Spring 2013

Automated RTL generator

Rohit Kulkarni
San Jose State University

Automated RTL generator

A project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements For the Degree

Master of Science

By

Rohit Kulkarni

May 2013

Rohit Kulkarni

The Designated Project Committee Approves the Project Titled

Automated RTL Generator

By

Rohit Kulkarni

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE
SAN JOSE STATE UNIVERSITY

MAY 2013

Dr. Robert Chun	**Department of Computer Science**
Dr. Soon Tee Teoh	**Department of Computer Science**
Kristopher Bechamp	**Design Engineer at Xilinx Inc.**

ABSTRACT

Code generation is a vast topic and has been discussed and implemented for quite a while now. It has been also been a topic of debate as to what is an ideal code generator and how an ideal code generator can be created. The biggest challenge while creating a code generator is to maintain a balance between the amount of freedom given to the user and the restrictions imposed on the code generated. These two seemed to be very conflicting requirements while designing the Automated RTL Code Generator. If the code generator tries to be rigid and sticks to well-defined paths and restricted code, the flexibility provided to the also reduces. It is a very interesting task to strike the right amount of balance and generate code of high quality and well-defined standards. Verilog code is a type of RTL (Register Transfer Level) that itself has fewer constructs and variety as compared to pure software languages like Java, or Python so it makes sense to generate it automatically so that the hardware designers are relieved from the mundane tasks of writing repetitive verilog code modules. Also code generator provides a nice introduction to the much wider topic of compiler design. This project also tries to delve deeper into the latest IPXACT XML standard IEEE 1865-2009 which is used for hardware description and will provide means of generating verilog code directly from it.

ACKNOWLEDGMENTS

I would like to thank my project advisor, Dr. Robert Chun, for his guidance, encouragement and unwavering faith in my project. He was completely supportive despite the fact that I was working on a hardware domain based project, which was a new domain for me. I would also like to thank my committee members: Dr. Soon Tee Teoh and for his feedback and suggestions and Kristopher Bechamp for introducing me to the domain while at Xilinx Inc. Finally I would like to thank my family who have always been with me through whatever I have set out to achieve.

Table of Contents:

1. Introduction:

Designing microprocessors involves circuits whose building blocks are registers and other data path components. It involves transferring data from registers through other data path components like adders and back to registers. Such design is thus called register transfer level design. Today most practice is at the Register Transfer Level. Improving tools continue to move design practice to higher levels. Higher levels deal with fewer and higher complexity building blocks, and thus can enable design of higher-complexity circuits with less time and effort. The Automated RTL generator is one such tool that aims to move the RTL design practice to a higher level by generating RTL (Verilog) code directly from an IPXACT xml file. The tool would take as input an IPXACT xml file and generate Verilog code (RTL) from it. This way it eliminates the need to manually write the code after analyzing at an IPXACT file.

2. IP XACT :

2.1 Introduction to IPXACT:

IPXACT is a simple XML file that adheres to standards set by the SPIRIT consortium. It describes in an understandable way, hardware components and the hardware designs. IP- XACT was created by the SPIRIT Consortium as a standard to enable automated configuration and integration through tools.

The goals of the standard are[1]:

• to help different vendors to share description of their hardware components

• to aid exchange of intricate and large design modules between electronic design automation

(EDA) designer for System on Chip design environments,

• to enable description of configurable modules using meta-data

• to facilitate creation of EDA tools that are not specific to a vendor, but can still generate components

Approved as IEEE 1685-2009 on December 9, 2009 published on February 18, 2010. this will be the standard input for the RTL generator.

2.2 IPXACT Background:

The IPXACT Schema technical committee was formed by Accellera in July of 2010. The standard was first established in 2003 by the SPIRIT consortium. After the initial version various changes were made to the standard. Each incremental change was aimed to accommodate various features of intellectual property (IP), that would be captured by the standard and realized into designs. Various features that the standard tries to include are the description of bit-fields, registers, modules, input output wires etc. IPXACT was submitted to the IEEE-SA and received industry approval in June 2009. The approval was given by way of ballot.

It became available in June 2010 as the IEEE 1685 IPXACT standard. IEEE 1685 was developed within the IEEE Standards Association Corporate Program in which each participating member entity (such as corporations or other institutions) has one vote.

The IPXACT forms that are standardized include: components, systems, bus interfaces and connections, abstractions of those buses, and details of the components including address maps, register and field descriptions, and file set descriptions for use in automating design, verification, documentation, and use flows for electronic systems. The XML schema finalized for IEEE 1685-2009 conforms to the requirements described by the World Wide Consortium (W3C) and addition to them imposes its own semantic rules[1].

2.3 Understanding IPXACT:

IPXACT is a simple XML document that contains a description of electronic components and their design. It contains like every other XML document, set of tags their attributes and information contained within those tags. These tags should represent correctly a component that is synthesizable, for example consider a register that has been described using the IPXACT format (Fig 1).

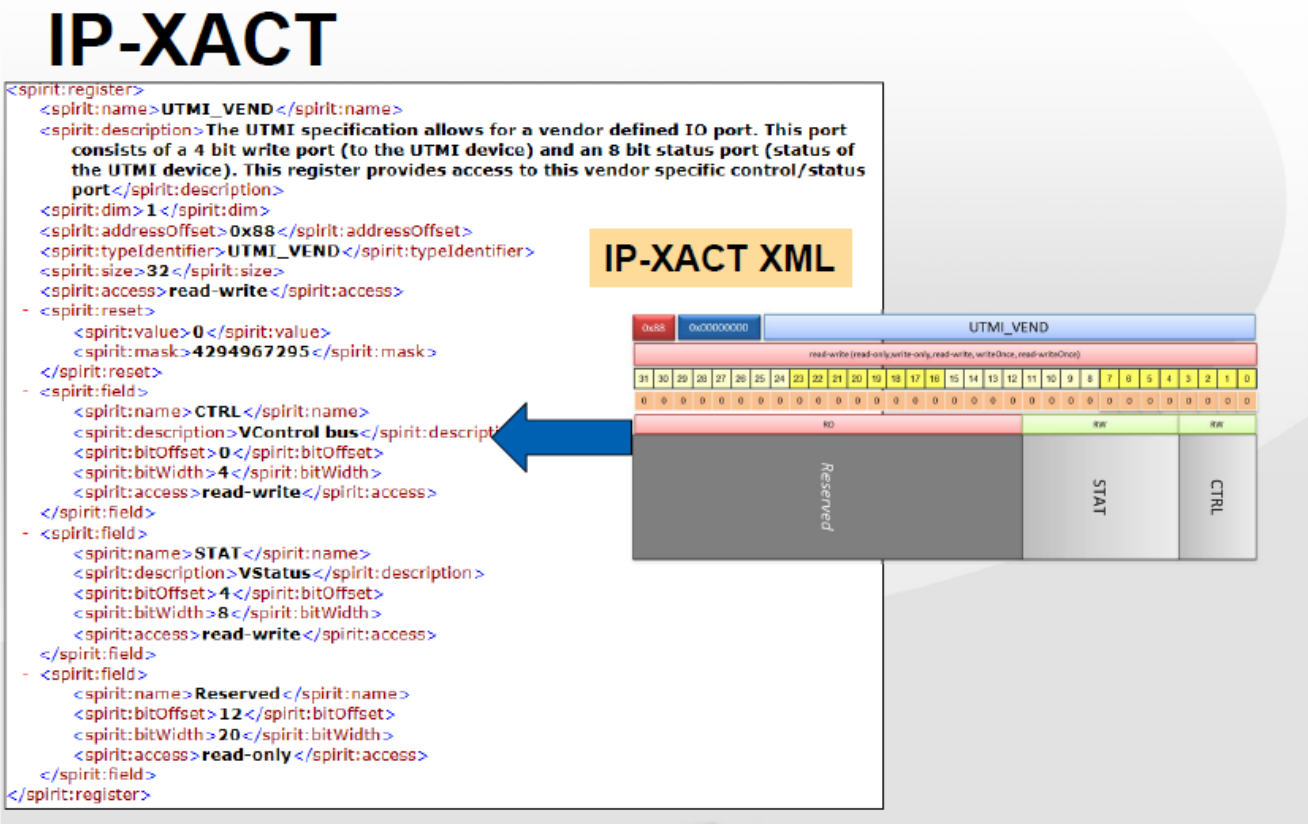


Fig 1

The above diagram shows how one of the components of the IPXACT (xml) document represents a component of the hardware (in this case a simple register along with its bit-fields).

2.4 Components of IPXACT XML document:

There are 7 top level components of an IPXACT XML document[2].

1. busDefinition
2. abstractDefinition
3. componentDescription
4. designDescription
5. abstractorDescription
6. generatorchainDescription
7. designConfiguration

These will be the top level tags in any IPXACT XML document. Each of these components have sub-components in the form of XML tags. These tags have further sub-components and so on. This leads to a complex hierarchy of XML tags in an IPXACT document.

3. XML parser creation:

XML is gaining a lot of popularity among enterprises as it provides a standard for exchange of information among heterogeneous platforms. Any application using XML as a source of information (IPXACT in this case) requires XML parsing that provides an interface for the user to access its content (then generate RTL from the content in this case). There are two major challenges when designing an XML parser:

- Code Size

  The code size should be constrained as there may be memory limitations

- Run time adaptability

  Parsers should be adaptable as diverse applications may be dependent on the XML syntax set.

Parsing can be done through two distinct approaches[5]:

- Event based parser (Eg. SAX)
- Object-based parser (Eg. DOM)

The below section takes a closer look at both the types of parsers. It also explains what approach was chosen for the automated RTL generator and the reasons for the same.

3.1 Event-based XML Parsers (SAX Parser):

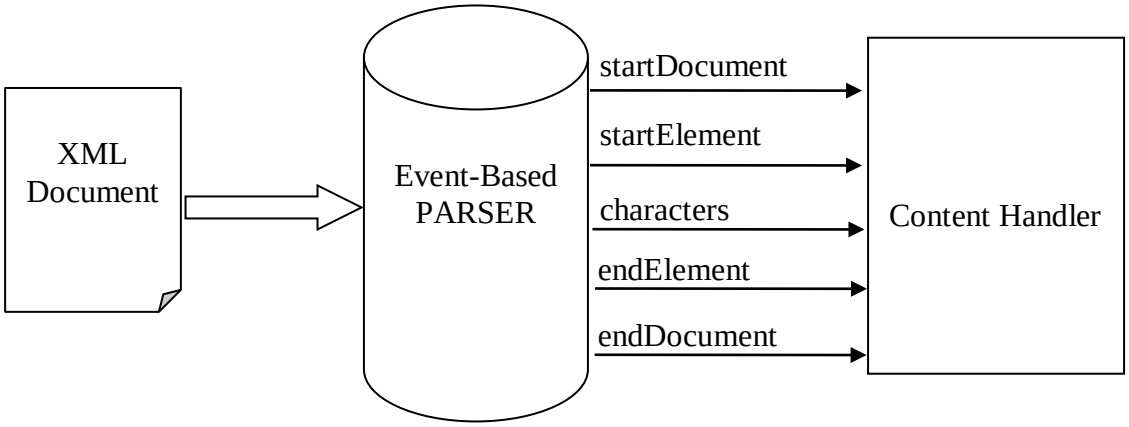


Fig 2

- They do not build internal trees to store parsed data[5]
- At no point of time is the entire xml data stored in the memory at once
- Implements event-handling handlers that deal with different events much like a graphical user interface (GUI events).
- Provides a more finer access to the XML elements
- Since does not store the entire XML data , parsing of very large documents is possible
- The SAX parser is an example of an Event-based parser
- Consider finding a single term in a large XML document. It would not be efficient to store the entire XML data in memory just to locate a single piece of information.
- Such a task would be done more effectively using an event-based parser by locating the information using a single pass over small part of the data.

3.2 Object-Based XML Parsers (DOM parser)

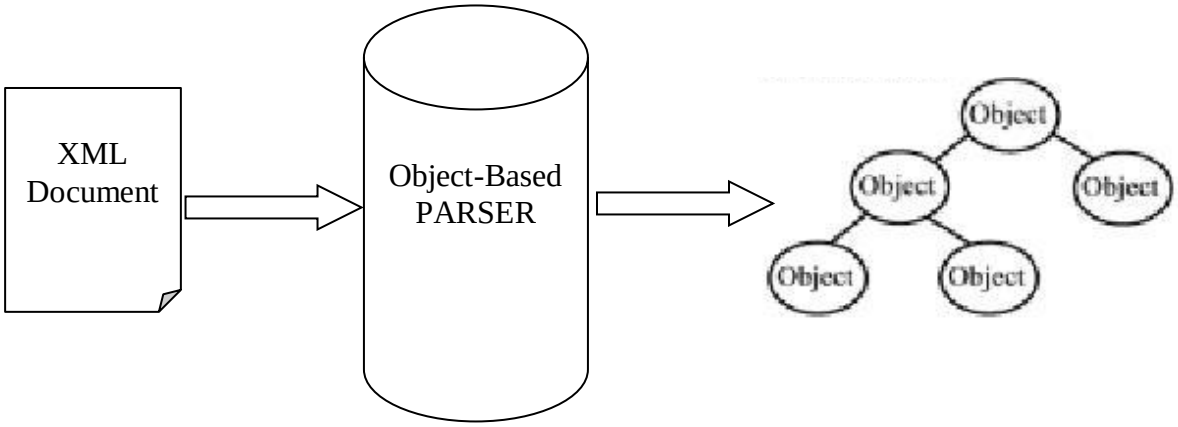


Fig 3

- Models the XML document as a tree of various nodes
- DOM parsers create a node object for each element of the XML, thus precisely models the structure and information of the XML document.
- Provides random access to the XML document
- Since it stores the entire document in memory its very inefficient for enterprise-wide applications[5]
- The time required for parsing a large is long and the application has to wait until the entire document is processed.

3.3 Approach adopted for Parsing IPXACT: Object-Based XML Parsers (DOM parser)

- Parsing an IPXACT file should yield all the information that is required to generate Verilog code that correctly describes the hardware specified in the XML.
- The DOM parser stores the entire XML information in a single object.
- This comes in very handy when generating the Verilog code because all the information that could possible used to generate the code is available at all times in the DOM object.
- Object based parsing thus seems more suitable to build auxiliary data structures that can be used for complicated processing to produce RTL.
- The python API used here is the python “minidom” API.
- The “minidom” API is a minimalistic version of the more complicated DOM API.
- It provides all the basic functionality that is required to parse an IPXACT document.
- There are some limitations when using the minidom API:
    - It is memory and CPU intensive
    - The size of the document that can be parsed is restricted
- Even then the flexibility provided by this object model is very convenient. Hence this approach has been choosing for the parsing part.

4. Traditional design process without RTL Generation without using IPXACT:

Traditionally, in the SoC design process, different engineers working on a single chip would have to work on the specifications given to them and create design in their own format. For different modules the engineers would come up with their own RTL for their own module. As the chips become complex there are two major concerns for facing the engineers:

- Standardized way of generating design
- Writing lengthy RTL

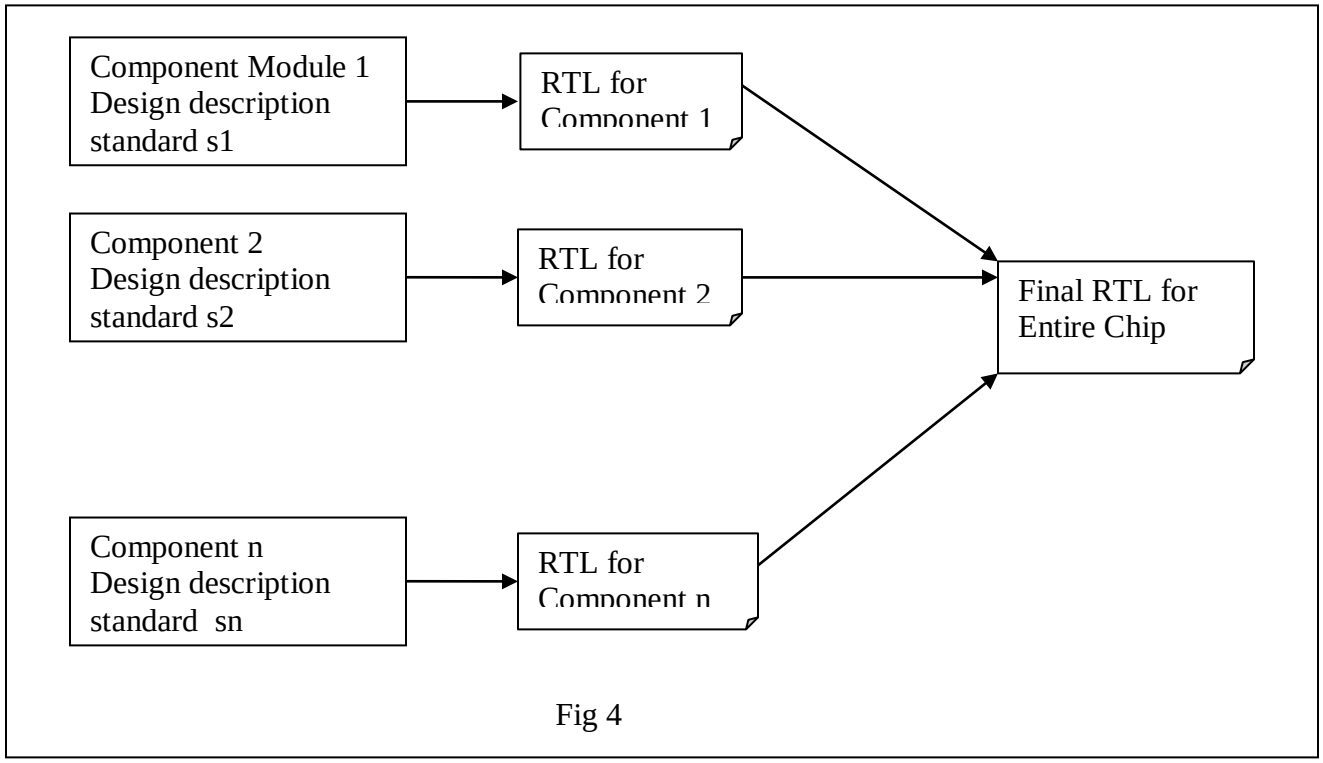


Fig 4

4.1 IPXACT for intra-organization design sharing:

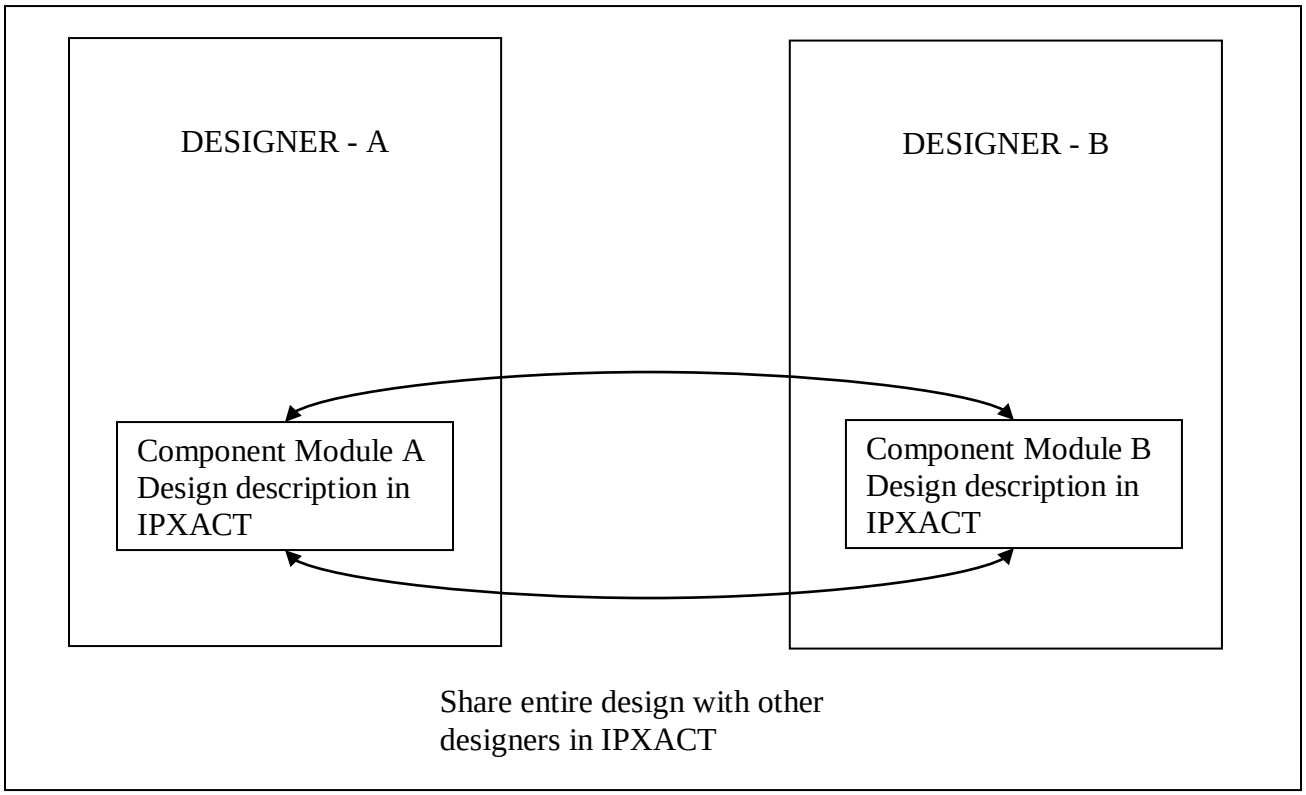


Fig 5

- Designers working on different components with a company may use different standards for designing their component.
- If there is a need to exchange component design description among designers working on different components in such a case, it becomes difficult as other designers might not be well versed with the design description standards of other teams.
- IP XACT is a very good solution in such a case, as designers can just use IPXACT as their standard of design description. Since an XML file is easy to create, edit and visualize it is not a time or effort consuming task to create an IPXACT description of a component.

4.2 IPXACT for inter-organization design sharing:

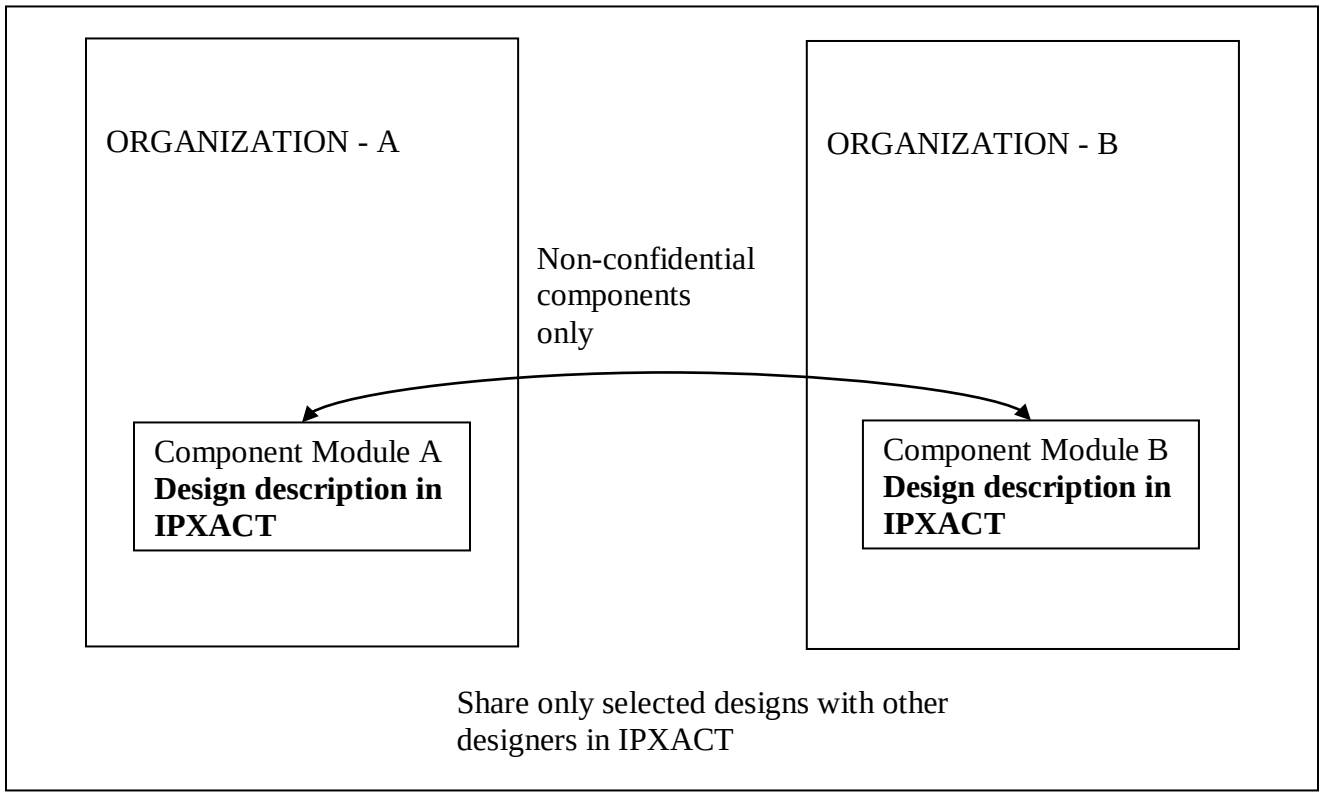


Fig 6

- Different vendors can use IPXACT to exchange design descriptions using the IPXACT standard
- Currently each vendor uses their own design description standard which makes it difficult to share designs with their clients or other vendors.
- This requires use to special applications that can understand design description coming in from other vendors
- IPXACT can be used in the same way as it can be used within an organization, but in case of sharing it with other vendors only those components which are NOT confidential can be easily using IPXACT.
- Since XML is a very easy format, non-confidential components can be easily identified and only those can be shared with external entities.

5. Automatic Code Generation Process:

5.1 Automatic code generation:

Automated code generation simply refers to “writing code that writes code”. The input for an automatic code generator can be any high level model or design that is easy enough to create. The input model just needs capture the requirements of the output that the user intends to generate. The basic purpose of any automatic code generator is to make the software development faster and more accurate.

5.2 Similarity to Language Compilers:

Compilers take computer programs as inputs and generate code that can be executed by the hardware. It provides the users with the luxury of writing of computer programs in high-level languages that are more suited to a human computer programmer. Automated code generators work on the same principle but at a higher level of abstraction. Like compilers they interpret a document or a piece of code written in a user-friendly way and output computer programs that are accurate, well-formed and would have otherwise taken a longer time if written manually. The different phases of a compiler are the Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Analysis, Machine Code Generation and Assembly and Linking. Any automatic code generator will have to go through similar steps though not necessarily all of the above[4].

5.3 Automatic code generation from XML:

XML documents can be ideal inputs for automatic code generators such as the RTL generator. The reasons for that are as follows:

- XML structure is simple and easy to understand.
- Every tag has an associated meaning
- Parsing an XML is relatively fast

– The structure of XML facilitates creation of Syntax Trees

5.4 Phases of Automatic code generation:

The figure below (Fig 7) represents the general architecture of any automated code generator.

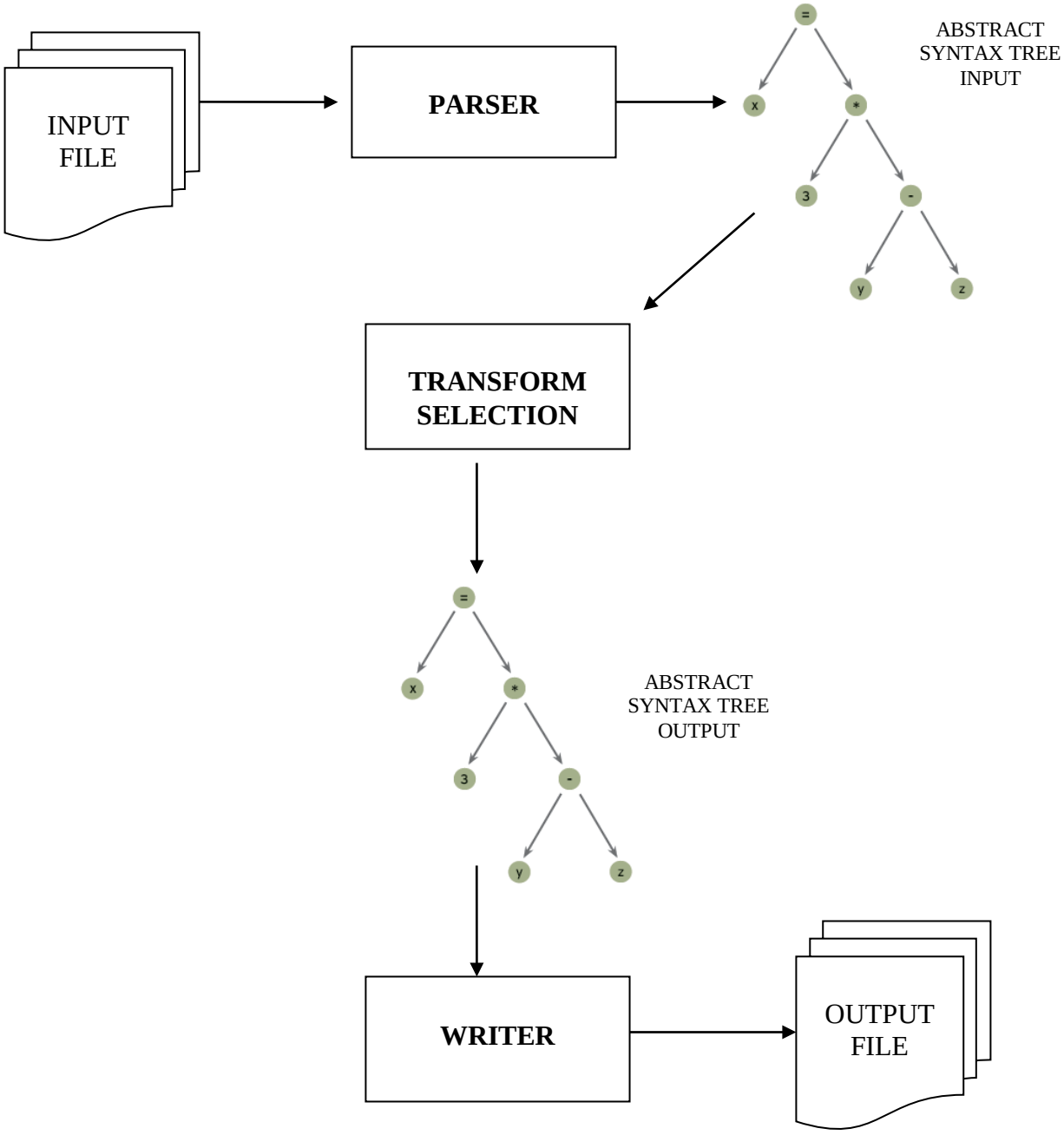


Fig 7

The phases of an automated code generator are[3]:

- **Parsing**
  - The input file is parsed by the parser token by token.
  - It creates an input syntax tree using the tokens
  - It checks whether the input file is syntactically correct
  - Parsing can be either tree-based or event-based.
  - Tree based parsing constructs the entire tree for the parsed document which is then used for further processing.
  - Event-driven parsers process any recognizable construct as soon as it is encountered.
  - The parse-tree is an in-memory entity that holds the information of the input file in a data-structure.
  - The parse-tree is absolutely vital as it is the data-structure on which all the further processing takes place.
- **Transformation and Selection**
  - This phase converts the input parse tree into some other form for instance optimization.
  - Selection is browsing the parse tree and actually generating the code based on the information from the tree.
- **Writer**
  - The writer interprets the output syntax-tree to produce the final code.
  - The writer is the reverse of the parser in the sense that it generates code from the tree.

7. Automatic RTL Generator:

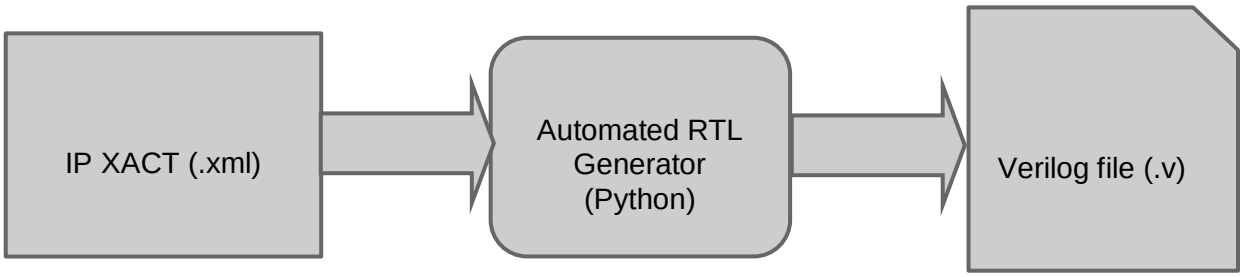


Fig 8

7.1 Need for automated RTL Generation:

Most of the processors designed today are complicated and involve huge intricate logic design.

Designers would therefore, like to focus in the logic than worry about writing the code to implement the design. IPXACT is a recognized standard (IEEE 1685-2009) which basically is an XML-schema for documenting the Intellectual Property (IP). Designers can document their design in IPXACT format, without worrying about writing the code and implementation details.

The RTL generator will parse this IPXACT document and generate Verilog code after interpreting the document. This would save the designers a lot of time and effort involved in writing the Verilog code.

It would also ensure that the code generated is error-free and follows certain set of standards. The need for an automated RTL generator from IPXACT thus arises from the requirement that the designer be focused only on the design (XML).

Figure above represents a high-level view of what the RTL generator is all about.

7.2 Phases of Automatic code generation as they apply to the RTL Generator:

The Automated RTL generator is similar to the one discussed in the previous section in its architecture and does through similar phases (Fig 9).

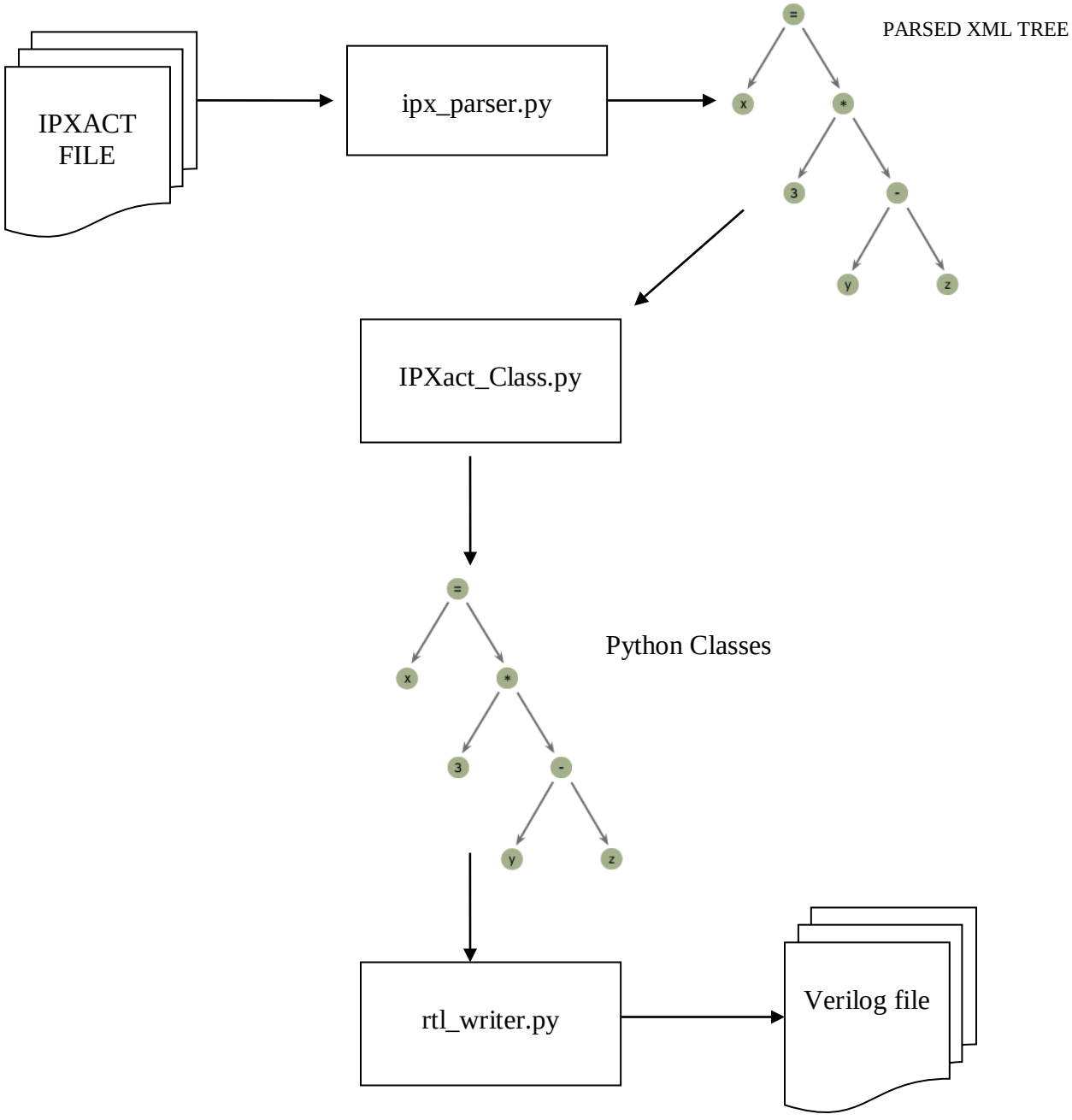


Fig 9

7.3 RTL Generator Modules:

The RTL generator is the main python module that will create the verilog file. The RTL generator will parse the IP XACT xml file and consolidates all the information from it. Depending upon the information in the IP XACT description file, corresponding details are extracted and stored in the IPXact Class.The RTL generator will follow a model view controller (MVC) model to separate the functionalities.

Model – Will be the storage for the RTL Module.

View – Will be the verilog code that will be generated.

Controller – Will be the parser that controls how the IP XACT is parsed

The model and view do not interact directly with each other. They can interact with each other only through the controller. This ensures that any changes to the view do not affect the model and vice-versa.

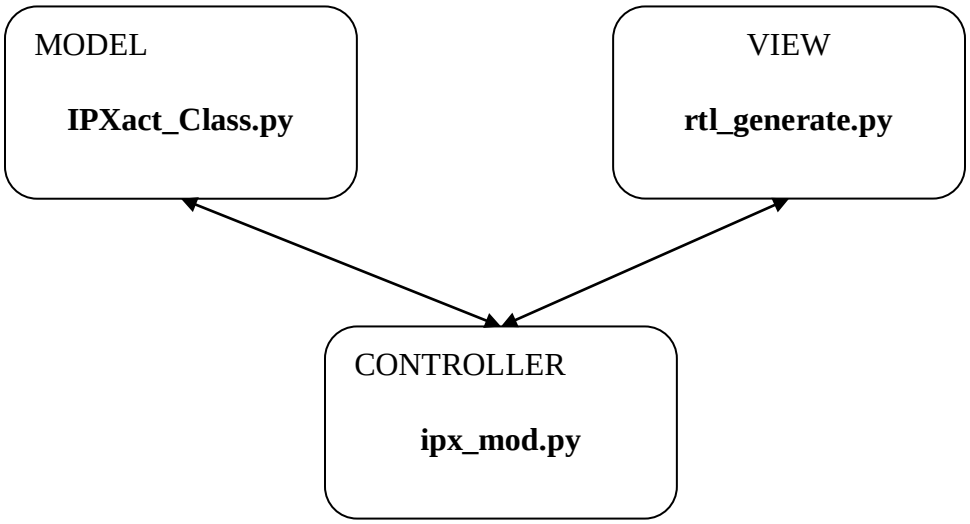


Fig 10

The above figure represents the model view architecture of the RTL generator.

The RTL generator will comprise of the following 3 python modules:

- ipx_mod.py
- IPXact_Class.py
- rtl_generate.py

7.3.1 ipx_mod.py :

This is the module that would parse the IPXACT file and interpret the design specified in the file. It will then create an instance of the 'IPXact' class and instantiate it with information extracted from the IPXACT file. It also instantiates the RTLWriter class which is responsible for writing the code. It will also perform housekeeping activities that will generate details that will be used in the final verilog code.

It is the link connecting the ‘View’ and the ‘Model’ and acts as the ‘Controller’ in the overall architecture of the system.

```
IPXactMod = getIpxactData(sys.argv[1])
Writer = RTLWriter(IPXactMod)
Writer.writeVerilog()
Writer.genVerilog()
```

The above snippet shows how the ipx_mod.py program instantiates the IPXact Class and then passes it to an instance fo the RTLWriter class. It takes care of the fact that there is no direct interaction between the writer and the model.

7.3.2 IPXact_Class.py:

This module contains a class that stores the module description. This class will act like the repository of the information parsed from the IPXACT file. As mentioned above it will be instantiated in rtl_generate.py.

All the information from the IPXACT file is stored in the IPXact class. There are 7 main classes representing the 7 top-level components in the IPXACT file. The information from each of these components in stored in their respective classes. The structure of the class looks as below:

```
class IPXact:
          def __init__(self):
                    self.bd = busDefinition()
                    self.adn = abstractDefinition()
                    self.cd = componentDescription()
                    self.dd = designDescription()
                    self.adr = abstractorDescriptor()
                    self.gc = generatorChainDescription()
                    self.dc = designConfigurationDescription()
```

7.3.3 rtl_writer.py

This is the module that performs formatting and writing of the verilog code. This instantiates a class of type 'RTLWriter' that takes as input an object of type ‘IPXact’ that has already been instantiated by rtl_parse. This class has the intelligence to browse through the ‘IPXact’ class and identify what code needs to be generated. Despite its intelligence, it is entirely dependent on the ‘IPXact’ class, if this class is not generated correctly the Verilog code generated will not be accurate.

7.4 RTL Generator Architecture (detailed):

7.4.1 Parsing – Creation of State Trees:

As mentioned earlier the parser is a a tree-based parser. The parsing tree is built in the form of a python class – IPXact_Class. Every sub-tree is a sub-class of the IPXact_Class. The seven top-level components will be direct subclasses of the IPXact_Class.

This high-level syntax tree can be represented as follows:

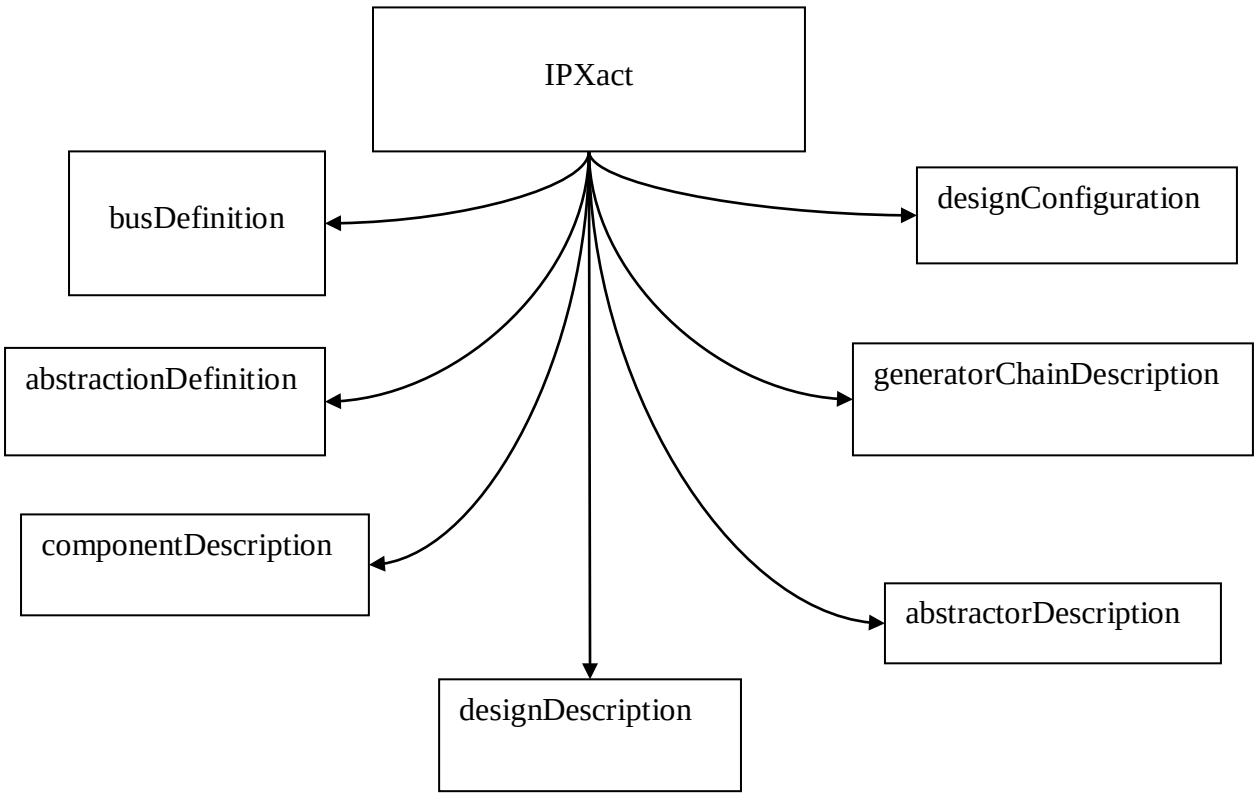


Fig 11

Each sub-tree in the above diagram has its own syntax sub-tree.

For instance consider the syntax-tree for busDefinition (Fig 12):

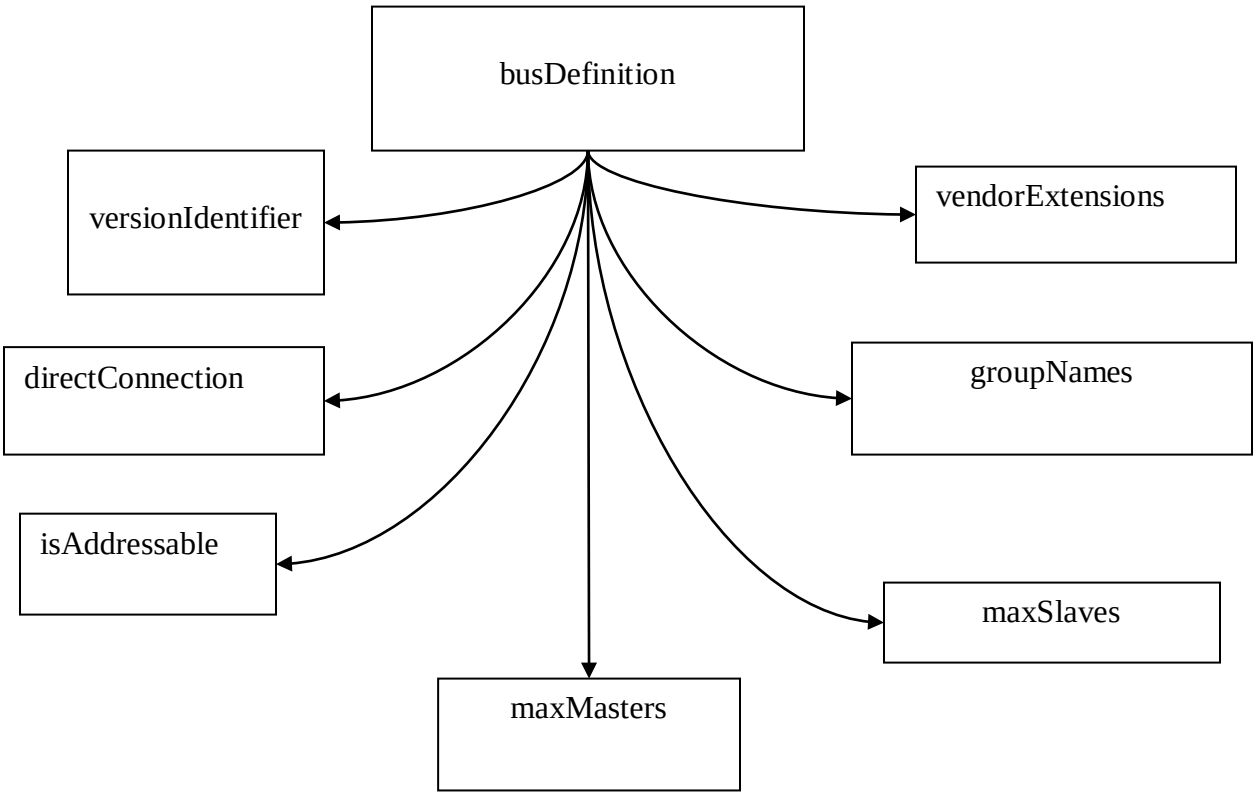


Fig 12

- Each sub-component is a syntax-tree in itself
- These syntax- trees could be single-nodes (leaf nodes) or they could be complex trees with sub-syntax-trees
- The leaf nodes carry information from the IPXact document
- This information is what gets finally transferred to the verilog code

7.4.3 Overview of syntax-trees generated for IPXACT documents:

Consider an IPXact XML document with all of its elements. The state tree created by such a document is so huge that it cannot be drawn diagrammatically. Below are some interesting statistics about the abstract-syntax trees for the top-level IPXACT components:

Top Level Component	Total Nodes	Max Depth
Bus Definition	11	2
Abstraction Definition	112	7
Component Description	235	9
Design Description	36	3
Abstractor Description	36	3
Generator Chain Description	22	3
Design Configuration Description	24	3

7.4.4 Comparison of syntax-trees generated for IPXACT documents:

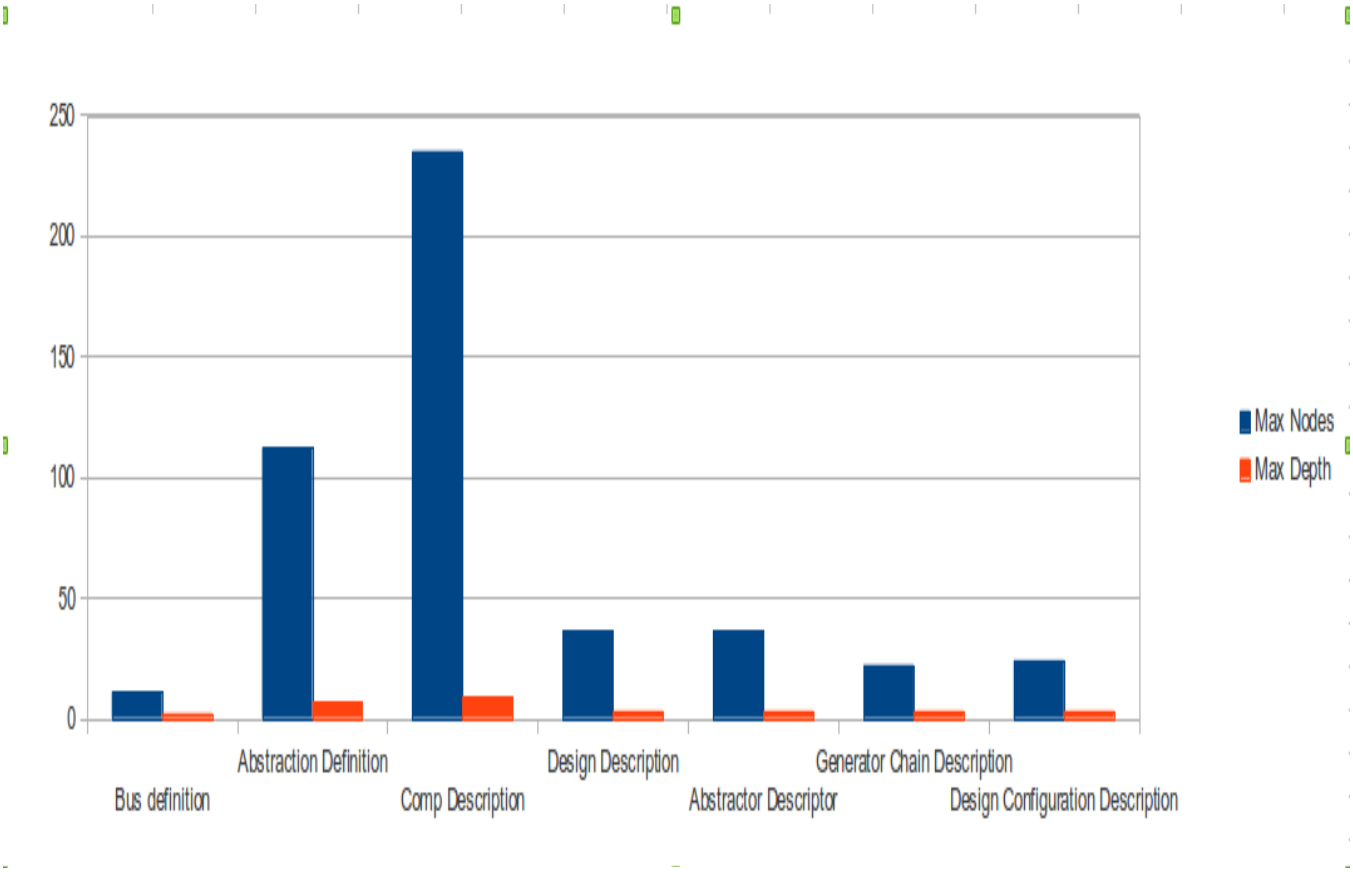


Fig 13

- X-axis represents various top-level components as the following tuple - (Max Node, Max Depth)
- Y-axis represents the values for the tuples on the X-axis:
    - Blue = Max Nodes
    - Orange = Max depth
- As can be observed from the graph, most of the information is stored in the syntax trees for the Abstraction and Component definition components
- Component Description contains the maximum information and hence is way above all other top-level components in terms of the depth of the syntax-tree and the total number of nodes in the tree.

7.4.5 Parsing the state trees using Finite State Machines (FSM):

To generate code based on the syntax trees generated, finite state machines (FSMs) are used. In this case the functional view of Finite State Machines is being considered. Each state of the machine is identified with a function. Consider the finite state machine of the highest level state-tree (Fig 14) for the seven top-level components.

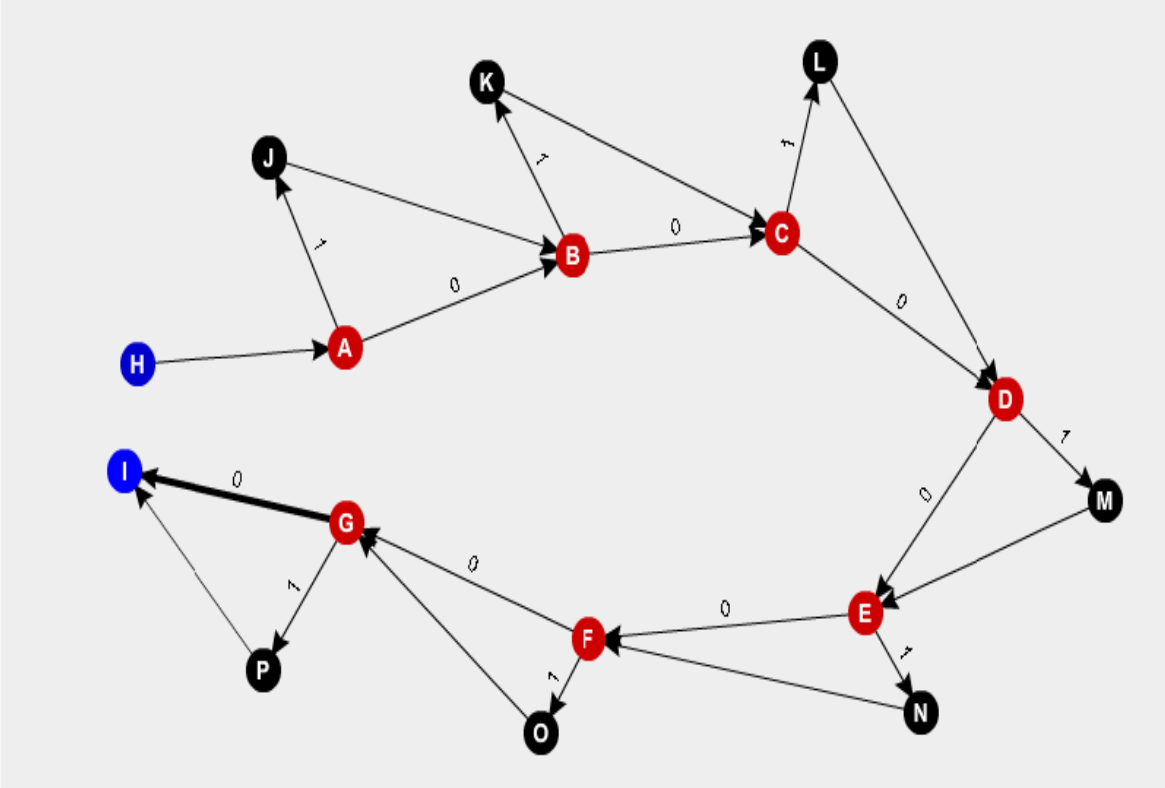


Fig 14

- The red nodes represent states of the FSM in which the presence of the top level component is checked.
- The black nodes represent the processing of the corresponding top level component.
- The blue nodes H and I represent the start and end states respectively.

- From the start state the machine transitions to each of the red state to check the presence of that component if its present (represented by 1) it transitions to the state corresponding to the state in which processing of the component is done.
- If the component is not present the machine transitions to the next red node that is state to check the presence of the next top-level component.
- From the black node the machine always transitions to the next red node to check if that component is present, except in the last case

The FSM explained above is one of the numerous FSMs that are used to generated the verilog code from the state trees that are generated after the parsing phase. The collated output of all the FSM's is the final verilog code.

7.4.6 Writing the verilog file based on the FSM outputs:

The writer makes use of the FSM to generate the verilog code. Each state that the FSM transitions to has the task of generating its piece of code that goes into the final verilog file.

For example, if the finite state machine transitions to the port generating state, the writer will generate the verilog code for various inputs and outputs for that port.

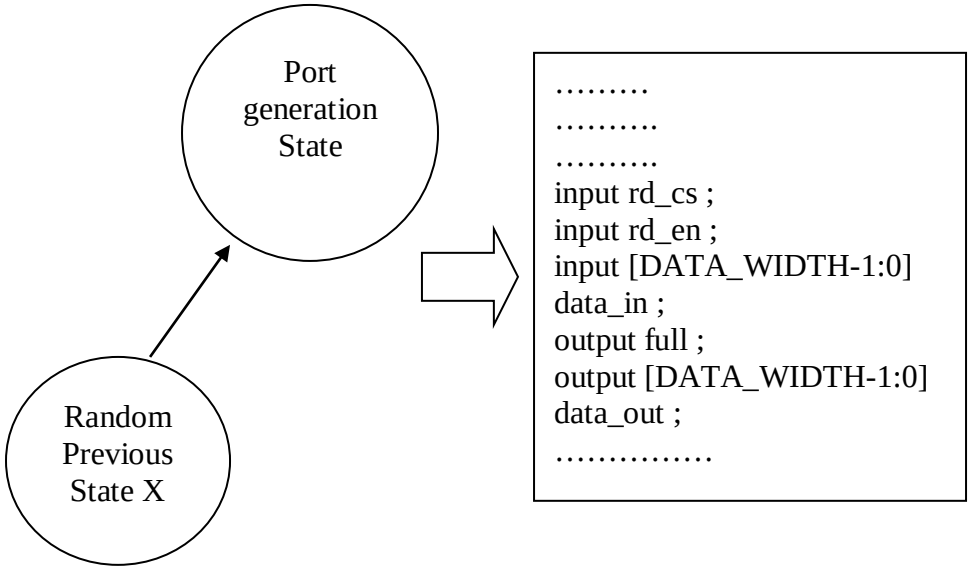


Fig 15

When all the FSM reach their final state after going through various cycles, the final verilog file is generated. Any FSM when implemented in code and in its simplest form will be a set of “if …else” statements. Consider the figure above, the port generation state is a FSM in itself, it works as follows:

- Is the port wire or transactional port?
    - If “wire”, process wire port
    - If “transactional” process transactional port

- If “processing wire port”
    - Loop through all ports
        - If “input” port , write “input” to file along with port details
        - If “output” port , write “output” to file along with port details

- If “processing transactional port”
    - Loop through all ports
        - If “input” port , write “input” to file along with port details
        - If “output” port , write “output” to file along with port details

- Check “End of portlist”
    - If “End of portlist”
        - Move to processing of next component in the AST
    - If “Not end of portlist”
        - Go Back to processing port state

The finite state machine explained above would output all the ports that have been collected from the IPXact then stored in the state trees. The source of all data for the FSMs are the state trees and state trees only.

8. Test Plan and Results:

The RTL generator was tested by running it with a number of IPXACT files and creating Verilog files. The generated Verilog files were simulated and ran using test benches. The goal was to ensure that the code generated was synthesizable and accurate.

8.1 Generation of Verilog code for a FIFO Module:

Using an IPXact version of a basic First In First Out (FIFO) module. The Verilog code for the FIFO module was generated using the Automated RTL Generator (ARG).

The FIFO module can be described using the below diagram (Fig 16):

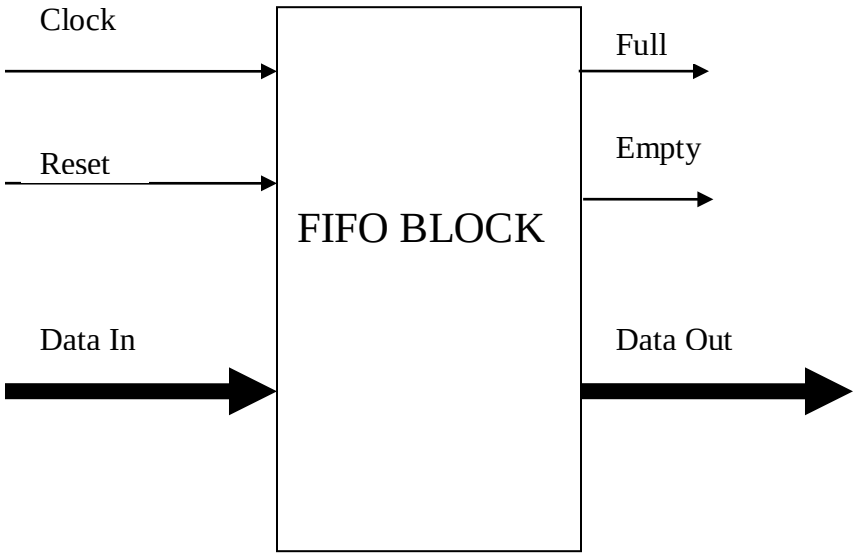


Fig 16

The FIFO Block in the diagram has 3 input signals:

- Clock – The FIFO block will function based on this clock signal
- Reset – Used to reset the FIFO
- Data In – Input data to the FIFO queue,8 bits [7:0]

The FIFO block has 3 output signals:

- Full – Signals if the FIFO is full
- Empty – Signals if the FIFO is empty
- Data Out – Output data from the FIFO queue, 8 bits [7:0]

All the above information is represented in an IPXact (IEEE 1685-2009) standard XML file. The name of the input file used is “fifo.xml”. For a complete description of the file. Please refer the appendix.

The verilog file generated after execution of the ARG program is “fifo.v”. Refer the appendix for the complete verilog file. Once the verilog file is generated it is tested using a test bench written for the file. The various test conditions that the test-bench verifies are as follows:

- Adding data to the FIFO
- Popping data from the FIFO
- Check for FIFO being full
- Check for FIFO being almost full
- Check for FIFO being empty

All the above conditions were tested with positive results.

The details of the test process are as follows:

The ARG is run as follows:

$python ipx_mod <*IPXact xml file name*>

Example:

$python ipx_mod fifo.xml

This creates the verilog file **fifo.v**. This file was run using the test bench file fifo_tb.v . The open source "icarus" tool was used for the same. Refer appendix for the complete test bench

$iverilog fifo_tb.v fifo.v

The above command generates the 'a.out' file. This file is the executable file which when executed generates the results.

It is executed as follows:

$./a.out

The execution of the command also leads to creation of the

Below is a screen shot (Fig 17) of the execution of test bench for the generated verilog. It also shows that a waveform file "**fifo_waveform.vcd**" is generated.

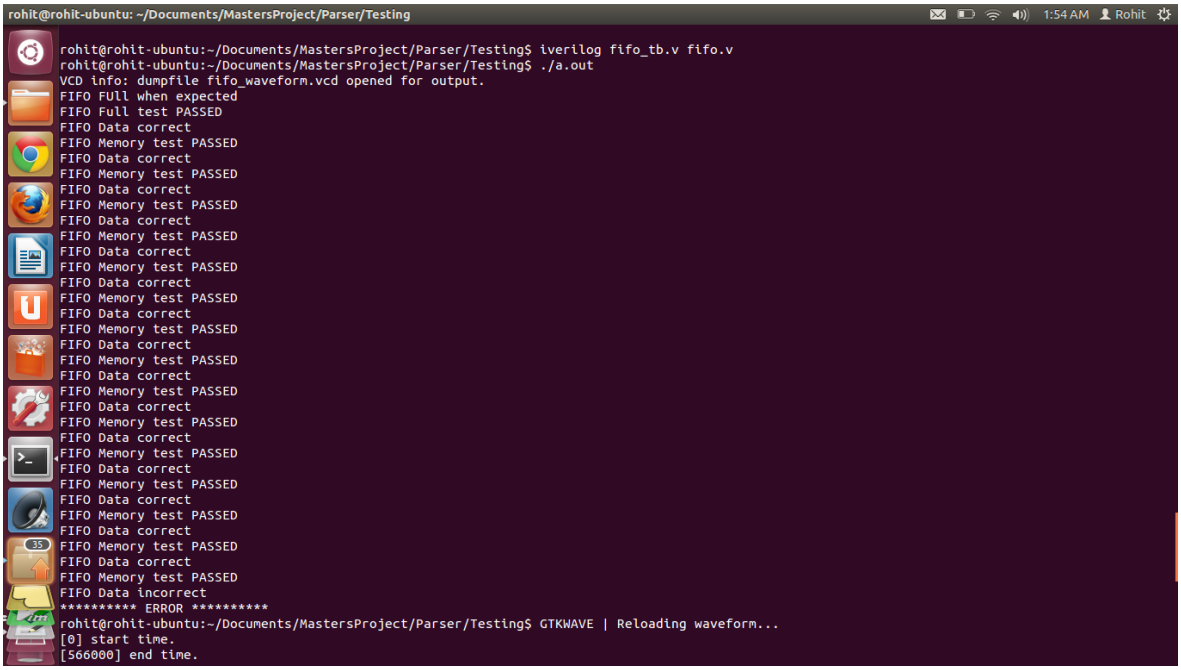


Fig 17

The open-source tool "gtk" was used to view the waveforms. The gtk tool is used as,

$gtk fifo_waveform.vcd

The above command when executed opens the gtk GUI, which shows the waveforms generated as a result of the tests conducted by the test bench. The GUI graphically demonstrates the different results that the verilog file fifo.v would produce from the inputs received from the test bench file fifo_tb.v

The waveform for the FIFO was as shown below (Fig 18):

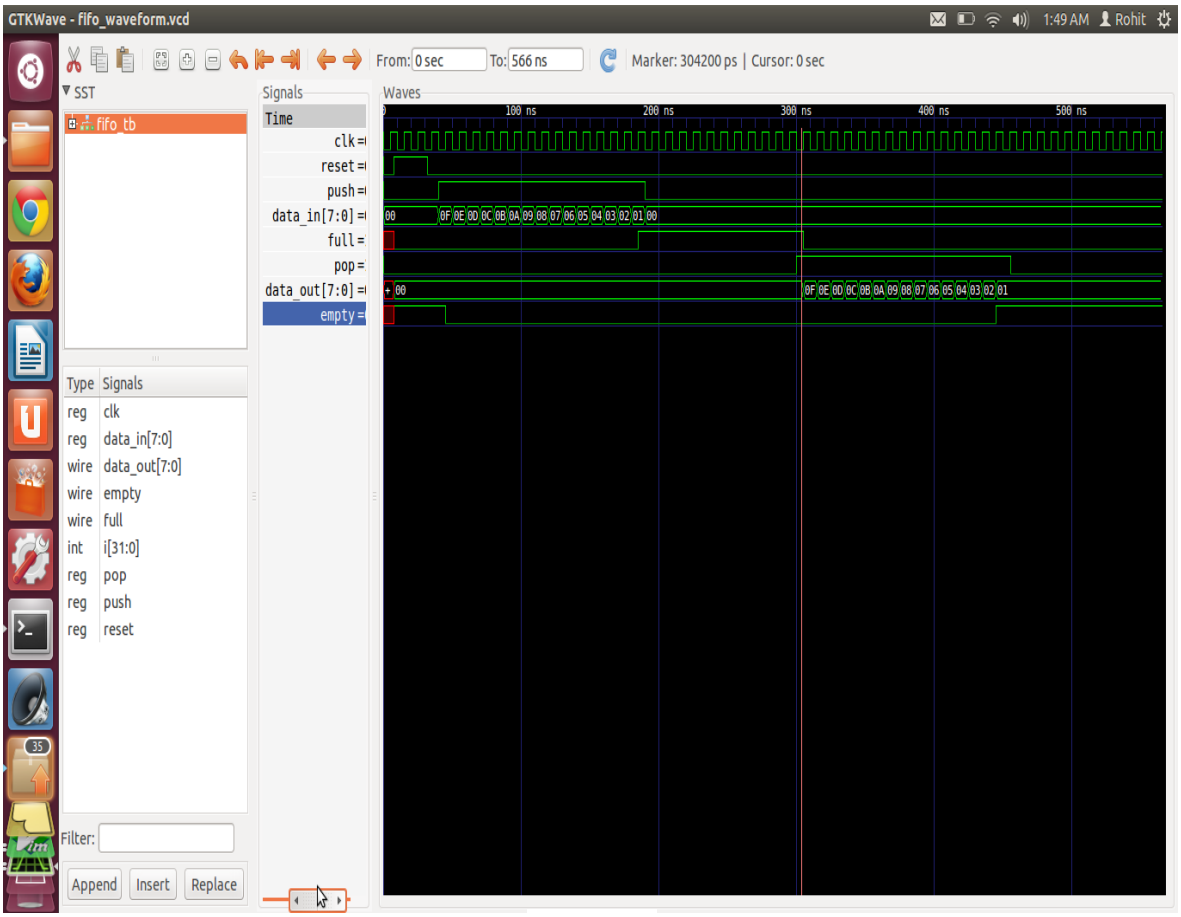


Fig 18

Observations in the waveforms:

- The first signal is the clock signal. Everything happens on the negative edge of the clock.
- The second waveform which is the reset signal, when high, causes the push signal to go low. It means the FIFO is reset and no data is pushed into the queue.
- The third waveform is the push signal, when it is high data gets pushed into the FIFO queue. This can be seen in the fourth waveform which is the data_in signal
- The fifth signal is the full signal, it goes high when the queue reaches its maximum size.

- The sixth waveform is the data_out signal it is high for 10 cycles and goes low when the queue is empty. It is at this time that the empty signal which is the seventh waveform, goes high.
- The seven waveforms are thus in accordance with the all the test cases that the test bench intended to test.
- This verifies the fifo code generated and also the test bench written for the same.

9. Existing tools for RTL generation from IPXACT

IPXACT is a new standard and is not yet a house-hold name in the world of hardware design. There are not many tools available for IPXACT pre and post-processing. Based on the research below are a few IPXACT processing tools. Two notable software tools are the

- Agnisys : IDesignSpec[6]
- Magillem Register View[7]

Their features , including strengths and weaknesses are also mentioned below.

9.1 Agnisys: IDesignSpecTM[7]

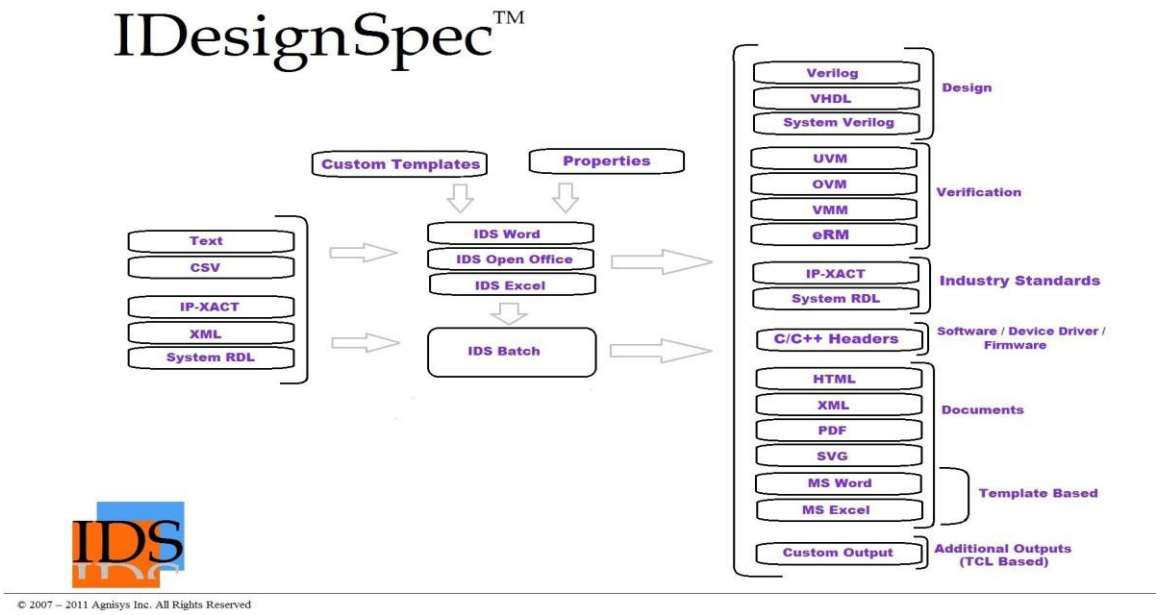


Fig 19

– Allows an IP, chip or system designer to create the register map specification once and

automatically generate all possible views from it.

- Various outputs are possible such as UVM, OVM, RALF, SystemRDL, IPXACT etc. User defined outputs can be created using Tcl or XSLT scripts.
- It is available as a plug-in for popular editors (MS-WORD, MS- EXCEL, Open Office etc)
- Works on a register level.

Advantages over this product:

- Since this only works for registers our tool can be used on a other levels like a chip or a module.
- Also we could just generate RTL directly from IPXACT
- This is a link of the demo:
- http://www.youtube.com/watch?feature=player_embedded&v=3AJlNgnC1Aw

9.2 MRV: Magillem Register View[8]

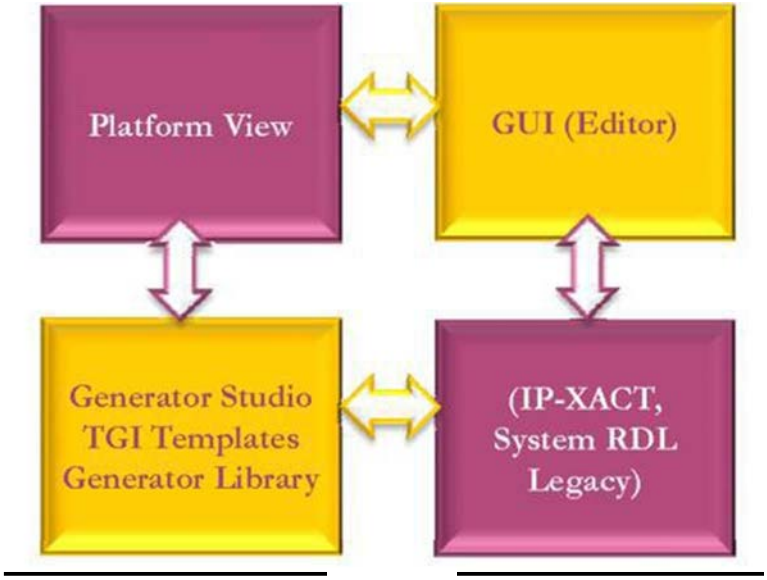


Fig 20

- This also works on a register level:
- Integration of configurable IP register descriptions (containing thousands of registers), delivered by multiple different vendors, in a hierarchical system
- Import and capture register descriptions defined in different formats (CSV,
- Excel, XML…) into a single database Reduce errors and misalignments using synchronized database and comprehensive consistency checks.
- Tight link with HW interface definitions and platform connectivity, to generate correct and aligned system map definitions for SW development
- Supports IEEE-1685.

10. Conclusions:

Code generators are really useful when designed correctly. The best use of code generators can be made if they are created with certain principles in mind. Below is the list if these principles accompanied by how the automated RTL generator tries to incorporate them in its design:

- **Regularity**[6] – All rules for generating code must be laid down beforehand and must be followed rigorously. The user of the code generator must be assured that if something is done in a way once, it will always be done in the same way always. Such consistency ensures the tool is easier to use and the learning curve consequently becomes less steeper.

  With respect to the automatic code generator this principle has been incorporated by strictly adhering to the IEEE 1685-2009 standard.

- **Orthogonality**[6] – Othogonality refers to division of responsibilities in the code generator. It should be possible to separate different concerns in the code generator. For instance a change in the parsing process should not affect the transformation, selection and writing process and vice versa.

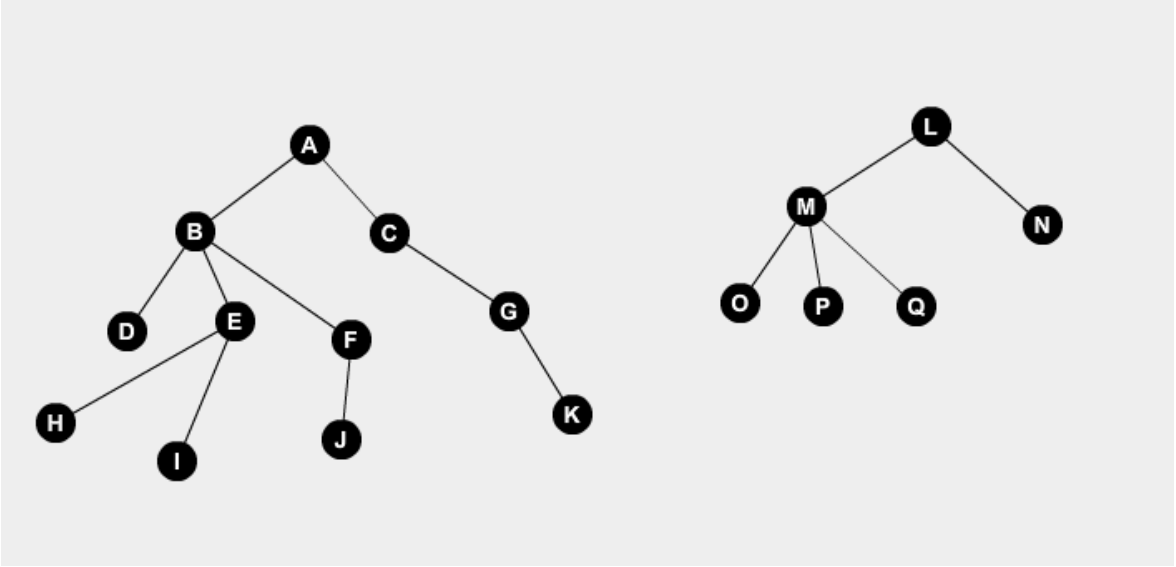
  The automatic code generator uses the MVC mode described in section 7.3 to separate responsibilities. For instance, any modification to the IPXact class does not affect the writer

- **Composability**[6] – It refers to the ability to perform the same function on different types of modules. For example, the function that displays the list of ports should be able to display a list of wire ports and also transactional ports. If the above two principles are taken care of, its easier to implement the composability principle.

11. Future Work:

11.1 A Faster way to perform the code generation process:

- IPXact document contains a lot of optional elements and attributes.
- These optional elements and attributes do not contribute to the final code, even if they do they are mostly cosmetic contributions in the form of comments and additional information.
- If these elements are not added to the state trees, the size of the state trees is greatly reduced and they are created faster.
- This effect is cascaded into the parsing and writing stages too.
- Smaller state-trees mean faster parsing and, hence, faster code generation.



Tree with optional elements Fig 21 Tree without optional elements

- The above figure shows two trees the left one is including optional attributes, while the right one is without.
- The right tree is obviously faster to create and then parse.
- This type of tree clipping is optional and is incorporated by providing an option to the user to clip the trees.

11.2 Regression testing using various IPXACT files:

- The project thus aims to substantially the reduce the work for designers and also improve the accuracy of design and lessen design errors.
- There haven't been particular efforts directed towards the efficiency of the process yet.
- Regression testing with different sizes of IPXACT XML files can indicate the performance of the application.

11.3 Optimization of Verilog Code:

- The Verilog generated is not guaranteed to be optimized.
- Further research needs to be done on ways to generate the most optimized code from the application itself.

References:

[1] http://www.accellera.org/activities/committees/ip-xact

[2] IEEE 1685-2009 IPXACT, Feb 18, 2010

[3] Code Generation using XML based document transformation, Craig Cleveland, Soumen Sarkar

[4] Basics of Compiler Design, Torben Ægidius Mogensen

[5] An Adaptive and Efficient XML Parser Tool for Domain Specific Languages, W. Jai Singh, S. Nithya Bala. International Journal of Scientific & Engineering Research Volume 2, Issue 4, April-2011

[6] Compiler Design Principles, William A. Wulf, July 1981

[7] http://www.agnisys.com/products/ids

[8] http://www.magillem.com/eda/mrv-magillem-register-view

Appendix:

A.1 Version Control:

The version control for the code was done using Git. The remote Git folder was then synced with the cloud on Dropbox.

A.2. fifo.v:

```
//FIFO Verilog

module fifo
(
        clk,
        reset,
        push,
        pop,
        data_in,
        data_out,
        full,
        empty
);
        parameter PTR       = 4;
        parameter SIZE      = 1 << PTR;
        parameter WIDTH = 8;

        input                   clk;
        input                   reset;
        input                   push;
        input                   pop;
        input   [(WIDTH-1):0]       data_in;
        output  [(WIDTH-1):0]       data_out;
        output                  full;
        output                  empty;

        reg     [(WIDTH-1):0]       memory      [0:(SIZE-1)];   // Memory of FIFO
        //reg   [7:0]   memory      [0:15]; // Memory of FIFO
        reg     [(WIDTH-1):0]       data_out;               // Data Output
        reg     [(PTR-1):0]     rd_ptr;                 // Read Pointer
        reg     [(PTR-1):0]     wt_ptr;                 // Write Pointer
        wire                    read;                   // Flag for valid read
        wire                    write;                  // Flag for valid write

        assign  full    = ((wt_ptr + 1'b1) == rd_ptr);
        assign  empty   = (wt_ptr == rd_ptr);
```

```
    assign  read    = !empty & pop;
    assign  write   = !full & push;

    always @ (posedge clk or posedge reset)
        if (reset == 1'b1)
            rd_ptr <= {(PTR){1'b0}}; //concatenation operator
        else
            rd_ptr <= (read == 1'b1) ? rd_ptr + 1'b1 : rd_ptr;

    always @ (posedge clk or posedge reset)
        if (reset == 1'b1)
            wt_ptr <= {(PTR){1'b0}};
        else
            wt_ptr <= (write == 1'b1) ? wt_ptr + 1'b1 : wt_ptr;

    always @ (posedge clk or posedge reset)
        if (reset == 1'b1)
            data_out <= {WIDTH{1'b0}};
        else
            data_out <= (read == 1'b1) ? memory[rd_ptr] : data_out;

    always @ (posedge clk or posedge reset)
        memory[wt_ptr] <= data_in;

endmodule
```

A.3. fifo_tb.v:

```
// FIFO Testbench

`timescale 1ns/10ps

module fifo_tb ();

    parameter PTR       = 4;            // Size of the read and write pointers
    parameter SIZE      = 1 << PTR;     // Depth of the FIFO. 2^(pointer_size)
    parameter WIDTH = 8;                // Width of the data

    reg                 clk;            // Clock signal. 100MHz
    reg                 reset;          // Reset signal. Resets the pointer value to 0
    reg                 push;           // Write enable signal
    reg                 pop;            // Read enable signal
    reg   [(WIDTH-1):0]     data_in;        // Input data
    wire  [(WIDTH-1):0]     data_out;       // Output data
    wire                full;           // Flag to indicate FIFO Full
    wire                empty;          // Flag to indicate FIFO Empty

    integer             i;
```

```
fifo fifo
(
        .clk            (clk),
        .reset          (reset),
        .push           (push),
        .pop            (pop),
        .data_in        (data_in),
        .data_out       (data_out),
        .full           (full),
        .empty          (empty)
);

// Block to generate Waveform file
initial
begin
        $dumpfile("fifo_waveform.vcd");     // System function to create a waveform file (.vcd)
        $dumpvars(0,fifo_tb);           // Name of the module that is to be plotted on the waveform
end

// Block to create a 100MHz clock
initial
begin
        clk = 1'b0;                     // Initialize the clock value to 0
        forever #5 clk = ~clk;          // Keep switching the clock after every 5ns
end

// Actual Test Cases
initial
begin
        // Initialize all the signals
        reset = 1'b0;
        push = 1'b0;
        pop = 1'b0;
        data_in = 0;

        // Apply reset pulse
        #8  reset = 1;
        #24 reset = 0;

        // Test to check the FIFO full signal
        i = 15;
        repeat (SIZE)
        begin
                @ (negedge clk);
                push = 1'b1;
                data_in  = i;
                i = i - 1;
        end
```

```
push = 1'b0;
@ (posedge clk) #1;
if (full !== 1'b1)
begin
        $display ("FIFO expected to be Full");
        $display ("********** ERROR **********");
end
else if (full === 1'b1)
begin
        $display ("FIFO FUll when expected");
        $display ("FIFO Full test PASSED");
end

#102

// Test to check the FIFO Empty signal
i = 15;
repeat (SIZE)
begin
        @ (negedge clk);
        pop = 1'b1;
        @ (posedge clk);
        #1
        if (data_out !== i)
        begin
                $display ("FIFO Data incorrect");
                $display ("********** ERROR **********");
        end
        else if (data_out === i)
        begin
                $display ("FIFO Data correct");
                $display ("FIFO Memory test PASSED");
        end
        i = i - 1;
end
pop = 1'b0;
@ (posedge clk) #1;
if (empty !== 1'b1)
begin
        $display ("FIFO expected to be Empty");
        $display ("********** ERROR **********");
end
else if (full === 1'b1)
begin
        $display ("FIFO Empty when expected");
        $display ("FIFO Empty test PASSED");
end

#100 $finish;
```

```
            $display ("Testbench Passed");
        end

endmodule
```