USING FAUST FOR FPGA PROGRAMMING

Robert Trausmuth, Christian Dusek, Yann Orlarey

USING FAUST FOR FPGA PROGRAMMING

Robert Trausmuth, Christian Dusek
Dept. of Computer Engineering
University of Applied Science
Wiener Neustadt, Austria
trausmuth@fhwn.ac.at

Yann Orlarey
GRAME
centre national de creation musicale
Lyon, France
orlarey@grame.fr

ABSTRACT

In this paper we show the possibility of using FAUST (a programming language for function based block oriented programming) to create a fast audio processor in a single chip FPGA environment. The produced VHDL code is embedded in the on-chip processor system and utilizes the FPGA fabric for parallel processing.

For the purpose of implementing and testing the code a complete System-On-Chip framework has been created. We use a Digilent board with a XILINX Virtex 2 Pro FPGA. The chip has a PowerPC 405 core and the framework uses the on chip peripheral bus to interface the core.

The content of this paper presents a proof-of-concept implementation using a simple two pole IIR filter. The produced code is working, although more work has to be done for implementing complex arithmetic operations support.

1. INTRODUCTION

Modern FPGAs (**F**ield **P**rogrammable **G**ate **A**rrays) allow the system engineer to design full systems in one chip by defining the desired logical functionality of the chip using a hardware description language. The inherent parallelism of the FPGA logic can be of advantage once it comes to heavy calculation tasks or to problems which use many parallel processes. Our motivation for this project was to create a system on chip design tool which can be used for audio signal processing. Although there are commercial products available, this implementation is part of an open source project and thus can be used and developed further by interested partners.

To ease the programming task we chose the **F**unctional **AU**dio **ST**ream programming language FAUST, developed at GRAME Centre National de Creation Musicale, Lyon, France. This language is based on a block-diagram algebra [1]. All program elements are described as building blocks, the final system consists of a hierarchy of those blocks. Each block has a defined interface (input and output). Once the desired signal processing algorithm is programmed using the FAUST syntax it gets compiled into VHDL logic. VHDL stands for **V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage, a programming language to develop on-chip logic designs. These designs are the input to the chip synthesis tool.

The FAUST compiler originally creates C++ code which can be put into a framework and later be used as plugin for a series of wellknown (software) audio processing programs. Our addition to the system is the possibility to generate on-chip logic to do the task and use the processor on chip for slow control and user interface. With this we are close to the current developments of hardware/software co-design tools (for a prominent example see [2]).

The choice of the target platform was quite easy because of the experiences with the POWERWAVE synthesizer [3]. The board uses an AC'97 compatible audio codec with 48 kHz sampling rate and 20 bits resolution. The XILINX Virtex 2 Pro has two PowerPC 405 processors on chip running at 300 MHz. One of those is used for implementing the slow control and user interface. The VHDL building blocks are embedded in an OPB module. The **O**n-chip **P**eripheral **B**us is an IBM standard bus intended to connect hardware extensions to the on-chip processor. This bus runs at 100 MHz. The framework implements several dual port RAMs for exchanging parameter data with the audio processor, granting parallel access for parallel running blocks.

2. FAUST EXTENSIONS

The FAUST compiler produces optimized C++ code. The optimization tries to find out what has to be computed and omits all unnecessary parts. For the purpose of FPGA programming we need VHDL modules. Since the implementation of logic on FPGAs has different needs, there are some alterations and extensions needed for the original FAUST compiler. The compiler works in several stages shown in figure 1.

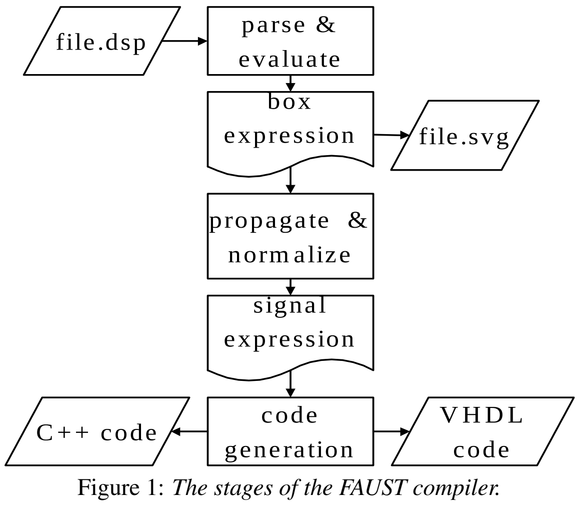

Figure 1: *The stages of the FAUST compiler.*

The role of the compiler is to translate from FAUST to C++ or from FAUST to VHDL, while preserving the mathematical semantic. To do that, the compiler must first discover the mathematical semantic of a FAUST program. This is the **propagation** phase of the compiler. It translates box expressions into normalized signal expressions that express the semantic of the box expressions. In theory (and in practice to some extent) two very different FAUST programs which actually compute the same thing from a mathematical point of view should result in the same signal expression after the propagation phase.

The **code generation** phase operates on signal expressions, not on box expressions. The current C++ generator translates these signal expressions into equivalent C++ code. It tries to generate the most efficient C++ code while preserving the mathematical semantic. The VHDL generator should do the same: it should operate on signal expressions and generate the most efficient VHDL code while preserving the semantic of the computation.

One of our main goals is keeping the VHDL code generation totally transparent to the end user. The final version of the compiler should work on any FAUST program (almost) without alterations on the code itself. So any considerations concerning the extensions of the FAUST compiler have to take into account this required transparency.

2.1. Fixed point number representation

The C/C++ code uses floating point number representation. Although VHDL suports floating point numbers, this floating point capability is not easily synthesizable to the chip logic. Therefore we need to implement the capability of fixed point number calculation, which implies a few extensions concerning the VHDL code generator. Fixed point numbers have a total width and a defined binary point. Each fixed point building block produces an output which can be of the same width or even larger. Due to signal propagation times on the chip, there are some limitations concerning the total width of adders and multipliers which also have to be taken into account.

Usually there is a need for truncating the big fixed point numbers after a calculation block. This will be done by a special primitive which incorporates stochastic rounding [4]. This well known method is used in DSPs and helps reduce the quantization error which might be a problem when several stages are calculated one after the other.

To grant transparency to the user all necessary parameters like bit width of the signals, the number of memory and MAC blocks and the VHDL default blocks are predefined by the compiler. However, there is the possibility to include a definition file into the FAUST code to overwrite those default settings.

2.2. Calculation costs

VHDL blocks are usually implemented in a synchronous manner, which implies that it takes at least one clock cycle to evaluate each block. Some blocks may have longer delays, especially if they consist of several other blocks. Since we have a defined timeframe for the signal calculation - roughly 2000 clock cycles in our case - the design has to be checked against this boundary condition.

Each design block is afflicted with the clock cycles it takes to evaluate the block logic. If several processes run in parallel and meet at some point, the design has to assure that the following block gets the input values it expects. After the FAUST code has been processed, the internal signal model is evaluated and a Gantt chart is created to find the critical path through the model. This path determines the total signal propagation time of the VHDL unit. At the merge points of the FAUST model the implementation model has to assure the synchrony of the temporary calculation results.

2.3. VHDL base blocks

The VHDL result is based on VHDL base building blocks, which have predefined calculation costs. Every junction adds to the total calculation cost. The base cost is part of the VHDL model library. If user specific VHDL blocks are added (like external C functions can be added at the moment), this cost value has to be stated in the VHDL model. The FAUST compiler creates a timing report showing all signal propagation times and the defined fixed point interfaces.

The VHDL top level interface block is called "process" and contains the interface to the framework. All other VHDL modules are logically contained in this toplevel module. Additionally, a C++ header file is produced containing the synonyms and addresses for all the registers of the design. This header file allows the interface implementation running on the PPC to access the proper memory locations for the parameters.

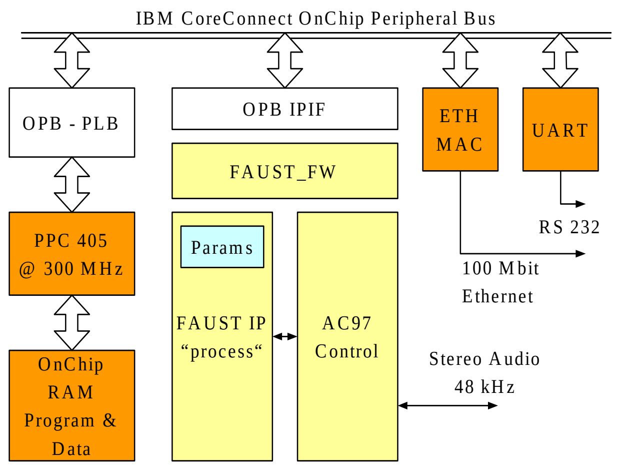

Figure 2: *Block diagram of the FAUST VHDL framework.*

The number of dual-port RAM blocks actually defines the maximum number of parallel data and address busses which can be used by the calculation cells. The number of parallel calculation cells (each one comparable to a small ALU with a command set optimized for DSP operations, see section 2.4) can also be specified.

Special care has to be taken concerning the width of the data bus as well as the duration and space cost of different block implementations. Although the bus width is set to 20 bits by default, this bus width can be changed. If

the 20 by 20 bit multiplier is too large to fit into the FPGA, another version (20 by 4 bits) is provided, which takes 5 clock cycles to calculate the result but only 1/4th of the FPGA fabric. Divisions on the FPGA logic are very costly and therefore not part of this implementation. Divisions by constants can be implemented using multiplications and reciprocal values.

A typical calculation step takes at least 6 clock cycles. The necessary operations are executed in a pipelined mode: the first 4 clock cycles are generally needed to fill the local registers with the

ALU command	mathematics	execution time
madd R1 R2	$A = A + R1 * R2$	1 (+ 4)
msub R1 R2	$A = A - R1 * R2$	1 (+ 4)
madd4 R1 R2	$A = A + R1 * R2$	5 (+ 4)
msub4 R1 R2	$A = A - R1 * R2$	5 (+ 4)
rounda	$A = rnd(A)$	1
clra	$A = 0$	1
mova R1	$R1 = A$	1
sra R1	$A = A \gg R1$	1 (+ 3)
srl R1	$A = A \gg R1$	1 (+ 3)
sla R1	$A = A \ll R1$	1 (+ 3)
sll R1	$A = A \ll R1$	1 (+ 3)
and R1	$A = A\&R1$	1 (+ 3)
or R1	$A = A\|R1$	1 (+ 3)
xor R1	$A = A\hat{\ }R1$	1 (+ 3)
not R1	$A = \tilde{\ }A$	1

Table 1: *ALU command set*

needed values, although register reuse and optimization can drop those steps. Then the ALU block takes at least one cycle to do the real calculations. After this another cycle may be necessary to store the ALU accumulator back to a local register. The ALU commands are explained in the next section.

2.4. ALU command set

All the FAUST calculations are finally done by a special ALU block which supports a few calculation operations. Since memory operations are handled by special bus controller processes, the ALU only needs the calculation commands shown in table 1. Logical decisions which are needed for branching during the execution of the program are not part of the language and therefore not part of the calculation units.

Execution times for the ALU commands are given in clock cycles, the additional values show the clock cycles used for loading the needed values into local registers, if necessary. The special multiplication used by the *madd4* and *msub4* commands uses less logic cells on the FPGA but takes longer to calculate. The use of these commands can be defined by a global setting in the FAUST program.

2.5. Code generation strategy

The FAUST compiler parses the process model and produces a tree of signal expressions optimized to the mathematical semantic. The VHDL code generator first takes these signal expressions and translates them into a Gantt chart. This chart reflects all memory and calculation tasks in a linear manner in the beginning.

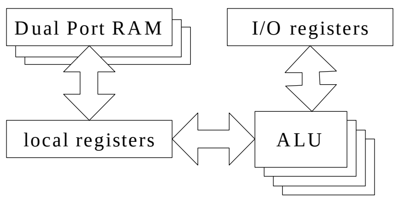

Figure 3: *VHDL architecture block diagram.*

The optimizer knows how many memory busses and ALU units are available and compacts the linear Gantt chart into a compact execution plan similar to the one shown in table 2. The optimizer determines the final number of local registers.

Each memory bus and ALU cell is then coded into one separate VHDL state machine, all being part of the FAUST IP "process" module. Figure 3 shows the block diagram of the synthesized architecture.

Each calculation is triggered by the AC97 module after receiving the next audio sample. This is the only synchronisation point in the design. After the start pulse all state machines run in parallel as designed by the optimizer. The optimizer has the responsibility to take care about execution times and synchronisation points throughout the VHDL design.

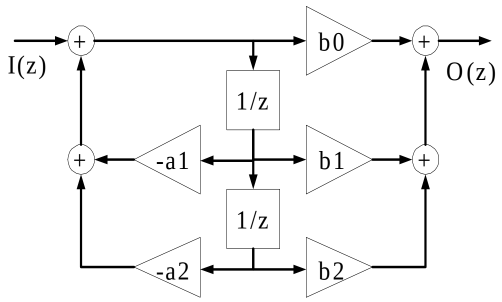

Figure 4: *block diagram of the 2 pole IIR filter used as reference design*

3. THE FRAMEWORK

The host for the FAUST VHDL module is a XILINX Virtex 2 Pro 30 with roughly 30.000 logic cells. The two PowerPC 405 CPUs use several chip busses to connect to the FPGA logic. Figure 2 shows the block diagram of the FPGA framework.

The VHDL module FAUST_FW is connected to the OPB bus. On the controller runs a small C program, so the user can talk to the system using ethernet or – in our first implementation – the serial RS 232 line. The OPB module provides the interface for the FAUST IP "process" module and also takes care of the audio codec. The AC'97 codec is initialized by the FPGA logic. Each sample is passed to the FAUST IP module using a simple handshake protocol. The FAUST module works on the audio data and provides the result for the next codec cycle.

There is an alternative operating mode (if we want to use block processing) in which 512 samples are stored in memory before the processing is called. In this case block operations can be used to calculate FFT or other block based algorithms. The block is shifted by 64 samples between two calls so the values keep overlapping. In this mode the recalculation is triggered every 64 samples, thus leaving 130000 cycles (or 1,3 ms) for one calculation run. The FFT algorithm as well as a few other block based signal algorithms will be one of the next extensions to the VHDL primitive library.

Parameters are stored in a dual port RAM section on chip. The FAUST module can access the parameters as needed and operates asynchronously to the PPC 405 control program. Several memory blocks can be used in parallel. In the default application, up to 4 busses can be used, each memory block containing 2 kbytes in 32 bit data width (512 parameter words). The FAUST optimizer takes care of parallel execution of the calculations. The master chip clock runs at 100 MHz, which gives about 2000 cycles per audio sample (still leaving 83 cycles for protocol handling between the process IP and the FAUST_FW IP).

4. THE IMPLEMENTATION

To show the implementation of the VHDL code generator we will use a simple example, the two pole IIR filter. This is used only as reference design in this paper, a detailed discussion can be found in [5]. The filter uses five parameters b0, b1, b2, a1 and a2 as well as two memory blocks. Figure 4 shows the block diagram.

The FAUST code to create this filter looks like this:

```
a1 = 1.73;
a2 = 1;
b0 = 1.25;
b1 = 1.73;
b2 = 1;

filter(b0,b1,b2,a1,a2) = + ~ conv2 : conv3
with
{
  conv2(x) = 0 - a1*x - a2*x';
  conv3(x) = b0*x + b1*x' + b2*x'';
};

process = filter(b0,b1,b2,a1,a2);
```

This code is parsed and normalized by the FAUST compiler to give a very short representation in C++. Basically, a few copy operations and the two additions in the main blocks conv2 and conv3 are left.

```
virtual void compute (int count,
 float** input, float** output)
{
  float* input0 __attribute__ ((aligned(16)));
  float* output0 __attribute__ ((aligned(16)));
  input0 = input[0];
  output0 = output[0];
  for (int i=0; i<count; i++)
  {
    float T0 = M0;
    M0 = R0_0;
    R0_0 = (input0[i] - (T0 +
           (1.730000f * R0_0)));
    float T1 = M1;
    M1 = R0_0;
    float T2 = M2;
    M2 = T1;
    output0[i] = (T2 + ((1.730000f * T1) +
                 (1.250000f * R0_0)));
  }
}
```

The translation into assembly code results in 54 CPU clock cycles for computing one filter cycle on a standard CPU (not on a DSP). The VHDL code makes use of parallel register loading and calculating and the calculations are done one per clock cycle. The total calculation time of the VHDL module is 11 clock cycles.

Putting the resulting VHDL code here would exceed the limits of this paper. However, table 2 displays the idea of the parallel processing used in the FPGA VHDL code.

The simple VHDL implementation uses one 42 bit MAC unit, one memory/bus controller and four 20 bit registers. The calculation and the memory access are done in parallel, thus saving time. Each row corresponds to one clock cycle, the result of each operation is available after one cycle. This result is comparable to an implementation on a modern DSP. However, when it comes to multiple filters in parallel, the FPGA gains advantage due to the parallel capabilities.

Reg0	**Reg1**	**Reg2**	**Reg3**	**MAC**
	load z^{-2}			0
		load z^{-1}		+input
			load a_2	
			load a_1	$-a_2z^{-2}$
			load b_2	$-a_1z^{-1}$
				round
copy MAC	load b_1			0
			load b_0	$+b_2z^{-2}$
		store as z^{-2}		$+b_1z^{-1}$
				$+b_0 * Reg0$
store as z^{-1}				round

Table 2: *execution plan for the IIR VHDL implementation*

If we put three IIR filters in parallel and use three MAC units and also three bus/memory units, the whole calculation will be done in 15 clock cycles.

5. CONCLUSIONS

The FAUST to VHDL compiler works for very simple problems where only basic mathematic functions are needed. If we want to use trigonometric functions, additional building blocks have to be provided in VHDL.

Complex mathematical calculations like the FFT algorithm can be implemented in VHDL as IP blocks and therefore improve the data throughput of the design. These IP blocks as well as the trigonometric function blocks will be provided in the future as external library blocks.

Since this is a proof-of-concept we are very positive that future work will produce a general purpose VHDL code generator.

6. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, “Syntactical and semantical aspects of Faust,” *Soft Computing*, vol. 8, no. 9, pp. 623–632, Sep. 2004.

[2] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, “hthreads: A hardware/software co-designed multithreaded RTOS kernel,” in *10th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, Catania, Italy, Sep. 2005.

[3] R. Trausmuth and A. Huovilainen, “POWERWAVE – a high performance single chip interpolating wavetable synthesizer,” in *Proc. Int. Conf. on Digital Audio Effects (DAFx-05)*, Madrid, Spain, Sep. 2005.

[4] C. Maxfield, *How Computers Do Math*. John Wiley & Sons, 2005.

[5] U. Zölzer, *DAFX – Digital Audio Effects*. John Wiley & Sons, 2002.