Published in
Proceedings of the International Conference on Signal Processing, Applications and Technology (ICSPAT), 1-5, 2000
which should be used for any reference to this work

ASIC DSP compiler for optimized synthesis

Andreas Drollinger, Alexandre Heubi, Peter Balsiger, Fausto Pellandini
Institute of Microtechnology, University of Neuchâtel
Rue A.-L. Breguet 2, 2000 Neuchâtel, Switzerland
URL: www-imt.unine.ch/esplab
e-mail: andreas.drollinger@imt.unine.ch

Abstract

This paper presents a high level DSP architecture compiler for cycle-constrained filters and datapath applications. The tool offers an easy way to get, from an equation representation of a filter, a synthetisable VHLD description of an application specific DSP architecture.
Inputs of the DSP compiler are an equation file to define the filter structure and a resource definition file to specify the available resource units. The equation syntax is very comfortable.
Resource mapping, scheduling, binding and furthermore the quantification of each operation is usually performed automatically, but can be controlled by the user. The result is a very fast filter synthesis time combined with highest flexibility for the users.

1. Introduction

Data flow graph description and simple mathematical equations are the two most commonly used filter representations. Many filter design tools exist, which automate the way or a part of it to get from such a representation an application specific DSP architecture. Nevertheless, the experience shows, that interface problems between the design tools, interaction limitations between tools and user, limitations for design constraints and the fact, that the tools are often optimized for a specific application domain raise the probability of unpredictable difficulties of the automated design flow. This is time-consuming and prevent from satisfying the increasing time to market constraints.
The aim of the presented work is to find out the real requests for a DSP compiler that automates the implementation phase of the development of a filter or datapath unit and to realize such a DSP compiler. Easy in the use, extendable, ability to produce efficient, low-power-oriented DSP architectures are the principal criteria for the DSP compiler.
The remainder of this paper describes the development of a high-level DSP compiler for the full automatic generation of low-power optimized DSP architectures for filter and datapath applications. Section 2 presents the request for the DSP compiler, while section 3 describes its realization. Section 4 shows an application example of the compiler with some results. Finally, conclusions are presented in Section 5.

2. Request for the DSP compiler

Most important of all, the DSP architecture compiler has to be very user-friendly and has to produce efficient implementation solutions. But, if it is needed for “non-standard” applications, it accepts also user interactions, can be programmed and user-specific extended. The following list shows more in detail the initial requests that are imposed on the development of the DSP compiler.

- The compiler can be used for cycle-constrained, low-power applications.
- It is OS and third party tool independent.
- Equations specify the data flow graph.
- The available resource units are user selectable.
- Synthetisable VHDL code is the compiler’s principal output.
- The way from the initial equations to the final VHDL is as simple as possible.
- It supports many kinds of design constraints.
- The tool supports low power implementation features like gated clocks.
- The quantification can be made automatically just by specifying the maximum allowed SNR (= quantification noise) on the outputs signals.
- The compiler supports multi-modes

description (an application block may work in different modes)
- It has a user friendly and scriptable user interface. Graphical user interfaces, utilities, applications and interfaces to other tools can be realized with scripts.
- User interactions are possible on each design compilation stage.
- The compiler assists test and integration of the produced DSP architecture by generation of VHDL testbenchs, simulation and synthesis scripts, Matlab and C models, etc.
- It is user extendable with plug-ins.

3. The realization of the DSP compiler

The design flow

The high-level synthesis can be divided into a series of compilation processes, which are executed in a well-defined order. This concept of long standing doesn't contain iteration loops on a global level and is therefore simple and fast [1]. It is also a good approach for the new DSP compiler. Figure 1 shows the chosen design flow. It follows a short description of each compilation step together with some indications about the compiler's realization.

- The *Equation Parser* reads the equation file that contains the filter description and design constraints. Listing 1 shows such a description together with all required design constraints. The application has two different working modes. The first part of the file is the functional description of the filter using equations that have a similar syntax to Matlab or DFL. Some nice features like a delay operator (known from DFL) or the use of "/2" (divide by 2) that is automatically transformed in a shift operation make the equation syntax comfortable. The second part defines all working modes. Each mode has a main function a periodicity definition and optionally some operation and mode constraints.
- The *Merger* maps the arithmetic operations to available resource units that are defined by a resource description file. Multi-parameter operations are splitted into basic operations, which have a fixed count of parameters.
  Listing 2 shows the resource description syntax using the multiplier unit as example. Each unit has a name, a set of possible operations, number of inputs, cost (size) values, timing specifications and other attributes. A resource unit is either logical or a functional resource unit.
  A logical resource unit has links to other resource units, which are used instead the original one if some conditions are satisfied (ex: *Mul*). A functional unit has a functional description for each output format. This is mainly a VHDL description, but also C descriptions for simulation and quantification.
- It follows optional *initial operation quantification*. This is used to assign to each operation a resource cost value, which depends on the operation word length and which is used by the following compilation

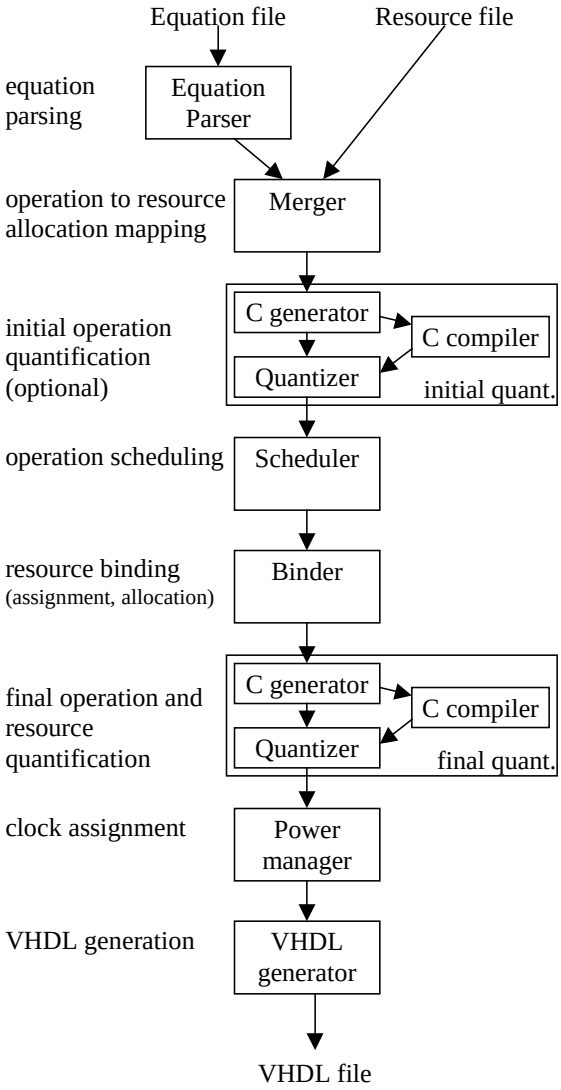

Figure 1: The architecture compilation flow

processes (*Scheduler* and *Binder*) to group similar operations in a better way. When this operation isn't performed, the following processes use generic resource cost values.
The quantification happens in 3 phases: First, floating point and fixed point C descriptions of the application are generated and compiled in a library, which is dynamically linked to the DSP compiler. Then, the word length of each operation is defined by simulation. Because the operations aren't yet allocated to resource units, each operation is considered as resource-independent.

- The *Scheduler* time schedules each operation in a way that the total resource cost is minimized. The execution cycle of each operation is exactly defined after this operation. The optimal solution is found in an iterative loop, using *tabu search* as optimization algorithm [4]. The optimization problematic has already been extensively discussed some years ago and is not the focus of this work [1], [3]. As it has been proved, *tabu search* guaranties a fast convergence in the case of cycle-constrained time scheduling problems [5].
- The *Binder* assigns a resource unit to each operation in a way that the total numbers of interconnections between the resource units are minimized. This happens also in an iterative optimization loop using *tabu search* as optimization algorithm.
- The *final quantification* quantifies the operations again. Now, it takes in account the resource dependency between the operations.
The user has just to define the input and output data formats as well as the expected signal processing quality given by a SNR value. Using these definitions as constraints, the *quantizer* search then by simulation for each algorithm operation the optimal data format. This happens in two steps. First, it defines the integer part, then the fractional part of the operations.
- The *power manager* assigns a clock signal on each clocked resource unit in a way that the total activity in the design is minimized. The total clock number is user definable and may be 1 if a synchronous architecture must be generated.

```
function [out1,out2]=adapt_3(in1,in2,const)
  s1=in1-in2
  s2=s1*const
  out1=s2-in2
  out2=s2-in1
function [out1,out2]=adapt_4(in1,in2,const)
  s1=in2-in1
  s2=s1*const
  out2=s2-in2
  out1=out2-s1
function Y=wdf(X)
  [N12,N11]=adapt_3(X@1, N11@2, 0.3…)
  [N15,N14]=adapt_4(N12, N14@2, 0.1…)
  [N5,N4]=adapt_3(X, N4@2, 0.0…)
  [N8,N7]=adapt_4(N5, N7@2, 0.3…)
  Y=(N15+N8)/2
function Y=add(X)
  Y=(X+X@1)/2
mode filter
  ModeFunction wdf
  OpInfo {X Y}.NbrBit=[16,15]
  OpInfo X.Cycle=-1
  OpInfo Y.Cycle=9
  OpInfo Y.SNR=83
  ModeInfo Period=5
mode add
  ModeFunction add
  OpInfo {X Y}.NbrBit=[16,15]
  OpInfo X.Cycle=-1
  OpInfo Y.Cycle=0
  OpInfo Y.SNR=83
  ModeInfo Period=1
GenInfo NbrClk=4
```

function declaration *function name* *function return values* *input parameters* *delay operator* *1st filter's working mode* *mode's main function* *bit format constraint* *cycle constraints* *output noise constraint* *mode periodicity* *2nd filter's working mode* *max. number of clocks*

Listing 1: The equation file for a filter block with two working modes.

```
#Resource
   Name MulReal
   Operation Mul
   NbrInput 2
   Cost "4*{nBit}+10*{nBitSrc0}*{nBitSrc1}"
   Cost 3200
   Delay 1
   Period 1
   Function C_DOUBLE "{Out}={In0}*{In1};"
   Function C_LONG "{Out}=MulShR({In0},{…
   Function VHDL "{Out}<=Sat(Trim(SIGNED(…
   Function Matlab "{Out}={In0}*{In1};"

#Resource
   Name ShftL
   Operation <<
   NbrInput 2 … (it follows all attributes)

#Resource
   Name Mul
   Operation *
   NbrInput 2
   IF IsPower2Cst(In1) USE << In0 Log2(In1)
   ELSIF IsPower2Cst(In0) USE << In1 Log2(…
   ELSIF always USE Mul In0 In1
```

Listing 2: The resource description of the multiplication

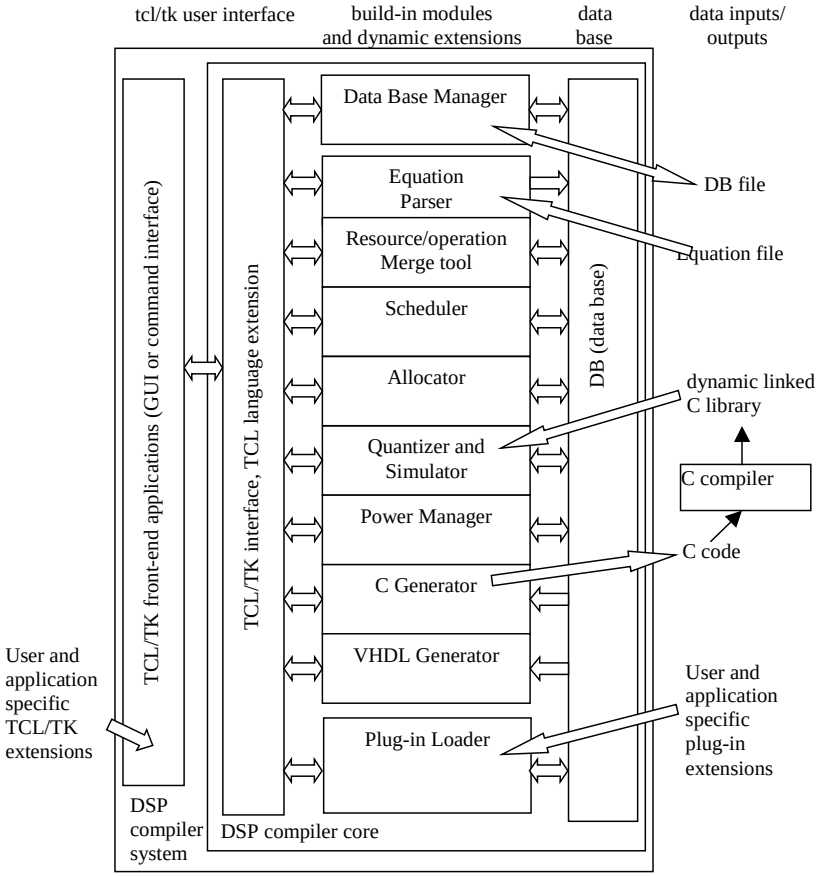

Figure 2: The DSP compiler's modular concept

- The *VHDL generators* produces finally an RTL-level VHDL description of the DSP architecture together with a testbench.

The DSP compiler

The DSP compiler system has 2 parts. The core of the compiler is C-programmed and contains the database, design flow related modules, a plug-in loader and a TCL/TK interface. TCL is a simple scripting language and can easily be incorporated into applications [6]. It is used to create the compiler's graphical user interface (GUI) and to facility user extensions. These TCL extensions and the GUI build the second part, or the front-end of the system.

The database saves all design data in an access-convenient form. The database manager contains file access functions for storing and loading of the database, extends the TCL language by compiler specific features and offers several functions to the other modules. The plug-in loader allows linking of user and application specific extensions into the compiler system. All design flow processes are realized with so-called modules.

The DSP compiler may be started in a command, in a graphical or in a mixed command/graphical mode. After start, it reads first an initialization script that contains user specific configurations and loads available plug-ins. The work with the compiler is simplified using the graphical front-end *DspC_GUI*, (Dsp Compiler's Graphical User Interface), which is entirely realized in TCL/TK. *DspC_GUI* helps executing correctly all design flow processes, gives an overview about the database state, manages the dynamic modules, etc.

The TCL/TK interface gives a free access to the internal database of the DSP compiler. This can be used to write useful applications, for example to visualize the final resource allocation.

4. Example and results

The DSP compiler is not only able to generate filter applications, but also complicated datapath units. An excellent application example is given by the datapath of the FFT processor, presented in [7]. The datapath is not able to perform just basic arithmetic operations, but also complex operations. The required operation set contains reel multiplication, reel MAC, radix-2 and radix-4 butterflies and radix-4 butterfly without complex coefficient multiplication. Consider that this example involves 205 operation, which have to be scheduled in an optimal way on at least 6 different arithmetic units, one sees that an implementation by hand requires quite a lot of time.

The DSP compiler resolves this problem in only some minutes (1 minute using Linux 2.2.14 on Pentium III @ 450 MHz, 5 minutes using Solaris2 on SUN Ultra 2 @ 180 MHz). As shown in Table 1, the solution produced by the DSP compiler is at least equal to handwritten

VHDL solutions. The interpretation of the table's comparison must be done carefully, because there are some small difference between the three solutions:

- The DSP compiler uses multiplicators, the handwritten solutions MAC units.
- The handwritten solutions may invert the coefficients in the MAC units. This feature hasn't been implemented with the compiler's solution.
- The DSP compiler uses arithmetic units from Synopsys' DesignWare™, the other solutions uses for each MAC a Booth coded Wallance tree, followed by a Brent&Kung adder.
- VHDL solution II includes output registers (2 times 16 bits), which aren't present in the other solutions.

		DSP compiler	VHDL solution I	VHDL solution II
design specification		1 day	1 day	1 day
implementation		5 minutes	5 days	5 days
architecture	# gated clocks	4	13	11
	# multiplicator	2	-	-
	# MAC	-	2	2
	# adder/substr	4	4	4
	# register[1]	11+12	21+14	27+14
	# MUX[2]	96+(11+12)	49+(21+14)	33+(27+14)
gate count	register	2294	3054	2712
	comb. logic	7786	8248	7806
	total	10081	11303	10519

Table 1: Comparison between an implementation using the DSP compiler and two hand-written VHDL solutions (reference technology: Alcatel-Mietec 0.35 μm CMOS).

5. Conclusion

The presented DSP architecture compiler allows very quick, efficient and robust implementations of filter and datapath structures. Once the filter's structure and the design constraints are fixed, it takes only some minutes to get a VHDL description of an application specific DSP architecture. A VHDL testbench as well as simulation inputs and references are also generated for verification purposes.

The use of this DSP compiler minimizes the risk of errors introduced by the user during the design flow because it checks the user interaction.

Many kinds of design constraints, which can be defined with an easy and convenient formulation, guaranties that the final filter or datapath block can easily be integrated in the destination environment.

The operation quantification can be automatically made. For that, the user has just to specify the maximum allowed SNR on the filter's output signals.

The DSP compiler generates either full synchronous filter blocks or solutions with individual or shared gated clocks for low-power applications.

If necessary, the user can control the design flow and access the database in each situation. That increases the flexibility of the tool.

The TCL/TK interface and the plug-in loader allow writing user applications like graphical user interfaces, database visualization application and interface applications to VHDL simulators and mathematical programs.

All these features make the DSP compiler system a powerful choice for efficient developments of high-quality DSP architectures for filter and datapath applications.

[1] The number of 16/18 bit registers: intermediate registers/ input registers of arithmetic units.

[2] The number of 16/18 bit multiplexers: The value in the brackets defines the additional registers if the design has to be synchronous (without gated clocks).

References:

[1] M. C. Mc Farland, A. C. Parker, R. Camposano, "The high-level synthesis of digital systems", Proceedings of the IEEE, vol 78. pp. 301-318, February 1990.

[2] S. Amellal, B. Kaminska, "Scheduling Algorithm in Data Path Synthesis Using the Tabu Search Technique", Proc. of EDAC_EUROASIC93, Paris, France, February 22-25 1993.IEEE Computer Society Press, 1993.

[3] S. Amellal, B. Kaminska, "Functional Synthesis of Digital Systems with TASS", IEEE transactions on computer-aided design of integrated circuits and systems, vol 13., 5 May 1994.

[4] F. Glover, "Tabu Search, Part I+II", ORSA J. Computing, pp. 4-32/190-206, 1989.

[5] A. Heubi, P. Balsiger, F. Pellandini, "An Automated Design Methodology for the Mapping of DSP Algorithms into Low Power VLSI Architectures", Proc. of ISIC-97, Singapore, September 10-12, 1997.

[6] J. K. Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesley Publishing Company,Inc, ISBN 0-201-63337-X

[7] A. Drollinger, C. Wälchli, D. Sun, L. Grisoni, C. Calame, A. Heubi, P. Balsiger, F. Pellandini, R. Brennan, T. Schneider, "An Ultra Low Power WOLA Filterbank Implementation in Deep Submicron Technology", Proc. of COST 254, , Neuchâtel, CH, May, 5-7, 1999.