Structured Design Methodology For High-Level Design

Polen Kission Hong Ding Ahmed A. Jerraya
System-Level Synthesis Group/TIMA/INPG
46 avenue Félix Viallet, 38031 Grenoble Cédex, France

Abstract - This paper deals with hierarchical design, component re-use and more generally structured design methodology at the behavioral level. This methodology has two main concepts: hierarchy and regularity. Hierarchies are used for decomposing a complex design into more manageable sub-parts. Regularity is aimed at maximizing the re-use of already designed components and sub-systems.

The methodology is illustrated using AMICAL, an architectural synthesis tool, and a design example, namely a Proportional Integral Derivative. The design is decomposed hierarchically into a top controller using a set of sub-systems. AMICAL is then used for the design of one sub-system which is used in a second step as a functional unit, by AMICAL, for the synthesis of the full design.

I. INTRODUCTION

In order to fully exploit the huge integration capabilities provided by the new sub-micro technologies, we need to improve design quality and designer productivity.

This can be achieved by two ways that can be combined: 1) Using a more structured design methodology allowing for hierarchical design and extensive re-use of existing components and sub-systems, and 2) Providing higher level design tool allowing to start from a higher level of abstraction.

This paper presents methods and tools aimed to extend structured design methodologies to the behavioral level.

1.1. Motivation

The structured design methodology for VLSI was introduced first by Mead and Conway [1]. This methodology has two main concepts: hierarchy and regularity [2].

This kind of design methodology has proven its efficiency in several other domains of system design [3]. Structured design methodology has been adopted since the starting of the domain of VLSI design [1]; it has been applied to the physical level with structured physical design tools [2,4], as well as to logic and register transfer level design [5].

1.2. Objectives

The long term goal of this work is the definition of a structured VLSI design methodology acting at the behavioral level. In order to achieve this, we need to solve two closely related problems:

1) How to structure the design in order to allow for hierarchical design and synthesis at the behavioral level.

2) The re-use of existing components (resulting from an early architecture design process) for the design and synthesis at the behavioral level.

This paper deals with hierarchical design and component reuse at the system-level. The architectural synthesis system AMICAL will be used to illustrate these concepts. The next section introduces the concepts underlying the structured design methodology. The application to high-level design shall be illustrated using AMICAL, an architectural synthesis system that allows for hierarchical design and component reuse; AMICAL will be presented in section III. A design example, detailed in section IV, will be used in section V to illustrate hierarchical design within architectural synthesis using AMICAL. Section VI concludes this paper and includes an overview of future work.

II. STRUCTURED DESIGN METHODOLOGY FOR HIGH-LEVEL DESIGN

Structured design methodology allows to handle very complex design with hierarchical approach. Hierarchical design proceeds by partitioning a system into modules. The implementation details of these modules are hidden. Proper partitioning allows independence between the design of the different parts. The decomposition is generally guided by structuring rules aimed to hide local design decisions, such that only the interface of each module is visible.

II.1. Principles

The main steps involved in structured design methodology for high-level design are shown by fig. 1. They include a system-level analysis and partitioning step and a high-level design step.

The system-level analysis and partitioning step starts with the high-level specification; its aim is:

1) To structure the design in order to produce a hierarchical decomposition of the initial specification. This leads to the isolation of sub-systems that will be designed independently as well as that of sub-functions that will be executed on specific functional units.

2) To select the components to be used; these may either be standard existing functional units or specific modules that have to be designed.

These two aspects are inter-related. The hierarchical decomposition may be influenced by the set of already existing components. On the other side, the selection of the components is influenced by the hierarchical decomposition of the design.
II.2. Structured Design Methodology and VLSI

As stated above, lots of work have already been done to provide environment for structured methodologies at the physical or circuit [2], logic and register transfer [5] levels. At each level, different techniques are used for hierarchical decomposition and for component specification and re-use.

At the register transfer level a typical environment will provide methods for modelling generic operators (such as adder, ALU) able to execute basic operations. These operators, acting as black boxes in the netlist (input description), are a hybrid between arithmetic operator and library cell. They are the link between the HDL operator and the components of the final library. Each operator corresponds to a unit that may perform one or several functions. Since the global description is given at the clock cycle level, the execution timing delay of these operations cannot exceed a clock cycle.

In order to apply structured design methodologies at the behavioral level, we need to scale up the concept of the operator in order to allow the use of behavioral components.

II.3. Behavioral Components

A behavioral component is an entity able to execute a set of operations invoked in the behavioral description. The component acts as a black box linking the behavioral and register transfer levels. The operation(s) executed by the behavioral component may be as simple as predefined operations (+,-,*,...) or as complex as input/output operations with handshaking or memory access with complex addressing functions. A component may correspond to a design produced by external tools and methods or to a sub-system resulting from an early design session.

Complex operations can be invoked through procedure and function calls in the behavioral description. Allowing the use of procedures and functions within a HDL is a kind of extension of this HDL. This concept is similar to the concept of system function library in programming languages [6].

II.4. Previous Work

Recent HLS (high-level synthesis) systems [7,8,9] handle multi-function units and multi-cycle operations; only few of these systems tackled the more general concept of re-using complex components at the behavioral level.

More advanced techniques have been used for structuring design in some DSP (digital signal processing) synthesis environments. For DSP systems [10,11], the inherent regularity of the system description is exploited to cluster its behavioral operations.

The system behavior is sometimes "logically partitioned" into processes so that the system can be implemented with multiple controllers. The system synthesis thus converts a system description into a set of algorithmic behavioral descriptions. All synchronisations between the different processes are handshakes [12]. This technique allows for hierarchical design, but not for easy re-use and resource sharing of the sub-modules.

However none of these systems, to our knowledge, provides a general environment that allows for structured methodology design handling true hierarchical design and component re-use at the behavioral level.

III. AMICAL: A DESIGN METHODOLOGY FOR HIERARCHICAL DESIGN

AMICAL is an interactive high-level synthesis system targeted towards control-flow dominated circuits [13]. It starts with two kinds of information: a behavioral specification given in VHDL and an external functional unit library. This corresponds to the second step of the methodology introduced above. The first step, system-level analysis and partitioning, is performed manually.

III.1. The Design-Flow

The AMICAL design-flow is illustrated by fig. 2. The behavioral description may make use of complex sub-systems through procedure and function calls. However for each procedure or function used, the library must include at least one functional unit (FU) able to execute the corresponding operation. During the different steps involved in the high-level synthesis, the functional units are used as black boxes.

The different steps involved in the synthesis process are: scheduling, allocation and architecture generation. During the first step, the scheduler reads in the VHDL description and produces a finite state machine. Each transition corresponds to the execution of a control step under a given condition. All the operations of a given transition may be executed in parallel. An operation may correspond to a standard VHDL operation or to a procedure call.

![Diagram of AMICAL design-flow](attachment:fig2.png)
After scheduling, architectural synthesis starts with two kinds of information, namely the scheduled description and an external functional unit library. The functional unit allocation step associates a functional unit with each operation in the state table. The micro-scheduling is then generated according to the execution scheme for each operation (the synthesis view of the functional unit specification is given by fig. 4). Each operation is decomposed into a set of transfers, which are scheduled into micro-cycles. Each micro-cycle contains a set of parallel transfers that take one basic clock cycle to execute. The datapath synthesis includes the component (functional unit and register) placement and the connection allocation (buses and switches).

III.2. Target Architecture

The target architecture of AMICAL is shown in fig. 3. It is composed of a top controller, a set of functional units and a communication network. These last two constitute the datapath. The communication network is composed of buses and registers. The network is built in order to allow the communication between functional units, and with the external world. The number of buses is fixed according to the parallel transfers required by the architecture.

The top controller sequences the operations executed by the functional units and the communication network. The controller is generated automatically during the synthesis process.

The architecture may include several functional units that may run in parallel. The amount of parallelism is fixed during the synthesis process.

III.3. Modelling Behavioral Components

Behavioral components are called functional units. They allow the use of existing macro-blocks in the behavioral specification. A functional unit may execute standard operations or new customized operations introduced by the user (section II.3). A functional unit can be called from within a behavioral description through procedure and function calls. It can accept and return parameters.

Each functional unit can be specified at four different abstraction levels: the conceptual view, the behavioral view, the implementation view and the high-level synthesis view. Fig. 4 shows these four views for a memory cell that can achieve the 2 operations: read and write.

From the conceptual point of view, the functional unit is an object that can execute one or several operations which may share some data (M). At the behavioral level, the functional unit is described through the operations that can be called from the behavioral description. These may correspond to standard operations, procedures or functions. The behavioral view in fig. 4 is a VHDL package that includes the procedures read and write. The implementation shows an external view of a possible realization of the functional unit.

![Fig. 3. Target architecture](image)

The high-level synthesis view of the functional unit links the behavioral and implementation views. It includes the interface of the functional unit, its call-parameters (corresponding to the operation parameters), the operation set executed by the functional unit as well as the parameter passing protocol for each operation. This protocol is expressed through static clock cycles, each operation needs to have a fixed predictable execution time. In order to overcome this constraint and to enable the use of complex functional units that may execute operations with data-dependent execution time, the methodology used consists in splitting the operation in a set of atomic operations with fixed execution time. The behavioral description will then be written according to the atomic operations introduced. During synthesis, the designer may need to ensure the coherence of the synthesis results through hints within the specifications.

Memory with complex addressing function and specific embedded computation and control can be easily described with this scheme. The addressing functions may be realized by an independent functional unit or may be integrated within the memory unit. In the same way complex I/O units may be used. They are also accessed through function and procedure calls. These may execute complex protocol or data conversion.

According to the characteristics of the application, a set of functional units will be provided before starting the synthesis process. The correspondence between the operations of the behavioral description and the functional units is made during the synthesis process.

IV. A DESIGN EXAMPLE

In order to illustrate hierarchical design and component reuse at the behavioral level we will use a design example: a PID (Proportional Integral Derivative).

IV.1. The PID

A PID controller applies a control function to an analog input and generates an analog output. The input signal
measures a process condition, while the output signal causes an actuator to either fix the process conditions. The PID used in this paper forms part of a speed control system detailed in [14]. The speed control system includes an ALU which performs elementary and logic operations, and memories to store the state variables and coefficients. The speed reference is supplied by the host computer and the PID speed algorithm is executed each time a rotor position change has occurred.

The control loop making call to the PID is shown in fig. 5. The PID is executed by the processor once every n position interrupts. It is assumed that before execution, the motor parameters and control coefficients have already been loaded in the main processor. The algorithm calculates a current reference (\(I_{ref}\)) as a linear expression of the rotational speed error (\(E_k\)), its time integral (\(\int E_k dt\)) and its rate of change (\(dE_k/dt\)).

IV.2. Specifications

The PID algorithm is given by:

\[ I_{ref} = (K_p \cdot E_k) + K_i \cdot \int E_k dt + K_d \cdot dE_k/dt \]

where \(K_p\), \(K_i\), \(K_d\) are constants and \(E_k\) is the error change.

However only the close approximation given as:

\[ I_{ref} = (K_p \cdot E_k) + K_i \cdot \int E_k dt + K_d \cdot dE_k/dt \]

will be developed in order to be synthesized at the behavioral level by AMICAL, for digital implementation. The integral is approximated by a sum of products while the derivative is replaced by successive position interrupts.

IV.3. System-Level Analysis and Partitioning

The goal of this step is to structure the description in order to allow hierarchical description and component re-use. The result is a behavioral description and its corresponding library of functional units. The above PID description makes use of complex operations (\(*\), \(/\)) that are not in the standard library.

The data types required by the speed control algorithm are numeric. Reals are used to represent the values of state variables while integers are used for time intervals and clock frequency constants. The decision was made to use fixed-point arithmetic for numerical calculations on reals. Two's complement representation are used for both integer and fixed-point real numbers. The word length is 32 bits. Fixed-point reals have a 12-bit integer part and a 20-bit binary point part.

As the PID to be designed has no severe timing constraint, the multiplication and division operators will be implemented by sequential procedures using basic operators from the library (+, -, shift). In order to share the basic operators, we decided to gather all the fixed-point operators (*, /, +, -) within a fixed-point unit that will be synthesized by AMICAL and re-used in order to build the PID.

V. Hierarchical Design

For the synthesis of the speed control design example, we will proceed in a hierarchical way. The fixed-point unit will be synthesized first and the results will be used as a functional unit for the design of the whole PID processor.

V.1. Design of the Fixed-Point Unit as a Behavioral Component

As stated above, the fixed-point unit executes addition, subtraction, multiplication as well as division. The multiplication algorithm selected is an add-and-shift based one, while the division (reciprocal as the numerator is equal to one) makes use of the restoring division algorithm [17].

The goal is to design the fixed-point unit as a functional unit. As explained in fig. 4, several views of the component should be defined. All these views are inter-related. An important step when defining new functional units is the definition of the behavioral interface and of the operations executed by the functional unit.

The execution time for division and multiplication are data-dependent and thus not fixed. The fixed-point unit should then provide a protocol in order to allow the use of these operations through actions with fixed execution time. However addition and subtraction need an exact number of clock cycles to execute (1 clock cycle). When using these algorithms, the multiplication takes 126/127 (according to the parity) clock cycles for execution. The execution time for the reciprocal can vary between 82 and 102 according to the input values, assuming that the fixed-point unit is synthesized with limited resources (1 shifter and 1 ALU (+ and -)).

We decided to use a two-step protocol for the execution of both the division and the multiplication: the first step is the operation call with the corresponding parameters. The second step involves the "waitresult" operation which enables the result to be recovered. Both operation call and "waitresult" are described as one clock-cycle operations in the high-level description of the fixed-point unit.

The structure of the fixed-point unit can be summarized by the VHDL description of fig. 6.

Two different views of the interface of the fixed-point unit are shown in fig. 7. The behavioral view includes 5 operations, making use of 2 input (A and B) and 2 output parameters (Z and done). The protocol for parameter exchange is detailed in the synthesis view.

The use of the fixed-point unit as a component of the functional unit library results in expanding the multiplication or division call according to the protocol of the fixed-point unit. The lines:

\[ \text{Dek} := \text{Ek} \cdot \text{Ek}_1; \]
\[ \text{Iref} := \text{Kp} \cdot \text{Ek}; \]

of the initial PID description are thus written as the sequence:

\[ \text{Dek} := \text{Ek} \cdot \text{Ek}_1; \]
\[ \text{multiplication_call} (\text{Kp}, \text{Ek}) \]
\[ \text{waitresult} (\text{Iref}, \text{done}) \]
\[ \text{while} (\text{done} = 0) \text{loop} \]
\[ \text{waitresult} (\text{Iref}, \text{done}) \]
\[ \text{end loop} \]

The protocol for the reciprocal operation is similar to the multiplication operation.

(*) The controller device performs the 3 functions: speed control, current control and communication with a host computer. The host interface communicates control coefficients and internal state variable between the controller and the host computer.

Fig. 5. Control system
entity fixedpointunit is
  port (A, B : in integer; -- input values
       com : in integer; -- operation asked
       sel : in bit; -- enable signal
       Z : out integer; -- output value
       done : out bit); -- output validation signal
end fixedpointunit;
architecture behavior of fixedpointunit is begin
  process
    variable tmp : integer; -- result buffer
    variable input1, input2 : integer; -- input value buffers
    procedure mul(A,B:in integer) is
      begin -- shift and add algorithm; tmp := A * B;
      end mul;
    procedure rep(A:in integer)is
      begin -- restoring division algorithm; tmp := 1/A;
      end rep;
    begin
      wait until sel='1';
      case com is
        when 1 => -- reciprocal_call
          done<=0'; input1:=A; rep(input1);
        when 2 => -- multiplication_call
          done<=0'; input1:=A; input2:=B; mul(input1,input2);
        when 3 => Z <= A + B; -- "+
        when 4 => Z <= A - B; -- "-
        when 5 => Z <= tmp; done <= '1'; -- waitresult
        end case;
    end process;
end behavior;

Fig. 6. VHDL description of the fixed-point unit

V.2. The Design Process

The full design process is shown in fig. 8. The AMICAL process is run twice; once for the fixed-point unit synthesis and a second time for the PID synthesis.

The library of functional units used for the fixed-point unit synthesis is made up of an ALU which executes either addition or subtraction and a shifter (undertaking right shift as well as left shift operations).

Fig. 7. Fixed-point unit interface

The architectural synthesis of both fixed-point unit and PID are realized by AMICAL. The datapath obtained, by the high-level synthesis with AMICAL, for the fixed-point unit is shown in fig. 9(a). The final controller is made up of 28 states. The results of the synthesis of the fixed-point unit will be used twice:

1) To create the corresponding behavioral component that will be used for the synthesis of the PID.
2) During the logic synthesis of the PID as the required structural description of the functional unit: fixed-point unit.

The functional unit library used as input for the PID synthesis includes the fixed-point unit developed above and an ALU that may execute addition and subtraction. The behavioral description of the PID is written according to the protocol used by the fixed-point unit.

The synthesis of the PID produced an architecture where the controller is a 22-state and 33-transition FSM. The datapath obtained after some interactive architectural transformations is made up of 3 functional units and 3 buses. The PID datapath is represented in fig. 9(b). The component FU_2 is an instance of the fixed-point unit compiled previously.

Fig. 10 outlines the synthesis results obtained when running the whole process in the automatic mode. The comparison table gives the complexity of the different sub-systems composing the PID. The full design stands on 13 mm square when mapped onto a 0.8 CMOS technology using commercial logic synthesis and place and route tools.

<table>
<thead>
<tr>
<th>DESIGN</th>
<th>No of transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportional Integral Derivative</td>
<td>48 962</td>
</tr>
<tr>
<td>PID - Data Path</td>
<td>47 492</td>
</tr>
<tr>
<td>PID - Controller</td>
<td>1 470</td>
</tr>
<tr>
<td>Fixed-Point Unit</td>
<td>21 248</td>
</tr>
<tr>
<td>FPU - Data Path</td>
<td>19 366</td>
</tr>
<tr>
<td>FPU - Controller</td>
<td>1 882</td>
</tr>
</tbody>
</table>

Fig. 10. Synthesis results
VI. CONCLUSIONS AND FUTURE WORK

This paper dealt with structuring design in order to allow for hierarchical design and synthesis at the behavioral level. The use of behavioral design to build more complex design corresponds to the re-use of existing components for the design and synthesis at the behavioral level.

An example has been used to illustrate the structured design methodology using high-level synthesis. The architectural synthesis AMICAL has been used for the design of a fixed-point unit which has been, itself, used in a second step as a functional unit, by AMICAL, for the synthesis of a full PID controller.

The scheme detailed above is powerful as it allows for hierarchical design enabling the use of complex sub-systems as functional units in the library during architectural synthesis.

Future works are mainly targeted towards the automatization of the system-level analysis and partitioning step. The main issue is the definition of a cost function allowing to decide which part of the initial specification should be transformed into specific operation and the definition of the set of specific functional unit that will be used for the synthesis of a given application. The latter include clustering operations in functional units and the design of the corresponding sub-systems.

ACKNOWLEDGMENT

Thanks to Professor D.J. Kinniment, Department of Electrical and Electronic Engineering of the University of Newcastle upon Tyne for his help for the specifications and design of the PID.

REFERENCES