16

High-Level Synthesis for Embedded Systems

Michael Dossis
Technological Educational Institute of Western Macedonia,
Dept. of Informatics & Computer Technology
Greece

1. Introduction

Embedded systems comprise small-size computing platforms that are self-sufficient. This means that they contain all the software and hardware components which are "embedded" inside the system so that complete applications can be realised and executed without the aid of other means or external resources. Usually, embedded systems are found in portable computing platforms such as PDAs, mobile and smart phones as well as GPS receivers. Nevertheless, larger systems such as microwave ovens and vehicle electronics, contain embedded systems. An embedded platform can be thought of as a configuration that contains one or more general microprocessor or microprocessor core, along with a number of customized, special function co-processors or accelerators on the same electronic board or integrated inside the same System-on-Chip (Soc). Often in our days, such embedded systems are implemented using advanced Field-Programmable Gate Arrays (FPGAs) or other types of Programmable Logic Devices (PLDs). FPGAs have improved a great deal in terms of integrated area, circuit performance and low power features. FPGA implementations can be easily and rapidly prototyped, and the system can be easily reconfigured when design updates or bug fixes are present and needed.

During the last 3-4 decades, the advances on chip integration capability have increased the complexity of embedded and in general custom VLSI systems to such a level that sometimes their spec-to-product development time exceeds even their product lifetime in the market. Because of this, and in combination with the high design cost and development effort required for the delivery of such products, they often even miss their market window. This problem generates competitive disadvantages for the relevant industries that design and develop these complex computing products. The current practice in the used design and engineering flows, for the development of such systems and applications, includes to a large extent approaches which are semi-manual, add-hoc, incompatible from one level of the design flow to the next, and with a lot of design iterations caused by the discovery of functional and timing bugs, as well as specification to implementation mismatches late in the development flow. All of these issues have motivated industry and academia to invest in suitable methodologies and tools to achieve higher automation in the design of contemporary systems. Nowadays, a higher level of code abstraction is pursued as input to automated E-CAD tools. Furthermore, methodologies and tools such as High-level Synthesis (HLS) and Electronic System Level (ESL) design entry employ established techniques, which are borrowed from the computer language program compilers and

mature E-CAD tools and new algorithms such as advanced scheduling, loop unrolling and code motion heuristics.

The conventional approach in designing complex digital systems is the use of Register-Transfer Level (RTL) coding in hardware description languages such as VHDL and Verilog. However, for designs that exceed an area of a hundred thousand logic gates, the use of RTL models for specification and design can result into years of design flow loops and verification simulations. Combined with the short lifetime of electronic products in the market, this constitutes a great problem for the industry. The programming style of the (hardware/software) specification code has an unavoidable impact on the quality of the synthesized system. This is deteriorated by models with hierarchical blocks, subprogram calls as well as nested control constructs (e.g. if-then-else and while loops). For these models the complexity of the transformations that are required for the synthesis tasks (compilation, algorithmic transformations, scheduling, allocation and binding) increases at an exponential rate, for a linear increase in the design size.

Usually the input code (such as ANSI-C or ADA) to HLS tool, is first transformed into a control/data flow graph (CDFG) by a front-end compilation stage. Then various synthesis transformations are applied on the CDFG to generate the final implementation. The most important HLS tasks of this process are scheduling, allocation and binding. Scheduling makes an as-much-as-possible optimal order of the operations in a number of control steps or states. Optimization at this stage includes making as many operations as possible parallel, so as to achieve shorter execution times of the generated implementation. Allocation and binding assign operations onto functional units, and variables and data structures onto registers, wires or memory positions, which are available from an implementation library.

A number of commercial HLS tools exist nowadays, which often impose their own extensions or restrictions on the programming language code that they accept as input, as well as various shortcuts and heuristics on the HLS tasks that they execute. Such tools are the CatapultC by Mentor Graphics, the Cynthesizer by Forte Design Systems, the Impulse CoDeveloper by Impulse Accelerated Technologies, the Synfony HLS by Synopsys, the C-to-silicon by Cadence, the C to Verilog Compiler by C-to-Verilog, the AutoPilot by AutoESL, the PICO by Synfora, and the CyberWorkBench by NEC System Technologies Ltd. The analysis of these tools is not the purpose of this work, but most of them are suitable for linear, dataflow dominated (e.g. stream-based) applications, such as pipelined DSP and image filtering.

An important aspect of the HLS tools is whether their transformation tasks (e.g. within the scheduler) are based on formal techniques. The latter would guarantee that the produced hardware implementations are correct-by-construction. This means that by definition of the formal process, the functionality of the implementation matches the functionality of the behavioral specification model (the source code). In this way, the design will need to be verified only at the behavioral level, without spending hours or days (or even weeks for complex designs) of simulations of the generated register-transfer level (RTL), or even worse of the netlists generated by a subsequent RTL synthesis of the implementations. Behavioral verification (at the source code level) is orders of magnitude faster than RTL or even more than gate-netlist simulations. Releasing an embedded product with bugs can be very expensive, when considering the cost of field upgrades, recalls and repairs. Something that

is less measurable, but very important as well, is the damage done to the industry's reputation and the consequent loss of customer trust. However, many embedded products are indeed released without all the testing that is necessary and/or desirable. Therefore, the quality of the specification code as well as formal techniques employed during transformations ("compilations") in order to deliver the hardware and software components of the system, are receiving increasing focus in embedded application development.

This chapter reviews previous and existing work of HLS methodologies for embedded systems. It also discusses the usability and benefits using the prototype hardware compilation system which was developed by the author. Section 2 discusses related work. Section 3 presents HLS problems related to the low energy consumption which is particularly interesting for embedded system design. The hardware compilation design flow is explained in section 4. Section 5 explains the formal nature of the prototype compiler's formal logic inference rules. In section 6 the mechanism of the formal high-level synthesis transformations of the back-end compiler is presented. Section 7 outlines the structure and logic of the PARCS optimizing scheduler which is part of the back-end compiler rules. Section 8 explains the available options for target micro-architecture generation and the communication of the accelerators with their computing environment. Section 9 outlines the execution environment for the generated hardware accelerators. Sections 10 and 11 discuss experimental results, draw useful conclusions, and propose future work.

2. Background and review of ESL methodologies

2.1 The scheduling task

The scheduling problem covers two major categories: time-constrained scheduling and resource-constrained scheduling. Time-constrained scheduling attempts to achieve the lowest area or number of functional units, when the total number of control steps (states) is given (time constraint). Resource-constrained scheduling attempts to produce the fastest schedule (the fewest control states) when the amount of hardware resources or hardware area are given (resource constraint). Integer linear programming (ILP) solutions have been proposed, but their run time grows exponentially with the increase of design size, which makes them impractical. Heuristic methods have also been proposed to handle large designs and to provide sub-optimal but practical implementations. There are two heuristic scheduling techniques: constructive solutions and iterative refinement. Two constructive methods are the as-soon-as-possible (ASAP) and the as-late-as-possible (ALAP) approach.

In both ASAP and ALAP scheduling, the operations that belong to the critical path of the design are not given any special priority over other operations. Thus, excessive delay may be imposed on the critical path operations. This is not good for the quality of the produced implementation. On the contrary, list scheduling utilizes a global priority function to select the next operation to be scheduled. This global priority function can be either the mobility (Pangrle & Gajski, 1987) of the operation or its urgency (Girczyc et al., 1985). Force-directed scheduling (Paulin & Knight, 1989) calculates the range of control steps for each operation between the operation's ASAP and ALAP state assignment. It then attempts to reduce the total number of functional units of the design's implementation, in order to evenly distribute the operations of the same type into all of the available states of the range.

The problem with constructive scheduling is that there is not any lookahead into future assignment of operations into the same control step, which may lead to sub-optimal implementations. After an initial schedule is delivered by any of the above scheduling algorithms, then iterative scheduling produces new schedules, by iteratively re-scheduling sequences of operations that maximally reduce the cost functions (Park & Kyung, 1991). This method is suitable for dataflow-oriented designs with linear control. In order to schedule control-intensive designs, the use of loop pipelining (Park & Parker, 1988) and loop folding (Girczyc, 1987), have been reported in the bibliography.

2.2 Allocation and binding tasks

Allocation determines the type of resource storage and functional units, selected from the library of components, for each data object and operation of the input program. Allocation also calculates the number of resources of each type that are needed to implement every operation or data variable. Binding assigns operations, data variables, data structures and data transfers onto functional units, storage elements (registers or memory blocks) and interconnections respectively. Also binding makes sure that the design's functionality does not change by using the selected library components.

Generally, there are three kinds of solutions to the allocation problem: constructive techniques, decomposition techniques and iterative approaches. Constructive allocation techniques start with an empty implementation and progressively build the datapath and control parts of the implementation by adding more functional, storage and interconnection elements while they traverse the CDFG or any other type of internal graph/representation format. Decomposition techniques divide the allocation problem into a sequence of well-defined independent sub-tasks. Each such sub-task is a graph-based theoretical problem which is solved with any of the three well known graph methods: clique partitioning, the left-edge technique and the weighted bipartite matching technique. The task of finding the minimum cliques in the graph which is the solution for the sub-tasks, is a NP-hard problem, so heuristic approaches (Tseng & Siewiorek, 1986) are utilized for allocation.

Because the conventional sub-task of storage allocation, ignores the side-effects between the storage and interconnections allocation, when using the clique partitioning technique, graph edges are enhanced with weights that represent the effect on interconnection complexity. The left-edge algorithm is applied on the storage allocation problem, and it allocates the minimum number of registers (Kurdahi & Parker, 1987). A weighted, bipartite-matching algorithm is used to solve both the storage and functional unit allocation problems. First a bipartite graph is generated which contains two disjoint sets, e.g. one for variables and one for registers, or one for operations and one for functional units. An edge between one node of the one of the sets and one node of the other represents an allocation of e.g. a variable to a register. The bipartite-matching algorithm considers the effect of register allocation on the design's interconnection elements, since the edges of the two sets of the graph are weighted (Huang et al., 1990). In order to improve the generated datapaths iteratively, a simple assignment exchange, using the pairwise exchange of the simulated annealing, or by using a branch-and-bound approach is utilized. The latter reallocates groups of elements of different types (Tsay & Hsu, 1990).

2.3 Early high-level synthesis

HLS has been an active research field for more than two decades now. Early approaches of experimental synthesis tools that synthesized small subsets of programming constructs or proprietary modeling formats have emerged since the late 80's. As an example, an early tool that generated hardware structures from algorithmic code, written in the PASCAL-like, Digital System Specification language (DSL) is reported in (Camposano & Rosenstiel, 1989). This synthesis tool performs the circuit compilation in two steps: first step is datapath synthesis which is followed by control synthesis. Examples of other behavioral circuit specification languages of that time, apart from DSL, were DAISY (Johnson, 1984), ISPS (Barbacci et al., 1979), and MIMOLA (Marwedel, 1984).

In (Casavant et al., 1989) the circuit to be synthesized is described with a combination of algorithmic and structural level code and then the PARSIFAL tool synthesizes the code into a bit-serial DSP circuit implementation. The PARSIFAL tool is part of a larger E-CAD system called FACE and which included the FACE design representation and design manager core. FACE and PARSIFAL were suitable for DSP pipelined implementations, rather than for a more general behavioral hardware models with hierarchy and complex control.

According to (Paulin & Knight, 1989) scheduling consists of determining the propagation delay of each operation and then assigning all operations into control steps (states) of a finite state machine. List-scheduling uses a local priority function to postpone the assignment of operations into states, when resource constraints are violated. On the contrary, force-directed scheduling (FDS) tries to satisfy a global execution deadline (time constraint) while minimizing the utilized hardware resources (functional units, registers and busses). The force-directed list scheduling (FDLS) algorithm attempts to implement the fastest schedule while satisfying fixed hardware resource constraints.

The main HLS tasks in (Gajski & Ramachandran, 1994) include allocation, scheduling and binding. According to (Walker & Chaudhuri, 1995) scheduling is finding the sequence of which operations to execute in a specific order so as to produce a schedule of control steps with allocated operations in each step of the schedule; allocation is defining the required number of functional, storage and interconnect units; binding is assigning operations to functional units, variables and values to storage elements and forming the interconnections amongst them to form a complete working circuit that executes the functionality of the source behavioral model.

The V compiler (Berstis, 1989) translates sequential descriptions into RTL models using parsing, scheduling and resource allocation. The source sequential descriptions are written in the V language which includes queues, asynchronous calls and cycle blocks and it is tuned to a kind of parallel hardware RTL implementations. The V compiler utilizes percolation scheduling (Fisher, 1981) to achieve the required degree of parallelism by meeting time constraints.

A timing network is generated from the behavioral design in (Kuehlmann & Bergamaschi, 1992) and is annotated with parameters for every different scheduling approach. The scheduling approach in this work attempts to satisfy a given design cycle for a given set of resource constraints, using the timing model parameters. This approach uses an integer linear program (ILP) which minimizes a weighted sum of area and execution time of the

implementation. According to the authors, their Symphony tool delivers better area and speed than ADPS (Papachristou & Konuk, 1990). This synthesis technique is suitable for data-flow designs (e.g. DSP blocks) and not for more general complex control flow designs.

The CALLAS synthesis framework (Biesenack et al., 1993), transforms algorithmic, behavioral VHDL models into VHDL RTL and gate netlists, under timing constraints. The generated circuit is implemented using a Moore-type finite state machine (FSM), which is consistent with the semantics of the VHDL subset used for the specification code. Formal verification techniques such as equivalence checking, which checks the equivalence between the original VHDL FSM and the synthesized FSM are used in the CALLAS framework by using the symbolic verifier of the Circuit Verification Environment (CVE) system (Filkorn, 1991).

The Ptolemy framework (Kalavade & Lee, 1993) allows for an integrated hardware-software co-design methodology from the specification through to synthesis of hardware and software components, simulation and evaluation of the implementation. The tools of Ptolemy can synthesize assembly code for a programmable DSP core (e.g. DSP processor), which is built for a synthesis-oriented application. In Ptolemy, an initial model of the entire system is partitioned into the software and hardware parts which are synthesized in combination with their interface synthesis.

The Cosyma hardware-software co-synthesis framework (Ernst et al., 1993) realizes an iterative partitioning process, based on a hardware extraction algorithm which is driven by a cost function. The primary target in this work is to minimize customized hardware within microcontrollers but the same time to allow for space exploration of large designs. The specialized co-processors of the embedded system can be synthesized using HLS tools. The specification language is based on C with various extensions. The generated hardware descriptions are in turn ported to the Olympus HLS tool (De Micheli et al., 1990). The presented work included tests and experimental results based on a configuration of an embedded system, which is built around the Sparc microprocessor.

Co-synthesis and hardware-software partitioning are executed in combination with control parallelism transformations in (Thomas et al., 1993). The hardware-software partition is defined by a set of application-level functions which are implemented with application-specific hardware. The control parallelism is defined by the interaction of the processes of the functional behavior of the specified system. The system behavior is modeled using a set of communicating sequential processes (Hoare, 1985). Each process is then assigned either to hardware or to software implementation.

A hardware-software co-design methodology, which employs synthesis of heterogeneous systems, is presented in (Gupta & De Micheli, 1993). The synthesis process is driven by timing constraints which drive the mapping of tasks onto hardware or software parts so that the performance requirements of the intended system are met. This method is based on using modeling and synthesis of programs written in the HardwareC language. An example application which was used to test the methodology in this work was an Ethernet-based network co-processor.

2.4 Next generation high-level synthesis tools

More advanced methodologies and tools started appearing from the late 90s and continue with improved input programming code sets as well as scheduling and other optimization

algorithms. The CoWare hardware-software co-design environment (Bolsens et al., 1997) is based on a data model that allows the user to specify, simulate and produce heterogeneous implementations from heterogeneous specification source models. This synthesis approach focuses on designing telecommunication systems that contain DSP, control loops and user interfaces. The synchronous dataflow (SDF) type of algorithms found in a category of DSP applications, can easily be synthesized into hardware from languages such as SILAGE (Genin et al., 1990), DFL (Willekens et al., 1994), and LUSTRE (Halbwachs et al. 1991). In contrast to this, dynamic dataflow (DDF) algorithms consume and produce tokens that are data-dependent, and thus they allow for complex if-then-else and while loop control constructs. CAD systems that allow for specifying both SDF and DDF algorithms and perform as much as possible static scheduling are the DSP-station from Mentor Graphics (Van Canneyt, 1994), PTOLEMY (Buck et al., 1994), GRAPE-II (Lauwereins et al., 1995), COSSAP from Synopsys and SPW from the Alta group (Rafie et al., 1994).

C models that include dynamic memory allocation, pointers and the functions malloc and free are mapped onto hardware in (Semeria et al., 2001). The SpC tool which was developed in this work resolves pointer variables at compile time and thus C functional models are synthesized into Verilog hardware models. The synthesis of functions in C, and therefore the resolution of pointers and malloc/free inside of functions, is not included in this work. The different techniques and optimizations described above have been implemented using the SUIF compiler environment (Wilson et al., 1994).

A heuristic for scheduling behavioral specifications that include a lot of conditional control flow, is presented in (Kountouris & Wolinski, 2002). This heuristic is based on a powerful intermediate design representation called hierarchical conditional dependency graph (HCDG). HCDG allows chaining and multicycling, and it enables advanced techniques such as conditional resource sharing and speculative execution, which are suitable for scheduling conditional behaviors. The HLS techniques in this work were implemented in a prototype graphical interactive tool called CODESIS which used HCDG as its internal design representation. The tool generates VHDL or C code from the HCDG, but no translation of standard programming language code into HCDG are known so far.

A coordinated set of coarse-grain and fine-grain parallelizing HLS transformations on the input design model are discussed in (Gupta et al., 2004). These transformations are executed in order to deliver synthesis results that don't suffer from the negative effects of complex control constructs in the specification code. All of the HLS techniques in this work were implemented in the SPARK HLS tool, which transforms specifications in a small subset of C into RTL VHDL hardware models. SPARK utilizes both control/data flow graphs (CDFGs) as well as an encapsulation of basic design blocks inside hierarchical task graphs (HTGs), which enable coarse-grain code restructuring such as loop transformations and an efficient way to move operations across large pieces of specification code.

Typical HLS tasks such as scheduling, resource allocation, module binding, module selection, register binding and clock selection are executed simultaneously in (Wang et al., 2003) so as to achieve better optimization in design energy, power and area. The scheduling algorithm utilized in this HLS methodology applies concurrent loop optimization and multicycling and it is driven by resource constraints. The state transition graph (STG) of the design is simulated in order to generate switched capacitance matrices. These matrices are then used to estimate power/energy consumption of the design's datapath. Nevertheless,

the input to the HLS tool, is not programming language code but a proprietary format representing an enhanced CDFG as well as a RTL design library and resource constraints.

An incremental floorplanner is described in (Gu et al., 2005) which is used in order to combine an incremental behavioral and physical optimization into HLS. These techniques were integrated into an existing interconnect-aware HLS tool called ISCALP (Zhong & Jha, 2002). The new combination was named IFP-HLS (incremental floorplanner high-level synthesis) tool, and it attempts to concurrently improve the design's schedule, resource binding and floorplan, by integrating high-level and physical design algorithms.

(Huang et al., 2007) discusses a HLS methodology which is suitable for the design of distributed logic and memory architectures. Beginning with a behavioral description of the system in C, the methodology starts with behavioral profiling in order to extract simulation statistics of computations and references of array data. Then array data are distributed into different partitions. An industrial tool called Cyber (Wakabayashi, 1999) was developed which generates a distributed logic/memory micro-architecture RTL model, which is synthesizable with existing RTL synthesizers, and which consists of two or more partitions, depending on the clustering of operations that was applied earlier.

A system specification containing communicating processes is synthesized in (Wang et al., 2003). The impact of the operation scheduling is considered globally in the system critical path (as opposed to the individual process critical path), in this work. It is argued by the authors in this work, that this methodology allocates the resources where they are mostly needed in the system, which is in the critical paths, and in this way it improves the overall multi-process designed system performance.

The work in (Gal et al., 2008) contributes towards incorporating memory access management within a HLS design flow. It mainly targets digital signal processing (DSP) applications but also other streaming applications can be included along with specific performance constraints. The synthesis process is performed on the extended data-flow graph (EDFG) which is based on the signal flow graph. Mutually exclusive scheduling methods (Gupta et al., 2003; Wakabayashi & Tanaka, 1992) are implemented with the EDFG. The graph which is processed by a number of annotations and improvements is then given to the GAUT HLS tool (Martin et al., 1993) to perform operator selection and allocation, scheduling and binding.

A combined execution of operation decomposition and pattern-matching techniques is targeted to reduce the total circuit area in (Molina et al., 2009). The datapath area is reduced by decomposing multicycle operations, so that they are executed on monocycle functional units (FUs that take one clock cycle to execute and deliver their results). A simple formal model that relies on a FSM-based formalism for describing and synthesizing on-chip communication protocols and protocol converters between different bus-based protocols is discussed in (Avnit, 2009). The utilized FSM-based format is at an abstraction level which is low enough so that it can be automatically translated into HDL implementations. The generated HDL models are synthesizable with commercial tools. Synchronous FSMs with bounded counters that communicate via channels are used to model communication protocols. The model devised in this work is validated with an example of communication protocol pairs which included AMBA APB and ASB. These protocols are checked regarding their compatibility, by using the formal model.

The methodology of SystemCoDesigner (Keinert et al., 2009) uses an actor-oriented approach so as to integrate HLS into electronic system level (ESL) design space exploration tools. The design starts with an executable SystemC system model. Then, commercial synthesizers such as Forte's Cynthesizer are used in order to generate hardware implementations of actors from the behavioral model. This enables the design space exploration in finding the best candidate architectures (mixtures of hardware and software modules). After deciding on the chosen solution, the suitable target platform is then synthesized with the implementations of the hardware and software parts. The final step of this methodology is to generate the FPGA-based SoC implementation from the chosen hardware/software solution. Based on the proposed methodology, it seems that SystemCoDesigner method is suitable for stream-based applications, found in areas such as DSP, image filtering and communications.

A formal approach is followed in (Kundu et al., 2010) so as to prove that every HLS translation of a source code model produces a RTL model that is functionally-equivalent to the one in the behavioral input to the HLS tools. This technique is called translation validation and it has been maturing via its use in the optimizing software compilers. The validating system in this work is called SURYA, it is using the Symplify theorem prover and it was used to validate the SPARK HLS tool. This validation work found two bugs in the SPARK compilations.

The replacement of flip-flop registers with latches is proposed in (Paik et al., 2010) in order to yield better timing in the implemented designs. The justification for this is that latches are inherently more tolerant to process variations than flip-flops. These techniques were integrated into a tool called HLS-1. HLS-1 translates behavioral VHDL code into a synthesized netlist. Nevertheless, implementing registers with latches instead of edge-triggered flip-flops is generally considered to be cumbersome due to the complicated timing behavior of latches.

3. Synthesis for low power

A number of portable and embedded computing systems and applications such as mobile (smart) phones, PDAs, etc, require low power consumption therefore synthesis for low energy is becoming very important in the whole area of VLSI and embedded system design. During the last decade, industry and academia invested on significant part of research regarding VLSI techniques and HLS for low power design. In order to achieve low energy in the results of HLS and system design, new techniques that help to estimate power consumption at the high-level description level, are needed. In (Raghunathan et al., 1996), switching activity and power consumption are estimated at the RTL description taking also into account the glitching activity on a number of signals of the datapath and the controller. The spatial locality, the regularity, the operation count and the ratio of critical path to available time are identified in (Rabaey et al., 1995) with the aim to reduce the power consumption of the interconnections. The HLS scheduling, allocation and binding tasks consider such algorithmic statistics and properties in order to reduce the fanins and fanouts of the interconnect wires. This will result into reducing the complexity and the power consumed on the capacitance of the inteconnection buses (Mehra & Rabaey, 1996).

The effect of the controller on the power consumption of the datapath is considered in (Raghunathan & Jha, 1994). Pipelining and module selection was proposed in (Goodby et

al., 1994) for low power consumption. The activity of the functional units was reduced in (Musoll & Cortadella, 1995) by minimizing the transitions of the functional unit's inputs. This was utilized in a scheduling and resource binding algorithm, in order to reduce power consumption. In (Kumar et al., 1995) the DFG is simulated with profiling stimuli, provided by the user, in order to measure the activity of operations and data carriers. Then, the switching activity is reduced, by selecting a special module set and schedule. Reducing supply voltage, disabling the clock of idle elements, and architectural tradeoffs were utilized in (Martin & Knight, 1995) in order to minimize power consumption within HLS.

The energy consumption of memory subsystem and the communication lines within a multiprocessor system-on-a-chip (MPSoC) is addressed in (Issenin et al., 2008). This work targets streaming applications such as image and video processing that have regular memory access patterns. The way to realize optimal solutions for MPSoCs is to execute the memory architecture definition and the connectivity synthesis in the same step.

4. The CCC hardware synthesis method

The previous two sections reviewed related work in HLS methodologies. This section and the following six sections describe a particular, formal HLS methodology which is directly applicable on embedded system design, and it has been developed solely by the author of this chapter. The Formal Intermediate Format (FIF)[1] was invented and designed by the author of this chapter as a tool and media for the design encapsulation and the HLS transformations in the CCC (Custom Coprocessor Compilation) hardware compilation tool[2]. A near-complete analysis of FIF syntax and semantics can be found in (Dossis, 2010). The formal methodology discussed here is based on using predicate logic to describe the intermediate representations of the compilation steps, and the resolution of a set of transformation Horn clauses (Nilsson & Maluszynski, 1995) is used, as the building blocks of the prototype HLS tool.

The front-end compiler translates the algorithmic data of the source programs into the FIF's logic statements (logic facts). The inference logic rules of the back-end compiler transform the FIF facts into the hardware implementations. There is one-to-one correspondence between the source specification's subroutines and the generated hardware modules. The source code subroutines can be hierarchical, and this hierarchy is maintained in the generated hardware implementation. Each generated hardware model is a FSM-controlled custom processor (or co-processor, or accelerator), that executes a specific task, described in the source program code. This hardware synthesis flow is depicted in Figure 1.

Essentially the front-end compilation resembles software compilation and the back-end compilation executes formal transformation tasks that are normally found in HLS tools. This whole compilation flow is a formal transformation process, which converts the source code programs into implementable RTL (Register-Transfer Level) VHDL hardware accelerator models. If there are function calls in the specification code, then each subprogram call is transformed into an interface event in the generated hardware FSM. The interface event is

[1] The Formal Intermediate Format is patented with patent number: 1006354, 15/4/2009, from the Greek Industrial Property Organization

[2] This hardware compiler method is patented with patent number: 1005308, 5/10/2006, from the Greek Industrial Property Organization

used so that the "calling" accelerator uses the "services" of the "called" accelerator, as it is depicted in the source code hierarchy as well.

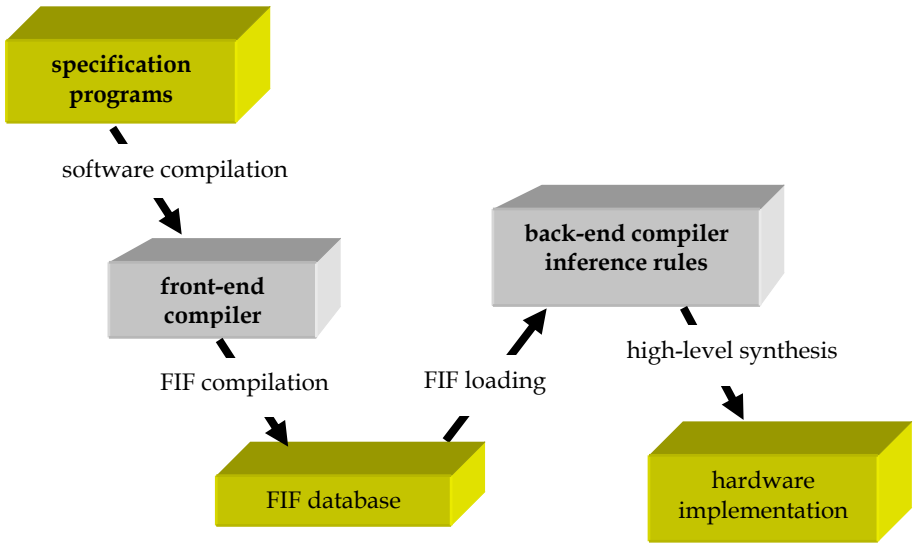


Fig. 1. Hardware synthesis flow and tools.

5. Back-end compiler inference logic rules

The back-end compiler consists of a very large number of logic rules. The back-end compiler logic rules are coded with logic programming techniques, which are used to implement the HLS algorithms of the back-end compilation phase. As an example, one of the latter algorithms reads and incorporates the FIF tables' facts into the compiler's internal inference engine of logic predicates and rules (Nilsson & Maluszynski, 1995). The back-end compiler rules are given as a great number of definite clauses of the following form:

$$A_0 \leftarrow A_1 \wedge \ldots \wedge A_n \text{ (where } n \geq 0) \qquad \text{(form 1)}$$

where $\leftarrow$ is the logical implication symbol ($A \leftarrow B$ means that if B applies then A applies), and $A_0, \ldots, A_n$ are atomic formulas (logic facts) of the form:

$$\text{predicate\_symbol(Var\_1, Var\_2, …, Var\_N)} \qquad \text{(form 2)}$$

where the positional parameters Var_1,…,Var_N of the above predicate "predicate_symbol" are either variable names (in the case of the back-end compiler inference rules), or constants (in the case of the FIF table statements). The predicate syntax in form 2 is typical of the way that the FIF facts and other facts interact with each other, they are organized and they are used internally in the inference engine. Thus, the hardware descriptions are generated as "conclusions" of the inference engine upon the FIF "facts". This is done in a formal way from the input programs by the back-end phase, which turns the overall transformation into a provably-correct compilation process. In essence, the FIF file consists of a number of such

atomic formulas, which are grouped in the FIF tables. Each such table contains a list of homogeneous facts which describe a certain aspect of the compiled program. E.g. all prog_stmt facts for a given subprogram are grouped together in the listing of the program statements table.

6. Inference logic and back-end transformations

The inference engine of the back-end compiler consists of a great number of logic rules (like the one in form 1) which conclude on a number of input logic predicate facts and produce another set of logic facts and so on. Eventually, the inference logic rules produce the logic predicates that encapsulate the writing of RTL VHDL hardware co-processor models. These hardware models are directly implementable to any hardware (e.g. ASIC or FPGA) technology, since they are technology and platform – independent. For example, generated RTL models produced in this way from the prototype compiler were synthesized successfully into hardware implementations using the Synopsys DC Ultra, the Xilinx ISE and the Mentor Graphics Precision software without the need of any manual alterations of the produced RTL VHDL code. In the following form 3 an example of such an inference rule is shown:

dont_schedule(Operation1, Operation2) ←

examine(Operation1, Operation2),

predecessor(Operation1, Operation2). (form 3)

The meaning of this rule that combines two input logic predicate facts to produce another logic relation (dont_schedule), is that when two operations (Operation1 and Operation2) are examined and the first is a predecessor of the second (in terms of data and control dependencies), then don't schedule them in the same control step. This rule is part of a parallelizing optimizer which is called "PARCS" (meaning: Parallel, Abstract Resource – Constrained Scheduler).

The way that the inference engine rules (predicates relations-productions) work is depicted in Figure 2. The last produced (from its rule) predicate fact is the VHDL RTL writing predicate at the top of the diagram. Right bellow level 0 of predicate production rule there is a rule at the -1 level, then level -2 and so on. The first predicates that are fed into this engine of production rules belong to level –K, as shown in this figure. Level –K predicate facts include of course the FIF facts that are loaded into the inference engine along with the other predicates of this level.

In this way, the back-end compiler works with inference logic on the basis of predicate relation rules and therefore, this process is a formal transformation of the FIF source program definitions into the hardware accelerator (implementable) models. Of course in the case of the prototype compiler, there is a very large number of predicates and their relation rules that are defined inside the implementation code of the back-end compiler, but the whole concept of implementing this phase is as shown in Figure 2. The user of the back-end compiler can select certain environment command list options as well as build an external memory port parameter file as well as drive the compiler's optimizer with specific resource constraints of the available hardware operators.

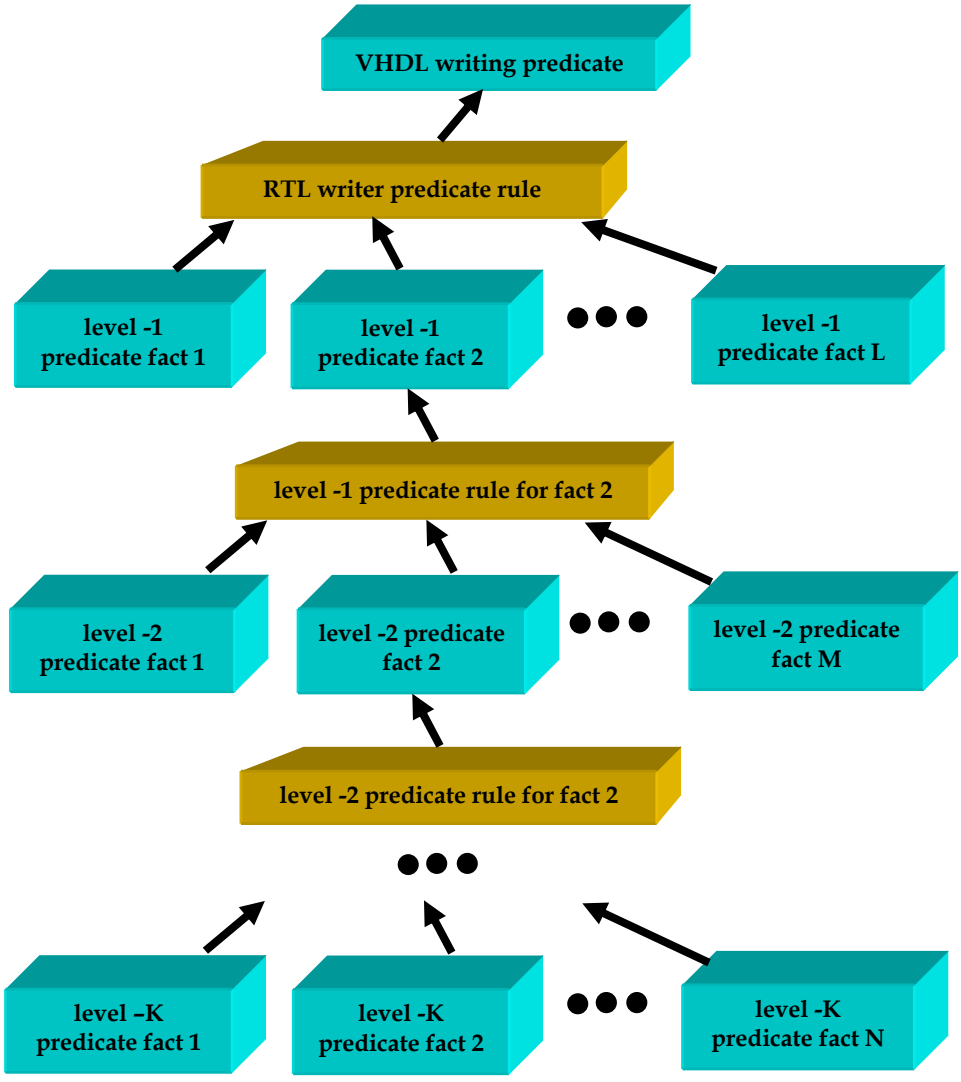


Fig. 2. The back-end inference logic rules structure.

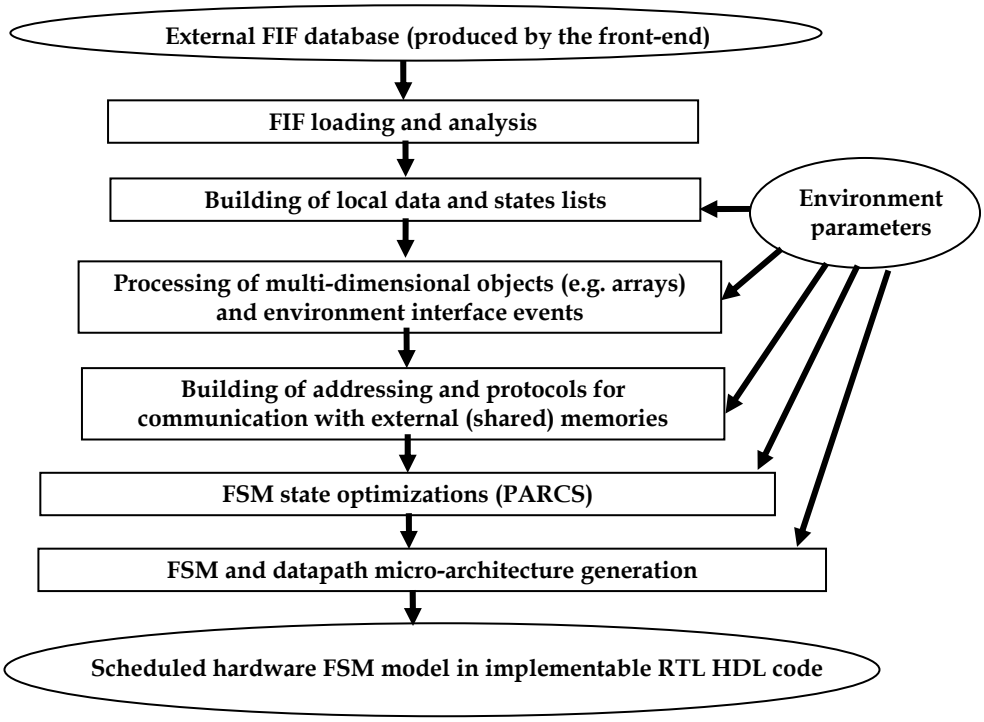


Fig. 3. The processing stages of the back-end compiler.

The most important of the back-end compilation stages can be seen in Figure 3. The compilation process starts with the loading of the FIF facts into the inference rule engine. After the FIF database is analyzed, the local data object, operation and initial state lists are built. Then the environment options are read and the temporary lists are updated with the special (communication) operations as well as the predecessor and successor dependency relation lists. After the complete initial schedule is built and concluded, the PARCS optimizer is run on it, and the optimized schedule is delivered to the micro-architecture generator. The transformation is concluded with the formation of the FSM and datapath implementation and the writing of the RTL VHDL model for each accelerator that is defined in each subprogram of the source code program.

A separate hardware accelerator model is generated from each subprogram in the system model code. All of the generated hardware models are directly implementable into hardware using commercial CAD tools, such as the Synopsys DC-ultra, the Xilinx ISE and the Mentor Graphics Precision RTL synthesizers. Also the hierarchy of the source program modules (subprograms) is maintained and the generated accelerators may be hierarchical. This means that an accelerator can invoke the services of another accelerator from within its processing states, and that other accelerator may use the services of yet another accelerator and so on. In this way, a subprogram call in the source code is translated into an external coprocessor interface event of the corresponding hardware accelerator.

7. The PARCS optimizer

PARCS aggressively attempts to schedule as many as possible operations in the same control step. The only limits to this are the data and control dependencies as well as the optional resource (operator) constraints, which are provided by the user.

1. start with the initial schedule (including the special external port operations)
2. Current PARCS state <- 1
3. Get the 1st state and make it the current state
4. Get the next state
5. Examine the next state's operations to find out if there are any dependencies with the current state
6. If there are no dependencies then absorb the next state's operations into the current PARCS state; If there are dependencies then finalize the so far absorbed operations into the current PARCS state, store the current PARCS state, PARCS state <- PARCS state + 1; make next state the current state; store the new state's operations into the current PARCS state
7. If next state is of conditional type (it is enabled by guarding conditions) then call the conditional (true/false branch) processing predicates, else continue
8. If there are more states to process then go to step 4, otherwise finalize the so far operations of the current PARCS state and terminate

Fig. 4. Pseudo-code of the PARCS scheduling algorithm.

The pseudo-code for the main procedures of the PARCS scheduler is shown in Figure 4. All of the predicate rules (like the one in form 1) of PARCS are part of the inference engine of the back-end compiler. A new design to be synthesized is loaded via its FIF into the back-end compiler's inference engine. Hence, the FIF's facts as well as the newly created predicate facts from the so far logic processing, "drive" the logic rules of the back-end compiler which generate provably-correct hardware architectures. It is worthy to note that although the HLS transformations are implemented with logic predicate rules, the PARCS optimizer is very efficient and fast. In most of benchmark cases that were run through the prototype hardware compiler flow, compilation did not exceed 1-10 minutes of run-time and the results of the compilation were very efficient as explained bellow.

8. Generated hardware architectures

The back-end stage of micro-architecture generation can be driven by command-line options. One of the options e.g. is to generate massively parallel architectures. The results of this option are shown in Figure 5. This option generates a single process – FSM VHDL description with all the data operations being dependent on different machine states. This implies that every operator is enabled by single wire activation commands that are driven by different state register values. This in turn means that there is a redundancy in the generated hardware, in a way that during part of execution time, a number of state-dedicated operators remain idle. However, this redundancy is balanced by the fact that this option achieves the fastest clock cycle, since the state command encoder, as well as the data

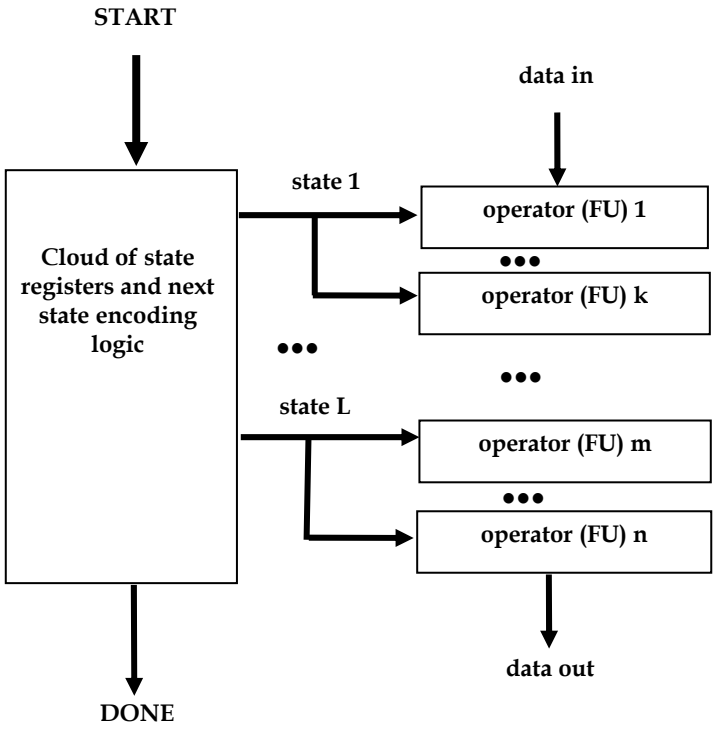


Fig. 5. Massively-parallel microarchitecture generation option.

multiplexers are replaced by single wire commands which don't exhibit any additional delay, and this option is very suitable to implement on large ASICs with plenty of resources.

Another micro-architecture option is the generation of traditional FSM + datapath based VHDL models. The results of this option are shown in Figure 6. With this option activated the generated VHDL models of the hardware accelerators include a next state process as well as signal assignments with multiplexing which correspond to the input data multiplexers of the activated operators. Although this option produces smaller hardware structures (than the massively-parallel option), it can exceed the target clock period due to larger delays through the data multiplexers that are used in the datapath of the accelerator.

Using the above micro-architecture options, the user of the CCC HLS tool can select various solutions between the fastest and larger massively-parallel micro-architecture, which may be suitable for richer technologies in terms of operators such as large ASICs, and smaller and more economic (in terms of available resources) technologies such as smaller FPGAs.

As it can be seen in Figure 5 and Figure 6, the produced co-processors (accelerators) are initiated with the input command signal START. Upon receiving this command the co-processors respond to the controlling environment using the handshake output signal BUSY

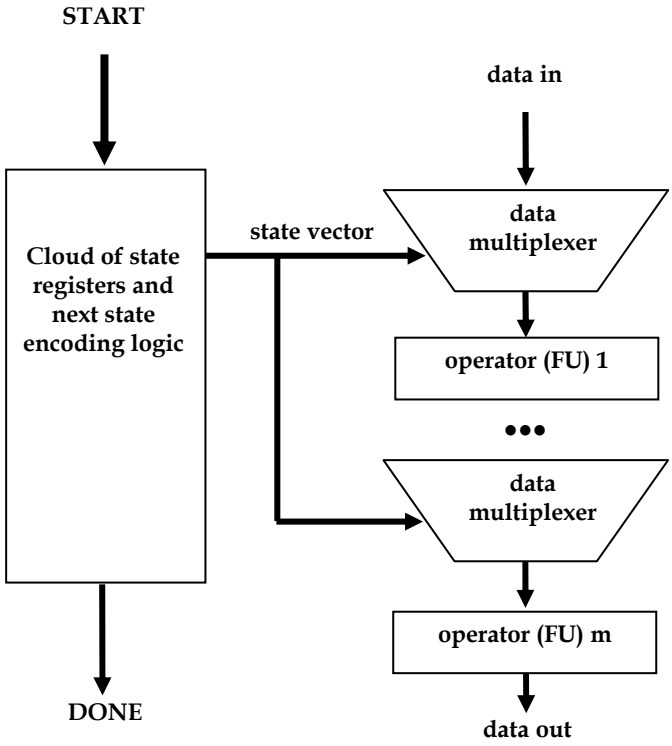


Fig. 6. The traditional FSM + datapath generated micro-architecture option.

and right after this, they start processing the input data in order to produce the results. This process may take a number of clock cycles and it is controlled by a set of states (discrete control steps). When the co-processors complete their processing, they notify their environment with the output signal DONE. In order to conclude the handshake the controlling environment (e.g. a controlling central processing unit) responds with the handshake input RESULTS_READ, to notify the accelerator that the processed result data have been read by the environment. This handshake protocol is also followed when one (higher-level) co-processor calls the services of another (lower-level) co-processor. The handshake is implemented between any number of accelerators (in pairs) using the START/BUSY and DONE/RESULTS_READ signals. Therefore, the set of executing co-processors can be also hierarchical in this way.

Other environment options, passed to the back-end compiler, control the way that the data object resources are used, such as registers and memories. Using a memory port configuration file, the user can determine that certain multi-dimensional data objects, such as arrays and array aggregates are implemented in external (e.g. central, shared) memories (e.g. system RAM). Otherwise, the default option remains that all data objects are allocated to hardware (e.g. on-chip) registers. All of the related memory communication protocols and

hardware ports/signals, are automatically generated by the back-end synthesizer, and without the need for any manual editing of the RTL code by the user. Both synchronous and asynchronous memory communication protocol generation are supported.

9. Co-processor execution system

The generated accelerators can be placed inside the computing environment that they accelerate or can be executed standalone. For every subprogram in the source specification code one co-processor is generated to speed up (accelerate) the particular system task. The whole system (both hardware and software models) is modeled in algorithmic ADA code which can be compiled and executed with the host compiler and linker to run and verify the operation of the whole system at the program code level. In this way, extremely fast verification can be achieved at the algorithmic level. It is evident that such behavioral (high-level) compilation and execution is orders of magnitude faster than conventional RTL simulations.

After the required co-processors are specified, coded in ADA, generated with the prototype hardware compiler and implemented with commercial back-end tools, they can be downloaded into the target computing system (if the target system includes FPGAs) and executed to accelerate certain system tasks. This process is shown in Figure 7. The accelerators can communicate with each other and with the host computing environment using synchronous handshake signals and connections with the system's handshake logic.

10. Experimental results and evaluation of the method

In order to evaluate the efficiency of the presented HLS and ESL method, many designs from the area of hardware compilation and high-level synthesis were run through the front-end and the back-end compilers. Five selected benchmarks include a DSP FIR filter, a second order differential equation iterative solver, a well-known high-level synthesis benchmark, a RSA crypto-processor from cryptography applications, a synthetic benchmark that uses two level nested for-loops, and a large MPEG video compression engine. The fourth benchmark includes subroutines with two-dimensional data arrays stored in external memories. These data arrays are processed within the bodies of 2-level nested loops. All of the above generated accelerators were simulated and the RTL behavior matched the input source program's functionality. The state number reduction after applying the PARCS optimizer, on the various modules of the five benchmarks is shown in Table 1.

Moreover, the number of lines of RTL code is orders of magnitude more compared with the lines of the source code model for each sub-module. This indicates the gain in engineering productivity when the prototype ESL tools are used to automatically implement the computing products. It is well accepted in the engineering community that the coding & verification time at the algorithmic program level is only a small fraction of the time required for verifying designs at the RTL or the gate-netlist level. There were more than 400 states in the initial schedule of the MPEG benchmark. In addition to this, manual coding is extremely prone to errors which are very cumbersome and time-consuming to correct with (traditional) RTL simulations and debugging.

The specification (source code) model of the various benchmarks, and all of the designs using the prototype compilation flow, contains unaltered regular ADA program code,

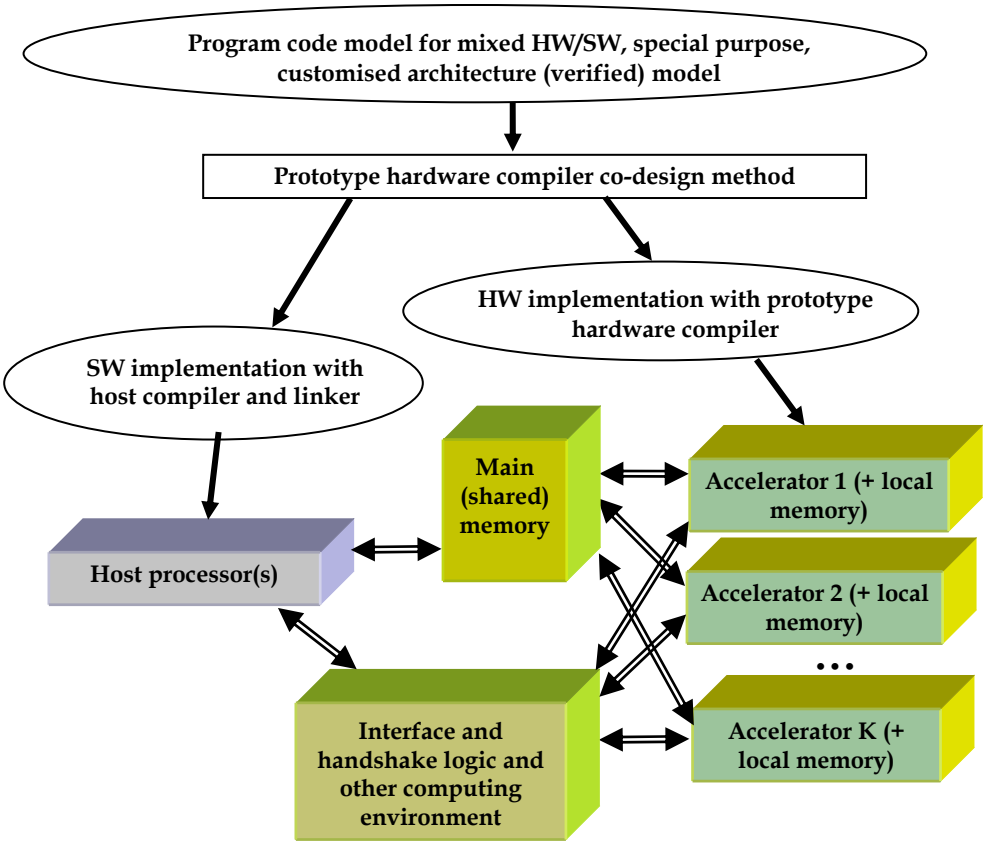


Fig. 7. Host computing environment and accelerators execution configuration.

without additional semantics and compilation directives which are usual in other synthesis tools which compile code in SystemC, HandelC, or any other modified program code with additional object class and TLM primitive libraries. This advantage of the presented methodology eliminates the need for the system designers to learn a new language, a new set of program constructs or a new set of custom libraries. Moreover, the programming constructs and semantics, that the prototype HLS compiler utilizes are the subset which is common to almost all of the imperative and procedural programming languages such as ANSI C, Pascal, Modula, Basic etc. Therefore, it is very easy for a user that is familiar with these other imperative languages, to get also familiar with the rich subset of ADA that the prototype hardware compiler processes. It is estimated that this familiarization doesn't exceed a few days, if not hours for the very experienced software/system programmer/modeler.

Module name	Initial schedule states	PARCS parallelized states	State reduction rate
FIR filter main routine	17	10	41%
Differential equation solver	20	13	35%
RSA main routine	16	11	31%
nested loops 1st subroutine	28	20	29%
nested loops 2nd subroutine (with embedded mem)	36	26	28%
nested loops 2nd subroutine (with external mem)	96	79	18%
nested loops 3rd subroutine	15	10	33%
nested loops 4th subroutine	18	12	33%
nested loops 5th subroutine	17	13	24%
MPEG 1st subroutine	88	56	36%
MPEG 2nd subroutine	88	56	36%
MPEG 3rd subroutine	37	25	32%
MPEG top subroutine (with embed. mem)	326	223	32%
MPEG top subroutine (with external mem)	462	343	26%

Table 1. State reduction statistics from the IKBS PARCS optimizer.

The following Table 2 contains the area and timing statistics of the main module of the MPEG application synthesis runs. Synthesis was executed on a Ubuntu 10.04 LTS linux server with Synopsys DC-Ultra synthesizer and the 65nm UMC technology libraries. From this table a reduction in terms of area can be observed for the FSM+datapath implementation against the massively parallel one. Nevertheless, due to the quality of the technology libraries the speed target of 2 ns clock period was achieved in all 4 cases.

Area/time statistic	massively-parallel, initial schedule	massively-parallel, PARCS schedule	FSM + datapath, initial schedule	FSM + datapath, PARCS schedule
area in square nm	117486	114579	111025	107242
equivalent number of NAND2 gates	91876	89515	86738	83783
achievable clock period	2 ns	2 ns	2 ns	2 ns
achievable clock frequency	500 MHz	500 MHz	500 MHz	500 MHz

Table 2. Area and timing statistics from UMC 65nm technology implementation.

Moreover, the area reduction for the FSM+datapath implementations of both the initial schedule and the optimized (by PARCS) one isn't dramatic and it reaches to about 6 %. This happens because the overhead of massively-parallel operators is balanced by the large amount of data and control multiplexing in the case of the FSM+datapath option.

11. Conclusions and future work

This chapter includes a discussion and survey of past and present existing ESL HLS tools and related synthesis methodologies suitable for embedded systems. Formal and heuristic techniques for the HLS tasks are discussed and more specific synthesis issues are analyzed. The conclusion from this survey is that the authors prototype ESL behavioral synthesizer is unique in terms of generality of input code constructs, the formal methodologies employed and the speed and utility of the developed hardware compiler.

One important contribution of this work is a provably-correct, ESL, and HLS method and a unified prototype tool-chain, which is based on compiler-compiler and formal logic inference techniques. The prototype tools transform a number of arbitrary input subprograms (for now coded in the ADA language) into an equivalent number of correct-by-construction and functionally-equivalent RTL VHDL hardware accelerator descriptions. Encouraging state-reduction rates of the PARCS scheduler-optimizer were observed for five benchmarks in this chapter, which exceed 30% in some cases. Using its formal flow, the prototype hardware compiler can be used to develop complex embedded systems in orders of magnitude shorter time and lower engineering effort, than that which are usually required using conventional design approaches such as RTL coding or IP encapsulation and schematic entry using custom libraries.

Existing HLS tools compile usually a small-subset of the programming language, and sometimes with severe restrictions in the type of constructs they accept (some of them don't accept while-loops for example). Furthermore, most of them are suited for linear, data-flow oriented specifications. However, a large number of applications found in embedded and telecommunication systems, mobile and other portable computing platforms involve a great deal of complex control flow with nesting and hierarchy levels. For this kind of applications most of HLS tools produce low level of quality results. The prototype ESL tool developed by the author has proved that it can deliver a better quality of results in applications with complex control such as image compression and processing standards.

Future extensions of this work include undergoing work to upgrade the front-end phase to accommodate more input programming languages (e.g. ANSI-C, C++) and the back-end HDL writer to include more back-end RTL languages (e.g. Verilog HDL), which are currently under development. Another extension could be the inclusion of more than 2 operand operations as well as multi-cycle arithmetic unit modules, such as multi-cycle operators, to be used in datapath pipelining. Moreover, there is ongoing work to extend the FIF's semantics so that it can accommodate embedding of IP blocks (such as floating-point units) into the compilation flow, and enhance further the schedule optimizer algorithm for even more reduced schedules. Furthermore, connection flows from the front-end compiler to even more front-end diagrammatic system modeling formats such as the UML formulation are currently investigated.

12. References

Avnit K., D'silva V., Sowmya A., Ramesh S. & Parameswaran S (2009) Provably correct on-chip communication: A formal approach to automatic protocol converter synthesis. *ACM Trans on Des Autom of Electr Sys (TODAES)*, ISSN: 1084-4309, Vol. 14, No. 2, article no: 19, March 2009.

Barbacci M., Barnes G., Cattell R. & Siewiorek D. (1979). *The ISPS Computer Description Language*. Report CMU-CS-79-137, dep. of Computer Science, Carnegie-Mellon University, USA.

Berstis V. (1989). The V compiler: automatic hardware design. *IEEE Des & Test of Comput*, Vol. 6, No. 2, pp. 8–17.

Biesenack J., Koster M., Langmaier A., Ledeux S., Marz S., Payer M., Pilsl M., Rumler S., Soukup H., Wehn N. & Duzy P. (1993). The Siemens high-level synthesis system CALLAS. *IEEE trans on Very Large Scale Integr (VLSI) sys*, Vol. 1, No. 3, September 1993, pp. 244-253.

Bolsens I., De Man H., Lin B., Van Rompaey K., Vercauteren S. & Verkest D. (1997). Hardware/software co-design of digital telecommunication systems. *Proceedings of the IEEE*, Vol. 85, No. 3, pp. 391-418.

Buck J., Ha S., Lee E. & Messerschmitt D. (1992). PTOLEMY: A framework for simulating and prototyping heterogeneous systems. *Invited Paper in the International Journal of Computer Simulation*, 31 August 1992. pp. 1-34.

Camposano R. & Rosenstiel W. (1989). Synthesizing circuits from behavioral descriptions. *IEEE Trans Comput-Aided Des Integr Circuits Syst*, Vol. 8, No. 2, pp. 171-180.

Casavant A., d'Abreu M., Dragomirecky M., Duff D., Jasica J., Hartman M., Hwang K. & Smith W. (1989). A synthesis environment for designing DSP systems. *IEEE Des & Test of Comput*, Vol. 6, No. 2, pp. 35–44.

De Micheli G., Ku D., Mailhot F. & Truong T. (1990). The Olympus synthesis system. *IEEE Des & Test of Comput*, Vol. 7, No. 5, October 1990, pp. 37-53.

Dossis M (2010) Intermediate Predicate Format for design automation tools. *Journal of Next Generation Information Technology (JNIT)*, Vol. 1, No. 1, pp. 100-117.

Ernst R., Henkel J. & Benner T. (1993). Hardware-software cosynthesis for microcontrollers. *IEEE Des & Test of Comput*, Vol. 10, No. 4, pp. 64-75.

Filkorn T. (1991). A method for symbolic verification of synchronous circuits, *Proceedings of the Comp Hardware Descr Lang and their Application (CHDL 91)*, pp. 229-239, Marseille, France 1991.

Fisher J (1981). Trace Scheduling: A technique for global microcode compaction. *IEEE trans. on comput*, Vol. C-30, No. 7, pp. 478-490.

Gajski D., & Ramachandran L. (1994). Introduction to high-level synthesis. *IEEE Des & Test of Comput*, Vol. 11, No. 4, pp. 44-54.

Gal B., Casseau E. & Huet S. (2008) Dynamic Memory Access Management for High-Performance DSP Applications Using High-Level Synthesis. *IEEE Trans on Very Large Scale Integr (VLSI)*, ISSN: 1063-8210, Vol. 16, No. 11, November 2008, pp. 1454-1464.

Genin D., Hilfinger P., Rabaey J., Scheers C. & De Man H. (1990). DSP specification using the SILAGE language, *Proceedings of the Int Conf on Acoust Speech Signal Process*, pp. 1056–1060, Albuquerque, NM., USA, 3-6 April 1990.

Girczyc E. (1987). Loop winding—a data flow approach to functional pipelining, *Proceedings of the International Symp on Circ and Syst*, pp. 382–385, 1987.

Girczyc E., Buhr R. & Knight J. (1985). Applicability of a subset of Ada as an algorithmic hardware description language for graph-based hardware compilation. *IEEE Trans Comput-Aided Des Integ Circuits Syst*, Vol. 4, No. 2, pp. 134-142.

Goodby L., Orailoglu A. & Chau P. (1994) Microarchitecture synthesis of performance-constrained low-power VLSI designs, *Proceedings of the Intern Conf on Comp Des (ICCD)*, ISBN: 0-8186-6565-3, Cambridge, MA , USA, 10-12 October 1994, pp. 323–326.

Gu Z., Wang J., Dick R. & Zhou H. (2005) Incremental exploration of the combined physical and behavioral design space. *Proceedings of the 42nd annual conf on des aut DAC '05*, Anaheim, CA, USA, June 13-17, 2005, pp. 208-213.

Gupta R. & De Micheli G. (1993). Hardware-software cosynthesis for digital systems. *IEEE Des & Test of Comput*, Vol. 10, No. 3, pp. 29-41.

Gupta S., Gupta R., Dutt N. & Nicolau A., (2003) Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits, *Proceedings of the IEEE Conf Comput Digit Techn*, ISSN: 1350-2387, 22 Sept. 2003, Vol. 150, No. 5, pp. 330–337.

Gupta S., Gupta R., Dutt N. & Nikolau A. (2004) Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis. *ACM Trans on Des Aut of Electr Sys*, Vol. 9, No. 4, September 2004, pp. 441–470.

Halbwachs N., Caspi P., Raymond P. & Pilaud D. (1991). The synchronous dataflow programming language Lustre, *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1305–1320.

Hoare C. (1985). *Communicating sequential processes*. Prentice-Hall, Englewood Cliffs, N.J., USA.

Huang C., Chen Y., Lin Y. & Hsu Y. (1990). Data path allocation based on bipartite weighted matching, *Proceedings of the Des Autom Conf (DAC)*, pp. 499–504, Orlando, Florida, USA, June, 1990.

Huang C., Ravi S., Raghunathan A. & Jha N. (2007) Generation of Heterogeneous Distributed Architectures for Memory-Intensive Applications Through High-Level Synthesis. *IEEE Trans on Very Large Scale Integr (VLSI)*, Vol. 15, No. 11, November 2007, pp. 1191-1204.

Issenin I, Brockmeyer E, Durinck B, Dutt ND (2008) Data-Reuse-Driven Energy-Aware Cosynthesis of Scratch Pad Memory and Hierarchical Bus-Based Communication Architecture for Multiprocessor Streaming Applications. *IEEE Trans on Comp-Aided Des of Integr Circ and Sys*, ISSN: 0278-0070, Vol. 27, No. 8, Aug. 2008, pp. 1439-1452.

Johnson S. (1984) *Synthesis of Digital Designs from Recursion Equations*. MA: MIT press, Cambridge.

Kalavade A. & Lee E. (1993). A hardware-software codesign methodology for DSP applications. *IEEE Des & Test of Comput*, Vol. 10, No. 3, pp. 16-28.

Keinert J., Streubuhr M., Schlichter T., Falk J., Gladigau J., Haubelt C., Teich J. & Meredith M. (2009) SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM *Trans on Des Autom of Electr Sys (TODAES)*, ISSN: 1084-4309, Vol. 14, No. 1, article no: 1, January 2009.

Kountouris A. & Wolinski C. (2002) Efficient Scheduling of Conditional Behaviors for High-Level Synthesis. *ACM Trans. on Design Aut of Electr Sys*, Vol. 7, No. 3, July 2002, pp. 380–412.

Kuehlmann A. & Bergamaschi R. (1992). Timing analysis in high-level synthesis, *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design (ICCAD '92)*, pp. 349-354.

Kumar N., Katkoori S., Rader L. & Vemuri R. (1995) Profile-driven behavioral synthesis for low-power VLSI systems. *IEEE Des Test of Comput*, ISSN: 0740-7475, Vol. 12, No. 3, Autumn 1995, pp. 70–84.

Kundu S., Lerner S. & Gupta R. (2010) Translation Validation of High-Level Synthesis. *IEEE Trans Comput-Aided Des Integ Circuits Syst*, ISSN: 0278-0070 ,Vol. 29, No. 4, April 2010, pp. 566-579.

Kurdahi F. & Parker A. (1987). REAL: A program for register allocation, *Proceedings of the Des Autom Conf (DAC)*, pp. 210–215 , Miami Beach, Florida, USA, June, 1987.

Lauwereins R., Engels M., Ade M. & Peperstraete, J. (1995). GRAPE-II: A system level prototyping environment for DSP applications. *IEEE Computer*, Vol. 28, No. 2, February 1995, pp. 35–43.

Martin E., Santieys O. & Philippe J. (1993) GAUT, an architecture synthesis tool for dedicated signal processors, *Proceedings of the IEEE Int Eur Des Autom Conf (Euro-DAC)*, Hamburg, Germany, Sep. 1993, pp. 14–19.

Martin R. & Knight J. (1995) Power-profiler: Optimizing ASICs power consumption at the behavioral level, *Proceedings of the Des Autom Conf (DAC)*, ISBN: 0-89791-725-1, San Francisco, CA, USA, 1995, pp. 42-47.

Marwedel P. (1984). The MIMOLA design system: Tools for the design of digital processors, *Proceedings of the 21st Design Automation Conf (DAC)*, pp. 587-593.

Mehra R. & Rabaey J. (1996) Exploiting regularity for low-power design. *Dig of Techn Papers, Intern Conf on Comp-Aided Des (ICCAD)*, ISBN:0-8186-7597-7, San Jose, CA, USA, November 1996, pp. 166–172.

Molina M., Ruiz-Sautua R., Garcia-Repetto P. & Hermida R (2009) Frequent-Pattern-Guided Multilevel Decomposition of Behavioral Specifications. *IEEE Trans Comput-Aided Des Integ Circuits Syst*, ISSN: 0278-0070, Vol. 28, No. 1, January 2009, pp. 60-73.

Musoll E. & Cortadella J. (1995) Scheduling and resource binding for low power, *Proceedings of the Eighth Symp on Sys Synth*, ISBN: 0-8186-7076-2, Cannes , France, 13-15 September 1995, pp.104–109.

Nilsson U. & Maluszynski J. (1995) *Logic Programming and Prolog*. John Wiley & Sons Ltd., 2nd Edition, 1995.

Paik S., Shin I., Kim T. & Shin Y (2010) HLS-l: A High-Level Synthesis framework for latch-based architectures. *IEEE Trans Comput-Aided Des Integ Circuits Syst*, ISSN: 0278-0070, Vol. 29, No. 5, May 2010, pp. 657-670.

Pangrle B. & Gajski D. (1987). Design tools for intelligent silicon compilation. *IEEE Trans Comput-Aided Des Integ Circuits Syst,* Vol. 6, No. 6. pp. 1098–1112.

Papachristou C. & Konuk H. (1990). A Linear program driven scheduling and allocation method followed by an interconnect optimization algorithm, *Proceedings of the 27th ACM/IEEE Design Automation Conf (DAC)*, pp. 77-83.

Park I. & Kyung C. (1991). Fast and near optimal scheduling in automatic data path synthesis, *Proceedings of the Des Autom Conf (DAC)*, pp. 680–685, San Francisco, USA, 1991.

Park N. & Parker A. (1988). Sehwa: A software package for synthesis of pipelined data path from behavioral specification. *IEEE Trans Comput Aided Des Integrated Circuits Syst*, Vol. 7, No. 3, pp.356–370.

Paulin P. & Knight J. (1989). Algorithms for high-level synthesis. *IEEE Des & Test of Comput*, Vol. 6, No. 6, pp. 18-31.

Paulin P. & Knight J. (1989). Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Trans Comput-Aided Des Integ Circuits Syst*, Vol. 8, No 6, pp. 661–679.

Rabaey J., Guerra L. & Mehra R. (1995) Design guidance in the power dimension, *Proceedings of the 1995 Intern Conf on Acoustics, Speech, and Signal Proc*, ISBN: 0-7803-2431-5, Detroit, MI , USA, 9-12 May 1995, pp. 2837–2840.

Rafie M., et al. (1994) Rapid design and prototyping of a direct sequence spread-spectrum ASIC over a wireless link. *DSP and Multimedia Technol*, Vol. 3, No. 6, pp. 6–12.

Raghunathan A. & Jha N. (1994) Behavioral synthesis for low power, *Proceedings of the Intern Conf on Comp Des (ICCD)*, ISBN: 0-8186-6565-3, Cambridge, MA , USA, 10-12 October 1994 pp. 318–322.

Raghunathan A., Dey S. & Jha N. (1996) Register-transfer level estimation techniques for switching activity and power consumption, *Dig of Techn Papers, Intern Conf on Comp-Aided Des (ICCAD)*, ISBN: 0-8186-7597-7, San Jose, CA , USA, 10-14 November 1996, pp. 158–165.

Semeria L., Sato K. & De Micheli G. (2001) Synthesis of hardware models in C with pointers and complex data structures. *IEEE Trans VLSI Systems*, Vol. 9, No. 6, pp. 743–756.

Thomas D., Adams J. & Schmit H. (1993). A model and methodology for hardware-software codesign. *IEEE Des & Test of Comput*, Vol. 10, No. 3, pp. 6-15.

Tsay F., & Hsu Y. (1990). Data path construction and refinement. *Digest of Techn papers, Int Conf on Comp-Aided Des (ICCAD)*, pp. 308–311 , Santa Clara, CA, USA, November, 1990.

Tseng C. & Siewiorek D. (1986). Automatic synthesis of data path on digital systems. *IEEE Trans Comput Aided Des.Integ Circuits Syst*, Vol. 5, No. 3, pp. 379–395.

Van Canneyt M. (1994). Specification, simulation and implementation of a GSM speech codec with DSP station. *DSP and Multimedia Technol*, Vol. 3, No. 5, pp. 6–15.

Wakabayashi K. & Tanaka H. (1992) Global scheduling independent of control dependencies based on condition vectors, *Proceedings of the 29th ACM/IEEE Conf Des Autom (DAC)*, ISBN: 0-8186-2822-7, Anaheim, CA , USA, 8-12 June 1992, pp. 112-115.

Wakabayashi K. (1999) C-based synthesis experiences with a behavior synthesizer, "Cyber". *Proceedings of the Des Autom and Test in Eur Conf*, ISBN: 0-7695-0078-1, Munich, Germany, 9-12 March1999, pp. 390–393.

Walker R. & Chaudhuri S. (1995). Introduction to the scheduling problem. *IEEE Des & Test of Comput*, Vol. 12, No. 2, pp. 60–69.

Wang W., Raghunathan A., Jha N. & Dey S. (2003) High-level Synthesis of Multi-process Behavioral Descriptions, *Proceedings of the 16th IEEE International Conference on VLSI Design (VLSI'03)*, ISBN: 0-7695-1868-0, 4-8 Jan. 2003, pp. 467-473.

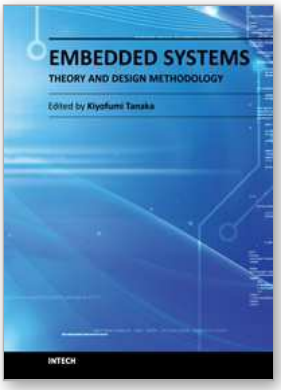
Wang W., Tan T., Luo J., Fei Y., Shang L., Vallerio K., Zhong L., Raghunathan A. & Jha N. (2003) A comprehensive high-level synthesis system for control-flow intensive behaviors, *Proceedings of the 13th ACM Great Lakes symp on VLSI GLSVLSI '03*, ISBN:1-58113-677-3, Washington, DC, USA, April 28-29, 2003, pp. 11-14.

Willekens P, et al (1994) Algorithm specification in DSP station using data flow language. DSP Applicat. 3(1):8–16.

Wilson R., French R., Wilson C., Amarasinghe S., Anderson J., Tjiang S., Liao S-W., Tseng C-W., Hall M., Lam M. & Hennessy J. (1994) Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIPLAN Notices*, Vol. 28, No. 9, December 2994, pp. 67–70.

Wilson T., Mukherjee N., Garg M. & Banerji1 D. (1995). An ILP Solution for Optimum Scheduling, Module and Register Allocation, and Operation Binding in Datapath Synthesis. *VLSI Design*, Vol. 3, No. 1, pp. 21-36.

Zhong L. & Jha N. (2002) Interconnect-aware high-level synthesis for low power. *Proceedings of the IEEE/ACM Int Conf Comp-Aided Des*, ISBN:0-7803-7607-2, November 2002, pp. 110-117.



Embedded Systems - Theory and Design Methodology
Edited by Dr. Kiyofumi Tanaka

ISBN 978-953-51-0167-3
Hard cover, 430 pages
Publisher InTech
Published online 02, March, 2012
Published in print edition March, 2012

Nowadays, embedded systems - the computer systems that are embedded in various kinds of devices and play an important role of specific control functions, have permitted various aspects of industry. Therefore, we can hardly discuss our life and society from now onwards without referring to embedded systems. For wide-ranging embedded systems to continue their growth, a number of high-quality fundamental and applied researches are indispensable. This book contains 19 excellent chapters and addresses a wide spectrum of research topics on embedded systems, including basic researches, theoretical studies, and practical work. Embedded systems can be made only after fusing miscellaneous technologies together. Various technologies condensed in this book will be helpful to researchers and engineers around the world.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821