The International Workshop
on Discrete-Event System Design, DESDes'01,
June 27÷29, 2001; Przytok near Zielona Gora, Poland

CHDL - AN APPROACH FOR HARDWARE DESIGN AT THE SYSTEM LEVEL

Miroslaw FORCZEK

Aldec-ADT, Compilers Division,
ul. Lutycka 6, 44-100 Gliwice, POLAND, *mirekf@aldec.katowice.pl*

Abstract. *Currently, designers turn to C/C++ instead of using HDL languages at the initial stage of their projects. Manual translation from C/C++ into an HDL is extremely time-intensive. Even with latest approaches such as C++ library of HDL classes, a designer still has invested a lot of time on re-writing the project code at the RTL level. Aldec's CHDL approach, a C subset, addresses precisely the problem of C to HDL conversion automation. The possibilities of an algorithm parallelism reconstruction from its software version on example of the DCT routine were shown.*

Key Words. *System-Level Hardware Design, Behavioral Synthesis, HDL Languages, C/C++ Language*

1. INTRODUCTION

As devices become more complex, their design processes take more time and become more expensive than before. One of the most important improvements, which was introduced over the past years, is an RTL synthesis tools, which automates the design transformation from RTL into gate-level process [10]. Since its introduction, most of a designer's efforts stop at the RTL stage of the design specification. Automated tools perform the rest of work (fig. 1). The synthesis tools have been improved since their first appearance, and it now makes no sense in terms of an economical aspect to try to make a better design manually than a synthesized one. A few decades ago the algorithm's distinction into either software or hardware was introduced. The possibilities of today's highly integrated chips cause, such a distinction is not so obvious now [8,11]. The algorithms also become more complex and change frequently. Making updates in the present hardware implementations is a costly and time-intensive process. The algorithms are now prototyped and verified in a software implementation version. Often they exist in software form for a longer time, acquiring stability before hardware implementation is required. This is a way hardware designers turn to classical programming languages such as C or C++ at the first stage of the project instead of using HDL languages. This causes new problems. One of them is making the transition from C/C++ implementation into HDL implementation. These two implementations are totally different:

- C/C++ implementation is a sequential process (in most cases) while HDL implementation is a parallel, multi-process design,
- - C/C++ implementation runs without any clock signals while HDL implementation must take into consideration system clock signal and must address the signals timing issues.

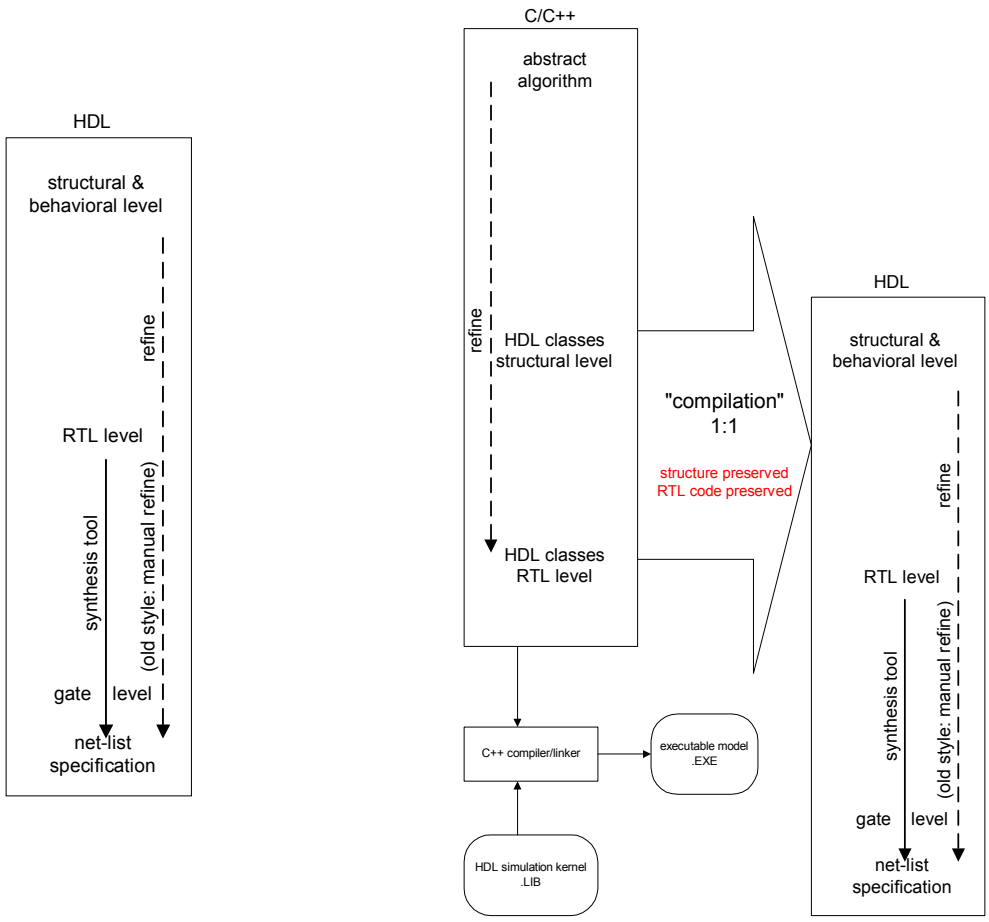

Fig. 1. Hardware design paths: with use of an HDL language (left) and with use of HDL classes in C++ (right)

The designer has to wait until the entire project is converted into HDL before he or she can validate the design again. This causes the conversion bugs to accumulate, making running the entire project much more difficult than in the case of having regular regression tests. A different approach has been proposed that address these problems. Currently, the most popular design solution is C++ library of HDL classes [2,3,9]. HDL classes enables a normal C/C++ environment with HDL constructs like modules, processes and signals. The designer performs a full conversion process in the same environment. As a result, the designer is able to make regular regression test at any stage of the conversion. Unfortunately, HDL classes library is not a solution that fully automates the hardware design path. The designer still has spent a lot of time on re-writing (refining) the project code from software form into HDL form (fig. 1).

Aldec's CHDL approach addresses exactly the problem of C to HDL conversion automation. Instead of enabling C/C++ environment with HDL features and pushing the user to go throught the refine steps until reaching RTL model, CHDL enables users to synthesize the HDL code directly from the C algorithm in its natural form.

2. CHDL NOTATION

CHDL is a subset of the C language [6]. The C constructs that made it impossible to perform full static data and analyze control flow were removed. The table 1 summarizes the C constructs included in or rejected from CHDL notation.

Table 1. Summary of C constructs included and rejected in CHDL notation

Constructs class	Included C constructs	Rejected C constructs	Comments
Built-in types	char, int, float, double	void, pointer types	
Complex types	structure, union		translated by fields expansion into HDL
Operators	most of C operators	* & (indirection and address of)	
Statements	if, switch, while, do while, for, function call		non-recursive functions are allowed only
Local scopes and visibility rules		cross-references over function boundary	

The only limitation for a C programmer when using CHDL is the reduced spectrum of available language features. There is no requirement to re-write the C algorithm in terms of modules, processes or registers like it is in case of HDL classes. The designer only needs to eliminate the forbidden constructs, but the algorithm structure remains untouched at the same level of abstraction. As a result, CHDL will offer true system level design capabilities (un-timed design).

While manual C code refine into HDL classes the designer explicitly specifies the algorithm's inherent parallelism (by decomposing the algorithm into processes). Also the hardware architecture and available resources are explicitly denoted. Consequently, the compilation from HDL classes model to HDL is very simple process, preserving all semantics of constructs used in design specification.

All of this additional information (regarding an algorithm's parallelism and its preferred implementation in hardware) is not present in the CHDL description of an algorithm. Instead, it utilizes a CHDL compiler task to take all required decisions while compiling the algorithm into HDL (fig. 2).

3. BEHAVIORAL SYNTHESIS FROM CHDL DESCRIPTION

Having a C algorithm written with the use of behavioral constructs from a small subset (CHDL notation) is very simple to make its transformation into HDL version. Each CHDL construct has a directly corresponding construct in HDL language. Of course, such a naive conversion would result in a single-process design. After synthesizing it into gate level, the designer will create a working design, but with very poor throughput to area size rate. The tool for real use must perform CHDL to HDL conversion in an intelligent way, which means that the behavioral synthesis approach must be implemented. The tool will take on the majority of decisions on its own. The user should only specify guidelines for preferred implementation architecture. The synthesized implementation consists of a two kinds of logical blocks, which are:

- the processing units,
- the control circuits.

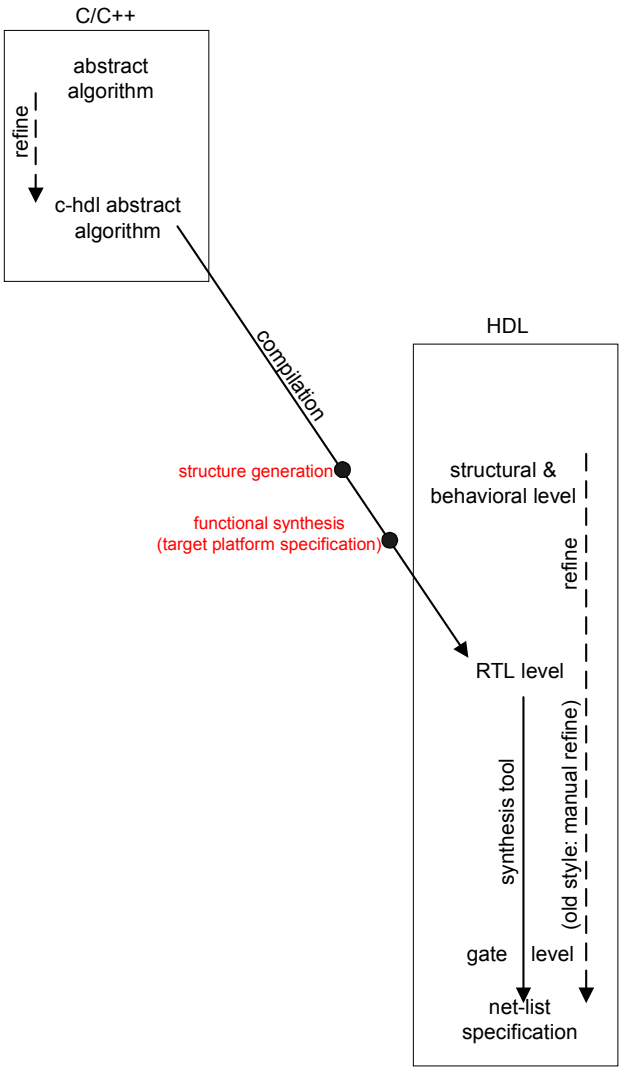

Fig. 2. The hardware design path with use of CHDL notation

The size of the control overhead depends on the proportion between the number of the algorithm's elementary sub-tasks units and the number of allocated resources for them. In the event there are less resources then sub-tasks to be processed, the control circuits must provide sharing the processing units in time, which translates into additional cycles for input data fetching, retrieving results from outputs and storage of intermediate results. As in any case, there is a trade-off between implementation size and its efficiency (power dissipation, throughput) [12]. In a normal design path, the designer has to make such decisions early in the design cycle, and the initial decision will make a large impact on the final result. With an automated path, designers can explore more than one architecture with relatively low costs or risks.

As an example, refer to a 2-D Discrete Consine Transform algorithm [4,5,7]. Assuming that the software implementation is already in place, the formula would appear as illustrated below [1]:

```
void fct2d(double f[], int nrows, int ncols) {
  int u,v;
  // ...
  for (u=0; u<=nrows-1; u++) {
    for (v=0; v<=ncols-1; v++) {
       g[v] = f[u*ncols+v];
```

```
    }
    fct(g,ncols);
  }
  for (v=0; v<=ncols-1; v++) {
    for (u=0; u<=nrows-1; u++) {
       g[u] = f[u*ncols+v];
    }
    fct(g,nrows);
    for (u=0; u<=nrows-1; u++) {
      f[u*ncols+v] = g[u]*two_over_sqrtncolsnrows;
    }
  }
}
```

This algorithm works as follows:

- 1-D DCT (**fct()** function) is performed for each row of the matrix **f**,
- the 1-D DCT is performed for each column from the result matrix after rows processing,
- the whole result matrix is scaled with a constant coefficient.

From the data and control flow analysis, it is possible to find and extract elementary sub-tasks that are independent of each other and can be processed in parallel. In this example, there are few groups of elementary tasks (fig. 3):

- **fct()** on each row of **f**,
- **fct()** on each column of result from previous processing,
- scaling each element of result from previous processing.

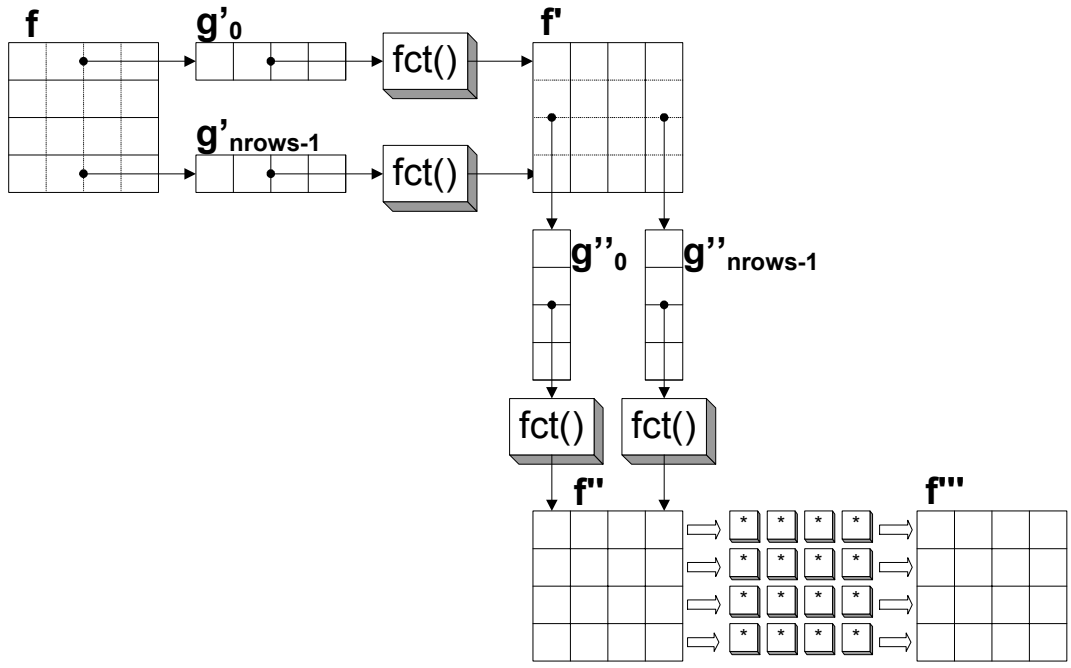

Fig. 3. The data and control flow diagram extracted from source code analysis

The designer now needs to decide what the synthesis mode to use for hardware implementation of this algorithm will be. The **blank array** mode can be used if the **f** matrix sizes are fixed. In this case, the fastest implementation as well as the larger one will allocate

its own processing unit. The **fixed resources** mode may be preferred, especially when the **f** matrix size varies in run-time. In this case, one will get an implementation that contains the limited number of processing units and the control block. It could be that the resulting size of the implementation will still not satisfy the requirements for the first time. If this is the case, the user has to try other tradeoffs by specifying various synthesis constraints.

4. CONCLUSION

The presented example clearly shows how the behavioral synthesis can be performed from the system level C algorithm. There is no need to manually re-write the algorithm in HDL manner to precise parallelism of the algorithm. The compromise made in the aforementioned approach is to reduce the flexibility of a source language (C in this case) in favor of a predictable construct for algorithm notation that allows its static analysis. From the very coarse version of the synthesis tool, new compilation techniques are applied incrementally for improvement results. There are still several problems to be researched and solved as a practical implementation in this synthesis tool. The most important issues are as follows:

- processing units synthesis or re-use of library units,
- automatic processing unit functionality selection based on the elementary sub-tasks code analysis.

REFERENCES

[1] *A Fast Discrete Cosine Transform*, Signal and Image Processing Group, The University of Bath, 1998

[2] "An Introduction to System-Level Modeling in SystemC 2.0", January 2001, *http://www.systemc.org*

[3] "Functional Specification for SystemC 2.0", January 2001, Synopsys Inc., CoWare Inc., Frontier Design Inc., 2000, *http://www.systemc.org*

[4] *Intel Image Processing Library. Reference Manual*, Intel Corporation, USA, 1999

[5] *Intel Signal Processing Library. Reference Manual*, Intel Corporation, USA, 1999

[6] *International Standard ISO/IEC 9899: 1999(E), Programming Languages – C,* ISO/IEC, 1999

[7] *Image Processing Toolbox. Users Guide. Version 2*, The MathWorks Inc., Natick, MA, 1999

[8] "QuickSilver: Technology Backgrounder", QuickSilver Technology, 2000, *http://www.quicksilvertech.com*

[9] "SystemC. Version 1.1 User's Guide", Synopsys Inc., CoWare Inc., Frontier Design Inc., 2000, *http://www.systemc.org*

[10] M.D.Ciletti, *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL*, Prentice Hall, New Jersey, 1999

[11] P.Master, K.Lane "Powering up 3G Handsets for MPEG-4 Video", Communication Systems Design, January 2001, *http://www.cdsmag.com/main/2001/01/0101feat2.htm*

[12] J.Phil, *Tradeoffs Between Parallel and Serial Architectures in High Performance Digital Signal Processing*, Norwegian University of Science and Technology, Faculty of Electrical and Computer Engineering, Trondheim, Norway, 1997