Formal Verification Using SLEC(Sequential Logic Equivalence Checker)

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

(Embedded Systems)

By

Shreya Panjvani

(15MECE15)

Electronics & Communication Engineering Department

Institute of Technology-Nirma University

Ahmedabad-382481

MAY 2017

Formal Verification Using SLEC(Sequential Logic Equivalence Checker)

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

(Embedded Systems)

By

Shreya Panjvani
(15MECE15)

Under the guidance of

External Project Guide:	**Internal Project Guide:**
Mrs Swati Garg	**Dr.N.P.Gajjar**
Member Of Consulting Staff, SLEC PV,	Assistant Professor, EC Department
Mentor Graphics India Pvt Ltd,	Institute of Technology,
Noida.	Nirma University, Ahmedabad.

Electronics & Communication Engineering Department

Institute of Technology-Nirma University

Ahmedabad-382 481

MAY 2017

Declaration

This is to certify that

a. The thesis comprises my original work towards the degree of Master of Technology in Embedded Systems at Nirma University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- Shreya Panjvani
15MECE15

Disclaimer

NIRMA UNIVERSITY
INSTITUTE OF TECHNOLOGY
NAAC ACCREDITED ‘A’ GRADE

Certificate

This is to certify that the Major Project entitled **“Formal Verification Using SLEC(Sequential Logic Equivalence Checker)”** submitted by **Shreya Panjvani (15MECE15)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge,haven’t been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Dr N.P.Gajjar
Internal Guide

Dr. N.P. Gajjar
Program Coordinator

Dr.N.P.Gajjar
Section Head, EC

Dr Alka Mahajan
Director, IT

Certificate

This is to certify that the Major Project (Phase- I) entitled **"Formal Verification Using SLEC(Sequential Logic Equivalence Checker)"** submitted by **Shreya Panjvani(15MECE15)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Mrs Swati Garg
Member Of Consulting Staff, SLEC PV,
Mentor Graphics India Pvt Ltd,
Noida

Acknowledgements

I would like to express my gratitude and sincere thanks to **Dr. N.P.Gajjar**, PG Coordinator of M.Tech Embedded Systems program for allowing me to undertake this thesis work and for his guidelines during the review process.

I take this opportunity to express my profound gratitude and deep regards to **Dr N.P.Gajjar**, guide of my major project for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I would take this opportunity to express a deep sense of gratitude to **Mrs Swati Garg**,Member Of Consulting Staff, SLEC PV,Mentor Graphics India Pvt Ltd. for her cordial support, constant supervision as well as for providing valuable information regarding the project and guidance, which helped me in completing this task through various stages.

I am obliged to team members of SLEC PV, Mentor Graphics India Pvt Ltd. for the valuable information provided by them in respective fields. I am grateful for their cooperation during the period of my assignment.

Lastly, I thank almighty, my parents, brother and friends for their constant encouragement without which this assignment would not be possible.

- Shreya Panjvani
15MECE15

Abstract

Verification of different designs using automated tools has become the widely used methodology for the Electronic Design Automation(EDA) industry. SLEC(Sequential Logic Equivalence checker) is one such tool which uses formal verification.
Formal verification is a method of proving or disproving the functionality of any design using Formal methods. Formal Verification does not require input vectors like simulation. It verifies two designs by comparing boolean equation of both the designs,generated using formal algorithms.
SLEC(Sequential Logic Equivalence Checker) is a sequential equivalence checker which compares two designs:specification design (SPEC) and Implementation design (IMPL) which may be structurally not equivalent. Formal Verification using SLEC in HLS(High Level Synthesis) is the main motto of this thesis.
The flow of HLS using Mentor's Catapult and SLEC(Sequential Logic Equivalence Checker) was studied and performed. Different features of SLEC HLS flow were tested. Bug finding and reporting was done in Bugzilla. Automation work required for the tool was done using scripting.
SLEC uses formal verification which leads to better coverage,better resource allocation,lower power consumption and lesser area. SLEC can prove two designs formally equivalent inspite of structural differences.

Contents

Declaration **iii**

Disclaimer **iv**

Certificate **v**

Certificate **vi**

Acknowledgements **vii**

Abstract **viii**

List of Figures **xiii**

1 Introduction **1**

1.1 Motivation 1

1.2 Objective 1

1.3 Requirements 2

2 An Introduction to Formal Verification **3**

2.1 An Introduction to verification 3

2.2 Formal Verification 4

2.2.1 Formal Equivalence Checking 4

3 Overview of SLEC 6

3.1 Introduction to SLEC . . . 6

3.2 Advantages of SLEC . . . 7

3.3 Ability of SLEC . . . 8

3.3.1 Equivalence Checking Compatibility . . . 8

3.3.2 Refinement Verification . . . 8

3.3.3 Basic operation of SLEC . . . 9

3.4 Setup File . . . 11

3.5 SLEC-HLS . . . 12

3.5.1 SLEC-HLS Flow . . . 13

4 Testing the all piped feature of SLEC HLS flow 21

4.1 An Introduction to pipelining . . . 21

4.2 Problem Statement . . . 23

4.3 Initation Interval value . . . 23

4.4 Algorithm For the Problem . . . 23

4.5 Problems . . . 24

4.6 Summary of the testing . . . 24

5 Testing the CCORE functionality of SLEC HLS flow 25

5.1 An Introduction to CCORE . . . 25

5.2 Synthesis of CCORE in Catapult . . . 26

5.3 Constraint on CCORE synthesis in Catapult . . . 27

5.4 Types of CCORE in Catapult . . . 27

5.5 Synthesis of CCORE in SLEC . . . 28

5.6 Verification Strategy Of CCORE in SLEC . . . 28

5.7 SLEC setup details for CCORE . . . 30

5.8 Problem statement . . . 30

5.9 An introduction to Self_checks . . . 30

5.10 Algorithm for the problem . . . 30

5.11 Example 31
5.12 Summary of the testing 33

6 Automation using PERL 34
6.1 Adding the top_name feature 34
6.1.1 problem statement 34
6.1.2 Algorithm 34
6.2 Cleaning of skiplist 35

7 Conclusion 36

8 Future Scope 37

Bibliography 38

List of Figures

2.1 Formal Verification Using Equivalence Checking 5

3.1 Overview Of SLEC [3] 6

3.2 Versions and usage of SLEC [2] 7

3.3 Basic Operation of SLEC [2] 10

3.4 Setup File [2] 12

3.5 Sample design for the HLS flow 13

3.6 Input files 14

3.7 Hierarchy step 15

3.8 Library step 16

3.9 Mapping step 17

3.10 Architecture step 18

3.11 Resource step 19

3.12 schedule step 20

4.1 Design Example 22

4.2 Algorithm For the problem 24

5.1 Implementation of CCORE functionality[4] 26

5.2 Step 1 Cut at CCORE port boundaries [4] 29

5.3 Step 2 Map CCORE port boundaries [4] 29

5.4 snippet of a testcase 31

5.5 snippet of self_checks added for the testcase 32

6.1 Algorithm For the problem 35

Chapter 1

Introduction

1.1 Motivation

Verification has become an inseparable part of any design process. Formal Verification has several advantages as compared to classical verification methods. Any design if verified properly would reduce cost as well as ensure quality and time to market. Formal verification on the other hand is the widely used method in EDA(Electronic Design Automation) as it has various advantages as compared to classical simulation.
SLEC(Sequential Logic Equivalence Checker) is a formal verification tool which has several advantages as compared to traditional logic equivalence checker. SLEC uses formal verification techniques which leads to lower capacity issues,lesser power requirements and better coverage

1.2 Objective

Objective for the project:Formal Verification using SLEC can be distributed as follows:

- Formal verify two designs using Mentor's Catapult and SLEC.

The SLEC HLS flow uses two tools Mentor's Catapult and SLEC. C++ is given as input to Catapult and produces output as Register Transfer Level(RTL). The RTL and c++ code is formally verified by SLEC.

- Testing various features of SLEC HLS flow

  The SLEC HLS has various features such as pipelining,resource scheduling,resuabality.

- Validating the tool and reporting bugs whenever required.

  If two designs do not prove to be formally equivalent then a falsification arrives. The falsification exists because of two reasons. One of them is when design is not coded properly and the other is when the tool does not prove it inspite of being functionally equivalent . A bug is reported in such a case.

1.3 Requirements

To complete this project knowledge of formal verification tool that is Sequential Logic Equivalence checker (SLEC) ,High Level Synthesis(HLS) tool that is catapult .In addition to this knowledge of c++ as well as knowledge of scripting languages like PERL and TCL is required.

Chapter 2

An Introduction to Formal Verification

2.1 An Introduction to verification

Verification and validation are two words which are used interchangeably but technically they are quite different. Verification is the method of evaluating your design in every stage of its development in order to ensure a error free design at the end of the design process. While Validation refers to the evaluation of final design to ensure that it meets the specification and requirements. There are various kind of verification methodology such as:

1) Functional Verification

2) FPGA emulation

3) Assertion based verification

4) code coverage

5) Formal Verification

There are various other methods for verification.Formal Verification is more popular in HLS due to its numerous advantages. Verification is generally performed during the development of design. It ensures that the right design is being developed.Validation generally comes after verification and it ensures that the final product developed is right or not.[1]

2.2 Formal Verification

Formal Verification is a method to prove correctness of any algorithm using various mathematical modeling. Formal verification can be used in verification of digital circuits whether sequential or combinational as well as for software expressed as source code. Earlier simulation was a very widely used method for verification.However Formal verification has a variety of advantages as compared to classical simulation:

1)Improves verification quality

2)Reduces verification effort

3)Saves verification time since process is faster

4) code coverage is better

5)It does not require input vectors

There are two popular methods for formal verification:

1)Formal Equivalence Checking

2)Formal Property Checking [1]

2.2.1 Formal Equivalence Checking

In the method of Formal Equivalence checking two designs are given as input and a output is produced by functional verification between two designs. In this method of formal verification Design Under Test(DUT) and golden reference are compared. They are reduced to boolean equations and then compared and output is produced.

Figure 2.1 shows the verification methodology for formal equivalence checking. There are two types of equivalence checking:

1) Combinational Equivalence Checking

2) Sequential Equivalence Checking

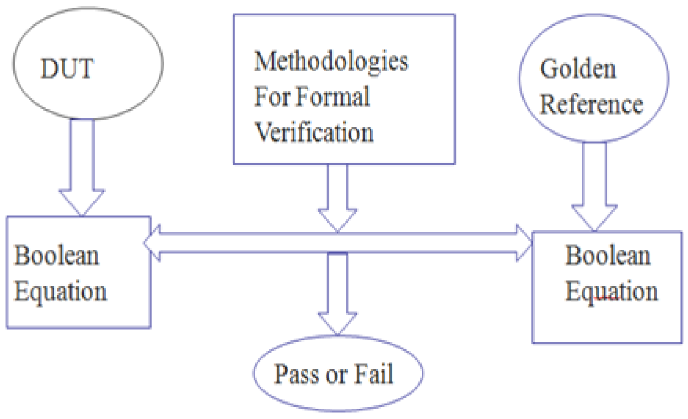


Figure 2.1: Formal Verification Using Equivalence Checking

- Combinational Equivalence Checking

  In combinational checking one to one mapping of flops is done between two designs that is the golden design and Device Under Test(DUT).

- Sequential Equivalence Checking

  In sequential checking two designs which are structurally different are compared without the use of one to one flop mapping.

  There are variety of tools available for logic equivalence checking in field of EDA.Some of them are

  1)FormalPro by Mentor Graphics

  2)Conformal by Cadence'

  3)Jasper Gold app by Cadence

  4)Formality by Synopsys

  5)SLEC by Calypto Design Systems(BU OF Mentor Graphics) [1]

Chapter 3

Overview of SLEC

3.1 Introduction to SLEC

SLEC stands for Sequential Logic Equivalence checker tool .It compares two designs - spec (specification) design and impl (implementation) design - to verify whether both designs are functionally equivalent or not. These designs may be at similar or different levels of abstraction. One of them may be be at abstraction or system level or RTL model and the other can be at RTL with deeper implementation.[2]

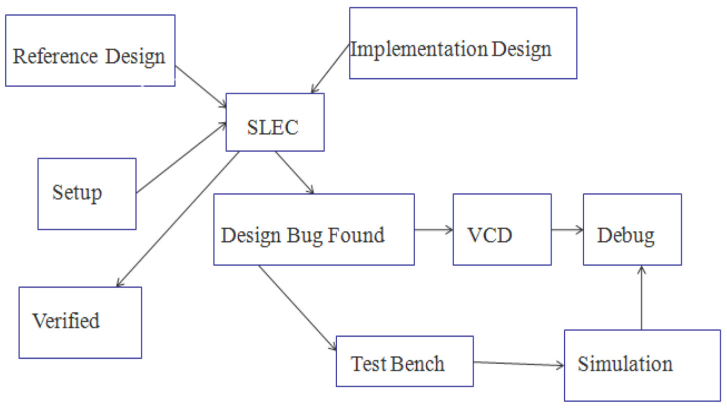


Figure 3.1: Overview Of SLEC [3]

SLEC provides:

- Formal Verification without the use of input test vectors.
- Generate Counter Example if designs are not functionally equivalent.
- Complex Bug Detection [2]

3.2 Advantages of SLEC

SLEC has several advantages as compared to other Logic Equivalence Checker.

- Easy and early detection of system-level and RTL functional bugs, without using test-benches.
- Reduces verification effort as there is no need for test-bench creation and modification
- Eliminates long simulation regressions and saves time

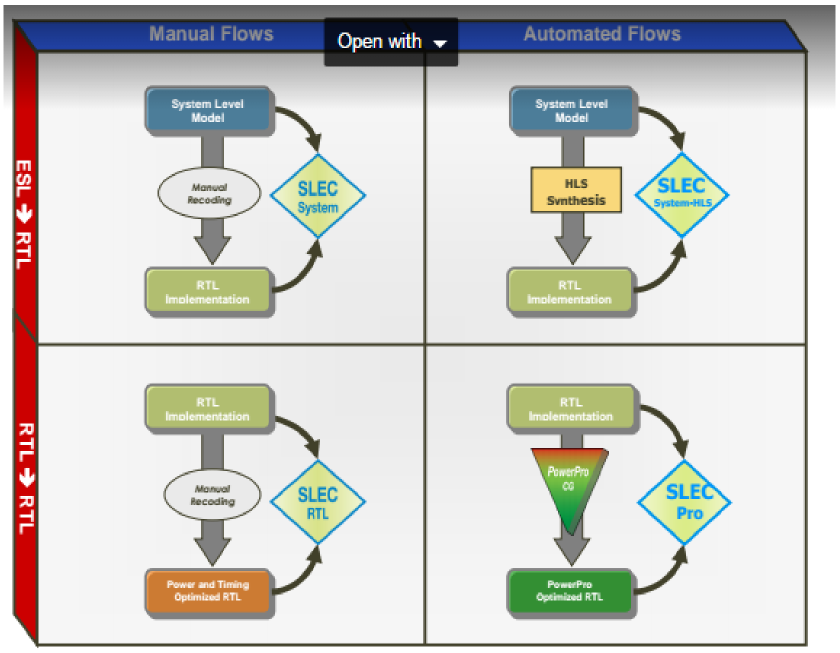


Figure 3.2: Versions and usage of SLEC [2]

3.3 Ability of SLEC

3.3.1 Equivalence Checking Compatibility

SLEC can perform equivalence checking between

- System-Level Models and manually created Register-Transfer Level (RTL) models.
- System-Level models and the RTL output produced by supported High-Level Synthesis (HLS) products such as Mentor Catapult.
- RTL models and the same RTL models with minor refinements in speed, area and power that may change their sequential behavior.
- RTL models and RTL models which have been optimized by various power optimization tools [2]

3.3.2 Refinement Verification

- Resource Scheduling: whenever a design is refined or enhanced, resources are allocated and scheduled to implement functional behavior in order to meet the cost and performance targets. It may happen that a computation which was of single cycle in a design specification may become multi-cycled during the implementation, changing the timing of an interface.
- State Recoding: State machine encodings may be changed in order to achieve better and optimized implementation area, timing, and/or dynamic power. A state recoding which was binary encoded in the specification design may change to one-hot in the implementation design.
- Pipelining: Pipelines are often added to a design in order to improve throughput and performance. Pipeline refinements means inserting or modifying the number of pipeline stages in a design's data and control paths.

- Register Retiming: Register retiming is a common method of RTL optimization which is used to balance the amount of logic between flip-flops. Even if the state of both the RTL models are different, the interface behavior of the impl and spec designs are equal.
- Clock Gating: Clock gating is an optimization technique used to reduce dynamic power. Sequential changes can affect the design state, due to which the use of combinational equivalence checkers are prevented. SLEC can verify designs with sequential differences thus it can highlight errors which are caused due to clock gating and verify designs inspite of the differences in clock gating.
- Interface Refinements: whenever designs are refined, block interfaces may change to abstract data types. In order to preserve core functionality, interface protocols and timings may change.
- Additional Modes of Operation: An implementation design may have additional modes of operation as compared to the specification design. In order to verify High-level behaviors implementation inputs are constrained so that the additional modes of operation are disabled.[2]

3.3.3 Basic operation of SLEC

The inputs given to SLEC are specification design and implementation design.Both of them are formally verified.
Figure 3.3 shows the basic steps involved:They are
1)Reading the designs
2)Providing clock to both the designs
3)providing reset states to both the designs
4)Providing Interface Alignment
5)State correspondence between both the designs
6)Finally verifying them[2]

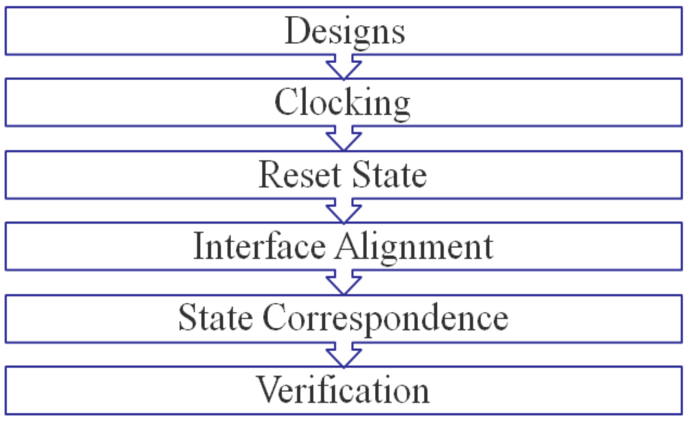


Figure 3.3: Basic Operation of SLEC [2]

- Design:The two designs spec and impl are given as input and are read in. Portions of the designs such as large memories which cannot be analyzed by SLEC, must be black-boxed. The design files and libraries used for each design have to be specified by the user[2].

- Clocking:A clock has to be specified to achieve a common frame of reference for timing relationships between the two devices. SLEC supports one clock per design, however different phase of the clock can be used and the clock may be gated. For the cases where both of the designs has a single clock, no setup information has to be added for the spec and impl designs in the tcl file[2].

- Reset state:Reset states in any design are required to begin equivalence checking. This information can be provided in the .tcl file through explicit state values,reset sequences or VCD dumps.Normally reset values are generally set to zero[2].

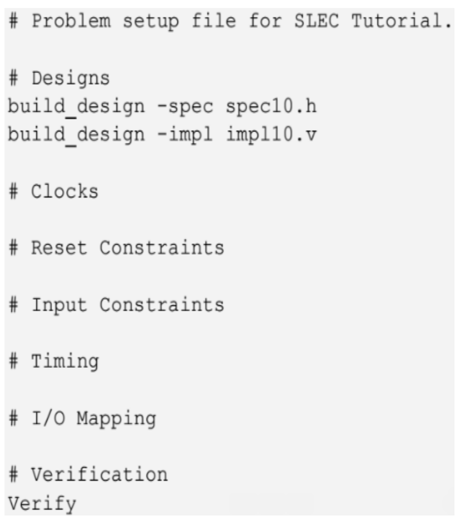
- Interface Alignment:To verify the designs formally correspondence must be established between the design interfaces of spec and impl. If both the designs are cycle accurate that is they use same clock cycles for operation and even use matching names for input and output ports, all of this information can be automatically interpreted. The Designs which have differences in timing, protocols, or data representation require additional setup information in the .tcl file. In that case the input and output ports in one design must be mapped to corresponding ports on the other design [2].

- State Correspondence: If the two designs have any common correspondence in state then it must be specified. This optional information helps SLEC in verification process. SLEC even tries to find as many state correspondences as possible between the two designs,before it attempts to prove the equivalence [2].

- Verification:The two designs can be verified using one of two modes: bug finding or full proof. Generally bug finding mode should is preferred to search for any functional differences between the designs. In this mode SLEC compares the designs over a bounded or fixed set of transactions. Full proof mode provides additional formal techniques which are to prove the designs equivalent which are not supported in the bug finding mode. If the designs are not equivalent, counterexamples are produced which demonstrate the differences. After SLEC completes its verification log files ,waveform files and test benches are generated for further analysis by the user [2].

3.4 Setup File

The operation of SLEC is controlled by a TCL command file. The command file provides problem setup information which identifies the designs and describes how the interface mappings and timings of the two designs should be compared. For equivalence checking, an instance of the top-level module of the specification design is named spec, while the comparable instance of the top-level module of the implementation design is named impl [2].



```
# Problem setup file for SLEC Tutorial.

# Designs
build_design -spec spec10.h
build_design -impl impl10.v

# Clocks

# Reset Constraints

# Input Constraints

# Timing

# I/O Mapping

# Verification
Verify
```

Figure 3.4: Setup File [2]

3.5 SLEC-HLS

HLS stands for High Level Synthesis (HLS). HLS has the ability to generate production quality RTL implementation from high level abstraction languages like c and c++. In High level synthesis Algorithmic behavior written in C/C++ or System C is given as input to the HLS tool which then automatically synthesizes the algorithmic behavior to register transfer level (RTL) design. HLS maps top-level c++ variables to resources that would implement RTL. In this, resource synthesis is the process of mapping top-level C++ variables to resources that implement a RTL. Whenever an algorithm is converted into RTL with the help of HLS,various constraints such as scheduling has to be taken care with the hardware point of view. SLEC-HLS is verification of the RTL generated by the HLS tool Catapult. The two inputs given to SLEC-HLS are RTL generated by the HLS tool and the system level design in c or c++. The two designs are verified formally stating that they are functionally equivalent or not. This would verify that whether the RTL generated by the HLS tool catapult is functionally correct or not. If the two designs are not found equivalent a counter example is generated which can pinpont design bugs.

The HLS flow of catapult is briefly described in the following stages:

1)Input Files

2)Hierarchy

3)Library

4)Mapping

5)Architecture

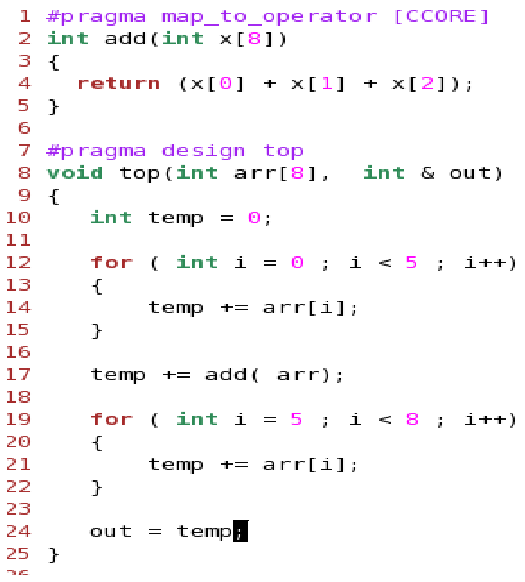
6)Resource

7)Generate RTL

3.5.1 SLEC-HLS Flow

This section would briefly describe the HLS flow used in SLEC

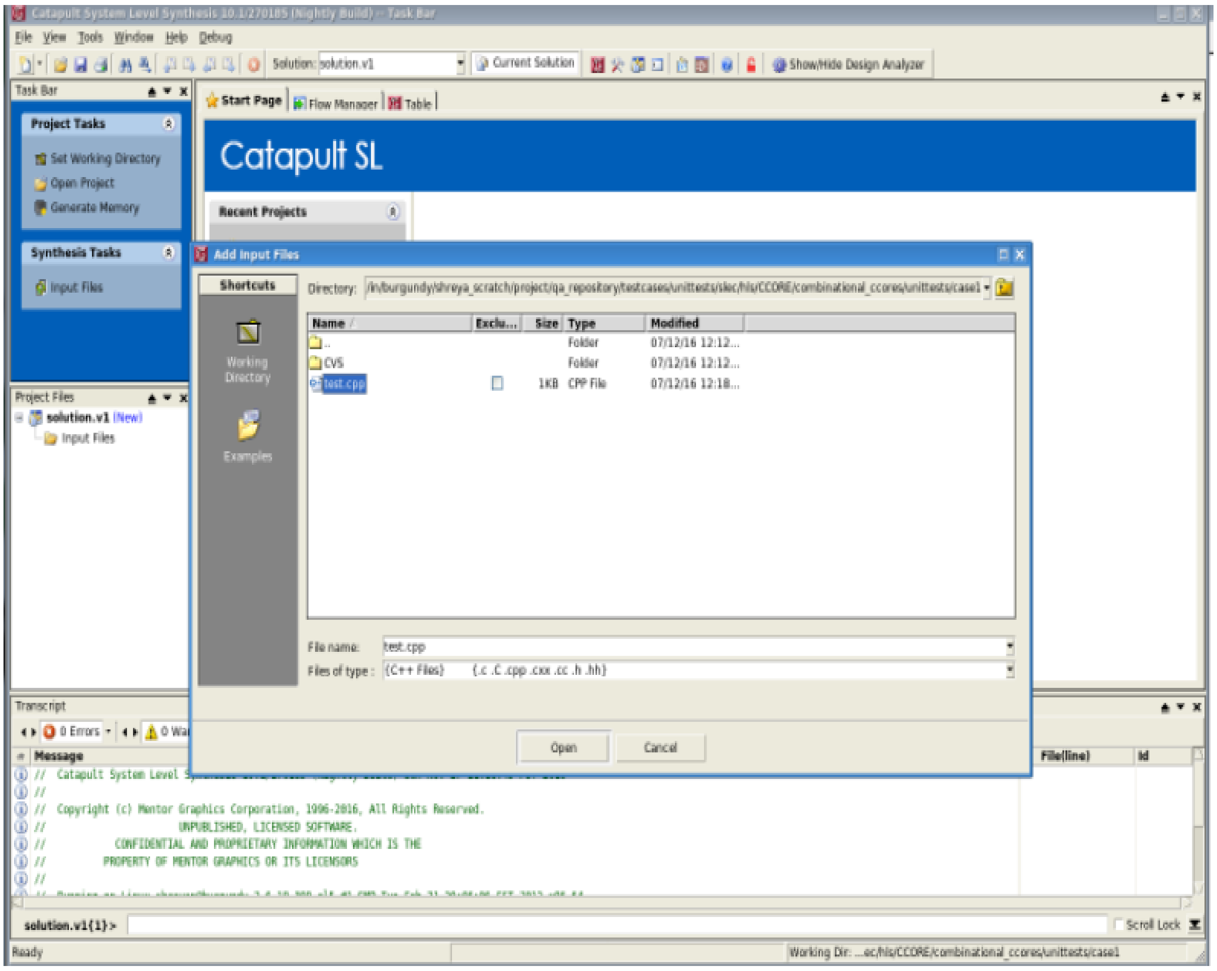
The Input design for the SLEC HLS flow is shown in figure:



```
#pragma map_to_operator [CCORE]
int add(int x[8])
{
  return (x[0] + x[1] + x[2]);
}

#pragma design top
void top(int arr[8],  int & out)
{
   int temp = 0;

   for ( int i = 0 ; i < 5 ; i++)
   {
       temp += arr[i];
   }

   temp += add( arr);

   for ( int i = 5 ; i < 8 ; i++)
   {
       temp += arr[i];
   }

   out = temp;
}
```

Figure 3.5: Sample design for the HLS flow

1) Input Files:This is the first step of the HLS flow. In this step the design file(c,c++ or system c) is given as input to the HLS tool. The following snippet depicts such a scenario:



Directory: /n/burgundy/shreya_scratch/project/qa_repository/testcases/unittests/slec/hls/CCORE/combinational_ccores/unittests/case1

Name	Exclu...	Size	Type	Modified
..			Folder	07/12/16 12:12...
CVS			Folder	07/12/16 12:12...
test.cpp	☐	1KB	CPP File	07/12/16 12:18...

File name: test.cpp
Files of type: (C++ Files) (.c .C .cpp .cxx .cc .h .hh)

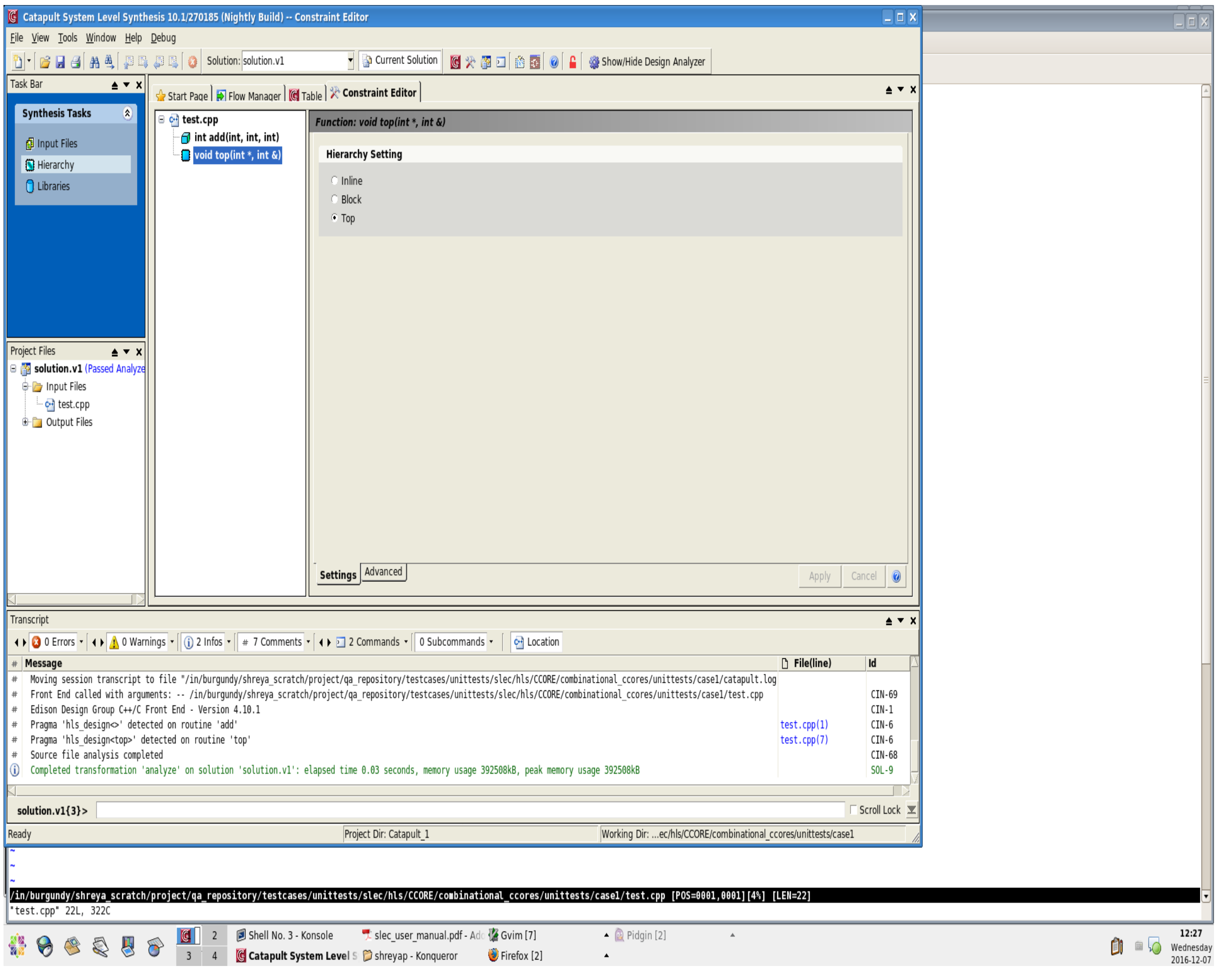
```
# Message                                                              File(line)  Id
// Catapult System Level S...
//
// Copyright (c) Mentor Graphics Corporation, 1996-2016, All Rights Reserved.
//                        UNPUBLISHED, LICENSED SOFTWARE.
//             CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//           PROPERTY OF MENTOR GRAPHICS OR ITS LICENSORS
//
```

solution.v1{1}>

Working Dir: ...ec/hls/CCORE/combinational_ccores/unittests/case1

Figure 3.6: Input files

2) Hierarchy:This step decides the hierarchy of the design. It would state which function would be the top level function and which functions will be the sub blocks.

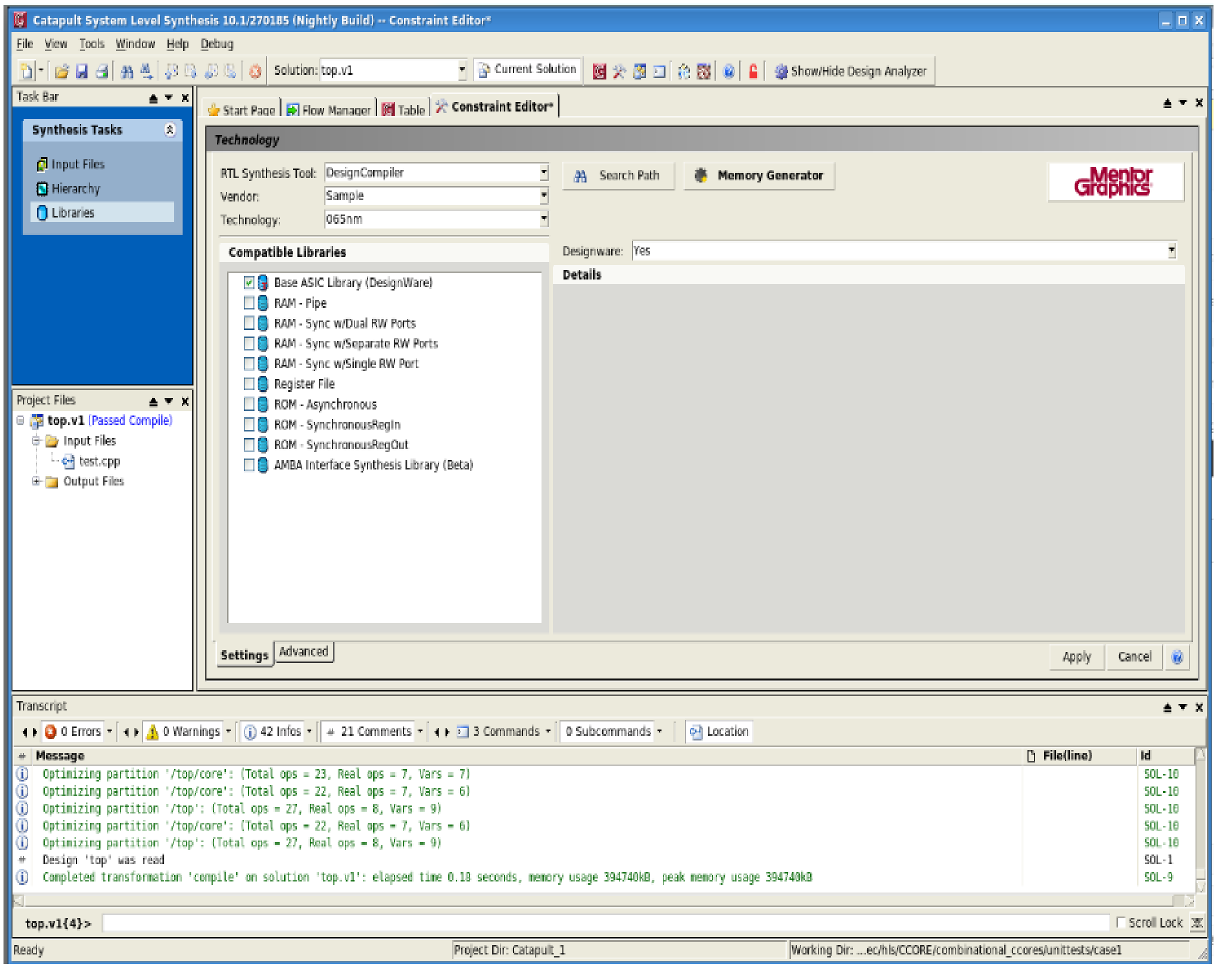


```
# Message                                                                                                                              File(line)   Id
# Moving session transcript to file "/in/burgundy/shreya_scratch/project/qa_repository/testcases/unittests/slec/hls/CCORE/combinational_ccores/unittests/case1/catapult.log
# Front End called with arguments: -- /in/burgundy/shreya_scratch/project/qa_repository/testcases/unittests/slec/hls/CCORE/combinational_ccores/unittests/case1/test.cpp          CIN-69
# Edison Design Group C++/C Front End - Version 4.10.1                                                                                                 CIN-1
# Pragma 'hls_design<>' detected on routine 'add'                                                                                  test.cpp(1)  CIN-6
# Pragma 'hls_design<top>' detected on routine 'top'                                                                               test.cpp(7)  CIN-6
# Source file analysis completed                                                                                                                    CIN-68
ⓘ Completed transformation 'analyze' on solution 'solution.v1': elapsed time 0.03 seconds, memory usage 392508kB, peak memory usage 392508kB           SOL-9
```

```
~
~
~
/in/burgundy/shreya_scratch/project/qa_repository/testcases/unittests/slec/hls/CCORE/combinational_ccores/unittests/case1/test.cpp [POS=0001,0001][4%] [LEN=22]
"test.cpp" 22L, 322C
```

Figure 3.7: Hierarchy step

3) Libraries:This step of HLS provides us various options such as technology(65 or 90nm) that has to be used for the verification. It also gives various other options of design constraints such as synthesis tool,target hardware technology,compatible libraries and design hierarchy.



RTL Synthesis Tool: DesignCompiler
Vendor: Sample
Technology: 065nm
Designware: Yes

Compatible Libraries
- [✓] Base ASIC Library (DesignWare)
- [ ] RAM - Pipe
- [ ] RAM - Sync w/Dual RW Ports
- [ ] RAM - Sync w/Separate RW Ports
- [ ] RAM - Sync w/Single RW Port
- [ ] Register File
- [ ] ROM - Asynchronous
- [ ] ROM - SynchronousRegIn
- [ ] ROM - SynchronousRegOut
- [ ] AMBA Interface Synthesis Library (Beta)

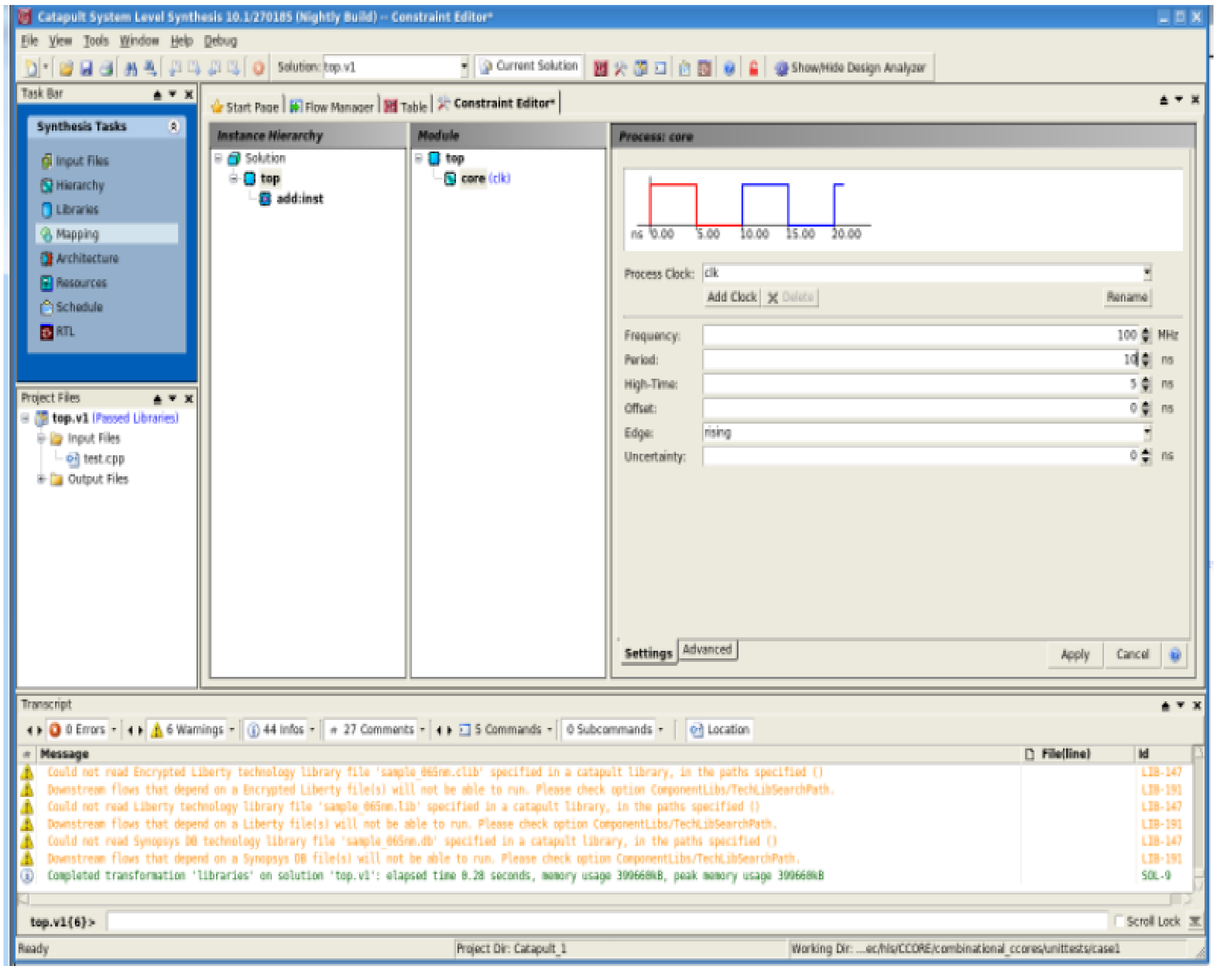
Project Files
- top.v1 (Passed Compile)
  - Input Files
    - test.cpp
  - Output Files

```
#  Message                                                                                                         File(line)  Id
   Optimizing partition '/top/core': (Total ops = 23, Real ops = 7, Vars = 7)                                                  SOL-10
   Optimizing partition '/top/core': (Total ops = 22, Real ops = 7, Vars = 6)                                                  SOL-10
   Optimizing partition '/top': (Total ops = 27, Real ops = 8, Vars = 9)                                                       SOL-10
   Optimizing partition '/top/core': (Total ops = 22, Real ops = 7, Vars = 6)                                                  SOL-10
   Optimizing partition '/top': (Total ops = 27, Real ops = 8, Vars = 9)                                                       SOL-10
#  Design 'top' was read                                                                                                       SOL-1
   Completed transformation 'compile' on solution 'top.v1': elapsed time 0.18 seconds, memory usage 394740kB, peak memory usage 394740kB   SOL-9

top.v1{4}>
```

Ready | Project Dir: Catapult_1 | Working Dir: ...ec/hls/CCORE/combinational_ccores/unittests/case1

Figure 3.8: Library step

4)Mapping: This step of HLS is used to set clock,reset,enable and handshaking signals. It is also used to select process in HLS. It is also used to specify handshaking signals.

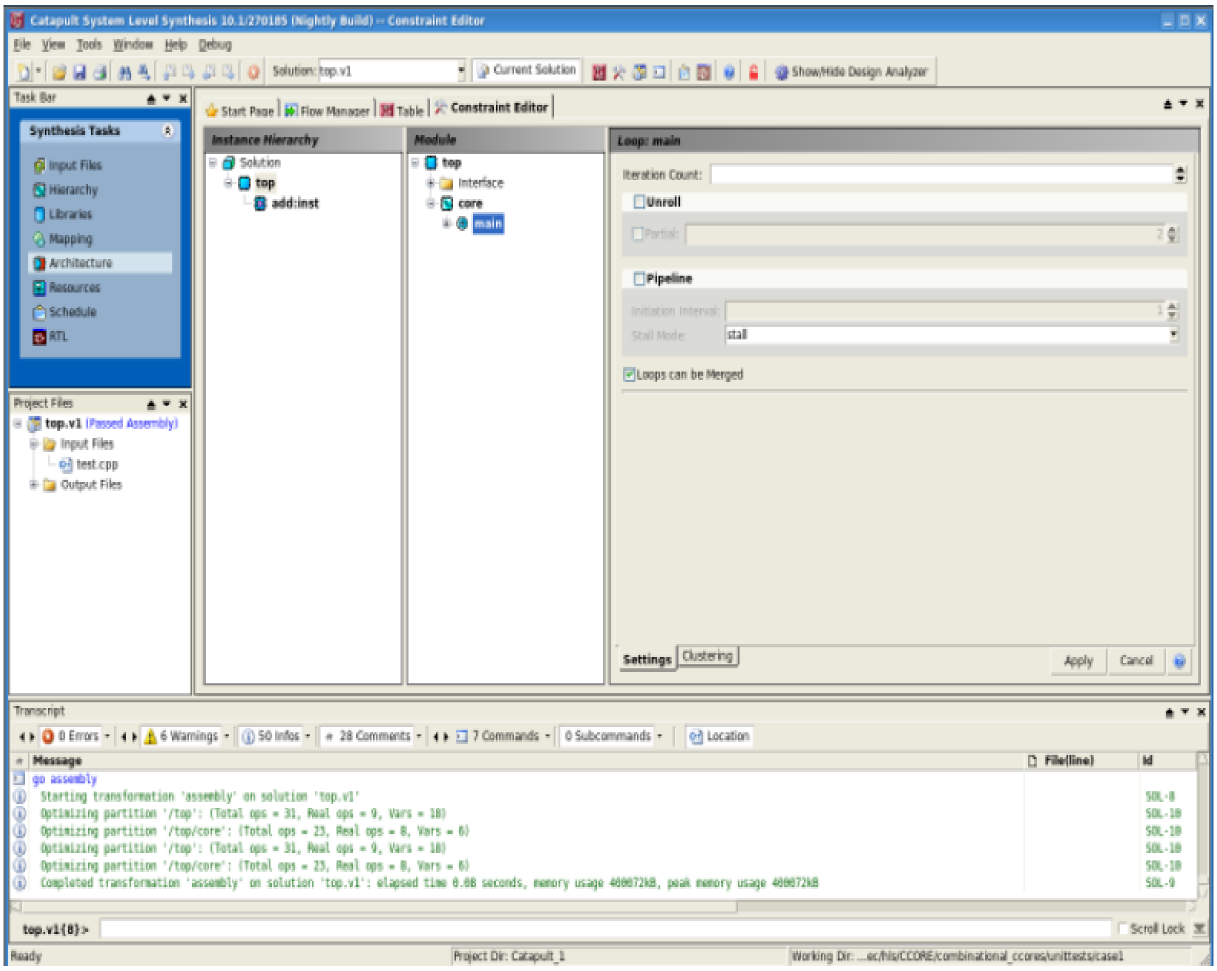


Process Clock: clk
Frequency: 100 MHz
Period: 10 ns
High-Time: 5 ns
Offset: 0 ns
Edge: rising
Uncertainty: 0 ns

```
Message                                                                                                                  File(line)  Id
Could not read Encrypted Liberty technology library file 'sample_065nm.clib' specified in a catapult library, in the paths specified ()   LIB-147
Downstream flows that depend on a Encrypted Liberty file(s) will not be able to run. Please check option ComponentLibs/TechLibSearchPath.  LIB-191
Could not read Liberty technology library file 'sample_065nm.lib' specified in a catapult library, in the paths specified ()   LIB-147
Downstream flows that depend on a Liberty file(s) will not be able to run. Please check option ComponentLibs/TechLibSearchPath.  LIB-191
Could not read Synopsys DB technology library file 'sample_065nm.db' specified in a catapult library, in the paths specified ()   LIB-147
Downstream flows that depend on a Synopsys DB file(s) will not be able to run. Please check option ComponentLibs/TechLibSearchPath.  LIB-191
Completed transformation 'libraries' on solution 'top.v1': elapsed time 0.28 seconds, memory usage 399668kB, peak memory usage 399668kB   SOL-9

top.v1(6)>
```

Figure 3.9: Mapping step

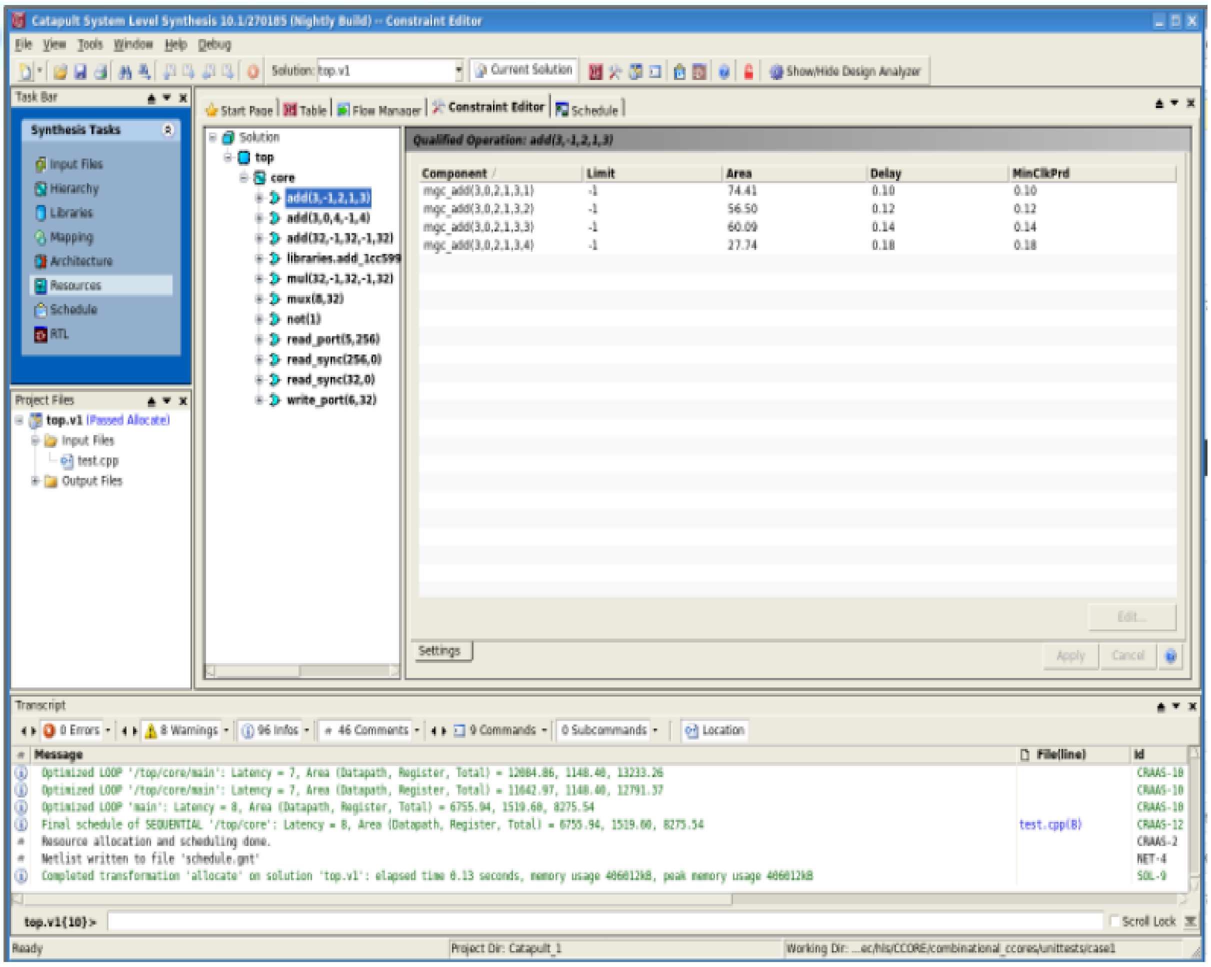
5)Architecture:This step of HLS performs design partition.This step performs optimization and applies constraint on designs. For example loops can be merged,unrolled or pipelined. It has low power options for low power designs. It also has various options such as Initation Interval value.



```
Message                                                                                                          File(line)   Id
go assembly
Starting transformation 'assembly' on solution 'top.v1'                                                                      SOL-8
Optimizing partition '/top': (Total ops = 31, Real ops = 9, Vars = 18)                                                       SOL-10
Optimizing partition '/top/core': (Total ops = 23, Real ops = 8, Vars = 6)                                                   SOL-10
Optimizing partition '/top': (Total ops = 31, Real ops = 9, Vars = 18)                                                       SOL-10
Optimizing partition '/top/core': (Total ops = 23, Real ops = 8, Vars = 6)                                                   SOL-10
Completed transformation 'assembly' on solution 'top.v1': elapsed time 0.08 seconds, memory usage 400872kB, peak memory usage 400872kB   SOL-9

top.v1{8}>
Ready        Project Dir: Catapult_1        Working Dir: ...ec/hls/CCORE/combinational_ccores/unittests/case1
```

Figure 3.10: Architecture step

6)Resource:This step of HLS allocates the resources needed for each and every operation. It also sets a limit for the number of instances that can be allocated.



Qualified Operation: add(3,-1,2,1,3)

Component	Limit	Area	Delay	MinClkPrd
mgc_add(3,0,2,1,3,1)	-1	74.41	0.10	0.10
mgc_add(3,0,2,1,3,2)	-1	56.50	0.12	0.12
mgc_add(3,0,2,1,3,3)	-1	60.09	0.14	0.14
mgc_add(3,0,2,1,3,4)	-1	27.74	0.18	0.18

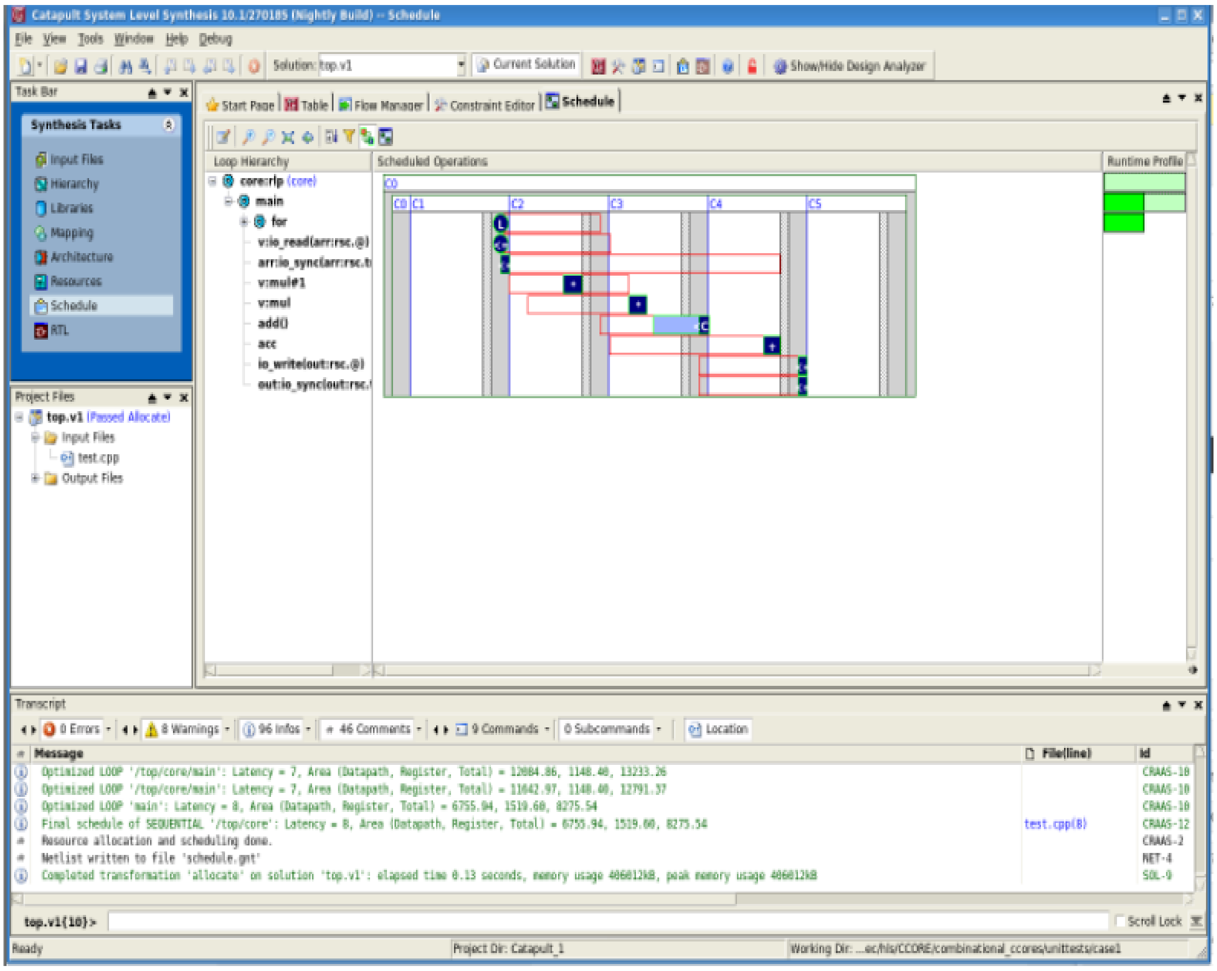
Message	File(line)	Id
Optimized LOOP '/top/core/main': Latency = 7, Area (Datapath, Register, Total) = 12084.86, 1148.40, 13233.26		CRAAS-10
Optimized LOOP '/top/core/main': Latency = 7, Area (Datapath, Register, Total) = 11642.97, 1148.40, 12791.37		CRAAS-10
Optimized LOOP 'main': Latency = 8, Area (Datapath, Register, Total) = 6755.94, 1519.60, 8275.54		CRAAS-10
Final schedule of SEQUENTIAL '/top/core': Latency = 8, Area (Datapath, Register, Total) = 6755.94, 1519.60, 8275.54	test.cpp(8)	CRAAS-12
Resource allocation and scheduling done.		CRAAS-2
Netlist written to file 'schedule.gnt'		NET-4
Completed transformation 'allocate' on solution 'top.v1': elapsed time 0.13 seconds, memory usage 406012kB, peak memory usage 406012kB		SOL-9

```
top.v1(10)>
```

Ready | Project Dir: Catapult_1 | Working Dir: ...ec/hls/CCORE/combinational_ccores/unittests/case1

Figure 3.11: Resource step

7)schedule:This step of HLS schedules the resources that have been allocated on the basis of timing constraints and usability. The whole scheduled process can be seen with the help of Gantt chart.



```
Transcript
0 Errors   8 Warnings   96 Infos   46 Comments   9 Commands   0 Subcommands   Location
Message                                                                                                              File(line)    Id
Optimized LOOP '/top/core/main': Latency = 7, Area (Datapath, Register, Total) = 12084.86, 1148.40, 13233.26                       CRAAS-10
Optimized LOOP '/top/core/main': Latency = 7, Area (Datapath, Register, Total) = 11642.97, 1148.40, 12791.37                       CRAAS-10
Optimized LOOP 'main': Latency = 8, Area (Datapath, Register, Total) = 6755.94, 1519.60, 8275.54                                    CRAAS-10
Final schedule of SEQUENTIAL '/top/core': Latency = 8, Area (Datapath, Register, Total) = 6755.94, 1519.60, 8275.54   test.cpp(8)  CRAAS-12
Resource allocation and scheduling done.                                                                                           CRAAS-2
Netlist written to file 'schedule.gnt'                                                                                             NET-4
Completed transformation 'allocate' on solution 'top.v1': elapsed time 0.13 seconds, memory usage 406012kB, peak memory usage 406012kB   SOL-9
top.v1(10)>
Ready    Project Dir: Catapult_1    Working Dir: ...ec/hls/CCORE/combinational_ccores/unittests/case1
```

Figure 3.12: schedule step

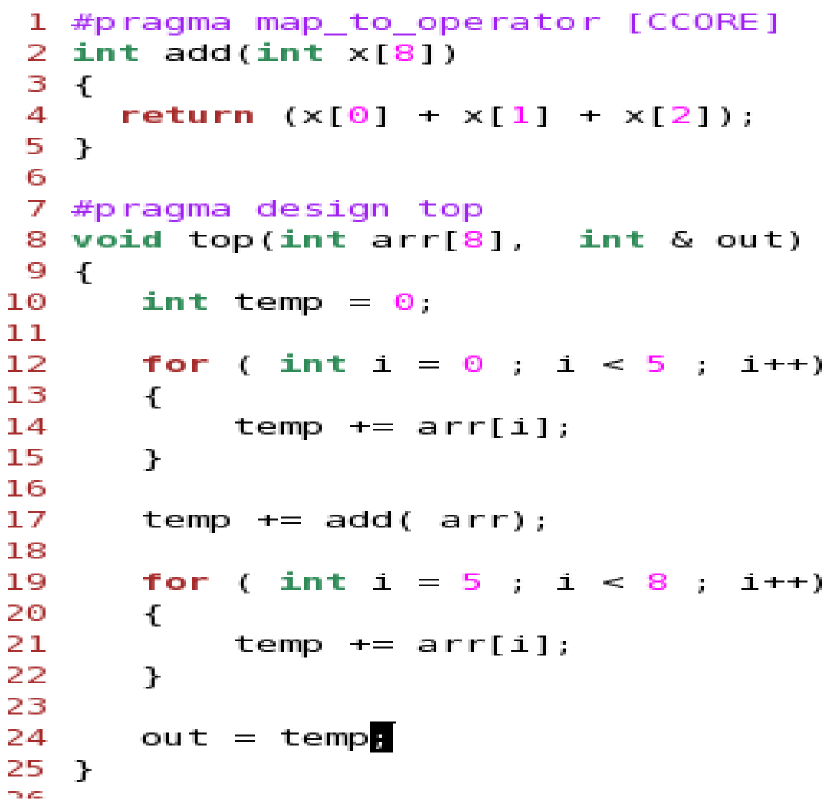
8)RTL:Finally an RTL is generated

Chapter 4

Testing the all piped feature of SLEC HLS flow

4.1 An Introduction to pipelining

Any algorithm would have data dependency in it which ultimately decreases the throughput and performance. Pipelining is a technique to reduce data dependency by parallel execution which however adds data or buffer storage. Pipelining allows fetching of the next instruction when the first one is performing any arithmetic operation. Thus it is an very important feature provided in HLS flow to increase throughput and performance. The designs in HLS flow are pipelined in order to speed up the synthesis process.
The SLEC HLS flow uses three modules for pipelining:
1)Top piped:If the design is top piped the function which is marked as top in the hierarchy is pipelined. The main function in c++ design is marked as top.
2)All piped:If the design is all piped all the other functions except the one which is marked as top are pipelined.
3)No piped:If the design is no piped no function is pipelined.
Figure below shows the design snippet of a sample design file. In this snippet in the



```
#pragma map_to_operator [CCORE]
int add(int x[8])
{
   return (x[0] + x[1] + x[2]);
}

#pragma design top
void top(int arr[8],  int & out)
{
     int temp = 0;

     for ( int i = 0 ; i < 5 ; i++)
     {
          temp += arr[i];
     }

     temp += add( arr);

     for ( int i = 5 ; i < 8 ; i++)
     {
          temp += arr[i];
     }

     out = temp;
}
```

Figure 4.1: Design Example

case of top piped only the main function that is top would be pipelined. In the case of no piped no function would be pipelined while in case of all piped both of the for loops would be pipelined. The throughput and performance would be different for each of pipelined case. For top piped the syntax would be

directive set /top/core/main -PIPELINE_INIT_INTERVAL 1

For all piped the syntax would be

directive set /top/core/main/for -PIPELINE_INIT_INTERVAL 1

directive set /top/core/main/for#1 -PIPELINE_INIT_INTERVAL 1

For no piped case the syntax would be missing.

4.2 Problem Statement

To reduce the Initation Interval value to the minimum possible value of all the all piped cases present in the qa repository.

4.3 Initation Interval value

Initation Interval value is the value after which the loop will start its next iteration. A minimum value of II is desirable since it would increase parallelism and hence decrease data dependency

4.4 Algorithm For the Problem

To test the all piped feature of SLEC HLS flow a certain algorithm was followed which is depicted in the figure below. First of all,all of the all piped cases were extracted from the qa repository. After extracting them the II value of all the cases were set to one. After setting them to one they were ran in regression which had a build which would error out for the all piped cases which do not showed pipelining behavior that is they do not have overlapping. After the regression ran some of the cases passed,some of them failed. The failing cases had two cases:
1)The cases which gave CTS-PRCM error
2)The cases which had Resource Competition error
The cases which gave CTS-PRCM error were the one which do not show any overlapping in any pipelining region and have to be removed from the qa repository since they show no meaning. While the cases which had resource competition error were the ones which could not complete its execution with II 1,so their II has to be increased to 2. Again regression was run on the cases which gave resource competition. After the regression finished the same scenario depicts as mentioned previously. The process has to be repeated till all the cases are resolved.

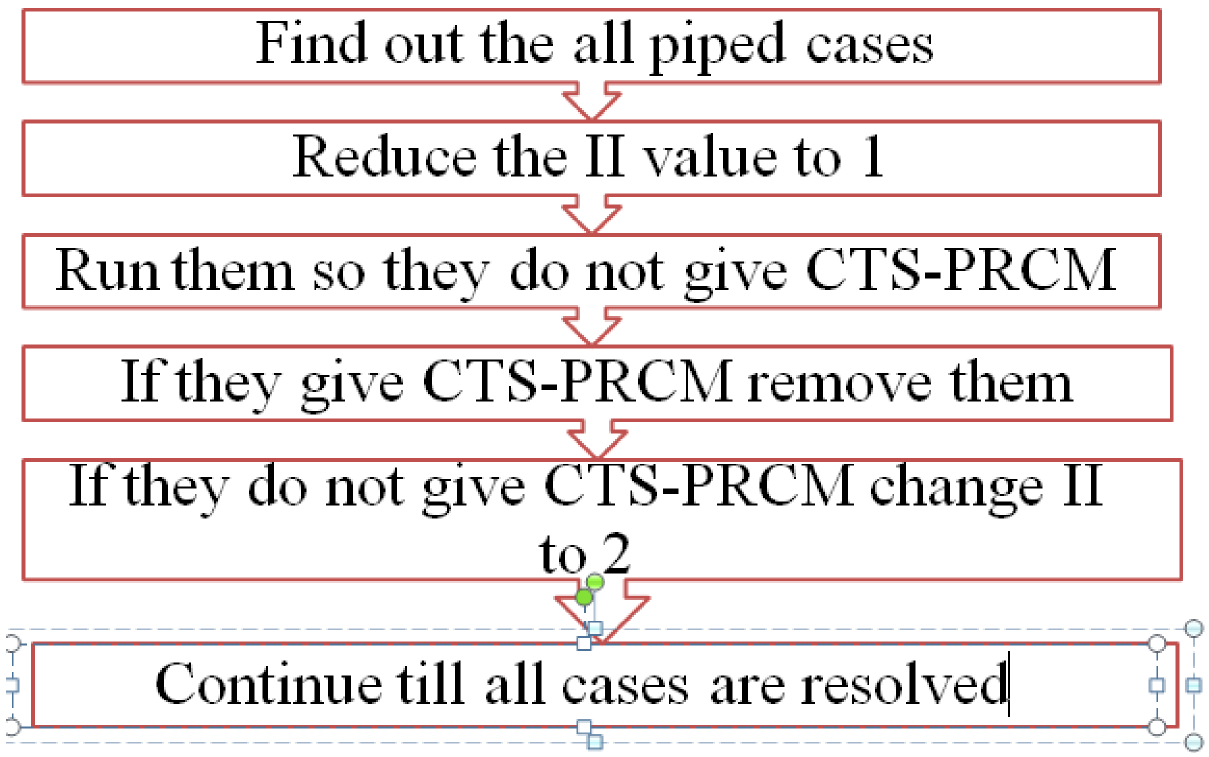

Find out the all piped cases
Reduce the II value to 1
Run them so they do not give CTS-PRCM
If they give CTS-PRCM remove them
If they do not give CTS-PRCM change II to 2
Continue till all cases are resolved

Figure 4.2: Algorithm For the problem

4.5 Problems

During the testing the major problem which was faced was there were test-cases which had multiple loops that were to be pipelined. So the test-cases had to be checked by providing different II value for the loops. Those test-cases were to be handled manually

4.6 Summary of the testing

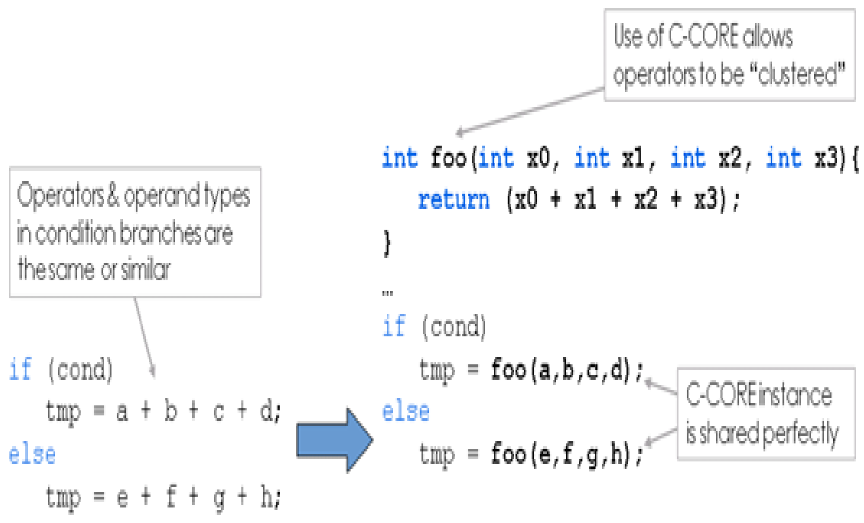
There were a total of 2111 all piped cases. After the testing finished 737 test-cases passed and had the minimum II value and 881 cases were removed. The remaining 1618 cases can be tested further.

Chapter 5

Testing the CCORE functionality of SLEC HLS flow

5.1 An Introduction to CCORE

CCORE is a user defined operation that consists of collection of one or more operators. CCORE is a functionality which is very much similar to user defined functions in C. CCORE is a concept defined by Catapult for the main purpose of reusability. The synthesis of CCORE in catapult is very much different as compared to SLEC.There are several advantages of using CCORE.They are:
1)Improves runtime
2)Reduces total number of variables since reusabality is introduced.
3)The functionality optimized once can be reused a number of times.
4)Minimizes mux sharing logic
5)Facilitates coarse gain sharing improving area
Figure 6.1 shows the difference of implementation of a functionality with and without the help of a CCORE. It also shows the advantage of using CCORE functionality.[4]


Operators & operand types in condition branches are the same or similar
Use of C-CORE allows operators to be "clustered"
C-CORE instance is shared perfectly

```
if (cond)
   tmp = a + b + c + d;
else
   tmp = e + f + g + h;
```

```
int foo(int x0, int x1, int x2, int x3){
   return (x0 + x1 + x2 + x3);
}
...
if (cond)
   tmp = foo(a,b,c,d);
else
   tmp = foo(e,f,g,h);
```

Figure 5.1: Implementation of CCORE functionality[4]

5.2 Synthesis of CCORE in Catapult

In Catapult CCORE can be synthesized in two modes:

1)Top-Down

In this mode CCORE can be synthesized in one step.The user has to simply specify the top functionality and the other CCORE functions. The synthesis of CCORE functionality using this method is simpler as compared to the bottom-up approach. Top-Down functionality is a simpler approach in implementation point of view.

2)Bottom-Up

In this method synthesis of CCORE is a multi-step process. First all the CCORE functionality are synthesized using separate Catapult runs and different CCORE libraries are used while synthesizing top. The bottom-up approach is tougher in implementation as compared to top-down but is easy to debug during falsifications.[4]

5.3 Constraint on CCORE synthesis in Catapult

- Inouts are not allowed
- Handshake interfaces are not allowed
- All outputs must be written unconditionally
- CCORE must be pipelined with II=1

5.4 Types of CCORE in Catapult

1)Combinational CCORE

In this type of CCORE output is not registered.This type of design does not contain any clock. The designs are purely combinational and does not contain any flip flop since they do not have to hold any value. This CCORE is generally scheduled in one Cstep.

2)Sequential CCORE

In this type of CCORE outputs are not registered. This type of design generally have one or more clock. This design contains flip flop since they need to hold values. In this type of CCORE registering of input and output can be controlled through directive. This type of CCORE may or may not be scheduled in one cstep.

3)Static CCORE

This type of CCORE contains static variables. Since this CCORE contains static variables they are known as static CCORE.

4)Variable latency CCORE

This type of CCORE contains variable latency.[4]

5.5 Synthesis of CCORE in SLEC

By default, SLEC inlines all function calls. User has to explicitly tell SLEC to synthesis it as a CCORE function hierarchy. SLEC only creates combinational CCOREs. SLEC will honor mark_hierarchy command or CCORE pragma only under a g̈lobal enable_hierarchy_synthesis:. For any function if mark_hierachy is specified, SLEC creates a module for that function and dumps all the information required for mapping in an XML file spec_ccore_info.xml.

Although, in most of the cases SLEC tries to create just one module for each CCORE, but sometimes due to the different ways of passing actuals to the same formals (e.g. pointer formal binded to two arrays of different sizes), SLEC will create different modules with different interfaces.[4]

5.6 Verification Strategy Of CCORE in SLEC

SLEC verify CCORE in the following steps :

1)SLEC identifies the number of CCORE functionality in the design

2) SLEC indentifies the top

3) The whole design is divided into different modules with the help of boundaries

4) The design is then cut at port of CCORE boundaries.

5) The next step is to map CCORE port boundaries accordingly

6) CCORE is then formally verified with the help of SLEC.[4]

The full process of verification is briefly explained in the following two figure. The two figure depicts the two main process of verification.

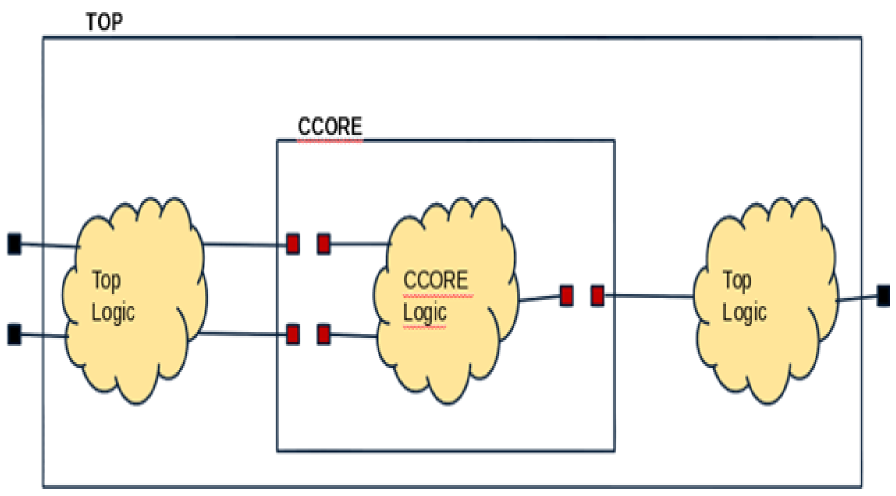


Figure 5.2: Step 1 Cut at CCORE port boundaries [4]

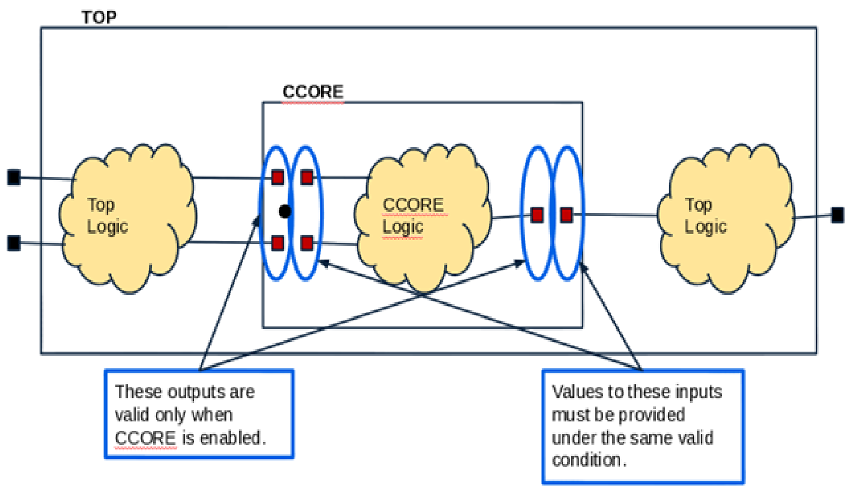

TOP
CCORE
Top Logic
CCORE Logic
Top Logic
These outputs are valid only when CCORE is enabled.
Values to these inputs must be provided under the same valid condition.

Figure 5.3: Step 2 Map CCORE port boundaries [4]

5.7 SLEC setup details for CCORE

SLEC in general inlines all the function calls.User has to explicitly specify to SLEC to synthesize it as a CCORE functionality. There are two methods for the user to explicitly specify the function as CCORE.
1) Using mark_hierarchy command
This command helps to point the function declaration which is to be synthesized as CCORE. Function Declaration can be pointed either by using source information or canonical names.
2)Use pragma
If function declaration is associated with following pragma, then also it is synthesized as a CCORE: #pragma map_to_operator [CCORE] [4]

5.8 Problem statement

To add self_checks to all the CCORE cases present in the qa_repository.

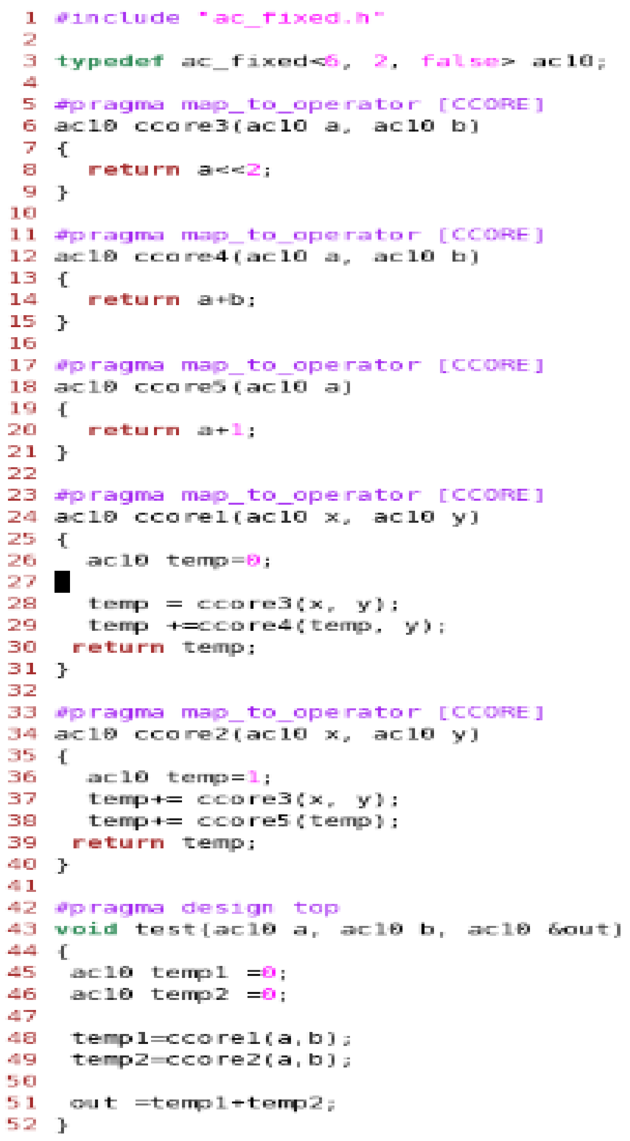
5.9 An introduction to Self_checks

Self_checks are the checks added by the tester to ensure the correct functionality of any design.

5.10 Algorithm for the problem

Self checks for any design are added in a seperate .tcl file whose name starts with self_checks.tcl. The following algorithm was followed for the solution of problem.
1)Find out all the CCORE cases present in the qa_repository.
2)Study the cases from design perspective that is study test.cpp
3)Add self_checks in .tcl file.

5.11 Example

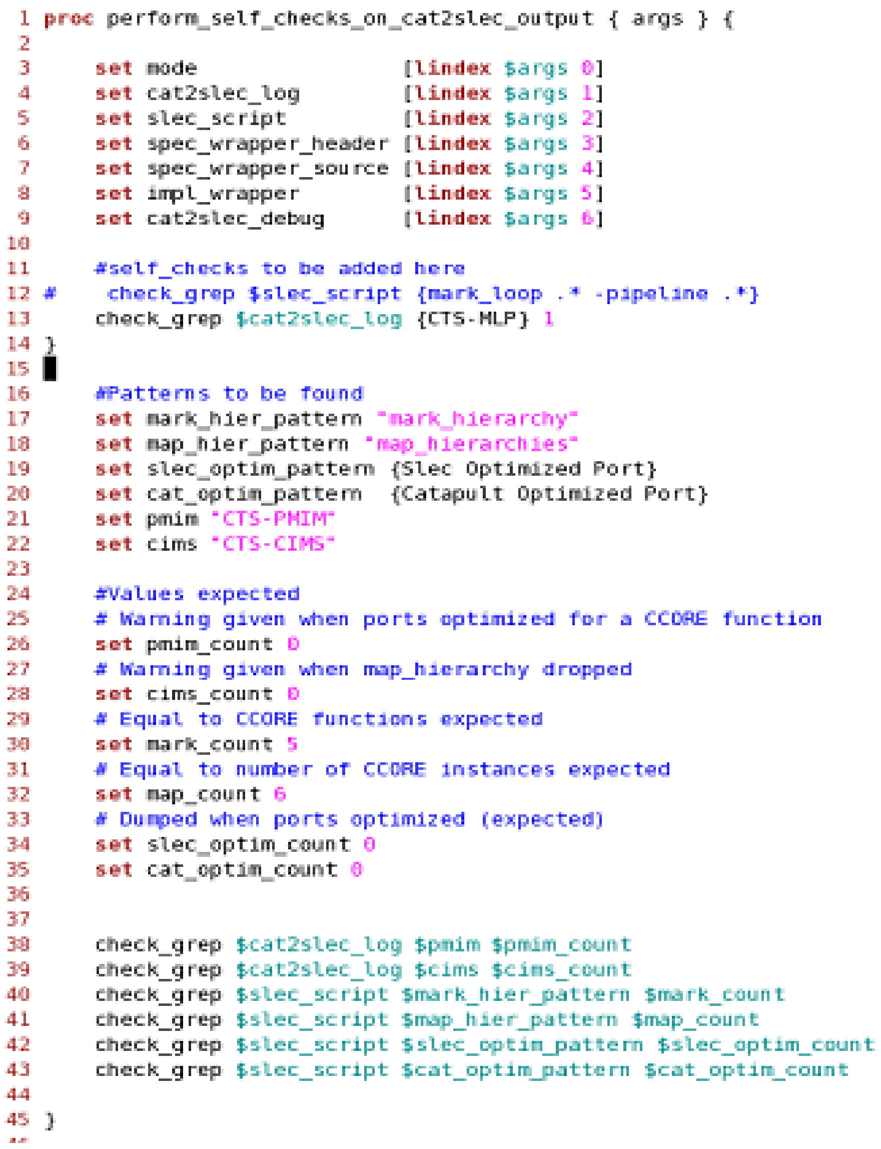


```
#include "ac_fixed.h"

typedef ac_fixed<6, 2, false> ac10;

#pragma map_to_operator [CCORE]
ac10 ccore3(ac10 a, ac10 b)
{
   return a<<2;
}

#pragma map_to_operator [CCORE]
ac10 ccore4(ac10 a, ac10 b)
{
   return a+b;
}

#pragma map_to_operator [CCORE]
ac10 ccore5(ac10 a)
{
   return a+1;
}

#pragma map_to_operator [CCORE]
ac10 ccore1(ac10 x, ac10 y)
{
   ac10 temp=0;

   temp = ccore3(x, y);
   temp +=ccore4(temp, y);
  return temp;
}

#pragma map_to_operator [CCORE]
ac10 ccore2(ac10 x, ac10 y)
{
   ac10 temp=1;
   temp+= ccore3(x, y);
   temp+= ccore5(temp);
  return temp;
}

#pragma design top
void test(ac10 a, ac10 b, ac10 &out)
{
  ac10 temp1 =0;
  ac10 temp2 =0;

  temp1=ccore1(a,b);
  temp2=ccore2(a,b);

  out =temp1+temp2;
}
```

Figure 5.4: snippet of a testcase

Figure 6.4 shows snippet of testcase.Self_checks have to be added for the testcase. Figure 6.5 shows the self-checks which are added for above testcase.

self_checks are the functionality explicitly added by the user to test the correct functionality of any design.



```
proc perform_self_checks_on_cat2slec_output { args } {

    set mode                 [lindex $args 0]
    set cat2slec_log         [lindex $args 1]
    set slec_script          [lindex $args 2]
    set spec_wrapper_header  [lindex $args 3]
    set spec_wrapper_source  [lindex $args 4]
    set impl_wrapper         [lindex $args 5]
    set cat2slec_debug       [lindex $args 6]

    #self_checks to be added here
#     check_grep $slec_script {mark_loop .* -pipeline .*}
    check_grep $cat2slec_log {CTS-MLP} 1
}

    #Patterns to be found
    set mark_hier_pattern "mark_hierarchy"
    set map_hier_pattern "map_hierarchies"
    set slec_optim_pattern {Slec Optimized Port}
    set cat_optim_pattern  {Catapult Optimized Port}
    set pmim "CTS-PMIM"
    set cims "CTS-CIMS"

    #Values expected
    # Warning given when ports optimized for a CCORE function
    set pmim_count 0
    # Warning given when map_hierarchy dropped
    set cims_count 0
    # Equal to CCORE functions expected
    set mark_count 5
    # Equal to number of CCORE instances expected
    set map_count 6
    # Dumped when ports optimized (expected)
    set slec_optim_count 0
    set cat_optim_count 0


    check_grep $cat2slec_log $pmim $pmim_count
    check_grep $cat2slec_log $cims $cims_count
    check_grep $slec_script $mark_hier_pattern $mark_count
    check_grep $slec_script $map_hier_pattern $map_count
    check_grep $slec_script $slec_optim_pattern $slec_optim_count
    check_grep $slec_script $cat_optim_pattern $cat_optim_count

}
```

Figure 5.5: snippet of self_checks added for the testcase

Self_checks are really heplful since if they fail there is a clear mismatch of functionality handling between the user and the tool. The self_checks added in the following functionalities are as follows:

- CTS-MLP

  This self check is added to check the pipeline functionality of the design. If the design is all_piped then atleast one should be pipelined. For no_piped no loop should be pipelined. For top_piped main loop should be pipelined.

- CTS-PMIM

  This self_check is added when user expects any port to be optimized by SLEC. When the functionality inside CCORE represents a dead logic or logic which is of no use such ports are to be optimized.

- CTS-CIMS

  This self_check is added when any map_hierarchy is dropped by SLEC. Map_hierarchy represents the number of hierarchy present in the design.

- map_count

  This self_check is added to check that the number of times CCORE instantiated by SLEC is as per user's expectation or not.

- slec_optim_count

  This self_check is added when any port is optimized by SLEC.

- cat_optim_count

  This self_check is added when any port is optimized by Catapult.

5.12 Summary of the testing

There were 500 cases present in the qa_repository. Self_checks were added to 500 testcases.Each design has to be viewed and functionality has to be added to them.

Chapter 6

Automation using PERL

6.1 Adding the top_name feature

6.1.1 problem statement

The top_name was to be added to the given testcases

6.1.2 Algorithm

Any test-case in the SLEC HLS flow runs with a setup file. This file is generally written in TCL.The following algorithm was followed to add the top_name option. The top_name option for a particular design has to be found from the slec.log after running the testcase. The value of top is then stored in a variable. The tcl file is now open and has to traverse line by line to find the build_design option. The top_name option has to be then added with the build_design option
The whole task was completed by a perl script. However some of the cases had to be treated manually.

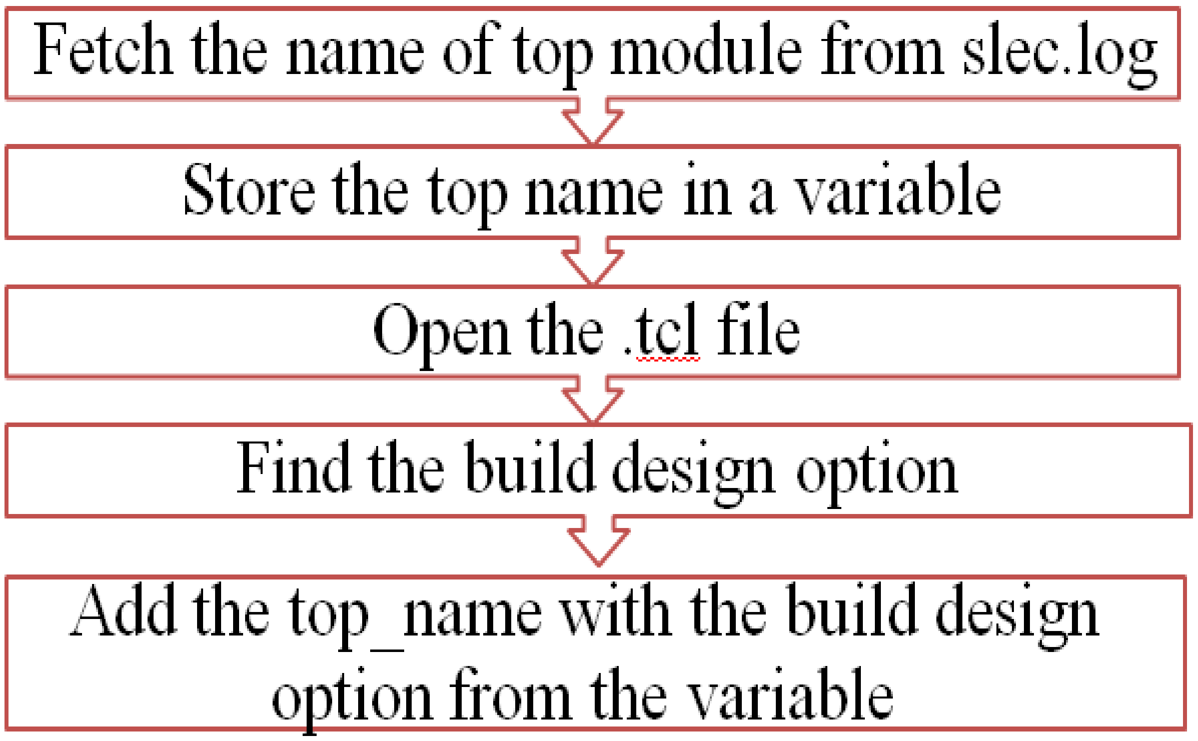

Fetch the name of top module from slec.log
Store the top name in a variable
Open the .tcl file
Find the build design option
Add the top_name with the build design option from the variable

Figure 6.1: Algorithm For the problem

6.2 Cleaning of skiplist

The testcases which ran in the central regression previously had only the top_piped version. Afterwards when they were converted to the all_piped and no_piped version,some of them started failing. These test-cases are added in the skiplist. Cleaning of skiplist is done by cleaning those failures

Chapter 7

Conclusion

Formal verification has several advantages and is highly flexible over simulation and other verification methods. Simulation requires input vectors while SLEC uses formal verification and hence it does not require input vectors. This leads SLEC to have lesser capacity issues and greater performance. HLS has eased the EDA since a specification design can be converted into RTL or implementation design which leads to decreased time to market. SLEC can prove two designs to be functionally equivalent inspite of structural differences.While testing different features of SLEC-HLS flow it was seen that SLEC even leads to capacity issues where two designs which need to prove formally equivalent had a simpler functionality.

Chapter 8

Future Scope

SLEC(Sequential Logic Equivalence checker) is a LEC which uses formal verification.In future it can be used for testing various features of SLEC HLS flow.The verification methodolody of SLEC is continuously improved by adding various features.The testing of this features has to be continuously done in order to make it robust.

Bibliography

[1] http://vlsi.pro/formal-verification-an-overview/

[2] SLEC user manual

[3] SLEC Product Family Datasheet

[4] SLEC CCORE Reference Manual