REPORT
ON

HARDWARE SYNTHESIS

Instructors:
Prof. M.Balakrishnan
Prof. Anshul kumar

SUBMITTED BY
A.Satyanarayana(2001vls025)
R.RamanjaneyaReddy(2001vls020)

ABSTRACT

In this we describe the Global Variable Localization and Transformation for Hardware Synthesis from High-Level Programming Language Description. And also the synthesis of hardware from object oriented specifications is presented. This approach utilizes the *e* language that has been proven to be highly efficient for the verification of hardware. The *e* language is similar to Java and provides additional constructs for specification and verification of hardware. We describe an automated design flow for the synthesis of object oriented descriptions that tightly integrates simulation based verification.

INTRODUCTION:

There is a growing gap between the number of gates that can be implemented on a single chip and the number of gates that can be designed by one person in one day. Additionally, short product life cycles, time to market, and changing standards require shorter design times. Therefore, new design methodologies must be applied. By specifying on a higher level of abstraction the productivity can be increased enormously. Commercial tools for synthesizing algorithmic descriptions are available already and gain more and more acceptance. Newer approaches raise the abstraction level even more by synthesizing object oriented descriptions. The object oriented paradigm can be applied in order to cope with the complexity of system-level designs.

System architects and system level designers typically use a C/C++ based development environment to specify and analyze the systems they are designing. Once the exact system behavior has been fixed, they hand-off the executable specification of the system to hardware designers and software developers. The hardware designer then translates the executable specification written in C/C++ into a hardware description language (HDL) and continues adding more detail until the HDL code can be synthesized into a gate-level netlist for IC implementation. One of the problems with this methodology is that rewriting the C/C++ code into an equivalent HDL description is both time consuming and error prone. It usually includes rewriting the original testbenches written in C/C++ into an HDL, which greatly adds to the task. In addition, each time a trade-off that might affect the overall behavior of the system is tried during the hardware design process, the different alternatives must be analyzed in the context of the C/C++ executable specification, and this requires the repeated involvement of the system architect or system level designer.

What's needed is a smooth and reliable methodology that allows the hardware designer to continue refining the C/C++ executable specification created by the system architect or system level designer into a form that is acceptable as input for hardware synthesis, without the need to translate the C/C++ code into an HDL. Eliminating the translation step enables the reuse of the original C/C++ testbench, decreases verification time and ensures compliance with the original specification.

CoCentric SystemC Compiler is the synthesis bridge between the system architect modeling and verifying SoC designs in C/C++ and hardware designers implementing those designs in silicon.

Using SystemC Compiler, hardware designers can implement SystemC code from a behavioral level of abstraction, which provides for quick and efficient modeling, small

code size, fast simulation/verification and the benefits of optimization at the architectural level, which can be far greater than those available at the RT-level.

Since hardware description languages like VHDL and Verilog are more suitable for RT level design than for higher levels of abstraction, software languages like C/C++ or Java are recently deployed for synthesis. These languages are already widely used for specifications that can be executed. Furthermore, hardware/software co-design can be simplified by using a single language for both domains.Verification typically consumes over half of the design effort. The languages currently used for synthesis provide only insuffi- cient support for verification. Therefore languages like Verisity's e (www.verisity.com) or Synopsys' Vera (www. open-vera.com) hardware verification language are deployed within a dedicated environment to improve the verification task. The elanguage is similar to Java enhanced by various verification and hardware related constructs. e provides all basic object oriented constructs, supports *multi-threading*, and it can be executed in the SpecmanTM environment. Therefore it is also well suited for specification. Synthesis on the system-level requires support for multi-threaded descriptions. Due to the complexity of such designs, we want to support verification appropriately in our approach. The powerful verification features of eand its capability for object oriented specification, encouraged us to investigate the applicability of the e language for synthesis. By making the synthesis of edescriptions possible, we allow to use one single language for specification, verification, and synthesis.

The remainder of the paper is organized as follows: In Section 2, we will present an overview of related work. Section 3 describes object oriented synthesis and details of the synthesis from e descriptions. In Section 4, we present verification within the Specman environment and the integration of this environment in our design flow. Section 5 presents examples and discusses the results. Section 6 concludes this paper and gives an outlook on further work.

2. PREVIOUSWORK

In the past, various approaches have been made to specify hardware and hardware/software systems on the basis of objects. Object oriented VHDL enhances VHDL with object oriented concepts. SystemC is based on C++ class libraries and is intended for design at the system/algorithmic and register-transfer levels. Recently, successful efforts have been made to specify and to synthesize from Java. For validation, simulation and formal verification have a long tradition and various commercial tools are provided by major EDA companies, e. g., Cadence, Mentor Graphics, and Synopsys.

3. OBJECT ORIENTED SYNTHESIS

The object oriented design paradigm has already gained broad acceptance in software engineering for the design of large systems. Object oriented synthesis allows hardware modeling on an abstraction level above the behavioral level. Designers can express their ideas in a very natural way by thinking of classes rather than of data and procedures. Encapsulating things that belong together andhaving data and behavior within one entity also simplifies re-use.

We developed an object oriented analysis system (OOAS) that allows to synthesize object oriented hardware descriptions. Our OOAS can be used to read in descriptions

written in different languages. Currently we support Java, SystemC, and eas input languages. In this paper we describe the synthesis from *e*.

There is already a variety of good synthesis tools on the market. These tools provide an automated design flow for RT and algorithmic descriptions, but they do not support object oriented concepts like object references, inheritance, or polymorphism. We utilize the algorithms deployed by these tools for the synthesis of object oriented descriptions by transforming such descriptions into equivalent descriptions on a lower level of abstraction that can be handled by commercial tools. The formats that we generate for those descriptions are VHDL, Verilog, and SystemC (figure 1). For the sake of simplicity this paper refers only to the output of Verilog.

3.1 Synthesis from e

Using one single object oriented language for specification, verification, and synthesis simplifies the design process. There is a variety of languages that could be used as such an unified language, each of them having advantages and disadvantages. Most of the languages do not support verification adequately. Verification of multi-threaded descriptions is mandatory for hardware design and particularly challenging. The e language is therefore especially suited as an unified language for hardware design. Today eis not only used for verifying hardware descriptions written in VHDL or Verilog, it is already used for writing a *golden model* of the hardware in an object oriented high-level language. This model is then manually translated into a synthesizable HDL description. The advantage of e over software languages like C/C++ or Java is that hardware related constructs like bit accurate data types, clocks, etc. are part of the language.

For the transformation from an object oriented description in e into an equivalent description in Verilog, the meta data that describe the object oriented structure is generated. The number and structure of objects have to be static in the realization in hardware. This should be considered in the description and is checked by preallocation during the analysis. Objects have to be allocated for variables and parameters of methods which reference sets of objects. The identification of such sets of static references is done during an alias analysis, which is part of the OOAS.

In e concurrent threads of control are described by so called *Time Consuming Methods* (TCMs). TCMs are similar to Threads in Java. TCMs are triggered by events. Events can be defined by temporal expressions that are part of the elanguage. TCMs have to be handled separately from the regular methods (non-TCMs) during the transformation

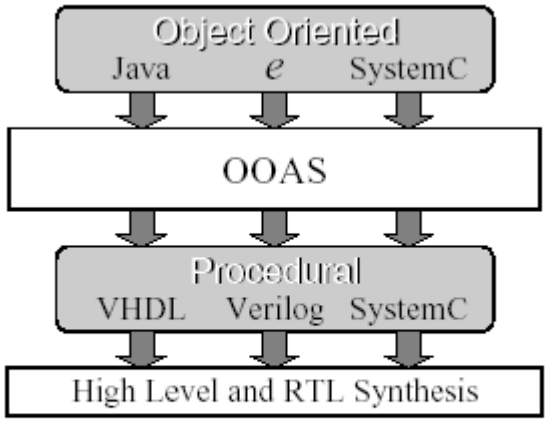


Figure 1: Languages supported by OOAS

process. Conflicts resulting from concurrent access to variables by different TCMs must be detected by a concurrency analysis and then be resolved. We now describe the control data flow analysis that is performed for the sequential case and the concurrency analysis that is performed for the parallel case.

3.2 Control Data Flow Analysis

The control data flow analysis applies techniques known from software compiler design [9] to the synthesis of hardware. After the lexical and semantical analysis done by the scanner and parser for e, a static control and data flow analysis is performed on the set of syntax trees for each eobject. The result of the analysis is a control flow graph (CFG) where the data flow information is stored in a scope table within each node of the graph. Because of the combination of data and control flow information, no separate data flow graph has to be generated and the analysis traverses the syntax tree only once. The CFG has two different types of nodes. The control flow nodes divide the CFG in multiple sub-trees if a node has multiple children. This is the case for nodes representing, e.g., *while* loops or *if* statements. The second type of nodes in the CFG are the data flow nodes, which do not change the control flow, like arithmetic operations or assignments.

The main problem of the transformation is the usage of references. Deciding which object is accessed when a method is called on a variable can only be done at runtime. Objects may have several references, or aliases at the same time, so it is hard to tell which statements affect which object. The analysis determines a set of possible objects for each variable within the scope of a statement. Each node of the CFG has one scope table, where all variables in the scope of the node are stored together with a set of objects that may be referenced by this variable. The scope tables are built together with the whole CFG. When a new node is created, the scope table of the parent node is cloned and the set of references ('reference set') of each variable changed by the statement is updated.

A reference set Rs(a) is the maximal set of all references on objects, which can be hidden by the alias a after the execution of statement s under consideration of the type of the variable and the preceding control flow. All reference sets Rs(a) of variables accessible in a CFG node n, build the scope table S(n). Figure 2 shows an example of a hardware component written in e. Classes are described using the keyword *struct*. A struct declares data fields and methods. Inheritance is supported by the keyword *like*. The

init method is similar to the constructor in Java. Instances of a struct are made by using the keyword *new*. A detailed description of the elanguage can be found in [10]. The constructs that we use for the specification of hardware components are described in [11]. The analysis for the TCM *runHWO* of the example code results in the CFG shown in figure 3. At node 5 in the CFG, the scope table includes three reference sets R5(x), R5(y) and R5(z). Each reference set has been initialized in the init method, where three objects of the struct S have been instantiated. The objects *S 1*, *S 2* and *S 3* are labeled with subsequent numbers. For each type of statement, a different algorithm is implemented to update the scope table of a node in the CFG. The algorithm for loop statements, used for the determination of the scope table for node number 7 in figure 3 is described in algorithm1. Table 1 shows the evolution of the reference sets for each iteration of the algorithm. The body of the *while* loop has to be reanalyzed four times to get the final reference sets for the *while* loop node (without determining the exact number of iterations during execution). The analysis terminates because there are no changes in the reference sets of step three and four. The merged reference sets in the last column of table 1 are the resulting sets contained in the scope table *S(7)* of node 7.

```
struct Test like HWO {
    x : S; y : S; z : S;

    init() is {
        x = new S;
        y = new S;
        z = new S;
    };

    runHWO() @clk is {
        var i : uint;
        i = 0;
        while (i < 10)
        {
            z = x;
            x = y;
            i = i + 1;
        };
        z.foo();
    };
};
```

```
struct S {
    foo() is {
        ...
    };
};
```

Figure 2: Struct *Test* and struct *S*

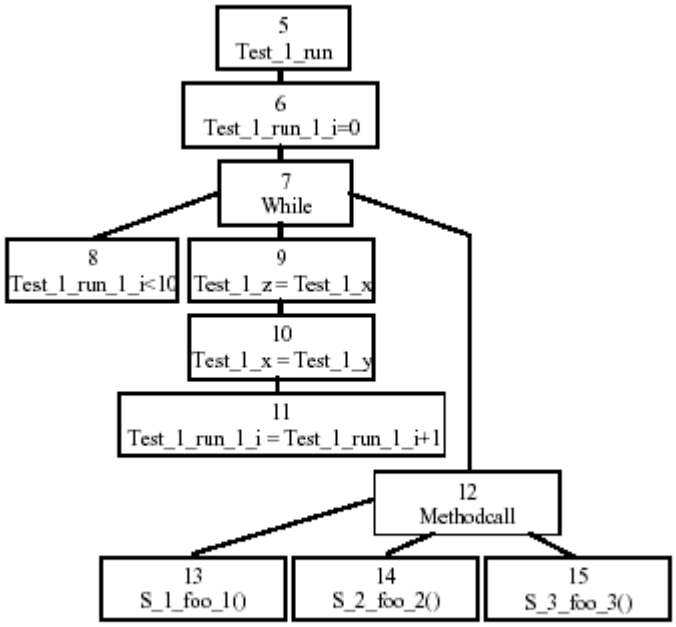


Figure 3: CFG for the TCM *runHWO*

Input: CFG node n, statement s of n, scope table $S(p)$ where p is the predecessor node of n

Output: Updated scope table $S(n)$

begin
  $S(n) \leftarrow$ copy of $S(p)$
  $S(tmp) \leftarrow$ copy of $S(p)$
  repeat
    $S(n) \leftarrow S(n) \cup S(tmp)$
    $S(tmp) \leftarrow$ analyze the block statements of the loop
  until $S(n) = S(tmp)$
end

Algorithm 1: Analysis of loops

Table 1: Reference sets for the while loop

	1.	2.	3.	4.	result
$R_7(x)$	{S_1}	{S_2}	{S_2}	{S_2}	{S_1, S_2}
$R_7(y)$	{S_2}	{S_2}	{S_2}	{S_2}	{S_2}
$R_7(z)$	{S_3}	{S_1}	{S_2}	{S_2}	{S_3, S_1, S_2}

The scope table *S(7)* is the input for the analysis of node 12. The method call *z.foo()* uses variable z where R12(z) contains the objects *S 3*, *S 1* and *S 2*. On which object the method *foo()* will be called depends on how often the *while* loop has been iterated at runtime. The analysis splits the CFG into three sub-trees, one for each object that may be referenced by the variable z.

For each method call, a new node in the CFG is created. The body of the method builds a sub-tree of the node. When the method is a TCM, the new CFG node builds an independent CFG with the method call node as root node. A copy of the reference sets of the actual scope is the initial scope of the new CFG. The analysis always terminates because of the constant number of objects in the whole system. The number of objects in a reference set can not exceed the maximum number of objects instantiated in the whole system. The constant number of objects instantiated in the eprogram is guaranteed since the instantiation of objects in loops is only allowed if the number of iterations can be determined at compile time. The instantiation of objects is not allowed within cyclic or recursive method calls.

When the analysis terminates, the CFG is used as input for the concurrency analysis, which is described in the next section.

3.3 Concurrency Analysis

During the concurrency analysis, the reference sets and the CFGs of the data and control flow analysis are used for creating a set of variables which are accessed by different TCMs. A variable can be accessed by a TCM in two different ways:

_ **write access:** The variable is on the left hand side of an assignment and becomes an alias for another object.

_ **read access:** The variable is on the right hand side of an assignment or is passed to another method as parameter or a method is called on the variable. If a node in the CFG accesses a variable the following cases have to be handled by the analysis:

1. There is no other TCM which reads or writes the variable. The variable is a normal variable that does not require any special treatment.
2. At most one TCM has write access to the variable and multiple TCMs have read access. This variable has to be declared in a global scope to enable multiple TCMs to access the referenced object.
3. More than one TCM writes to the variable. This case is reported to the user of the system. An arbiter to resolve the access must then be inserted. This arbiter is an adapted variation of the arbiter described in [12].

For the variables which are accessed by multiple TCMs (case 2) the CFG is extended by a set of global variables called *global*. To build up this set, the CFG is traversed to build a set of variables read by the TCM t called *read*t and a set of variables written by t, called *write*t. When a read access occurs in a node of the TCM t, the variable v is added to the set *read*t. If v is in the set *write*m or *read*m, where m is an already analyzed TCM, v becomes element of *global*. In case of a write access, v is added to the set *write*t. If v is already element of the set *write*m an error state is reached. When v is member of *read*m v becomes element of the set *global*. The actual TCMs are stored on a stack because a TCM can start another TCM. The actual TCM where a read or write access is executed, is always the top element of a stack called *tcmStack*. The described method that creates the global set of shared variables is shown in algorithm 2.

```
Input: node {root node of the CFG}
Output: global {set of global references}
begin
   tcmStack ← empty {stack with actual analyzed
TCMs}
   tcmSet ← empty {set of already analyzed TCMs}
   repeat
       switch node
       case start of TCM t
         tcmSet ← tcmSet ∪ tcmStack.top
         tcmStack.push(t)
      case end of TCM t
         tcmSet ← tcmSet ∪ tcmStack.pop
      case statement reads variable v
         TCM t ← tcmStack.top
         read_t ← read_t ∪ v
         for all TCM m in tcmSet do
            if v ∈ read_m or write_m then
               global ← global ∪ v
            end if
         end for
      case statement writes variable v
         TCM t ← tcmStack.top
         write_t ← write_t ∪ v
         for all TCM m in tcmSet do
            if v ∈ read_m then
               global ← global ∪ v
            end if
            if v ∈ write_m then
               report write access to the user
            end if
         end for
      end switch
      node ← node.next
    until node is empty
end
```

Algorithm 2: Concurrency analysis

After the complete concurrency analysis, the set *global* is added to the root node of the CFG. The CFG is traversed by the Verilog code generator that uses the scope information to generate the appropriate Verilog constructs. The TCMs that are used in e to specify multiple threads of control are transformed into Verilog *always* blocks. Each *always* block is initially idle and starts its actual execution at the same time as the corresponding TCM is started in the especification. In order to obtain this behavior and to ensure a correct reset behavior of the circuit, a skeleton is built implicitly around the actual description. Within that skeleton, the Verilog statements are generated according to the information in the CFG.

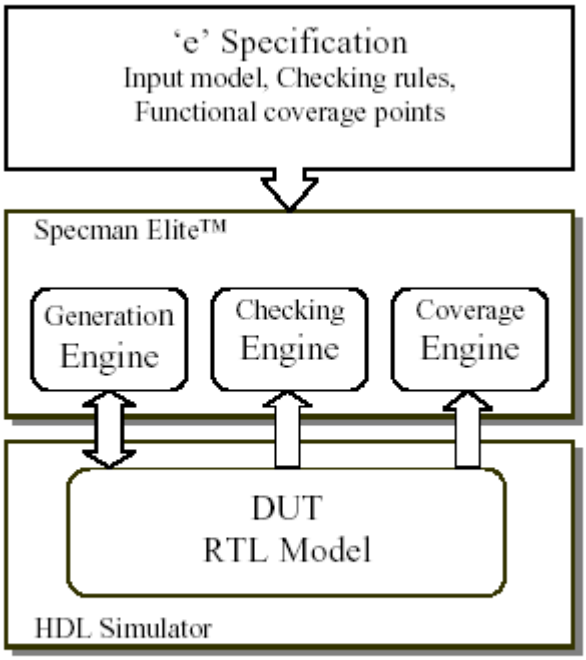


Figure 4: A typical e driven testbench

4. VERIFICATIONWITH e AND SPECMAN

Closely related to e is the verification environment Specman, both developed by Verisity Design. Simulation based verification requires the introduction of stimuli to the Device Under Test (DUT) being simulated, as well as the collection of DUT responses for the purpose of checking and coverage analysis (figure 4). Specman and the e language provide powerful support for these verification tasks. The supported verification methodology can be applied to DUTs implemented in e, those implemented in HDLs such as Verilog and VHDL as well as numerous other modeling languages.

We use this verification environment in three different phases of the design process depicted in figure 5. First, the design flow starts with the specification of the hardware module in e. This system specification may contain algorithmic descriptions as well as descriptions at the register-transfer level. The correctness of the specification can be verified by executing the ecode in the Specman environment. Second, existing Verilog

IP can be included for verification by attaching a simulator. The results of this first step in the verification process can be used to refine the test bench, also written in e, around the hardware module. Third, the especification of the hardware module is processed by our object oriented analysis system. Depending on the description style used in the specification (algorithmic or register-transfer level) the generated code is either behavioral Verilog, suitable for high-level synthesis with Synopsys Behavioral CompilerTM/Design CompilerTM, or a Verilog description at the register-transfer level that can be synthesized by Design Compiler. Additionally, the OOAS generates Verilog code that comprises several enhancements for simulation. The generated Verilog can be verified by attaching a simulator the same way as for the simulation of Verilog IP. The same environment is used for the verification of the results of the commercial tools.
All steps of the verification task are supported by Specman's sophisticated verification methodologies and the test bench can be re-used easily.

4.1 Input Modeling and Generation

In e, input stimuli are modeled as a hierarchy of objects with inter relating constraints. By defining object types the user specifies the universe of data elements or an alphabet for an input sequence. Constraints can both remove parts of the alphabet and restrict the composition of members of the alphabet into sequences. Constraints may depend on the state of the system.
In contrast to deterministic test inputs, the e input model normally contains degrees of freedom. The Specman test generation engine uncovers these degrees of freedom and creates input sequences using a controlled pseudo-random process. Constraints are

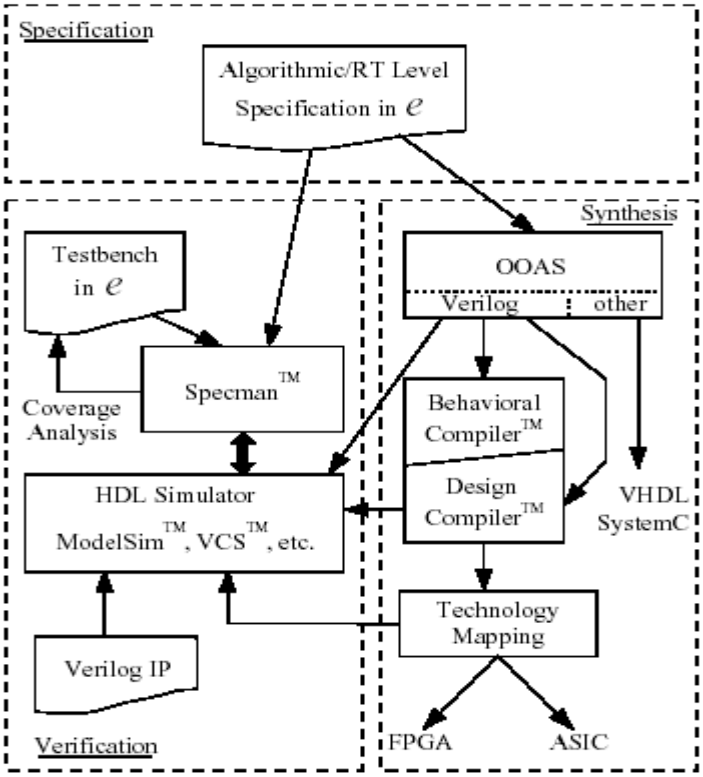


Figure 5: Design flow

conjunctive by nature, hence one can direct the generated input sequence by addition of constraints. These additions can be made per feature to be tested, or as a way to avoid areas of known defects and work in progress.

4.2 Driving and Checking

A directed random input generation methodology requires automated checking, since the input model does not predict a unique response. The e language offers three major features in support of driving and checking: Events, TCMs, and a complete temporal language. The temporal language uses events and state formulae as atomic entities. Temporal and logical operators are used for expressing protocol rules the DUT must adhere to. Specman features a temporal engine that interprets temporal expressions as runtime checkers.

4.3 Functional Coverage

The generation process ensures non-zero probability for any legal input sequence. However, given the huge input space and state space of modern devices, a practical approach would control the distribution of input sequences in view of the accumulated coverage.

The e language provides a way to define functional coverage metrics. Functional coverage points are user defined combinations of states, or sequences of states that have some architectural or micro-architectural significance. Because of its sequential nature, functional coverage is a more rigorous metric than code coverage. The accumulated functional coverage and its breakdown to architectural and micro-architectural features provide status information about the verification effort. This information is used for steering the process and to eventually certify that the DUT has a high probability of being functional.

5. CONCLUSIONS AND FUTUREWORK

Our approach provides an automated design flow for synthesis of object oriented specifications. Verification is tightly integrated in the design flow and all the design tasks can be carried out using the object oriented language e.

At present, no extensions have been made to the e language in order to distinguish hardware and software objects or to define the interfaces between them. Instead inheritance was used to augment object semantics as needed. We consider adding proper constructs to the e language to simplify the specification task for the designer.

So far we have deployed our synthesis system to improve and speed up the implementation flow. When applied to the verification flow, some components of the test environment may be subject to synthesis which may facilitate test bench acceleration and post silicon validation.

6.REFERENCES

1. http://sonata.kaist.ac.kr/Research/seminar/ppt/jylee20000408.ppt

2.Object Oriented Hardware Synthesis and Verification

T. Kuhn, T. Oppold,C. Schulz-Key,M. Winterholer,W. Rosenstiel, Univ. of Tuebingen, Sand 13. 72076 Tuebingen, Germany oase,rosen@informatik.uni-tuebingen.de

M. Edwards,Cisco Systems, Inc.7025 Kit Creek Road. RTP, NC 27709, jme@cisco.com

Y. Kashai. Verisity Design, Inc.2041 Landings Drive,Mountain View, CA 94043 yaron@verisity.com.

Application driven DSP Hardware Synthesis

ABSTRACT

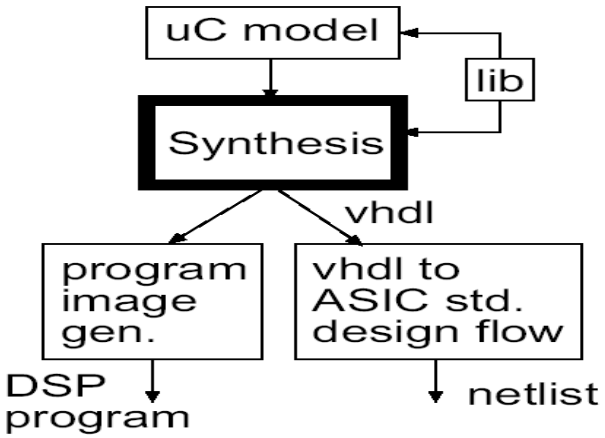
In this we present a synthesis tool aimed for application specific DSP processors. The purpose with the presented work has been to develop a tool where it is easy for a designer to try different approaches in order to achieve a well balanced architecture. In this we discuss the algorithms in the tool and show, by example, the intended way of operation.

1. INTRODUCTION

DSP processing in modern communication systems are today normally carried out either in programmable DSP processors or dedicated ASICs with little or no programmability. An ASIC solution offer high performance in terms of processing power and power consumption. The ASIC is targeted against one or a few tasks and can therefore be optimized to meet the desired computational requirements and memory bandwidth etc. We focus on applications that require a high degree of flexibility, but for a given application. Examples of applications include FFT processing, Viterbi and Reed-Solomon decoding. Each one of these examples exist in various variants, working with different block sizes etc. For these applications we want to be able to evaluate various instruction sets as well as different degrees of parallelism in a hardware-software co-design process. Instead of a automatic synthesis tool we need an interactive environment that gives the designer the opportunity to describe different architectures in an efficient way and then fast get the resulting netlist. In this paper we show a solution that makes it possible to synthesize a DSP processor from an executable cycle true, model of a processor and the application. This is done using a synthesis tool where most of the design choices are done by the designer.

2. SYNTHESIS FRAMEWORK

The synthesis tool has been designed to fit into the MDSP design flow which is a design methodology that allows the designer to use a C-like description language called μC for defining the DSP. The synthesis tool takes as input the cycle true μC model and gives as output an architecture that is able to execute the algorithms described in the model. The program memory image is then created using other tools in the framework. The generated architecture is later on passed to a VHDL compiler in order to generate a netlist suitable for the layout tool, Figure 1.



3. THE DSP SYNTHESIS TOOL

The synthesis tool takes as input the simulation model written in µC that contains all information about the desired instruction set as well as the desired parallelism. What not are given in the description is how many and which types of ALUs that are wanted. The description also lack explicit information about how the resources should be connected together.

3.1. Target Architecture

The DSP synthesis tool has a target architecture consisting of a number of ALUs, register files, memories, I/Os, busses, a control unit and a program memory, Figure 2. One difference between a general purpose DSP architecture and a dedicated one is how registers are used. A general purpose DSP have large register banks with general purpose register files and ALUs that can be used for everything from addressing to normal data processing. In a dedicated architecture it is possible to have dedicated registers and ALUs for addressing and different types of data. Therefore we have chosen to target an architecture with dedicated resources for different types of tasks in the DSP.

3.2. Synthesis Library

The synthesis tool maps the µC description to structural VHDL containing primitives found in a synthesis library. The library consists of registers, memories,

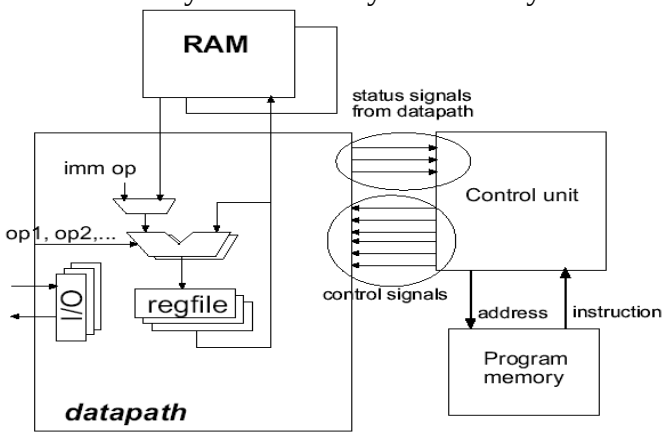


Figure 2. Target architecture.

various types of I/O blocks and arithmetic logic units (ALU). The I/Os are either plain registers, an asynchronous port that communicates using handshaking, or user defined I/O. The RAM have separate read and write buses which is common in many on-chip RAMs. There is also a control unit available in the library that support a set of instructions such as for instance jump, conditional jump and sequential execution. All control signals needed in the datapath is taken directly from the control unit. The instruction decoding is supposed to be made inside the control unit and is not done by the synthesis tool. Any kind of functional block can easily be included in the synthesis library by describing the block in VHDL and adding a description where the supported instructions are listed.

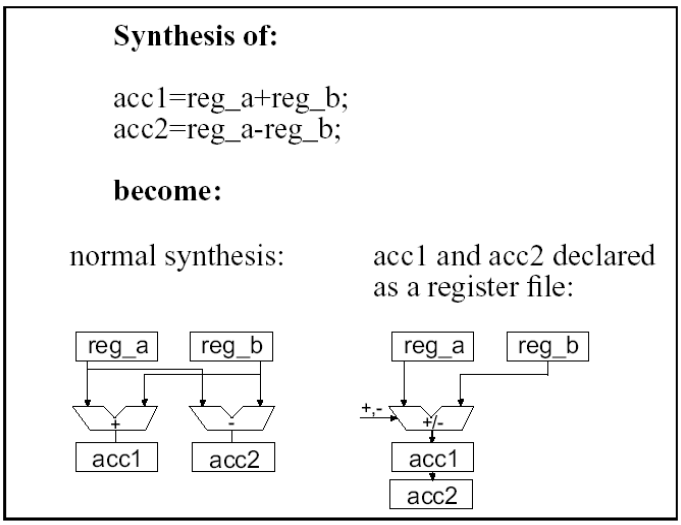


Figure 3. Synthesis of ALUs for register and register files.

3.3. Synthesis

The synthesis process is divided into a number of stages that analyze the resource need and then creates an architecture that is matched against the algorithms to implement. In the first stage the µC model is analyzed to find out which hardware that are explicitly declared, i.e. all memory and registers. Secondly, the tools analyze which operations that are made in the program flow. The target and destination registers for each instruction is also stored. In the third stage the operations are mapped to ALUs. This can basically be done in two ways realizing either a minimal architecture with as few ALUs as possible, or a maximal architecture where little or no resource sharing is made. A minimal architecture will require ALUs supporting many instructions, while a maximal architecture gives many, but simple ALUs. This tool creates an architecture where each destination register in the architecture gets a dedicated ALU. The ALU is chosen from the synthesis library by finding the ALU that supports all operations that has the given register as target register. Hence, the resulting architecture will be an architecture with one ALU attached to each register. In order to optimize the architecture the number of ALUs must be reduced. Typically, each ALU should have a number of registers attached to it, i.e. a register file. Therefore it is possible to define register files, telling the synthesis tool to attach the same ALU to each register in the register file, Figure 3. Normally the selected ALU is the ALU that most closely matches the needed instruction set, however it is also possible to control the tool such as the most power or area efficient or even as test ALU is chosen. This is accomplished by storing relative power, area and speed weights in the synthesis library. Finally the interconnections are created by analyzing the model in order to see which blocks that has to be able to communicate. Only necessary communication paths are created. The communication paths only contains multiplexers and wires, i.e. no tri-state buses are used.

3.4. User Control

The degree of inter activity during the design process is intended to be high. The main goal has been to provide a tool that makes it easy for the designer to get the intended architecture. Therefore the tool contains little inherent intelligence, but is easy to

control by flags fed to the synthesis tool, by modifying the synthesis library and/or rewriting the model. The type of ALUs that are chosen for a given register may not become the one the designer wants to have. The type of ALU can therefore be explicitly assigned using a configuration file as input to the synthesis tool. In this way it is possible to add a more powerful ALU that supports more instructions than required by the present application.

4. EXAMPLE

In this section an example how to use our synthesis tool in the design flow is given. In Figure 4 an example of μC code of a 32 tap FIR filter is given. Passing this description through the synthesis tool without any ALUs declared gives the architecture shown in Figure 5 (control unit excluded). The tool creates an architecture that can execute the given task, and nothing more. In order to achieve an implementation that are easier to reuse, for instance if we want to support any filter length up to 32 taps, the instruction set has to be extended. This has to be made such as it become possible to

```
1: // Declaration part
2: MDSP fir
3: {
4:
5:   INPUT inp(14, PARALLEL);// input port, 14 bits
6:   OUTPUT outp(14, PARALLEL); // output port, 14 bits
7:   REG acc(30), i(6), ca(5), da(5); // different registers
8:   RAM d(32,16);   // RAM with 32 16 bit words
9:   ROM c(32,16, "rom.data"); // ROM
10:
11:   PROCEDURE compfir ();// procedure declaration
12: }
13:
14: // Code part
15:
16: PROCEDURE main()
17: {
18:   for(;;){                          // loop forever
19:     do {;} while(!inpF) ; // While no input on the input
20:                            //  port inp do nothing
21:     inpF=0, d[da]=inp; // Reset input by setting inpF=0,
22:                           //  store inp in RAM. "," means that
23:                           // this is made in parallel
24:
25: compfir();         // call procedure compfir
26:     outp=acc;     // place the value of acc on outp port
27:   }
28: }
29: PROCEDURE compfir() // compute fir
30: {
31:   acc=0,ca=0;
32:
33: i=30;
34: do {
35:       acc+=d[da++]*c[ca++],
36:           i--;
37:     } while (i>0)
38:   acc+=d[da]*c[ca++];
39:   return;
```

Figure 4. μC model of 32 tap FIR filter.

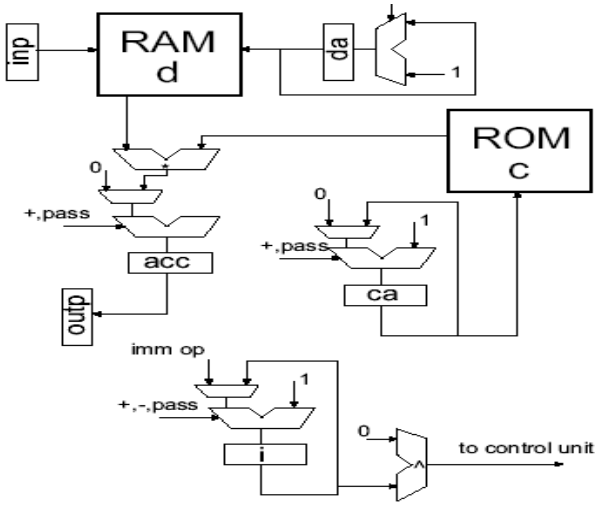


Figure 5. Datapath for 32 tap FIR filter.

realize an addressing scheme other than modulo 32. To realize this we may for instance include circular buffers for the calculation of data and coefficient addresses. Since a circular buffer may be useful in the future we decide to add a circular buffer ALU into our synthesis library and then instantiate it into the µC model. In Figure 6 it is shown how to change the µC model and what to add to the synthesis library. The new more general datapath is shown in Figure 7.

5. FUTURE WORK

The implemented heuristics with an ALU selection based on target registers lead to an architecture that normally works nice for dedicated DSPs. In modern general purpose DSPs there is normally a number of parallel ALUs that are connected to one

The µC model is changed such as

acc+=d[da++]*c[ca++]

is replaced by:

acc+=d[da]*c[ca], da=circ_add(1,da,firl),
ca=circ_add(1,ca,firl);

and

acc+=d[da]*c[ca++];

is replaced by:

acc+=d[da]*c[ca], ca=circ_add(1,ca,firl),

a register called *firl* that holds the wanted filter length is created and the line

firl=12;

is added to the model.

The VHDL description of the circ_add block is stored in the synthesis library and the library description file is added with:

```
ALU_circ_add: operations: circ_add
  weight:
    power=100,size=100,delay=100,default=100
```

Figure 6. Changes to support arbitrary circular

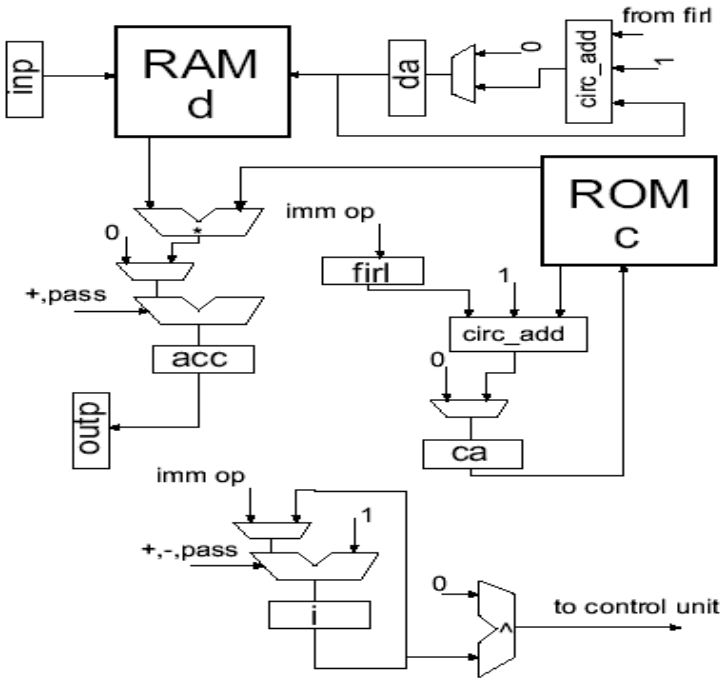


Figure 7. Datapath for a programmable FIR filter.

register file. This is an architecture that can not be supported in the present version of the synthesis tool. One of the problems with the synthesis of such an architecture is that it is difficult to decide which instruction to put in which ALU. In order to decide how many ALUs to attach to a register file the parallelism within the register file has to be analyzed. The parallel ALU problem can today be worked around by explicitly instantiate it into the uC model. But a more smooth way making it easier to elaborate with different solutions would be preferred. The instruction coding is today put into the control unit, which just is instantiated by the synthesis tool. A future extension would be to include an instruction coding stage in the tool in order to further reduce the design effort.

6. CONCLUSIONS

In this paper we have discussed a synthesis tool where a µC model is translated to a DSP processor. The nice thing with the tool is not the optimization routines in the tool since they do not contain anything advanced. Instead we have shown a design flow, using the synthesis tool, where it become easy for a designer to evaluate different architecture. We have, by an example, shown how an FIR filter can be synthesized and redesigned to support a wider application without too much work.There are things that can be improved, such as the user interface.

REFERENCE:

Application driven DSP Hardware Synthesis

Mikael Karlsson Rudberg[1,2] and Mikael Hjelm[1]

1) Ericsson Microelectronics AB, 164 81 Kista, SWEDEN

2) Department of Electrical Engineering, Linköping University, S-581 83 Linköping, SWEDEN Mikael.Rudberg@mic.ericsson.se, Mikael.Hjelm@mic.ericsson.se