Chapter 13

xHDL: EXTENDING VHDL TO IMPROVE CORE PARAMETERIZATION AND REUSE

Miguel A. Sánchez Marcos, Ángel Fernández Herrero, Marisa López-Vallejo
Dpto. Ingeniería Electrónica, E.T.S.I. Telecomunicación
Ciudad Universitaria s/n, 28040 Madrid, Spain
{masanchez,angelfh,marisa}@die.upm.es

Abstract Traditional hardware description languages are currently limited in their use to build complex systems through parameterization and reuse. In this chapter, we present xHDL, a meta-language designed to improve VHDL that provides flexible mechanisms for component customization, instantiation and interconnection. It has been conceived to ease the specification of highly parameterized cores and the reuse of already designed ones, keeping the currently available methodologies and synthesis tools. At the same time, it can help on parameter and component selection through the evaluation of functions that report on estimated characteristics of the design before the long synthesis phase. Finally, an FFT core illustrates the use of the meta-language for the specification of a complex design.

Keywords: HDL, IP, reuse, parameterization, component-based design

Introduction

Current VLSI designs are characterized by their increasing complexity and performance, while the design environment must ensure as short as possible development time. In this context, only the reuse of previous designs allows meeting such strong design constraints [Gajski, 1999]. Reused components, known as cores, may come from former designs or may be obtained from third parties. In the last case, they are qualified as Intellectual Property (IP), based on licensing reasons.

The design of systems can be significantly simplified by the proper assembly of already available components. However, integrating cores into a system is mostly manual, and consequently error prone. Designers must rightly know all

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 217–235.
© 2005 *Springer. Printed in the Netherlands.*

the characteristics of complex cores, as they are functionality, interfaces, and basic parameters.

Hence, there is a clear need of tools that help during component assembly and core generation tasks, while providing quality assessment measures (area, speed, power, accuracy) to ease the selection of cores and their key parameters. These goals can only be accomplished if the cores are specified with a language that allows subcomponent customization, instantiation and interconnection, with the corresponding simplification of the design of hierarchical collections of modules.

In this sense, conventional HDLs, as VHDL or Verilog, do not satisfy all the requirements that IP reuse may have. For instance, these languages exhibit serious limitations to parameterize a design or do not allow the specification of additional attributes to drive a design space exploration process.

To overcome these limitations, in this work we present xHDL (*Extended HDL*), a meta-language that helps the designer in the difficult task of IP core description and reuse. Our goal when creating xHDL was not to develop a new HDL, but to improve some aspects of existing ones (currently VHDL) to easy both the parameterization of designs and the reuse of previous works. In this way, xHDL provides a simple and clear way to describe complex systems, while being easy to learn and use. Since a traditional language is used as base, not only previous designs can be reused, but also tools and methodologies.

The main advantages of the proposed meta-language are the following:

- It is conceived to emphasize reuse.
- It allows a high degree of parameterization of the cores.
- It simplifies specification due to the definition of new constructs.
- No conditions are imposed to the HDL used to describe the designs.
- It is easy to learn, since it stays close to conventional HDLs.
- Design space exploration is allowed based on feedback information.

The meta-language is accompanied by a compiler that generates VHDL source code. In this sense, the designer can write customizable code in whatever style is necessary, to simulate or synthesize, both architectures or testbenches, as the meta-language is effectively independent of the target technology considered for a final implementation.

The conception of xHDL allows its use in many applications, as the implementation of core-generation tools, automated design space exploration or generation of testbenches.

The chapter is organized as follows. Next, other related works will be reviewed. Section 2 provides the description of the basic constructs of xHDL,

while section 3 illustrates the use of the meta-language through an example. Finally, some applications are described and some conclusions are drawn.

1. Related work

Previous work on languages for IP reuse has been addressed from very different points of view, tackling many different aspects. In this section, we will contrast the key points of these approaches with the present work.

First, some languages conceived for other purposes have been previously used in IP environments. For instance, SystemC [Panda, 2001] can support modelling at the register-transfer, behavioral and system levels. However, these languages do not provide the parameterization facilities that xHDL exhibits. XML has also been used for IP-based design [Zhang et al., 2001], but this language was not devised to be directly used by a designer. It requires a long learning time when used to describe hardware, and is not the best way to express some particularities required for IP specification. Finally, other languages have also been used targeting reuse, as is the case with SpecC [Dömer and Gajski, 2000], which deals specially with the protection of IPs.

Important efforts have been carried out in the field of interface description and synthesis. Rowson et al. [Rowson and Sangiovanni-Vincentelli, 1997] introduced the concept of “interface-based design”. The authors state that IPs should be designed in two parts, behavior and communication. Regular expressions are used to describe interface circuits, and an algorithm for their synthesis is presented in [Passerone et al., 1998]. Another interesting work on interface specification and verification is presented in [Suzuki et al., 1999]. Here, O_wL is an interface language defined for IP reuse, providing multiple applications. It is is devised to work at a very low level, what makes difficult its application for the specification of complex systems.

Confluence [Confluence, 2004] is a declarative, functional language that eases RTL code generation. This language combines the dataflow and component-based methodologies of HDLs with the expressiveness of modern functional programming. Some key ideas behind the language are shared by our proposal, but Confluence does not allow the definition of functions to evaluate internal parameters of the IP core and provide feedback about its suitability.

Finally, the component composition framework BALBOA [Doucet et al., 2003] is a bottom-up approach for SOC construction using reusable IPs. The framework includes a component integration language (*CIL*) which can be translated into a C++ implementation. In this case, as happened with Confluence, a new language is fully defined, making necessary to learn the new syntaxis and features and provide a complete and new set of tools to deal with.

Nowadays, there are workgroups upgrading traditional HDLs as Verilog or VHDL [EDA, 2004]. However, the time required to change a standard is too

long, and designers need tools to overcome language limitations even before the changes required by the language have been addressed.

In this chapter, we present a meta-language defined on top of VHDL to specify and generate IP cores. Our approach supports hierarchical implementation of complex designs, with emphasis on providing parameterization, module interconnection and reuse capabilities.

2. Fundamentals of xHDL

Source files for xHDL are known as *templates*. They contain the code for the available cores as embedded VHDL. The meta-language uses a reduced set of well defined types to symplify the learning process. This set is based in VHDL types, extended with new ones to implement specific functionalities. Arithmetic is performed by function calling, and a mechanism is provided to let the designer implement new functions and access them through the templates.

To allow reuse, xHDL implements a flexible way of communication between subcomponents, a calling mechanism. This helps in parameter fixing (e.g. a module can report to the core on the number of iterations needed for some computation), and also in getting feedback information from a subcomponent (e.g. multiplier latency that determines some buffer latency).

In the meta-language there are two different concepts for functions: as elements for expressions (introduced above) or as feedback information from a template. The last will be described later, and both can be identified in each context.

Similarly to VHDL, templates are divided into three sections: `generics`, `ports` and `architecture`.

2.1 Generics

This section allows the designer defining the basic data items for the core:

1. **Parameters**. They hold input values to customize the core. They can be provided directly, but also by an upper level calling core (if the current one is needed as a subcomponent), what allows passing parameters along the hierarchy. An example declaration with default value is:

   ```
   parameter width = 8;
   ```

   Differently from VHDL, where the generics are interpreted by the synthesizer, in xHDL the parameters will disappear after code generation, being substituted with their values. This greatly helps in a synthesis stage.

2. **Functions**. These are a set of output values to report on characteristics of the core, previously to the generation process. They can provide

bounds or recommended values for parameters, depending on the values chosen for some others, or they can also be simple estimates on core characteristics that help in the selection of parameter values:

```
function max_nstages = Add(width, 2);
```

As it happens to properties, their calculation is determined by the designer by using (expression) functions with parameters as arguments.

3 **Properties**. They are similar to functions, but calculated during the generation process, and reported afterwards. They can be used for estimators that strongly depend on how the core is created, or which subcomponents are finally selected for it, as throughput, timing, area, power or accuracy (expected quality magnitudes).

In the template, this type behaves as a global variable, so that, when declared, properties must be initialized with proper values:

```
property latency = 0;
```

In this example, the latency takes a null initial value and will be subsequently updated in the template, probably depending on specific generation options. The resulted value is stored to be recovered when needed.

2.2 Ports

The ports section is devoted to declare the interface for the core. In this sense, the statement:

```
XI: IN word;
```

is equivalent to the VHDL one:

```
XI: IN STD_LOGIC_VECTOR (word-1 DOWNTO 0);
```

where the type STD_LOGIC is considered as a base to interface definitions, being unnecessary its insertion in the declaration statements.

The meta-language introduces several facilities over VHDL, as is the possibility to conditionally declare ports:

```
( PIPELEVEL != 0 ) CLK: IN 1;
```

or the use of arrays, which are declared and accessed by using brackets:

```
DIR [Log2(word)]: IN 1;
```

2.3 Architecture

The meta-language aims to simplify the tasks that are difficult or limited in VHDL. For instance, even with the use of generate sentences, VHDL parameterization and generalization possibilities are awful and complex. In this sense, xHDL can be used to automate VHDL code insertions and subcomponent interconnection during core generation.

In xHDL, signals and components are declared only when needed, while variables allow carrying references to out-of-scope objects between different control structures, and can be used in embedded VHDL source code.

In this section, the architecture description takes place, and thus, it is where actual VHDL source code will be inserted. The constructions that can be used are within several types: *declarations*, *control structures* and *references*.

Declarations. In a template, it is possible to use data items declared in the `generics` or `ports` sections, but it is also possible to declare new ones for internal use. Differently from VHDL, declarations can be placed anywhere, but always before the object is used. *Variables*, *signals*, *entities* and *components* can be declared.

When a core reuses another one as subcomponent, it can use information provided by feedback functions and facilities for conditional declaration to declare only the types really needed at each moment.

Variables The variables allow storing values or references in a section:

```
variable var_word = word;
```

Variables are internally stored as strings, but their actual types depend on the use (integer, floating, etc.). They are similar to properties, as their values can be updated along the template:

```
latency = Add(latency, 1);
```

However, they are conceived as intermediate holders during component specification and are declared as needed in the `architecture` part, while properties are introduced in the `generics` section, as they keep important characteristics of the core that will be exposed to the user.

Variables in xHDL, which have no correspondence in VHDL, allow both store intermediate values in a template and keep references to ports and signals, which helps in the generalization of a core specification.

As mentioned before, it is possible to use functions, both tool predefined (`Add`, `Sub`, `Max`, `Log2`, `Dec2Bin`, etc.) and user defined. For calculations, constant values are also allowed.

Signals This type, together with entities, is closely related to the corresponding in VHDL. The keyword `signal` will insert a signal declaration into the declarative part of the VHDL component generated, keeping at the same time a reference to that signal into the meta-language:

```
signal mem2add[radix] var_word;
```

Variable and signal declarations have scope, the control structure where they are defined. Hence, out of there, references are lost. This simplifies name tracking during specification, differently from VHDL, where the signals must be declared at the beginning of the architecture. If needed, variables can be used to keep references to out-of-scope signals.

At the same time, signals are only generated if their control structure is accessed during component generation. Moreover, the same xHDL signal can produce several VHDL ones, as is the case with iterative sentences, simplifying declarations.

Entities and components Entity declarations are key in the reuse context. They allow refering to already described subcomponents.

To reuse a component, it is first necessary to provide its name and the library where it is stored. Then, we can initialize some of its parameters, differently from their default values, and get the value of feedback functions for the combination of parameters that results:

```
entity cordic lib.arith.Cordic;
cordic.generics (NVBITS = 16);
variable N = cordic->function ("NROTS_MAX");
```

The last two statements can be performed as many times as necessary, allowing an iterative process to search for the right values in each application (a process of inter-component communication for negotiation of parameter values). The component will not be generated. The strength of xHDL to perform design space exploration is partly due to this ability.

Then, it is possible to set again the same parameter or others, and finally declare the component, triggering its source code generation. This also inserts the declarative part of the subcomponent into the current one:

```
cordic.generics (NROTS = N);
component i_cordic = cordic->generate;
```

Now, we can access component properties defined by the designer:

```
variable L = i_cordic->property ("NREGS");
```

In summary, several entities can be evaluated within the meta-language to finally choose only the most convenient of them, or even several ones to be placed in different locations into the architecture.

Since our final goal is core reuse and automatic generation, in xHDL there is a way to nest templates for architectures, similar to function calling for expressions. It allows automatically generating code for the port-to-signal mapping for a subcomponent, ensuring at the same time the correct use of conditional ports:

```
i_cordic.ports ( CLK = CLK, ..., Y0 = rot_y0 );
var_temp_y = rot_y0;
```

The use of variables in the meta-language is very advantageous, giving a great flexibility to the generation process, as they are not declared in the finally generated component, what is different from the case of signals and subcomponents. Variables can transport signal references between control structures at the meta-language level.

Control Structures. In VHDL, the `if` construction does not allow `else`, being therefore necessary to reevaluate conditions. Moreover, the VHDL `for` has a closed range, which must be fixed from the beginning.

In this sense, conditional structures in xHDL are `if-else` based:

```
if ( ... ) { ... }
if ( ... ) { ... } else { ... }
if ( ... ) { ... } else if ( ... ) { ... }
```

On the other hand, iterative structures are based in `while` constructs, with condition evaluation previous to loop execution:

```
while ( ... ) { ... }
```

Conditions have been already introduced in port declarations, and can be simple (relational operators: ==, != , >, >=, <, <=) or composed (logical operators: &&, ||). Finally, it is possible to use meta-language functions to evaluate complex conditions.

Code insertions. As was previously stated, xHDL is not a new language, but it is conceived to ease the creation of parameterizable designs based on other description languages. In this sense, control structures and data types available in xHDL have already been described, but to properly describe a component, it is also necessary to perform source code insertions in the middle of the meta-language. Those insertions can be customized by using data types, through the substitution of their actual values during generation.

Source code insertions are performed by using reserved words in the meta-language, one for each possible location. In this way, to insert code into the VHDL library declarations section, we will use:

```
lib { LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL; }
```

To insert code into the declarative part of VHDL architectures:

```
decl { CONSTANT MEM_[index] : STD_LOGIC_VECTOR ([var_msb] DOWNTO 0) :=
   "[Dec2Bin(lib.arith.cordic.ATRCoeff(index, word))]"; }
```

Finally, to insert code into the implementation part of VHDL architectures:

```
code { MEM_[index] WHEN DIR = "[Dec2Bin(index, word_index)]" ELSE ... }
```

In code insertions, VHDL is interpreted as plain text, but accepts substitutions anywhere, triggered with brackets around meta-language expressions, which use formerly defined types (e.g. parameters, functions, properties, variables). This allows getting extensive code customization.

In this sense, the `decl` example above illustrates the use of variable substitutions and function calling, both tool predefined and user defined.

3. Design Example

To check the usefulness of xHDL, showing its customization and interconnection capabilities, a set of cores has been implemented and integrated into a meta-lenguage library. This contains from simple cores, as adders, registers, etc., to more sophisticated ones, as general KCM's or CORDIC rotators.

The library can be used as the base for new cores, as it is with the selected example for this chapter, the FFT. This core has been developed both reusing the library and building new components.

3.1 FFT specification

The chosen implementation follows Cooley-Tukey's algorithm [Rémondeau, 1999], with an online pipelined and parallel architecture. This architecture results from a regular repetition of several non-identical stages, so it is good to illustrate hierarchical design and parameterizable reuse of the components needed in the stages.

The VHDL code embedded in the templates for this example has been optimized to target the Xilinx Virtex II architecture, as is the case with the feedback information provided in properties.

The main parameters of the FFT core are the number of stages and the radix, which determine the final length. Other basic parameters are wordlength for input samples and scaling, providing the last one the growing policy after butterfly calculations:

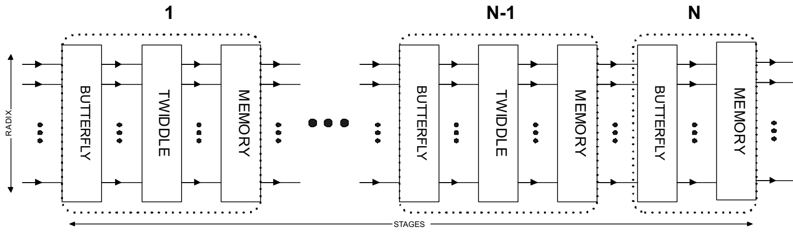

Figure 13.1. Structure of the FFT

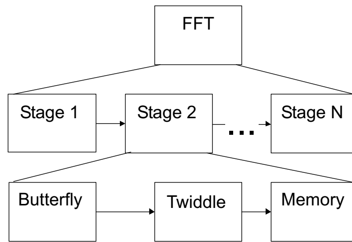

Figure 13.2. Hierarchy of the FFT

```
generics {
     parameter STAGES = 4;
     parameter RADIX = 4;
     ...
     }
```

These parameters are introduced in the top template of the design, which is based on the structural description of the selected algorithm (see figure 13.1) and built around three sub-cores: memory, butterfly and twiddle multipliers. These new cores are also implemented as xHDL templates with their own set of parameters, though related to those of the top template.

The top template merely specifies where and when the sub-cores are inserted, and how to customize (with xHDL functions and properties) and interconnect them (with xHDL variables and signals). Figure 13.2 shows the way these sub-cores are hierarchically instantiated into the specification.

The sub-cores are instantiated within a `while` control structure, which is used to unroll the algorithm into parameterized stages. In each one, first a butterfly is customized and inserted, then a twiddle multiplier and the memory core are generated, if needed:

```
variable fft_index = STAGES;
while ( fft_index > 0 ) {
    entity butterfly = fft.online.butterfly.dif;
    ...
    if ( fft_index > 1 ) {
       entity twiddle = fft.online.ffwd.twdarray;
```

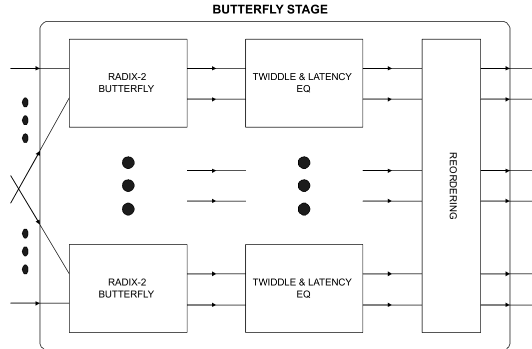

Figure 13.3. Components in a general butterfly element

```
        entity memory = fft.online.ffwd.datamem;
        ...
        }
    ...
    fft_index = Dec(fft_index);
    }
```

3.2 Butterfly

In this subcomponent, a key parameter is `RADIX`, which can only take power-of-two values. This parameter not only defines the inner structure but also provides the number of ports needed:

```
ports {
      ...
      IN_REAL  [RADIX] : in  WIDTH;
      IN_IMAG  [RADIX] : in  WIDTH;
      OUT_REAL [RADIX] : out OUT_WIDTH;
      OUT_IMAG [RADIX] : out OUT_WIDTH;
      }
```

The template is internally implemented as `Log2(RADIX)` stages, and hence, two variable arrays are declared to store the references to intermediate signals:

```
variable data_real [RADIX];
variable data_imag [RADIX];
```

The mapping of input ports into these variables is made as in:

```
variable index = 0;
while ( index < RADIX ) {
    data_real [index] = IN_REAL [index];
    data_imag [index] = IN_IMAG [index];
    index = Inc(index);
}
```

Now, they are used inside a main while loop, which creates the structure of the butterfly (shown in figure 13.3):

```
while ( index < RADIX ) {
    pipereg.ports (
          CLOCK = CLOCK,
          RESET = RESET,
          ENABLE = ENABLE,
          DATA_IN = data_real [index],
          DATA_OUT = real_pipereg [index] );
    data_real [index] = real_pipereg [index];
    index = Inc(index);
}
```

Finally, variables are again used to connect the output of the last component to the output ports:

```
while ( index < RADIX ) {
     code {
          [OUT_REAL [index]] <= [data_real[index]];
          [OUT_IMAG [index]] <= [data_imag[index]];
     }
     index = Inc(index);
}
```

In each stage of the main loop, radix-2 butterflies are firstly performed between pairs of ordered input samples. In these butterflies, the arrays are used as inputs, whereas local signals are declared for outputs. When the components are inserted, the variables are updated with these output signals. In this process, a bit is added if the scaling option is set, otherwise the result is truncated:

```
signal real_r2a [RADIX] : WIDTH;
   ...
// radix-2 butterflies
   ...
if ( SCALE != 0 ) {
   data_real [i] = real_r2a [i];
   ...
} else {
   data_real [i] = vhdl.Range(real_r2a[i], width, Dec(width), 1);
   ...
}
```

Next, a twiddle for each sample is calculated using xHDL functions, and the most suitable rotator for each one is chosen, together with the pipeline level for every radix-2 butterfly stage.

```
variable twiddle_term = fft.Twiddle(base_radix, index_radix);
if (( twiddle_term == "360.0" ) || ( twiddle_term == "0.0" )) {
   ...
} else if ( twiddle_term == "315.0" ) {
   ...
} else { // general twiddle rotator
   ...
}
```

Finally, the samples, which are stored in the array, are reordered by changing their positions:

```
while ( index < Div(RADIX, 2) ) {
   variable i1 = index;
   variable o1 = Mult(index, 2);
   ...
   tmp_real [o1] = data_real [i1];
   tmp_imag [o1] = data_imag [i1];
   index = Inc(index);
   }
```

During component generation, the template updates their own properties, as latency (which is used in the top FFT to synchronize all the sub-cores), with those of the subcomponents that built it (adders, subtracters, etc.).

3.3 Twiddle multipliers

This template acts a wrapper for the rotators which will be inserted in each FFT stage during source code generation. The choice of the rotator is based on the value of the angle associated with the twiddle position into the FFT. When the rotators are inserted, the template also has to equalize every output so that all of them have the same final wordlength, gain and latency.

For trivial rotations, a set of simplified sub-cores is available that implement them with great savings of resources with respect to a general rotator.

On the other hand, the general rotator chosen is based on CORDIC [Andraka, 1998]. This component is already available in the xHDL library, so that it only has to be instantiated with proper values for its parameters, some of them with recommended values obtained by using feedback information functions:

```
cordic.generics (KWIDTH = WIDTH, ...);
variable cordic_nrots = cordic->function ("NROTS_MAX");
cordic.generics (NROTS = cordic_nrots);
```

The CORDIC lets implement a twiddle rotator in two ways. First, as a completely general rotator with a register and an adder to internally generate the twiddle angle. Second, for fixed angles, the rotation sequences can be externally supplied from a ROM, which is a new sub-core whose coefficients are calculated by meta-language functions at generation time (figure 13.4).

Selection among the different rotator alternatives is determined by an input parameter to the core. This parameter can be fixed to one of the possible alternatives, or it can take the value `automatic`, in which case the template tries to minimize resources:

```
// TYPE 0 Automatic
// TYPE 1 Fixed-angle rotations
if (( TYPE == 1 ) || (( NUM_DATA < 512 ) && ( TYPE == 0 ))) {
   entity cordic = arith.cordic.online.fixrots;
   ...
} else {  // General rotator
   entity cordic = arith.cordic.online.basic;
   ...
}
```

```
decl {
   CONSTANT rom_[i] :
   STD_LOGIC_VECTOR ([Dec(WIDTH)] DOWNTO 0) :=
   [fft.CordicRots(i, POINTS, STAGE, WORDLENGTH)];
   }
```

a) source xHDL code for a ROM

```
CONSTANT rom_0 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "011110100"; -- "0.0"
CONSTANT rom_1 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100010001"; -- "-5.625"
CONSTANT rom_2 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100010111"; -- "-11.25"
CONSTANT rom_3 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100011101"; -- "-16.875"
CONSTANT rom_4 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100100100"; -- "-22.5"
CONSTANT rom_5 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100101011"; -- "-28.125"
CONSTANT rom_6 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100110001"; -- "-33.75"
CONSTANT rom_7 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100110111"; -- "-39.375"
CONSTANT rom_8 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100111110"; -- "-45.0"
...
```

b) generated VHDL code for a ROM

Figure 13.4. Example of code generation from xHDL

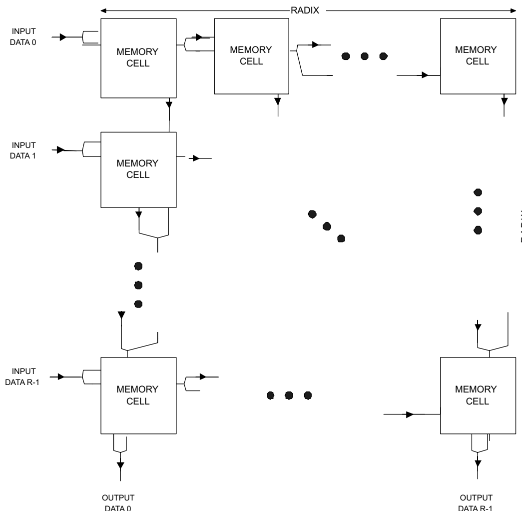

Figure 13.5. Structure of the memory template.

3.4 Memory

The function of this element is to keep and reorder intermediate samples between stages. Consequently, it needs a set of parameters that determine the amount of space which is necessary, and are passed from the upper template depending on their own parameters and variables.

The memory sub-core collects input samples from an FFT stage and sends ordered ones to another stage. To implement this function, the memory is arranged as $R \times R$ matrices, and the template describes this structure by using a double loop and a sub-template (memory cell):

```
variable index_in_radix = 0;
while ( index_in_radix < RADIX ) {
      ...
      variable index_out_radix = 0;
      while ( index_out_radix < RADIX ) {
              memory_cell.ports(...);
              ...
      }
      ...
}
```

The template for the memory cell provides the type of memory (block or distributed RAM, registers) through an internal parameter. The template for the top memory performs both cell interconnection and generation of the read/ write and addressing signals from the available set of input control signals.

4. Applications

The formerly described meta-language is mainly oriented to the specification and reuse of IP-cores, but it has many more applications in the IP domain:

- Design space exploration,
- Description of general IP-cores,
- Generation of testbenches,
- RTL code generation.

For instance, a *system designer* can make design space exploration from the very early stages of the development cycle, especially if it is based on the reuse of components from a library. If these components have well defined feedback functions and properties, they can be instantiated into a structural design and characteristics like estimated area, speed, etc. can be obtained even without a synthesizable design.

At the same time, design space exploration lets a *core designer* to select between different alternatives for subcomponents. Once the final components and their parameters have been chosen, the design can be fully specified with the meta-language to obtain a new parametric IP-core. Now, it can be added to the library to be reused in further designs.

On the other hand, during hardware design, it is also mandatory to take into consideration a set of testbenches to verify correct operation. Using xHDL, several parametric testbenches can be automatically added to the core and generated within the synthesizable source code.

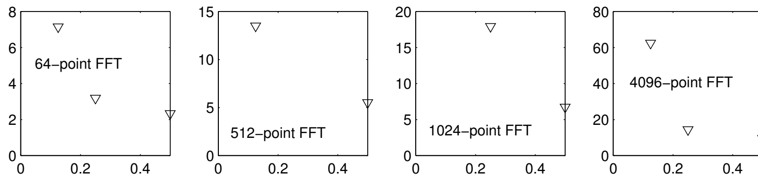

Figure 13.6. Design points of different FFT implementations (area vs. performance).

We address some considerations about the design space exploration and a core generation tool in the following sections.

4.1 Design Space Exploration

A design process is usually iterative, where different options need to be checked to finally get the best option. In this sense, the meta-language allows getting several core configurations by parameterization, while providing design feedback by functions and properties.

Based on these concepts, the designer can choose parameters within the bounds fixed by some of the functions, while checking the values from others to decide if the core meets restrictions. Finally, he can perform several tentative generations to check properties, which can give a more accurate or elaborated information, and then choose the best option. This is far cheaper than several complete synthesis stages.

Figure 13.6 illustrates the process of design space exploration for different FFT implementations. Four figures are shown for different N-point FFTs. The horizontal axis shows the inverse of the radix, which is proportional to the throughput for the same clock frequency, while the vertical one depicts area related values (thousands of LUTs for our FPGA implementation) for the resulted architecture.

The typical area-time tradeoff can be observed, and these results can be employed by the designer while choosing a core architecture.

4.2 Core Generation Tool

We have implemented an interactive tool to provide easy access to every possible configuration of a target core described with xHDL, and also to manage the available feedback functions [Fernández et al., 2004]. The tool collects the necessary information for the generation of a subcomponent from two main sources:

- Meta-language templates and some configuration files.
- Parameter values obtained from the user through the interface.

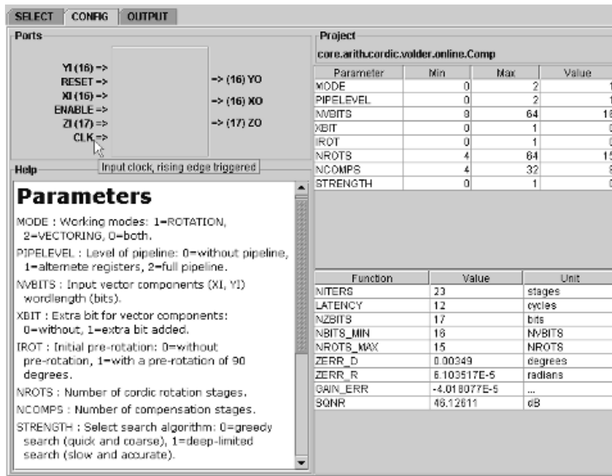

Figure 13.7. Second panel of the interface window for a CORDIC core.

The GUI is composed of a window with three panels, selectable by tabs placed in the upper part (figure 13.7). The first panel (*Select*) gives a list of the available cores by using a tree graph, while also shows some description information for the selected one.

The second panel (*Config*), shown in figure 13.7, consists of two parts. On the right, it displays the available component parameters, with bounds and a field to provide the desired value, and the feedback functions, with name, current value and unit. On the left, a block diagram for the core instance, together with HTML formatted help information. Both are automatically generated using the template contents.

The last panel (*Output*) includes fields to provide the final instance name and destination email address for the generated files, and buttons to begin core generation.

5. Conclusions

This chapter has dealt with the definition of xHDL, a meta-language for IP core description and reuse. This meta-language allows extensive VHDL source code parameterization and simplifies the specification phase, automating many awful tasks, as subcomponent instantiation and interconnection. It also provides several complex code manipulations, as conditionals and loops, many of them dependent on input parameters.

The underlying template concept is general enough to cope with guided source code generation of any hardware component whose implementation, parameters and feedback functions are available. In fact, the modular descrip-

tion allows the reuse of whatever component previously implemented as templates, and this is widely used to instantiate simpler components, as registers, multiplexers, arithmetic, or more complex sub-cores.

There are many interesting applications for xHDL. In this sense, a tool for core generation with parameter selection has been built as demonstrator. The meta-language can also be used to perform design space exploration, which is guided by the evaluation of feedback functions and properties that report on selected characteristics of the resulted components.

Acknowledgements

This work has been supported by the Spanish Government under Research Projects TIC2003-07036 and TIC2003-09061-C03-02.

References

Andraka, R. (1998). A survey of CORDIC algorithms for FPGA based computers. In *Proc. ACM/SIGMA 6th Int. Symposium on FPGAs*.

Confluence (2004). Confluence language. `http://www.confluent.org`.

Dömer, R. and Gajski, D. (2000). Reuse and Protection of Intellectual Property in the SpecC System. In *Proc. ASP-DAC*.

Doucet, F., Shukla, S., Otsuka, M., and Gupta, R. (2003). BALBOA: a Component-based Design Environment for System Models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*.

EDA (2004). Electronic Design Automation Industry Working Groups. `http://www.eda.org`.

Fernández, A., Sánchez, M. A., and López-Vallejo, M. (2004). A Web-based Environment for the Evaluation and Generation of Complex IP Cores. In *IP-SOC*.

Gajski, D. (1999). IP-based Design Methodology. In *Design Automation Conference (DAC)*.

Panda, P. R. (2001). SystemC – A Modeling Platform Supporting Multiple Design Abstractions. In *14th Intl. Symposium on System Synthesis*.

Passerone, R., Rowson, J. A., and Sangiovanni-Vincentelli, A. (1998). Automatic Synthesis of Interfaces between Incompatible Protocols. In *Design Automation Conference (DAC)*.

Rowson, J. A. and Sangiovanni-Vincentelli, A. (1997). Interface-based Design. In *Design Automation Conference (DAC)*.

Rémondeau, J.-M. (1999). Scalable parallel architecture for ultra fast FFT in an FPGA. In *Proc. ICSPAT*.

Suzuki, K., Ara, K., and Yano, K. (1999). $O_{w}l$: An interface description language for IP reuse. In *IEEE Conf. on Custom Integrated Circuits*.

Zhang, T., Benini, L., and Micheli, G. De (2001). Component selection and matching for IP-based design. In *Design Automation and Test in Europe (DATE)*.