A Tool for Mapping a C Description into a Hardware Configuration using VHDL at RTL

Anton Velinov Chichkov
Alcatel Microelectronics
Westerring 15, B-9700 Oudenaarde Belgium
Tel: +32 55 33 27 72 Fax: +32 55 33 22 64
E-mail: anton.chichkov@mie.alcatel.be

Carlos Beltrán Almeida
INESC/IST
R. Alves Redol 9, 1000 LISBOA-Portugal
Tel.: +351 1 3100240 Fax: +351 1 3145843
E-mail: cfb@inesc.pt

Summary
This paper introduces a software package developed to assist the hardware/software co-design of digital systems. The main purpose of this work is the transformation of a high-level specification using C programming language into RTL VHDL. In the last years several translators from C and C++ to VHDL have been developed but all of them are using a subset or a superset of C in order to include timing structures into the language. The present method performs the translation of a program written in standard C language into two files, a VHDL file at Register Transfer Level and a C program that, together, will perform the same function as the input program but with a significant performance boost. A special attention is devoted to the system synthesis module where the VHDL file at Register Transfer Level is generated. Two FPGA implementation examples are presented, leading to sequential and combinational circuits, derived from the analysis of the data and control structures of C functions used to validate the hardware/software co-design methodology.

1 Introduction

Computationally intensive applications typically spend most of their running time within small parts of the executable code whereas the rest of the program is executed in a short time period. In many of these applications substantial improvements can be achieved by adapting the configuration of the processor to these frequently accessed parts of code. These computing systems, called *Custom Computing Machines*, are composed by a processing platform expanded with FPGA based segments which can be reconfigured in order to customise the architecture to some specific tasks. The compilation of a program for such a platform becomes an hardware/software co-design problem. The goal of the work, where this paper describes only one last step of the design flow, was the development of a configuration compiler [1][2]. The input of this software tool is a program, specified through a high level programming language, and the outputs are a hardware description and a software program that, together, will perform the same function as the input program but with increased performance.

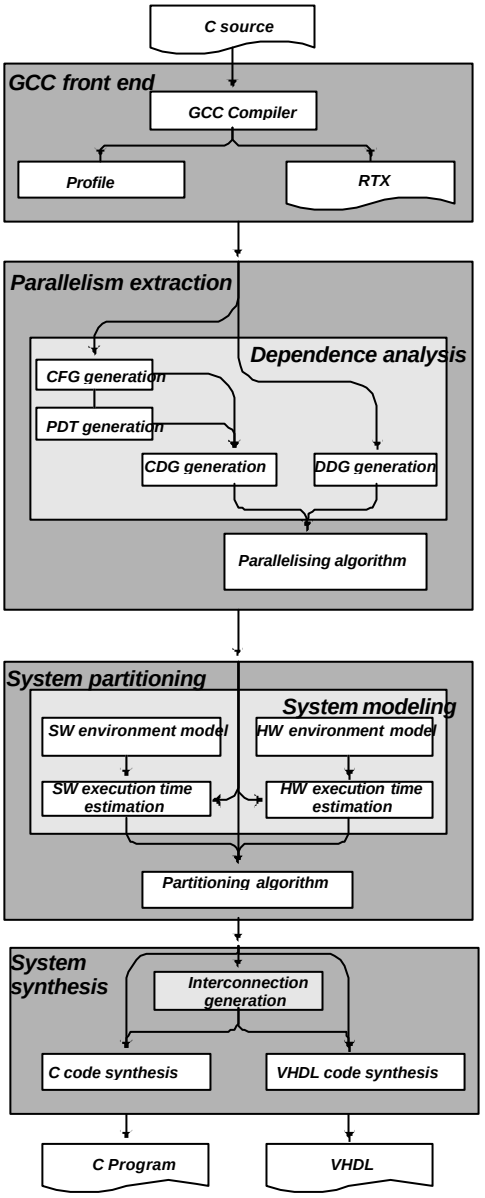


Figure 1 - Configuration compiler architecture.

The complete methodology, shown in figure 1, consists of the following main parts: a front-end GNU C compiler [3], a parallelising compiler [4], a hardware/software partitioning algorithm [5] and an implementation tool. This paper is focused to the generation of one FPGA configuration that implements the functionality of the system.

A brief description of the entire co-design flow is provided for better understanding of the importance of the last phase, the system synthesis, treated in this paper.

Specification of the system

The ANSI C programming language is used as input specification language. During the compilation of the C program corresponding to the initial specification, the intermediate three-address code (RTX) is generated. This code is used during the modelling and the analysis of the system. In order to capture the behaviour of the system the flow graph model is used. In fact, the flow graph model has enough power to describe sequential and parallel structures, corresponding to the classical SW and HW descriptions. All further analysis and transformations are performed over this graph model and not on the initial specification.

Partitioning

Partitioning is accomplished in two steps: extraction of the implicit parallelism and final partitioning.

- *Implicit parallelism extraction*

The extraction of the implicit parallelism is performed for two main reasons. First, the partition obtained considering the parallelism results in a system with a small overhead caused by the interface. Second, the resulting parallel model is better suited for the HW implementation. The parallelising compiler explores the acyclic code regions of the source program in order to find useful parallelism [4]

- *Final partitioning*

The final partitioning is achieved using an iterative algorithm. A cost function was developed for evaluation of the resulting partitions. The decision to move a branch to hardware is based on the value of the cost function, which is calculated using estimations of the branch performances corresponding to the hardware and to the software implementations. Estimation methodologies were developed and incorporated in the hardware/software partitioning algorithm [5].

System implementation

Finally, the implementation of the blocks resulting from the partitioning in hardware and software is accomplished. The software implementation requires some additional functions providing the interface between the partitions. This step will be treated in detail in this paper.

2 System implementation

The main purpose of this work is the transformation of a high-level specification using C programming language into RTL VHDL.

In the last years, several translators from C and C++ to VHDL have been developed but all of them are using a subset or a superset of C in order to include timing structures into the language. In the case of using a subset the assignment of resources and the timing definition is left to the designer. In the present work standard C language was used. Some small limitations were imposed concerning to the hardware implementation of the typically software oriented structures of the language, for example, the interface with the user.

The choice of specification language

The choice of the input language was determined by the applications targeted of this work: acceleration of intensive calculation programs using *Custom Computing Machines* (CCM). For CCM the input language should be an existing programming language. C was chosen not only because of its flexibility but also because this language is widely accepted by the scientific community and used for the implementation of intensive calculation algorithms.

The choice of implementation language

The choice of VHDL RTL as implementation language was mainly determined by the availability of mature commercial tools. The possible use of the behavioural VHDL was not considered due to the early stage of development associated to the synthesis tools for this language.

The implementation method

The lack of timing information in the initial specification defines the implementation of one branch of a C program into hardware as a high-level synthesis problem. This transformation includes three main steps: the resource assignment, the translation from intermediate GCC code to VHDL and the timing definition. The resource assignment consists in the definition of the exact places for the storage components (registers). The timing definition is the assignment of an appropriate clock period to the system. The realisation of these two tasks requires the analysis of the control and data dependencies, the analysis of the variables lifetimes and the performance estimation. The analysis of the control and data dependencies provides the information for the data path implementation and the controller structure. The analysis of the variable lifetimes determines the resource assignment, whereas the performance estimation defines the clock cycle for the system.

The front-end of the GCC compiler was used, as part of the developed configuration compiler, in order to:

- ensure the lexical and syntactical correctness of the source
- parse the entire input file when invoked, generating the intermediate three-address code
- take advantage of the compiler optimising algorithms

Some of the optimisations done during the compilation pay attention to *if conditions, jumps, loops, common sub-expression elimination, data flow analysis and scheduling*. The result of these actions is a three-address code: one operation and two operands. The translation software presented in this paper uses this representation as input. Specially developed parsers analyse the input and build the control flow and the data dependence graphs of the function. After the construction of those graphs, the analysis of the variable lifetimes is performed. The control dependence and data control dependence graphs are the input for the implementation tool that translates the graph structure to the RTL VHDL or back to the C program language.

2.1 Variables lifetime and dependence analysis

The variable lifetime and dependence analysis in the entire input program is a prohibitively time consuming operation. The approach adopted in this work consists in the analysis of the program function by function.

The first step of the analysis, after the GCC intermediate code generation, is the identification of the program functions and the division of the initial specification into separated files corresponding to each function.

Control flow analysis

The statements implementing the control flow in C programming language are *if, case, goto, for, while, do, continue* and *break*. In the intermediate code these statements are represented by *jump* commands. There are two types of *jumps*: conditional (*if then else*) and unconditional (*go*). These statements divide a program into basic blocks [6] and establish the control dependencies between them. There are two ways of representing the control dependencies: through the Control Flow Graph (CFG) showing all possible control paths between the basic blocks of the function and by means of the Control Dependence Graph (CDG) describing all the control dependencies between the basic blocks of the function. For the implementation of the system only the CFG is needed. However, in this work the CDG was also considered by the parallelising algorithm [4].

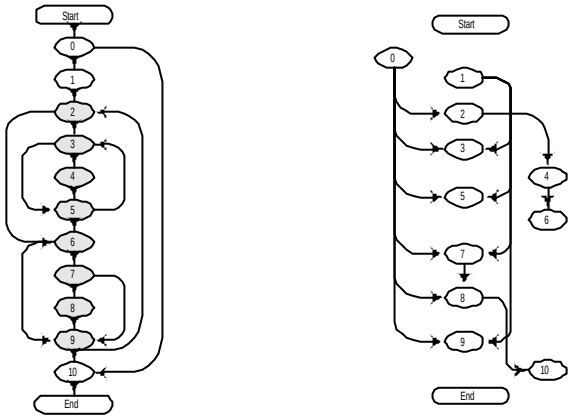
Generation of the CFG

The generation of the CFG is achieved by parsing the input, searching for jump statements and their target labels. An example of CFG is shown in figure 2-a). A generic graph tool, DOTTY [7], is used for interactive edition and manipulation of graphs like those presented in figure 2.

Data dependency analysis

Another information needed for the final implementation of the system concerns the data dependencies between the basic blocks. The data dependence analysis is performed in two steps: analysis of the variables lifetimes and generation of the Data Dependence Graph (DDG). The GCC compiler, used as a front-end of this package, performs a partial data dependence analysis and assigns the registers being alive in the beginning of each basic block. Using the information provided by GCC and CFG, the variables are analysed to define their scope and their sources, i.e., the basic blocks setting the values of the variables. In the second step, the tool generating the DDG performs the bottom-up recursive analysis of the CFG searching for the sources of the data used by the block under analysis. The program outputs are presented in the Figure 2-b).

The parsers were implemented using LEX and YACC and they run on a SUN SPARC4 UNIX environment.



a) b)

Figure 2- a) Control flow graph.

b) Data dependence graph.

2.2 Graph to RTL VHDL translation

The implementation of a graph into RTL VHDL can be done through sequential or combinational circuit blocks, depending on the branch architecture. The difference between the combinational and the sequential implementations consists in the explicit order of execution enforced by the control structure in the sequential implementation. The combinational implementation requires fewer resources and has

better performance so, whenever it is possible, the combinational implementation is used.

Combinational implementation

Non-hierarchical graphs, i.e., graphs corresponding to branches of one program without calling sub routines, without internal variables (registers) or deprived of internal loops, can be implemented using only combinational blocks. Thus, a combinational circuit implementation corresponds to graphs using only input/output, data or control signals stored in registers. Therefore, the graph body is implemented as a combinational data path and will produce the correct output after a finite period of time. Multiplexers using the model shown in figure 3 accomplish the implementation of the conditional vertices. The final translation is straightforward, once the multiplexers are included in the data path, mapping the three-address expressions of the intermediate code into correspondent VHDL expression.

However, it is necessary to emphasise that the final circuit implementation will always be a sequential one since the connection of these combinational branches with the higher levels of the program hierarchy is done using registers.

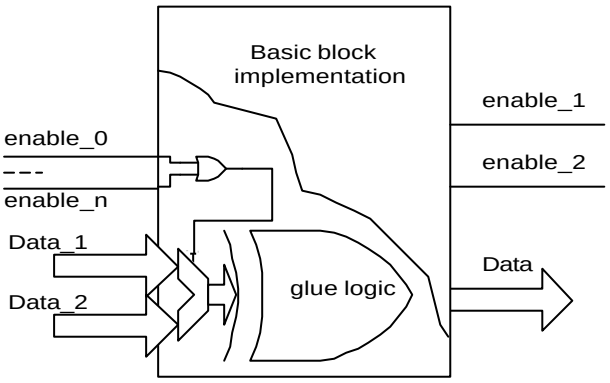


Figure 3 - Combinational basic block.

Sequential implementation

There are two possible architectures for a sequential circuit implementation: using a separated data path and control module or using a mixed implementation. In this work, the mixed implementation was selected. The operations are ordered by the scheduling mechanism of the GCC. The corresponding implementation is achieved through the inclusion of control structures in each vertex of the graph. These structures allow the execution of the vertex in a pre-defined time slot.

Control structures definition and implementation

The implementation of the control structure is based on the division of the functionality of the vertex into control and data statements. The vertex architecture may present all possible combinations of vertexes: those that implement only control operations, only data operations or both. It is impossible to implement separated architectures for the control and data only by inspection of the basic blocks (vertex) because the data and control tracks are using the same statements. In the used graph model different edge types represent the control flow and the data flow. The CFG and DDG clearly distinguish the data edges from control edges. Using this definition the implementation of the edges and vertices is carried on.

Edge implementation

The data edges are implemented through registers and the control edges are implemented like a semaphore system controlling the input load of the data registers. All fork and join vertexes in the graph are disjoined [8]. When more then one control signal is directed to the same vertex it should be connected to the load signal through an OR gate. The initially used C compiler assumes the existence of only one processor and, consequently, defines the disjoint characteristic of the graph. The conjoined vertexes are only possible in parallel architectures.

Vertices implementation

a) Vertex body implementation

The vertex body is implemented by mapping the three-address expressions of the intermediate code into correspondent VHDL expression. The same parser used for the combinational implementation also implements the body of the basic blocks (vertices).

b) Vertex control structure implementation

In addition to the enable signals produced by the previous vertexes, some extra control signals were used in the sequential basic block presented in figure 4: *clock*, *reset*, *done_out*, *done_in*. Usually the signal *done_out* is connected to *done_in*, disabling the block after finishing its execution. In this case, only the *reset* signal can enable the block again.

Block execution sequence

The hardware implementation of the vertexes has three states: *ready*, *busy* and *done*. One vertex goes from the *ready* to the *busy* state when one of its enable inputs is active. It will stay *busy* until the last statement of the body of the vertex is executed. After that, the vertex goes to the state *done*, where it will stay until the *reset* signal is obtained. In order to allow a pipeline execution, the *reset* signal does not affect the data registers.

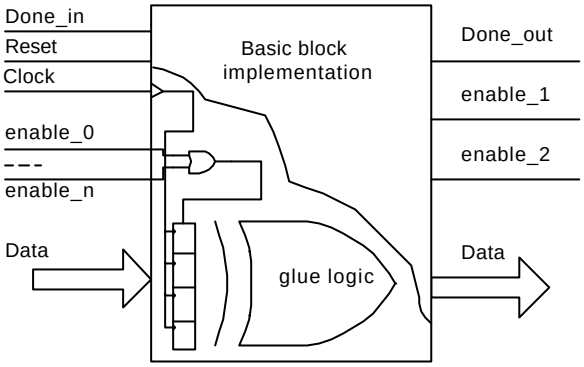


Figure 4 - Sequential basic block.

Branch implementation

The branch is implemented by putting the blocks, sequential or combinational, in the correct order and connecting their data and control structures.

Branch execution sequence

The branch implementation shown in the example of figure 5, have two states: *ready* and *busy*. In this case, the architecture goes from ready to *busy* when the *enable* signal is received. The system will stay *busy* until the execution of the last vertex or until *reset* signal is available.

3 Examples and results

Several examples were used to illustrate different aspects of the proposed methodology. In this paper, two C programs were used to verify the efficiency of the synthesis module.

- The first one called Hierarchical Clustering is a partitioning algorithm developed at INESC that divides a large circuit into sub-circuits with a given complexity.
- The second program, called Tmndecode, is a decoder and player for H.263 bit-streams developed by Telenor R&D. This last program displays images of a decoded sequence of 8, 16, and 24(32) bits colour palette for both, X11 and Microsoft Win32 windows systems. The program Tmndecode contains sixteen C files, which were all analysed in order to select the most appropriated functions to accelerate the execution of the program via its hardware implementation. The branches of the function *Reconstruct* were the candidates for a FPGA implementation.

Tables 1 and 2 list some of the implemented functions in the XC4005 component. The hardware execution time is obtained from XILINX XDelay/timing analyser. The delay time represents the worst case clock to output pad delay for the sequential implementation (table 1) and the worst case input pad to output pad delay for the combinational implementation (table 2). The CLB count is obtained from XILINX XACT tool. The software time refers to the execution on a SUN SPARC 4.

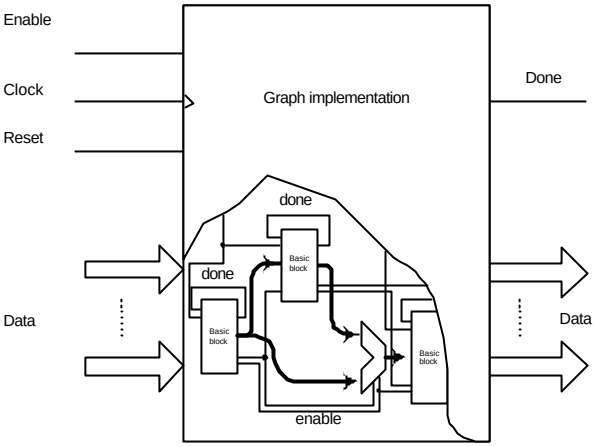


Figure 5 - Sequential implementation of a branch

Table 1 - Sequential implementation

Tmndecode			
Function	#CLBs	HW execution time	SW execution time
Reconstruct	48	294 ns 14 HW clock cycles	2360 ns 59 SW clock cycles
Hierarchical clustering			
Function	#CLBs	HW execution time	SW execution time
V_max1	78	140 ns 4 HW clock cycles	920 ns 28 SW clock cycles

Table 2- Combinational implementation

Tmndecode			
Function	#CLBs	HW execution time	SW execution time
Reconstruct (part)	26	45.6 ns	2360 ns
Hierarchical clustering			
Function	#CLBs	HW execution time	SW execution time
V_max1 (part)	11	35.5 ns	920 ns

4 Conclusions

A computer aided hardware/software co-design tool that transforms C code deprived of timing information, into VHDL at Register Transfer Level was introduced. In the developed methodology that transformation is achieved through the assignment of resources, the translation of the three-address code generated by GCC to VHDL and the definition of the system clock. The examples demonstrated an acceleration of hundreds to thousands times of the hardware implemented function compared with the original software function. However, due to limitations on the size of the re-configurable platforms to accommodate a complete program, in general, the overall acceleration will not exceed one order of magnitude.

The continuos increase of gate count in FPGA devices, verified in the last years, will certainly remove, in a near future, all the actual limitations found to implement a complete program in an hardware platform.

5 References

[1] P. Athanas and H. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis: Architecture and Compiler”, Computer, vol 28, no. 3, pp. 11-18, March 1993.

[2] P. Athanas, “An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration“, Technical Report LEMS-101 Division of Engineering, Brown University, February 1992.

[3] Free Software Foundation, ”Using and Porting GNU CC”, Cambridge, MA, 1992.

[4] Anton Chichkov, C. Beltrán Almeida, "Identification and Optimisation of Parallelism in Hardware/Software Partitioning", International Workshop on Logic and Architecture Synthesis, Grenoble, France, December 1996.

[5] Anton Chichkov, Carlos Beltrán Almeida, “An Hardware/Software Partitioning Algorithm for Custom Computing Machines”, Proceedings of FPL97, London, September 1997.

[6] Alfred V. Aho, Ravi Sethi and Jefferey D. Ullman, "Compilers: Principles, Techniques and Tools", Addison Wesley, 1986.

[7] Elftherios Koutfios, Stephen C. North, "Editing Graphs with DOTTY", User’s Manual, version 94b, 1994.

[8] Rajesh Kumar Gupta, "Co-Synthesis of Hardware and Software for Digital Embedded Systems", Ph.D. dissertation, Stanford University, December 10, 1993.