Implementing Signal Processing Algorithms on FPGAs

Ruben Bartholomä, Frank Kesel
University of Applied Sciences Pforzheim
Tiefenbronner Str. 65, D-75175 Pforzheim, Germany
Email: kesel@fh-pforzheim.de, Tel.: ++49/7231/286567

Abstract: *In this contribution we will show how high-level synthesis can be applied to solve the problem of mapping signal processing algorithms on FPGA hardware. The algorithms will be described in C++ and mapped onto a Xilinx FPGA using a high-level synthesis tool.*

1 Introduction

Today digital signal processing applications (DSP), like e.g. telecommunications or image processing, need considerable computing performance in order to work in real time. In some cases the performance of DSPs (Digital Signal Processor) is not sufficient to guarantee real time behavior. In these cases the problem is often solved by using dedicated hardware to do some pre-processing of the data or even implement the whole signal processing system. Since hard-wired ASICs (Application-Specific Integrated Circuits) are too expensive and not flexibel enough, FPGAs (Field-Programmable Gate-Arrays) have proven to be a viable alternative for implementing DSP algorithms in hardware in the last decade. FPGAs belong to the class of programmable logic devices (PLD) and are reprogrammable or reconfigurable. The reason why hardware implementations of DSP algorithms show a superior performance in many cases compared to a software implementation on a DSP is a different computing paradigm. Compared to a DSP, which implements basically the sequential “von Neumann” computing principle (computing-in-time), FPGAs can implement algorithms with much stronger parallelism, since the hardware is inherently parallel (computing-in-space). Unfortunately the design of FPGA hardware needs normally more effort than a software implementation on a DSP. Furthermore the development tools like hardware description languages (HDL, e.g. VHDL), simulators and synthesis and the way in which the implementation is described differs much from what the system or software engineers are used to, like e.g. describing the problem in tools like Matlab [6] or in languages like C/C++. On the other side tools have been developed in the last decade to describe hardware on a higher abstraction level, the so-called algorithm level, see e.g. [1], and to automatically map the algorithmic description onto a register-transfer-level-model and finally to a gate-level-implementation of an ASIC- or PLD-library. A commercial tool which implements this so-called “high-level-synthesis” (HLS) or “behavioral synthesis”, see e.g. [2] or [3], is the “CoCentric SystemC Compiler” from Synopsys [10]. It accepts algorithmic descriptions in VHDL [4] or C++/SystemC [5]. According to the constraints given by the designer it tries to find an implementation for the given algorithmic description. In this contribution we will show how HLS can be used to implement DSP algorithms on FPGAs and we will evaluate the results with respect to computing performance and resource usage. In section 2 we will show how HLS basically works. Section 3 describes the FIR (Finite-Impulse Response) filter structures used for the evaluation. Section 4 presents the results and compares the HLS results with results from Matlab/Simulink [6].

2 High-Level-Synthesis

Digital hardware design today usually starts with a description of the circuit on register-transfer-level (RTL), using languages like VHDL. The RTL description models the circuit in terms of its registers and the combinational transfer functions, which are described with control structures similar to high-level programming languages, like “if”, “case” or “for”. The RTL-model is bit-correct and clock cycle accurate. An algorithmic description, according to the Gajski-Y-diagram [1], describes the circuit one abstraction level above the RTL. In this case the model is still bit-correct but not clock cycle accurate. This gives the freedom to implement the description with a variable number of clock cycles, which in turn enables the time-sharing of resources, like adders or multipliers. Therefore by using HLS one can trade-off the number of clock cycles (performance) against the number of adders or multipliers (resources) for different RTL-implementations of the same algorithmic description. This is not possible by using standard RTL synthesis tools.

This can be shown by the simple example in figure 1. Assuming that the variables are implemented with 16 bit, we need an internal bit width of 32 bit. If we synthesize this code with a standard RTL synthesis tool, like e.g. “Leonardo” from Mentor [7], for an Altera Flex10K20 FPGA [8], it will consume 95% of the resources, since we need 2 adders, 1 subtractor and 2 multipliers. The resulting logic is fully combinational, i.e. no registers are used.

```
int design(int a, int b, int c, int d, int e, int f)
{
        int y;

        y = a * b + c + ( d - e ) * f;

        return y;
}
```

Figure 1: Algorithm example

In contrast to standard RTL synthesis HLS can transform this description also into a sequential implementation which uses 1 adder/subtractor and 1 multiplier. Registers and a finite state machine are generated such that the circuit needs 3 clock cycles to calculate the result. We will describe in the following briefly how HLS achieves this transformation. In the first step a data flow graph (DFG) is extracted from the algorithmic description, which models the data dependencies. Each node represents an operation and two nodes are connected by an edge if there is a data dependency. Control structures like “if” can be represented in an extension, the Control DFG (CDFG) see [3]. In the following phase (“resource allocation”) the type and number of resources or (arithmetic) components will be determined. First one needs the library for the ASIC or FPGA device and second the designer has to define the number of components to be used. This step basically determines if we will have a more parallel (more resources, faster) or a more sequential implementation (less resources, slower). The following “scheduling” assigns the operations to clock cycles with respect to the DFG dependencies and results in a scheduled DFG according to figure 2. Scheduling is the basic technology for HLS and is usually implemented by heuristic algorithms, see e.g. [3]. If an edge crosses a clock cycle boundary then a register is needed to hold the (intermediate) result (“register allocation”). Lifetime analysis tries to minimize the number of necessary registers by sharing them in different clock cycles, e.g. v1 and v4 in figure 2 could use the same register. Next the operations will be assigned to components (“binding”), which determines the number of necessary multiplexers, and finally a finite state machine (FSM) is constructed, which is necessary to control the datapath. Our simple example does not contain loops, like e.g. in the description of an FIR filter. In standard RTL synthesis loops in the code are always unrolled. This can result in considerable hardware, since it is a fully parallel solution. Therefore loops are normally used with care in RTL design. Since HLS constructs a FSM or sequencer it is possible to not unroll (fully sequential) or only partially unroll the loops.

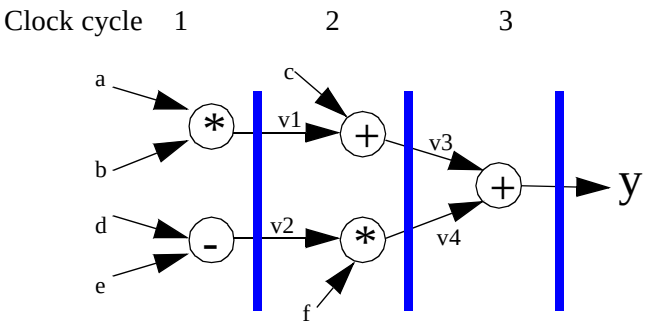

Figure 2: DFG with scheduling informations

3 FIR Filter Structures

FIR filters are one of the most important operations in digital signal processing systems. The output signal *y* of a FIR filter can be determined by the convolution of the input signal *x* with the impulse response *h* of the filter, as shown in equation (1). *N* denotes the number of taps used in the filter.

$$y(n) = \sum_{i=0}^{N-1} h(i) \cdot x(n-i) \qquad (1)$$

The signal flow graph for the direct form of an FIR filter is shown in figure 3(a). It can be directly constructed from equation (1). The signal flow graph of the transposed direct form, as shown in figure 3(b), can be obtained by applying the transposition theorem to the signal flow graph of the direct form [9]. A fully parallel hardware implementation of both filter structures requires *N* multipliers, *N-1* adders and *N-1* registers. Due to the fact, that the direct form of the FIR filter merely needs to hold samples of the input signal in registers, there are only $(N-1)\cdot W_x$ flipflops necessary. W_x specifies the wordlength of the input samples x. However, the transposed direct form of the FIR filter requires more flipflops, because the registers hold the results of previous sums of products. An addition of two products may produce a carry bit, which has to be hold for preserving full precision. An upper bound for the total number of flipflops can be expressed as $(N-1)\cdot(W_x + W_h + \lceil ld(N) \rceil)$, where W_h denotes the wordlength of the filter coefficients *h*. Considering the critical path, which is the maximum delay between two registers, of both structures, we can see significant advantages of the transposed direct form. Let *T* be the delay of an adder and *3T* the delay of a multiplication, then the total delay of the critical path of the transposed direct form is *4T*. In the case of the direct form the total delay of the critical path is $(N+2)\cdot T$, which reflects a linear dependency on the number of taps.

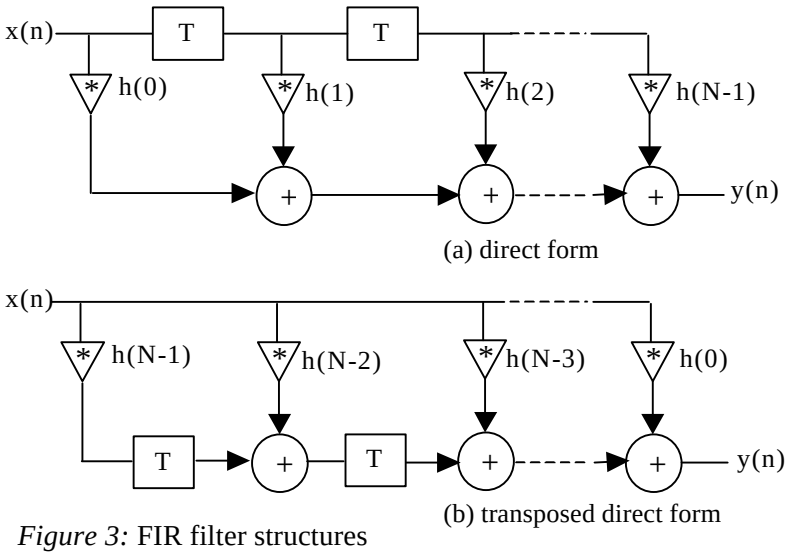

Figure 3: FIR filter structures

4 Results

This section presents the results of our study. We have generated different hardware architectures of a FIR filter, using the HLS tool "CoCentric SystemC Compiler" from Synopsys and targeting a Xilinx FPGA XC2V250 [11]. We have also created the same FIR filters with the "System Generator" from Xilinx, in order to compare the efficiencies of both approaches, according to figure 4, in terms of the tradeoff between resource usage (measured as equivalent gate count) and throughput (measured as maximum sampling rate). The "System Generator" can be used with Matlab/Simulink. It enables the designer to model a system, using Xilinx IP Cores and lower level elements like e.g. multipliers and registers. The system can then be simulated and mapped onto the Xilinx FPGA. It must be noted that the "System Generator" approach does not use HLS, but is based on a library approach using pre-designed and optimized hardware blocks for a certain FPGA family. Therefore the approach is hardware dependent. The architectures generated with "CoCentric SystemC Compiler" on the other hand stem from one single C++ hardware independent algorithmic description of the filter. Figure 4 depicts the total equivalent gate count depending on the data throughput for different architectures of a 16 tap FIR filter, which has a data and coefficient wordlength of 16 Bit. All architectures make use of the embedded Xilinx Virtex multipliers. The results show that the tradeoff between area and speed can be approximated as a linear function in both cases. The fastest architectures in both approaches implement the tranposed direct structure with 16 multipliers and 15 adders, as mentioned in section 3. Slower architectures share adder and multiplier resources, which require additional logic for implementing control

logic (e.g. multiplexers) and additional registers. Figure 4 shows that there is nearly no difference in both approaches in the case of the fastest solution. In case of the slower solutions the “System Genera-

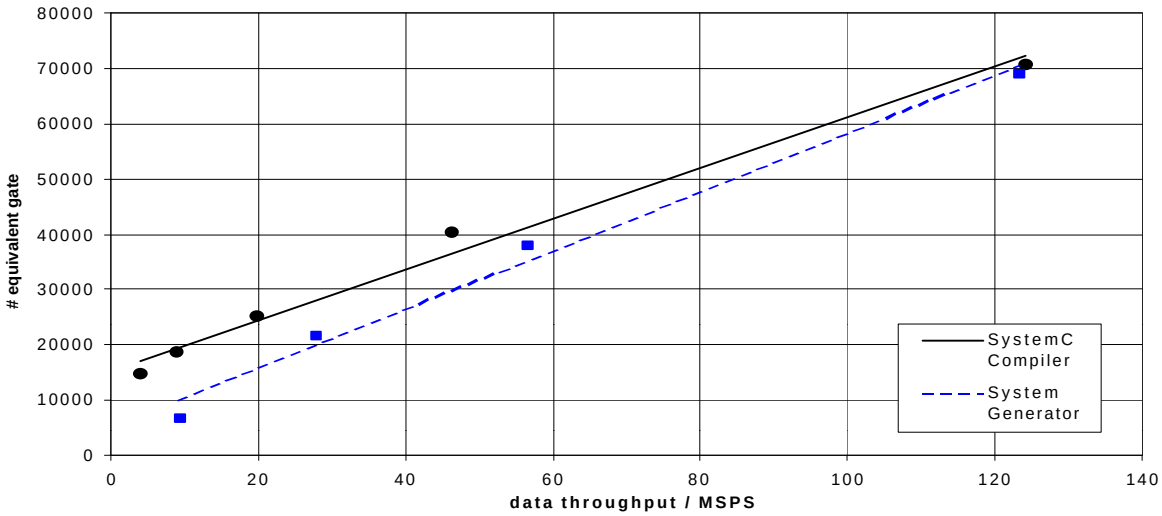

Figure 4: Efficiency of FIR filter implementations

tor” approach shows better efficiency, which can be attributed to a better control logic implementation. Further speedup of the throughput could be achieved in both approaches by pipelining the datapaths, consisting of multipliers and adders.

5 Conclusion

In this paper we have presented how HLS tools can be used for mapping signal processing algorithms onto FPGAs. We have implemented FIR filter architectures, using the HLS tool “CoCentric SystemC Compiler“ and compared the results with similar architectures, designed with “System Generator”. The comparison shows that both approaches are equally efficient with respect to the fastest or fully parallel solutions. With decreasing parallelism or increasing sequentialism the “System Generator” solutions are more efficient. One advantage of the “System Generator“ is its possibility to model hardware architectures within the Matlab environment, which is often used for designing signal processing algorithms. However modelling hardware with the “System Generator” is similar to RTL modelling and thus time consuming and in this case also hardware dependent. In order to have a design path from a Matlab model to SystemC and HLS we are currently working on generating SystemC descriptions from Matlab models.

6 References

[1] Gajski, D.D., Dutt, N.D., Wu, A.C.; High-Level Synthesis: Introduction to Chip and System Design; Kluwer Academic Publishers, 1992

[2] Knapp, D.W.; Behavioral Synthesis - Digital System Design Using the Synopsys Behavioral Compiler; Prentice Hall, 1996

[3] Elliot; J.P.; Understanding Behavioral Synthesis; Kluwer Academic Pub., 2000

[4] Ashenden, P.J.; The Designer's Guide to VHDL; Morgan Kaufmann, 1996

[5] Grötker, T., Liao, S., Martin, G., Swan, S.; System Design with SystemC; Kluwer Academic Publishers, 2002

[6] Matlab-Homepage “www.mathworks.com“

[7] Mentor-Homepage “www.mentor.com”

[8] Altera-Homepage “www.altera.com”

[9] Wanhammar, L.;DSP Integrated Circuits; Academic Press, 1999

[10] Synopsys-Homepage “www.synopsys.com”

[11] Xilinx-Homepage “www.xilinx.com”