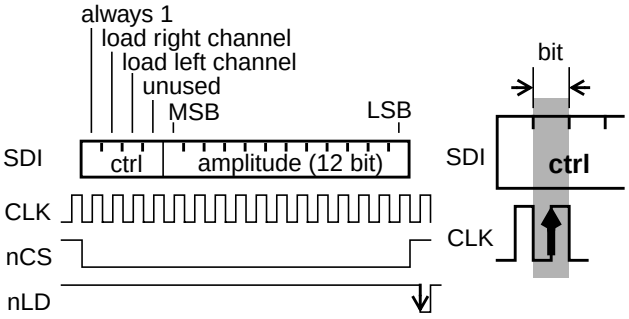


Figure 3: Interface to the Digital Analog Converter

The SPDIF decoder in the FPGA environment described above may not be a common design. However, we found that certain design requirements of the SPDIF decoder are typical for ASIC designs in the networking area such as ATM, SDH/SONET, MPEG, 1394, etc.:

- Recognition of header patterns and synchronization (Bit layer)
- Parsing a structured data stream (Block and Frame layer)
- Generating a waveform (DAC interface)
- Interface issues between modules such as synchronization or stall.

We therefore think that the solutions and conclusions described below can be applied there as well.

5. DESIGN IMPLEMENTATION

The design is broken into four modules, one for each layer of the SPDIF format and one for the DAC interface. Figure 4 shows the block diagram including all important signals.

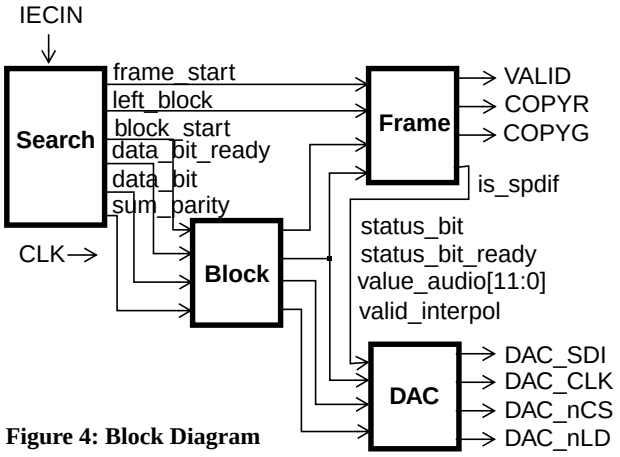


Figure 4: Block Diagram

Signals frame_start, block_start, and data_bit_ready are flags, which means they are set high for only one clock cycle to indicate the corresponding event. For example, when a data bit '1' has been detected, then data_bit is set high and signal data_bit_ready is set high for one clock cycle. Please note that the decoder runs with a clock speed of 16 MHz, several times faster than the data bit rate of the SPDIF source, so during most cycles no header or data bit will be detected.

A snapshot of the specification in Protocol Compiler is given in figure 5. The figure shows the specification in exact the same way

(besides color) that one sees it in Protocol Compiler's graphical frame editor. Each of the four boxes is a *reference frame*, abstracting a complete module of the design modeled in the Protocol Compiler language. (Please see [Sea96] for a detailed description of individual language elements, the so called "frames"). By double-clicking, one "jumps" into such a reference frame.

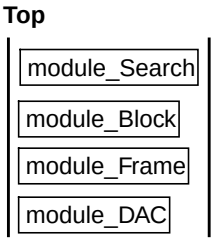


Figure 5: Specification in Protocol Compiler (Top Level)

5.1 Block Module

Figure 6 shows the definition of the module_Block reference frame implementing the Block layer. Please compare Figure 2 how the sequence {reserved..parity} directly reflects the structure of the block layer. The close similarity between specification and implementation is a strong feature of Protocol Compiler's high-level language. Understanding *what* is implemented or incorporating a specification change is very easy thus making the designer more productive.

The fields reserved, amplitude, ... are abstracted into individual frame definitions shown in Figure 7. All fields expand to *terminal "1" frames* describing a delay of one clock cycle each. A *repeat frame* $[\,]^N$ means to repeat the inner frame N times. Some terminal frames have *actions* (the rounded boxes) attached to them which are executed in the corresponding clock cycle. Most actions like valid_interpol=data_bit just store the received data bit into internal registers. The concat action "{ }" shifts the audio value into val_audio[11:0] for later use by the DAC-Interface module. The audio value is transmitted with 20-bit resolution but the DAC has only 12 bits, so the first 8 bits shifted in (the least significant ones) are overwritten, which means simply ignored. The last of the four control bits is the parity which should always be even. In case the parity check is successful, the conditional action "!sum_parity: set(status_bit_ready)" executes, telling the Frame module that the next status bit is available.

The two *template frames* RunIdle and Proc_StartAfter surrounding the sequence describe internal interface issues between the Search and Block module. Since the decoder runs at a higher clock speed than the SPDIF source, data bits are only available when the Search layer detects them and indicates so with signals block_start and data_bit_ready.

Template frames are, like reference frames, defined by sub-frames but can be customized. The RunIdle template basically stalls its inner frame when the condition data_bit_ready || block_start is

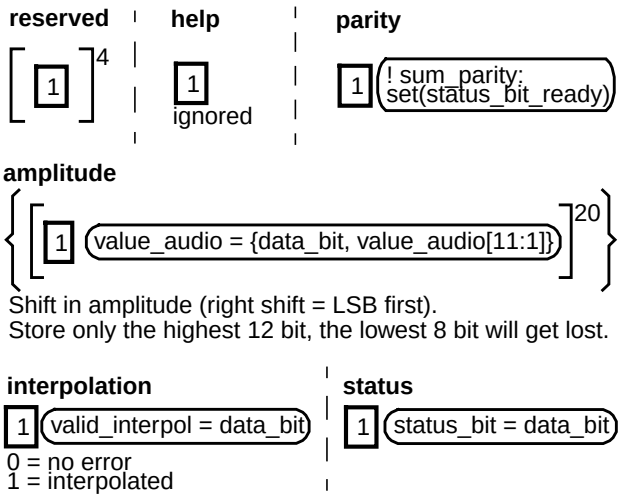


Figure 7: Submodules of the Block Module

false, so time proceeds only during cycles when one of the flags is set. Template Proc_StartAfter is an endless loop, waiting for condition block_start to be true, then executing its body (the sequence) and then waiting again. Modification of sequence {reserved..parity} with the two template frames allows us to substitute "1 cycle = 1 data bit" with respect to the sequence.

Please note that the Protocol Compiler description clearly distinguishes between the internal interface issues and the pure structure of the Block layer as defined by the SPDIF protocol. This not only increases the readability but also allows easy re-use of frames in similar designs. Furthermore, the orthogonal description of inter-block interface and protocol structure isolates any changes between these two parts, thus speeding up design changes. An RTL or schematic implementation doesn't allow a clear separation and would make later re-use of the design far more difficult.

5.2 Frame Module

The Frame module is very similar to the Block module and due to limited paper space not explained. Its task is to extract certain status information and set the ports VALID, COPYR, and COPYG which drive status LEDs.

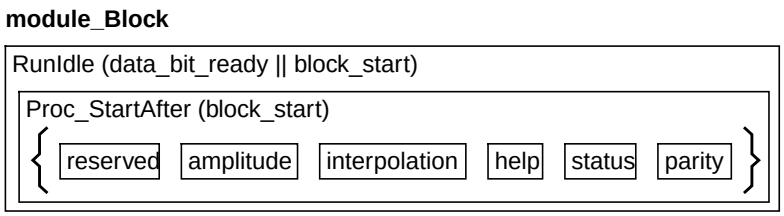
5.3 DAC-Interface Module

The task of the DAC interface is to sequentially send the audio value plus a few control bits to the DAC. The interface (Figure 3) requires that the data signal SDI must be stable during the rising edge of CLK. A simple way to guarantee this is to spend two clock cycles per bit as done here (see the two terminal frames within the $[\,]^{16}$ frame in Figure 8). During the first cycle, SDI is set and CLK is cleared. In the next cycle, SDI stays but CLK is set, so SDI is stable during the rising edge of CLK.

Besides the 12-bit audio value, four control bits must be transmitted

Figure 6:

Implementation of the Block Module



as well. In this implementation, the shift register is extended to 16 bits and the four control bits are loaded into it before the shifting starts. This way the underlying controller stays very simple.

Port DAC_nLD is set every clock cycle by a *default action*. When all 16 bits have been sent and the block is valid, then DAC_nLD is cleared, overriding the default set and thus generating the negative edge required to load the audio value into the DAC internal converter.

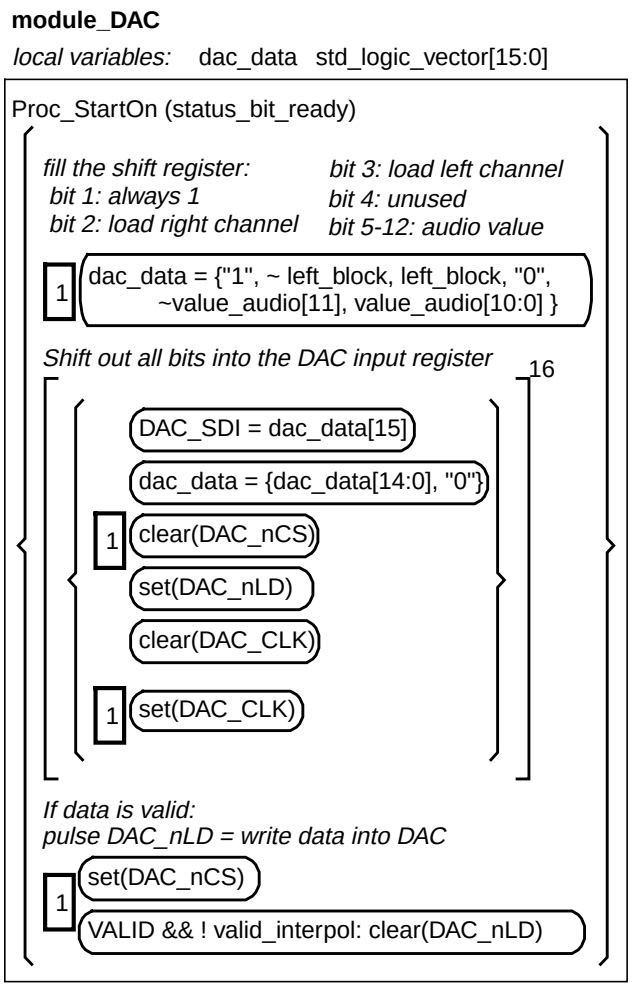


Figure 8: Implementation of the DAC Module

5.4 The Search Module

The Search module processes the Bit layer and indicates received headers and data bits, and calculates the parity bit. The Search module directly receives the IECIN input and must detect headers and data bits of the Bit layer. Its main task is to search the input stream for certain patterns which are derived from the SPDIF protocol. Figure 9 shows how to compute the search pattern for the M header. Since the receiver runs on a higher clock speed than the SPDIF source and both clocks are not synchronized, one has to allow for variances in the pattern at the receiver side. For example, a header lasts exactly 8 cycles at the source but may need either 22 *or* 23 cycles at the receiver. Determining good search patterns is a s complicated task and not explained in this paper. It has to be done by the designer and we merely describe here the chosen patterns.

A “1” (“0”) in the pattern means that the input stream must be high (low) during the corresponding cycle, while a don’t care “-” means that the input stream is not checked.

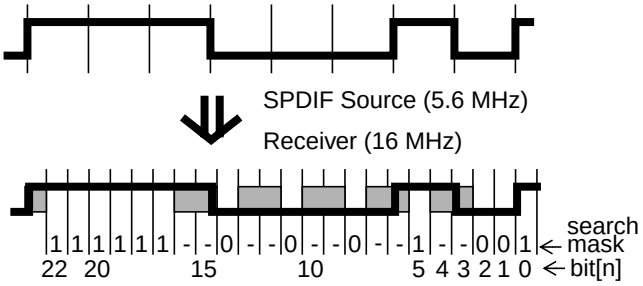


Figure 9: Search Pattern for the M Header

There are several ways to implement the Search module. The chosen architecture uses a 22-bit wide shift register and compares the register contents in two consecutive cycles. This way only 6 bits at a time must be compared and the underlying FSM is very simple. Figure 10 shows the continuous pattern search modeled with a [1]* frame and Figure 11 shows the pattern broken into two consecutive comparisons. The action pattern={pattern[20:0],IECIN} realizes the 22-bit shift register. Other implementations with shorter shift registers and more compare cycles were tried as well but due to limited space are not discussed in this paper.

Specifying the pattern search in Protocol Compiler is a straightforward task due to its ability to understand *pipelined control flow*. For example, when a M header is received, then the terminal frame pattern[21:13]=="111111--0" will match first, and terminal pattern[11:0]=="0--0--1--001" matches in the next cycle. The complete comparison consist of a two-cycle sequence. Because it is not known when the first match will occur, the sequence must be tried every clock cycle thus requiring to overlap, or pipeline, two instances of the sequence. In Protocol Compiler, pipelined control is specified with a leading [1]* frame followed by terminals containing search patterns as seen in Figure 10.

Manually implementing pipelined control as an FSM or connection of stage registers is a tedious and error prone task. Fortunately, Protocol Compiler does this task automatically and furthermore can guarantee with its distributed encoding style [Sea96] a very efficient implementation.

After a header has been detected, 28 data bits will follow and must be detected as well. Like the header detection, the designer must determine patterns recognizing the bi-phase codes. Due to limited space these bi_phase_pattern frames are not shown in the paper.

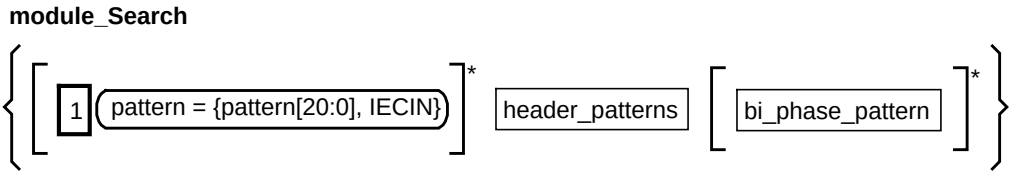


Figure 10: Implementation of the Search Module

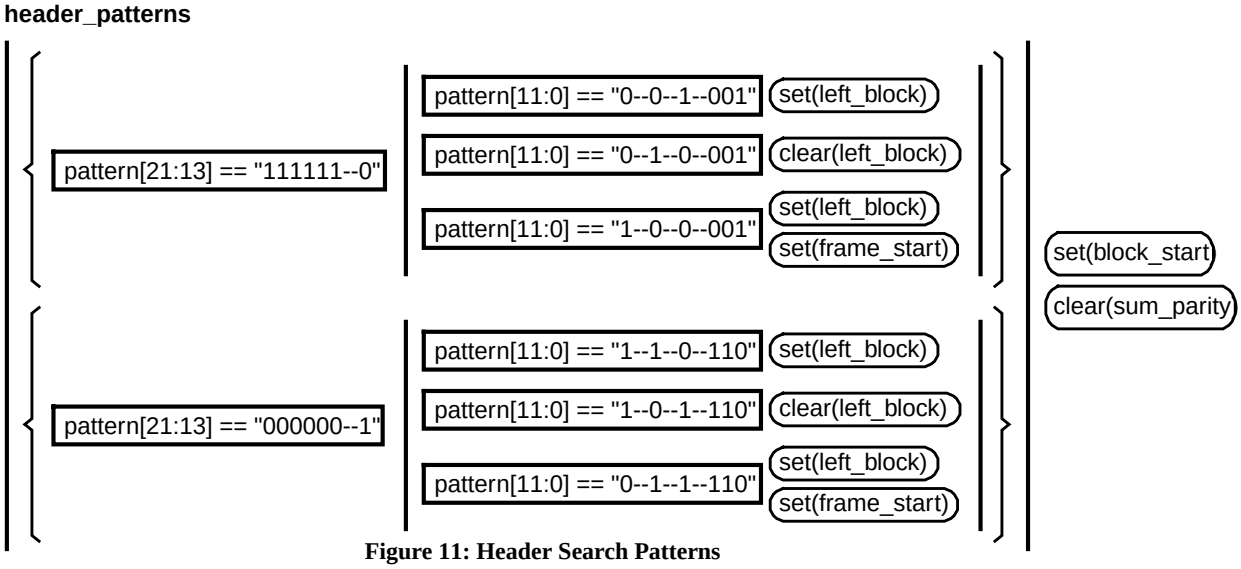


Figure 11: Header Search Patterns

6. RESULTS

The design flow of the decoder involved Protocol Compiler (Synopsys), Verilog-XL (Cadence), Design Compiler v3.5a (Synopsys) and XACT 5.2.1 (Xilinx). Design entry was done with Protocol Compiler. Simulation used Verilog-XL + Protocol Compiler for annotating the simulation results back onto the source. Compilation into an FPGA required protocol synthesis with Protocol Compiler first, then logic synthesis with Design Compiler and finally P&R by XACT.

Protocol Compiler compiles the frame description into a set of FSMs during protocol synthesis. By defining the set of FSMs and their individual coding style, the user can easily explore different architectures without the need to change the input description. For example, repeat $[\ ]^N$ frames can either be unrolled or implemented by an LFSR counter resulting in different architectures.

The architectural exploration of different coding styles and counter usage is demonstrated by the results in Table 1 for the DAC-Interface module. The repeat $[\ ]^{16}$ frame was either unrolled or implemented by an LFSR counter. All results refer to mapping onto an XC30xx target with 64 CLBs.

Table 1: Different Architectures of DAC Module

Counter	Coding Style	Area CLBs	Critical Path
LFSR	distributed	48	74.2ns
LFSR	min-encoded	39	46.1ns
LFSR	1-hot	46	64.4ns
unrolled	distributed	59	49.4ns
unrolled	min-encoded	35	49.3ns
unrolled	1-hot	57	75.7ns

Table 2 shows the results of the whole receiver for different target devices. The receiver is split into 4 modules according to the block structure, all repeat frames are unrolled, modules Block and DAC are min-encoded, the Frame module is 1-hot encoded, and the Search module is distributed encoded. These protocol synthesis settings were found by trying out different settings for each module and using that one with the best results, e.g. unrolled and min-encoding for the DAC module as Table 1 shows. Protocol and logic synthesis are done for each module individually, followed by a final logic synthesis run with "-ungroup all". Area and delay values for FPGAs were obtained after P&R with XACT; values for the LCA10K library were obtained after logic synthesis with Design Compiler.

Table 2: Protocol Compiler Results for Different Target Architectures

Target	Area	Critical Path
XC3042A-7	103 CLBs	53.4ns
XC4003-6	81 CLBs	57.6ns
LCA10K	1350 gates	34.8ns

Table 3 shows results for different design styles. The values in the table refer to the whole receiver and also refer to experienced designers. Schematics are derived from the Verilog RTL solution,

Table 3: Results for Different Design Styles

Design Style	Area	Critical Path	Design Time
Protocol Compiler	103 CLBs	53.4ns	2-3 days
Verilog-RTL	90 CLBs	53.2ns	1 week
Verilog-RTL +Schematic	114 CLBs	33.6ns	4-5 weeks

therefore the 4-5 weeks include 1 week for RTL. About 2/3 of the

Protocol Compiler frames describing the design are shown in this paper. The Verilog RTL solution consists of 912 lines, 570 of them comments or blank lines. The results show that the design time with Protocol Compiler compared to RTL methodology was improved by 1.5x -2x and compared to RTL+schematic by 10x.

7. SUMMARY AND CONCLUSIONS

This paper described the design of a digital audio receiver for the SPDIF format used by CD-ROM or DAT devices. The design was done with Protocol Compiler, a high-level synthesis tool for the design of controllers. By comparing Protocol Compiler with a traditional design style of writing a Verilog RTL file and then either automatically (logic synthesis) or manually (schematic entry) mapping it onto the FPGA the following conclusions were obtained:

- The Protocol Compiler description is far more readable than Verilog RTL, thus allowing faster entry, debugging, and changes. While the Protocol Compiler design is suitable for later re-use in a similar project, the RTL and schematic solutions are not.
- Protocol Compiler cut the design time compared to RTL design style by 1.5x - 2x.
- Protocol Compiler designs can always be synthesized and their quality (area, timing) is acceptable. Furthermore, design exploration is much simpler.

Altogether, applying Protocol Compiler to this design showed that it is a productive high-level synthesis tool for controllers. We think that our conclusions apply to other ASIC designs in the networking area such as (ATM, SDH/SONET, MPEG, 1394) as well because the design requirements are similar.

8. REFERENCES

[Ber92] G. Berry and G. Gainsayer, “The ESTEREL synchronous programming language: design, semantics, implementation”, in *Science of Computer Programming*, Nov. 1992, vol. 19 (no. 2): pp. 87-152.

[Cre96] A. Crews and F. Brewer, “Controller Optimization for Protocol Intensive Applications”, in Proceedings of the European Design Automation Conference 1996, Geneva, Switzerland, September 1996, pp. 140-145

[Har87] D. Harel, “Statecharts: A Visual Approach to Complex Systems,” in *Science of Computer Programming*, Aug. 1987, vol. 8 (no. 3), pp. 231-275.

[IEC958] “Digital Audio Interface”, International Standard IEC958, First edition 1989-03.

[Mey97] Wolfgang Meyer, Andrew Seawright, Fumiya Tada: “Design and Synthesis of Array Structured Telecommunication Processing Applications.” in Proceedings of the Design Automation Conference 1997, Anaheim, June1997, pp. 486-491.

[Sea94] A. Seawright and F. Brewer, “Clairvoyant: A Synthesis System For Production-Based Specification,” in *IEEE Trans. on VLSI Systems*, June 1994, pp. 172-85.

[Sea96] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck, “A System for Compiling and Debugging Structured Data Processing Controllers”, in Proceedings of the European Design Automation Conference 1996, Geneva, Switzerland, September 1996, pp. 86-91

[Son83] Sony/Philips: “Digital Audio Interface for Domestic Use”, September 1983.

[Tou93] H. Touati and G. Berry, “Optimized Controller Synthesis Using Esterel”, in *Proc. International Workshop on Logic Synthesis IWLS’93*, Lake Tahoe, 1993.