Using the Protocol Compiler for a critical FPGA Design

Peter Blinzer
Technical University of Braunschweig
Department of Integrated Circuit Design (E.I.S.)
Postfach 3329, Gaußstraße 11
38023 Braunschweig, GERMANY
blinzer@eis.cs.tu-bs.de

Abstract:
The Protocol Compiler is a new tool focused on the design of controller circuits for processing data protocols. We used it for the design of an FPGA based circuit, which transformed a serial stream of digital audio data conforming to the SPDIF protocol into audible sound by controlling a digital to analog converter. Having already obtained solutions for this design problem by both logic and RTL synthesis, we not only knew what was feasible, but also had strict constraints implied by an existing hardware testbed. Experiences in designing this circuit using the Synopsys Protocol Compiler are at the center of this paper.

1 Introduction

At the Department of Integrated Circuit Design of the Technical University of Braunschweig, we design integrated circuits for research and educational purposes. One of these circuits, which we used successfully in the past two years as a demanding FPGA design exercise, is a protocol converter circuit for digital audio transmission using the SPDIF protocol. The basic function of this circuit is to receive and analyse an incoming digital audio data stream and to forward the included audio sample data to a digital to analog converter by a simple write access protocol, producing audible sound as a result (a more detailed description is given in [1] and [2]).

The design of this circuit offers several challenges, since an input data rate of 5.6MBit/s has to be processed for bit synchronization, protocol analysis and data reformatting in real time, within a relatively small and slow FPGA. The basic structure consists of several communicating and parallel working finite state machines to control processing of the incoming data, which were highly optimized both for speed and area.

When we got the chance to work with one of the early versions of the Protocol Compiler, it was a natural decision for us to try it out with a data protocol oriented problem, for which we already had a solution, so we could classify the results properly. For the digital audio converter we had designs created by both logic and RTL synthesis, which were implemented with great care. We also had a hardware testbed for it allowing expressive testing in a real world, real time environment. So we decided to use this converter as our first project for the Protocol Compiler.

2 Getting acquainted with the Protocol Compiler

Being used to work with schematic entry or synthesis of register transfer level Verilog or VHDL the design technique of the Protocol Compiler seems a little bit strange at first sight. But the high abstraction of the modelling methodology, its capabilities and graphical visualization are very straightforward oriented towards the intended application area. These characteristics are well focused on the design of synchronous controllers performing at certain trigger conditions hierarchically grouped patterns of actions, as usually implied by data communication protocols [3]. Therefore getting acquainted to the tool is not really complicated. Within a short time the personal cognitive skills adapt to the graphical visualization of the design elements (“frames”), allowing designs to be grasped with a glimpse.

The application of the Protocol Compiler for our digital audio converter design was fairly easy, since this tool fitted well in the design flow we had already used for Verilog HDL synthesis. It actually worked as a substitute for manual register transfer level Verilog design (figure 1).

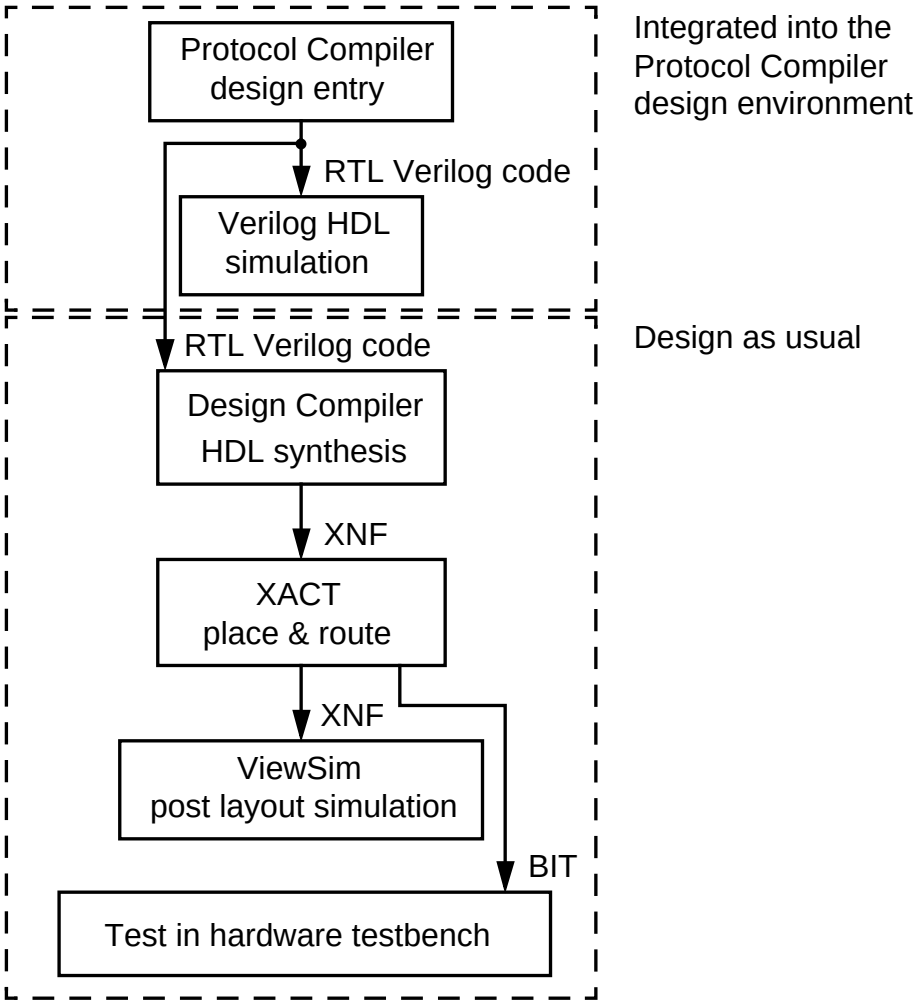

Figure 1: Protocol Compiler based design flow for the digital audio converter

The complete pre-synthesis simulation could be done conveniently within the environment of the Protocol Compiler, due to tool linkage by the Verilog Programming Language Interface (PLI). Another comfortable feature was the ability to reuse an already existing Verilog test-bench for this simulation.

The smooth adaption of a known and feasible design flow allowed us to concentrate on modelling and design space exploration with the new tool, rather than to fight tool interfacing problems. This is also supported by the Protocol Compiler with its modelling methodology, which always leads to synthesizable HDL code (Verilog or VHDL) for the following steps in the design flow.

3 Basic modelling issues

The digital audio transmission protocol SPDIF consists of three hierarchical layers for bits, data blocks and data frames respectively. Due to this structure of the protocol, an implementation with several parallel communicating state machines is in order. With the additional need for handling a digital to analog converter with a simple write access protocol this leads to a top-level structure of four parallel Protocol Compiler frames (figure 2).

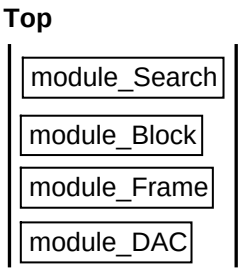

Figure 2: Top level of the SPDIF converter model using the Protocol Compiler

Protocol synchronization and bit decoding is done in the frame module_Search, whose output is further processed by the frame module_Block, collecting the 28 payload bits of each data block. The status data, spread up over several data blocks (a data frame of 192 block pairs), is monitored by module_Frame. If everything turns out to be correct, the received audio sample data is forwarded to an digital to analog converter (DAC) for audio playback by module_DAC.

In contrast to schematics or register transfer HDL models the single modules do not use wires and registers for internal data handling and exchange, but variables of one or more bits. These are globally declared for a Protocol Compiler model. Data input and output of the controller is done via ports, which have the same function of unidirectional ports on the gate or register transfer level.

Below the top level there are of course more frame references in the design hierarchy. The actions of this model have been coupled to elementary frames, located in the basement levels of this hierarchy. Since there is a relatively straight data flow within the model, the following descriptions will focus on this flow. Special attention will be given to modelling techniques used both for common protocol processing and critical design implementations.

4 Protocol synchronization and bit decoding

The input data stream comes as a single bit signal, which transports bits at a rate of 5.6MBit/s. The information in the input stream is self clocking by means of the biphase code, which creates signal edges both at the boundaries of encoded data bits and right in the middle of encoded 1-bits. By knowing roughly the bit rate of the code bits, one can easily extract the clock and the data from this signal using a phase locked loop (PLL) and a XOR-function.

To ensure synchronization to the correct signal edges even in a stream consisting only of encoded 1-bits, the SPDIF protocol specifies certain header patterns to be used at the beginning of each data block and in the first block of a data frame, which violate the biphase scheme.

Since we wanted to avoid use of an analog PLL outside our FPGA chip, we aimed at the implementation of a purely digital PLL inside the chip. This implied a sampling of the input data stream at more than two times the code bit rate to ensure a minimum of one reliable sample per code bit. Therefore we chose finally an FPGA clock rate of 16MHz, yielding almost three bit samples per code bit.

Due to this requirements the frame module_Search was built as shown in figure 3.

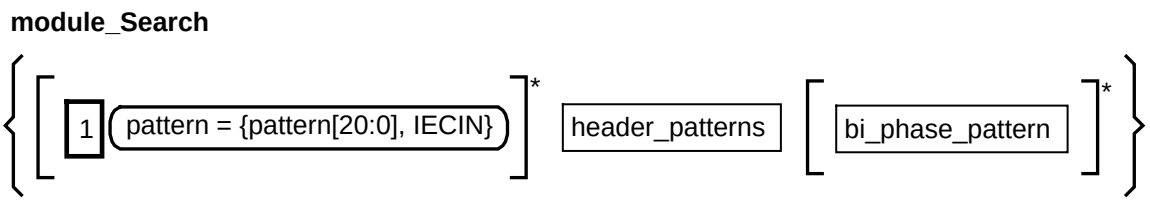

Figure 3: Implementation of the protocol synchronization and bit decoding frame

The frame consists of a sequence of one elementary frame coupled with an action and two frame references. The elementary frame is executed in every clock cycle, since it is always accepted (1-frame) and surrounded by a repeat operator. It samples data from `IECIN`, an input port of the complete model, into the lowest bit of the bit array variable `pattern` while shifting its contents up one index position, creating an upward shift register.

Because of the pipelined execution capability of the Protocol Compiler the execution of module_Search is not at a dead end. After each execution of the shift register action the frame header_patterns is activated to search for synchronization headers within the shift register. Only when a header pattern is found the frame header_pattern is accepted and the execution is continued with bi_phase_pattern for bit decoding in the remaining part of the data block.

For the recognition of header patterns a bit pattern comparison spanning two clock cycles was used (figure 4). This resulted in a much shorter and faster combinatorial logic path than a one cycle implementation, allowing us to keep in the timing constraints for the selected FPGA (a Xilinx 3042A-7) at the required clock speed of 16Mhz. The dashes in the patterns are used to exclude the positions of possibly instable signal values (in direct vicinity to expected edges) from the comparison.

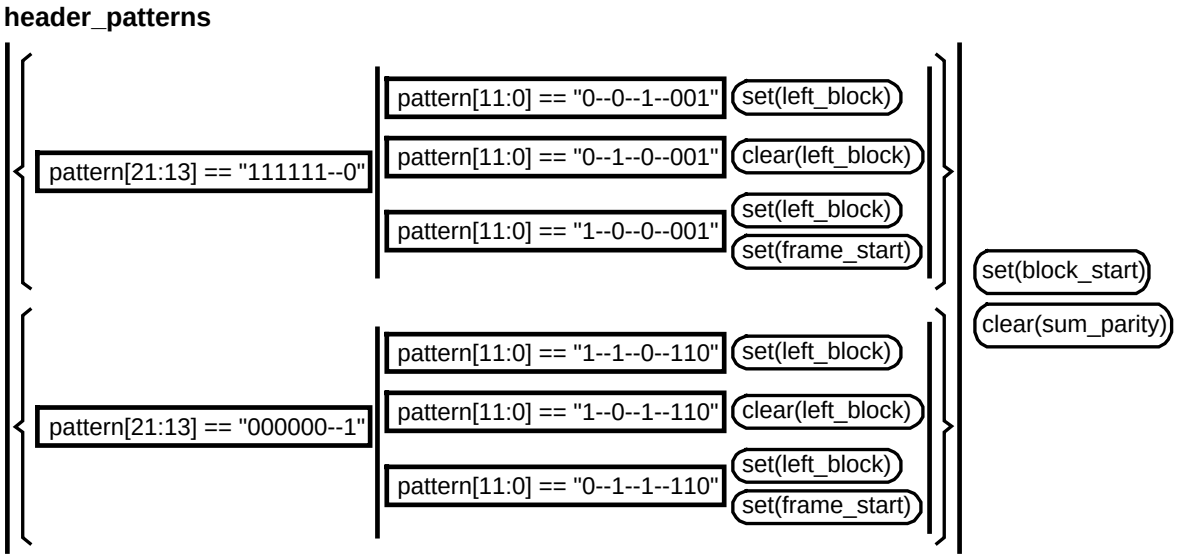

Figure 4: The synchronization pattern search

In the first comparision cycle `pattern[21:13]` is searched for the first part of the SPDIF header pattern, which can be "`11111--0`" or "`000000--1`". Only if this search is successful, the comparison is continued in the second cycle for the possible remainders of the patterns in `pattern[11:0]`. When a search path completes, the appropriate actions are taken, such as flagging the header type and the beginning of a new block via variables of the model. Since this module is required to be started in each clock cycle by the referencing frame, it will also make use of the pipelined execution capability without any additional intervention.

After successful completion of `header_patterns` bit decoding takes over in the frame `module_Search` with `bi_phase_pattern` (figure 5).

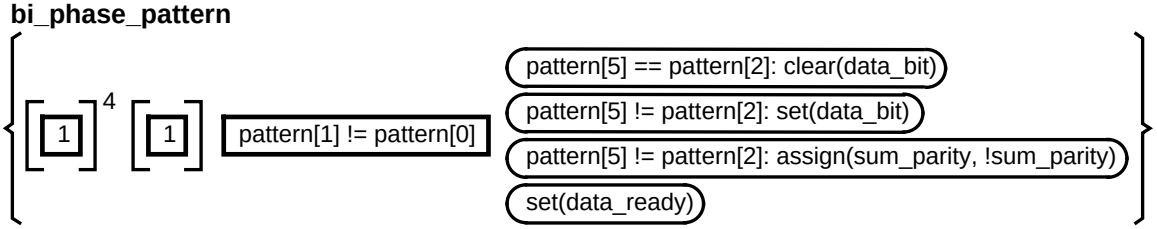

Figure 5: The digital PLL combined with the biphase decoder

The elementary frames of bi_phase_pattern actually form a digital PLL locking on to the data bit transmission rate, which is half the code bit transmission rate. Due to the input bit sampling rate of 16Mhz and the code bit transmission rate of 5.6MBit/s there are about 2.8 samples per code bit or 5.6 samples per biphase encoded data bit. So there will be 5 or 6 samples for a data bit depending on the actual alignment of the samples within the bit transmission. The easiest way not to lose bit synchronization in this situation is to lock on to the signal edges of the data bits.

When bi_phase_pattern is activated the leading edge of the first data bit lies between index position 0 and 1 of the `pattern` variable, because of the header_pattern timing. The first elementary frame then spends four clock cycles in its repeat operator, while in parallel the shift register is updated with four more samples. The next 1-frame is optionally executed, when with the fifth cycle no signal edge appears, that would allow proceeding to the edge detection frame and its associated actions. On this way, the fluctuation of the number of samples per data bit is compensated very easily, resulting in a very reliable PLL locking mechanism.

If execution continues with the third frame in the fifth or sixth cycle, depending on the exact signal timing, a signal edge is mandatory for the frame to succeed and its actions to be executed. Otherwise, the execution of bi_phase_pattern is completely aborted, since the input bit stream does not conform to the biphase scheme (this is also true for header patterns). In case of success the frame actions decode stable sample positions to a data bit, update the parity control for the data block and mark the presence of a new valid data bit in `data_ready`.

From a modelling point of view the complete protocol synchronization and bit decoding could be implemented in a very simple and elegant way with the Protocol Compiler. But the very concise and expressive construction possible for the digital PLL was really astonishing, especially when compared to the much more complex implementations on gate or register transfer level.

5 Processing of data blocks

After being refined on bit level by module_Search the SPDIF protocol is further processed on its block level by module_Block (figure 6).

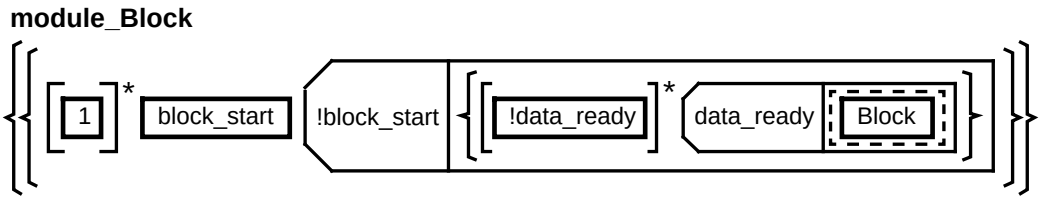

Figure 6: The processing of data blocks

This frame polls in every clock cycle the value of the bit variable `block_start`, which is set by header_patterns as a marker for a detected block synchronization and is otherwise reset by a Protocol Compiler default action declared in the default actions list. During the processing of one data block it is normally not possible for a new block to begin, so a conditional operator around the other block handling frames was used in order to avoid the unnecessary use of logic resources for a pipelined execution capability. This operator also guarantees the immediate abortion of the current block processing, if a synchronization failure should unexpectedly result in such a faulty restart condition, enabling resynchronization with minimum penalty.

After the gap between the block start and the first data bit occurrence has been bridged by a repeat frame looping in the condition of data_ready being 0, block processing advances step by step through the referenced frame Block with each activation of `data_ready`. This is done with a runidle operator, which stalls execution of Block as long as `data_ready` is 0.

Inside of the frame Block everything looks almost like a plain paper protocol specification sheet, specifying structure and order of the contained bits (figure 7).

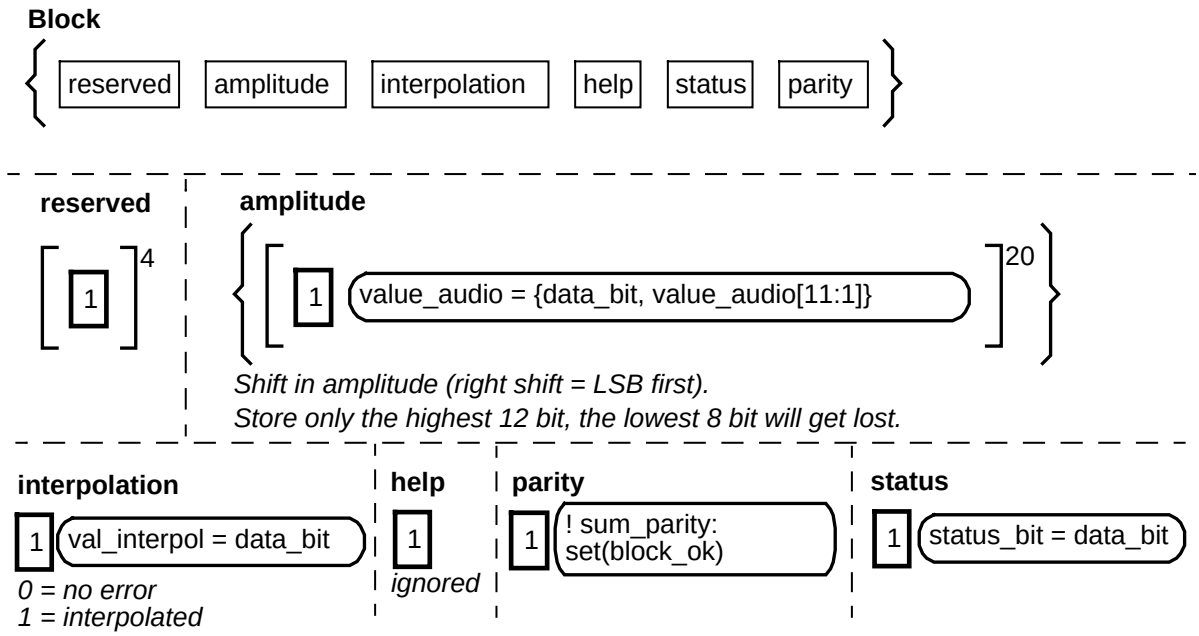

Figure 7: The block processing and its subframes

In contrast to a plain paper specification sheet this Protocol compiler code does also specify actions to be taken for each single bit of a block, which put every data at the right place for other frames to work upon.

6 Processing of data frames

The SPDIF protocol level of data frames constitutes the highest organizational level of this protocol. A frame consists of 192 pairs of data blocks, with each pair consisting of two data blocks, one for the left audio channel and one for the right audio channel. There is also some important status data spread over the status bits of the 384 data blocks in a frame, which are equal in a pair, so it is sufficient to monitor status bits for the blocks of one audio channel. This sounds very different from block processing, but as figure 8 shows, there are great structural similarities between the two processing levels.

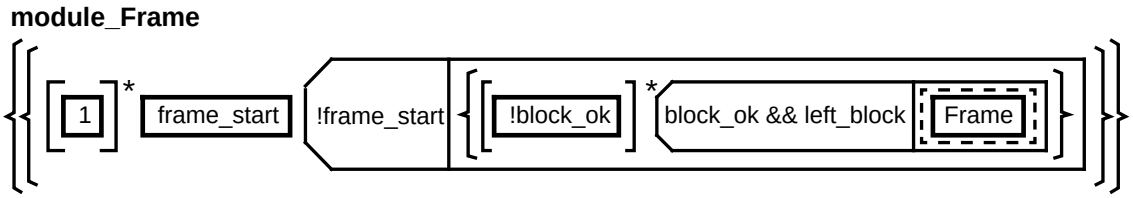

Figure 8: The processing of data frames

Again a variable (`frame_start`) is polled in every clock cycle for a value signalling proper synchronization within the processed protocol level (in this case data frames). The conditional operator then ensures, that no unnecessary pipelining logic is created and resynchronization from error conditions is done as fast as possible. The timing gap between the start of the data frame and the availability of the first complete block is another time bridged by a repeat operator, until finally execution passes over to the runidle operator for a step by step execution of Frame with each correctly received data block for the left audio channel.

As can be seen in figure 9, the similarities also extend to the subframes of this module.

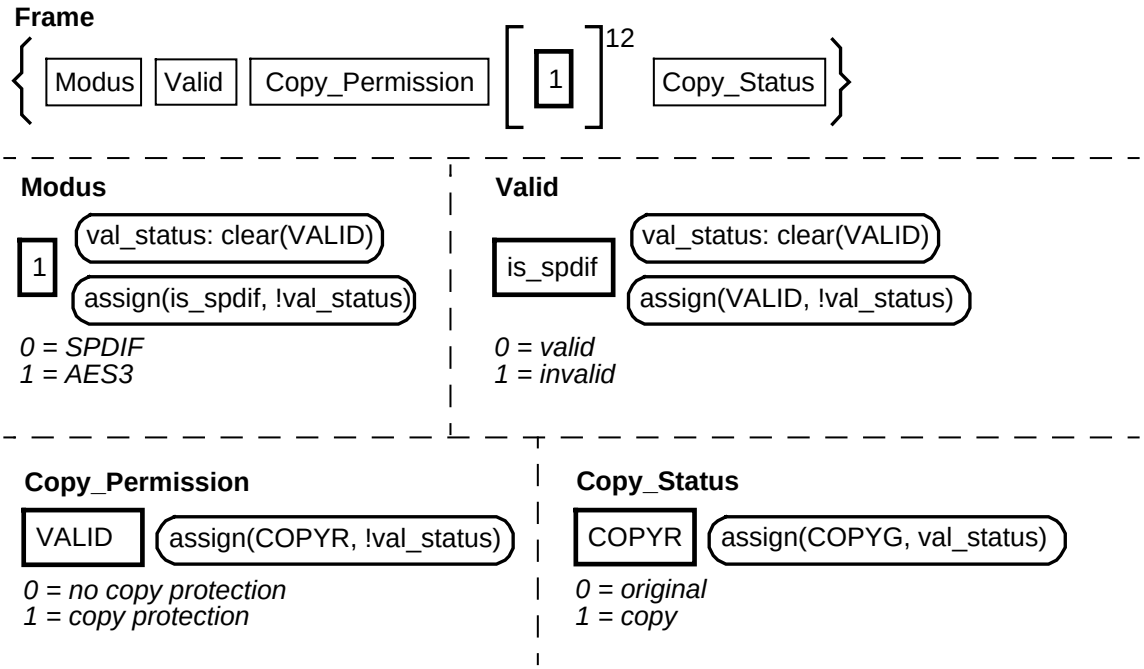

Figure 9: The data frame processing and its subframes

The collected status data is written out to ports (`VALID`, `COPYR`, `COPYG`) for status display and also handed over the to final top level frame of the model, module_DAC.

7 Sending audio samples to the DAC

The digital to analog converter in the testbench featured two output channels and was accessible through a four-wire serial interface (figure 10).

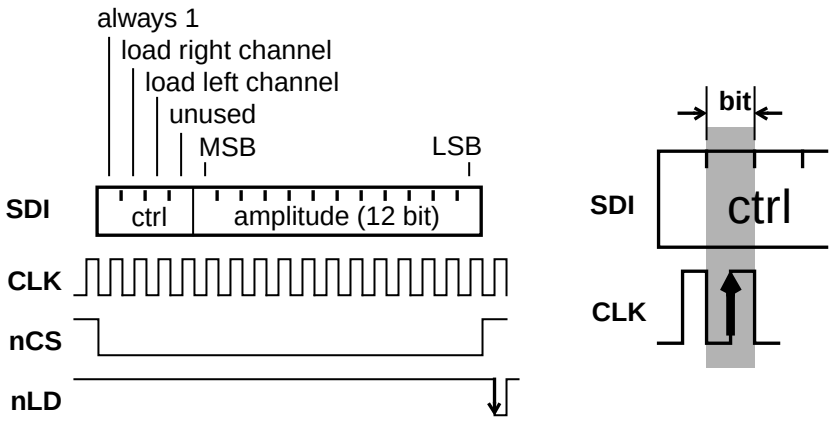

Figure 10: Write access protocol for the digital to analog converter

The audio sample data has to be sent to the DAC embedded in a 16 bit command frame, which includes enable bits for the output channels to be updated with the 12 bit sample value. The DAC data stream is transferred synchronously with the rising edge of a data clock signal, also to be generated by the data output controller. A race free transmission is ensured by placing the rising clock edge in the center of the transmitted data bit. After transmission of a DAC data frame the channel output is activated by a low pulse on the load signal (`nLD`). The only thing to be done in module_DAC, which is not fully specified by the DAC access protocol, is the need to invert the highest bit of the sample data to convert bipolar values to unipolar values.

The implementation of this frame follows in a very straightforward way as shown in figure 11, but is in contrast to the other frames displayed vertically.

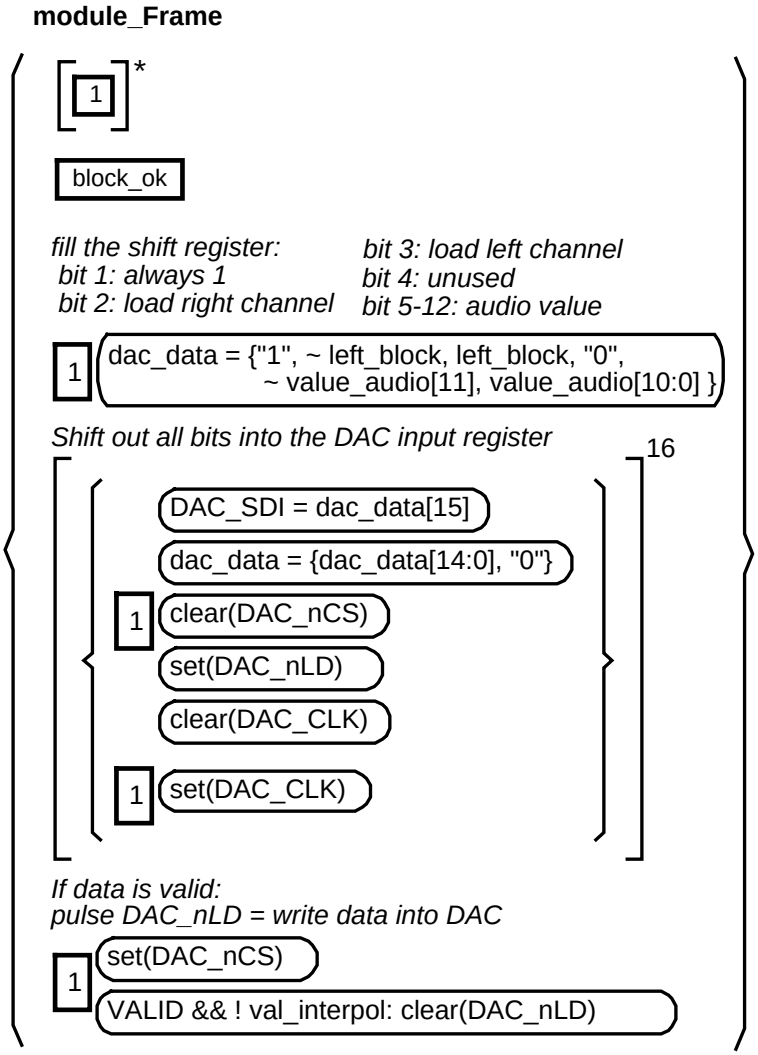

Figure 11: The DAC output protocol frame

The frame begins with a sampling phase, waiting for complete data block to be available by the model variables. Then the DAC command frame is assembled by a concatenation into an output shift register, whose contents is subsequently transmitted serially to the DAC. Finally a pulse on the load pin is used to convert valid audio data back to analog audio samples, ending the long digital way of the represented sound.

8 FPGA implementation

For FPGA implementation a Xilinx XC3042A-7 was used, which is a relatively small and slow FPGA nowadays. But this FPGA was installed in the existing hardware testbed, so we had not much choice.

The only design synthesis method of the Protocol Compiler for the complete design usable due to the FPGA constraints was the distributed FSM style. All other methods yielded code requiring too much logic resources after RTL synthesis with the Design Compiler.

At this point we had a working solution requiring 139 of 144 CLBs, with a critical path about 1ns faster than required (61.4ns, at 16Mhz 62.5ns would have been possible). This was however, far from being as fast and small as our existing solutions.

So we invested a little more design time, finding additional optimization techniques. First we made use of the poweron reset feature of the FPGA, by carefully changing the reset logic of the RTL Verilog netlist produced by the Protocol Compiler. This was fairly easy and reduced the amount of logic required by about 10%. Then we broke up the controller model into four single models (one for each top level frame) and used the optimum controller synthesizing method found for each of them. The distributed FSM style was now only for module_Search the best method. After all these manual optimizations we finally obtained the following results, which show that the results obtained with the protocol compiler can compete very well with other design styles after some finetuning:

Design Style	Area	Critical Path
Protocol Compiler	103 CLBs	53.4ns
Verilog-RTL	90 CLBs	53.2ns
Verilog-RTL +Schematic	114 CLBs	33.6ns

Table 1: Results for Different Design Styles

Conclusions

The Protocol Compiler proved to be a effective tool for synchronous controller design in our first project. There is a good potential for it, to make protocol oriented controller design a lot more easier and more effective for other designs and designers. The tool itself fits well into current design methodologies, which makes its application sure worth a try. Some optimizations done in the final stage of our project manually, however, should be included in the tool to make it more effective and more competitive.

References

[1] P. Blinzer; *Design of a Digital Audio Receiver in a VLSI Lab*; First Electronic Circuits and Systems Conference, September 4-5, 1997, Bratislava, Slovakia

[2] P. Blinzer; *Entwurf eines Digital-Audio-Receivers als Praktikumsaufgabe*; 8. E.I.S.-Workshop, April 8-9, 1997, Hamburg, Germany

[3] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe and J. Buck; *A System for Compiling and Debugging Structured Data Processing Controllers*; European Design Automation Conference 1996, Geneva, Switzerland