BEHAVIORAL LANGUAGE CONVERTER TO IMPROVE HLS QUALITY OF RESULTS

by

Himanshu Patra

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Benjamin Carrion Schaefer, Chair

Dr. William (Bill) Swartz

Dr. Joseph S. Friedman

Dedicated to my loving Family, Friends and Teachers.

BEHAVIORAL LANGUAGE CONVERTER TO IMPROVE HLS QUALITY OF RESULTS

by

HIMANSHU PATRA, BE

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

May 2019

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor, Dr. Benjamin Carrion Schaefer, for his continued support, motivation and believing in my ability to complete the thesis. I have benefited considerably from his immense theoretical knowledge and practical experience in my research. I especially thank him for carrying such great patience in helping the students under his guidance; I had an extraordinary self improvement. Thank you Professor.

I would also like to thank Dr. William (Bill) Swartz and Dr. Joseph S. Friedman for being on my thesis evaluation committee. I would like to thank the Department of Electrical Engineering and Computer Science at The University of Texas at Dallas and the Design Automation & Reconfigurable Computing Lab (DARC Lab) for providing me such an amazing platform to show my talent. I heartily thank my lab mates Jiyanqi, Farah, Mahesh, Monica for their help. I am also Immensely thankful to my closest friends Subhendu, Ruchul, Sujit, Amrit, Mona, Ashish, Anand, Ajay, Yash.

My heartfelt thanks to my family members, Aunty, Mama, Papa, Papu, Jyoti, Priti and Ani bhai. My biggest thanks to the one and only hero in my life, KunaMamu, without you none of this would ever be possible. My loving thanks to the love of my life, Krishna Kabi, I wouldn't be where I am today, if it's not because of and for you.

BEHAVIORAL LANGUAGE CONVERTER TO IMPROVE HLS QUALITY OF RESULTS

Himanshu Patra, MSEE
The University of Texas at Dallas, 2019

Supervising Professor: Dr. Benjamin Carrion Schaefer, Chair

High Level Synthesis (HLS) is one of the most emerging fields of research that is helping the industries to reduce the time-to-market constraint. With the advancement of the HLS, till now approximately 100 companies have adopted this for IP(Intellectual Property) design. The higher we go in abstraction level, the lesser time it takes to design and verify an IP or a design. It will take tremendous amount of time in comparison to HLS, to implement an IP in verilog or VHDL. This thesis focuses on improving the Quality of Result of HLS by an automation flow with added security(Obfuscation) to the IP.

If it's Hardware Description Language (HDL), the input type does not have any impact in the Quality of Result, because the architecture are being defined in the low level HDL coding. In HLS, as we are coding in higher abstraction, the architecture are being defined by the HLS tools. This might vary based on the optimization techniques that the tools follow as a front end parser, which converts the input to a CDFG (Control Data Flow Graph), which in turn is the input to HLS. So, different input languages will end up generating non-identical architecture for the same design, one of them being the best and one of them being the worst. We need to decide which input language should be considered for a particular design, which will generate the best QoR.

One of the issues that also needs to be addressed is SECURITY. The IPs must be made secure or encrypted to prevent unlawful usages. Usually, consumers buy IPs for a trial or evaluation purpose for a few days/weeks to verify the efficiency of the IP to their requirements. If the IPs will be made visible to the consumers in trial phase, they won't need to end up buying it, which will be a huge loss for the vendor. To prevent this kind of scenario, we need to encrypt the IP. This thesis proposed a method "Obfuscation in HLS", which is the most easy, inexpensive and efficient way to encrypt an IP. However, the QoR(Quality of Result) of HLS is greatly impacted with Obfuscation.

This thesis, proposes a unique conversion tool which converts the input language to SystemC from ANSI-C to save time, resources and will also automatically generate which input language to choose for the best QoR while adopting HLS.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF NOMENCLATURE

ADPCM	Adaptive Differential Pulse-Code Modulation
AES	Advanced Encryption Standard
ALAP	As Late As Possible
ASAP	As Soon As Possible
AVE8	Average of eight numbers
BDL	Behavioral Descriptive language
BIP	Behavioral Intellectual property
C2SC	ANSI-C to SystemC conversion tool
CDFG	Control Data Flow Graph
CPU	Central Processing Unit
CWB	CyberWorkBench
DUT	Design Under Test
EDA	Electronic Design Automation
FD	Forced Direct
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FU	Functional Unit
HDL	Hardware Description Language

HLS	High Level Synthesis
IFF	Internal File Format
IP	Intellectual Property
LS	List Scheduling
RTL	Register Transfer Level
S2CBench	Synthesizable SystemC Benchmarks
SoC	System-on-Chip
SW	Software
VHDL	Very High Speed IC Hardware Description Language
VLSI	Very Large Scale Integration

CHAPTER 1

INTRODUCTION

VLSI (Very Large Scale Integration) is a process of integrating hundreds of thousands or millions of transistors to a single chip called IC (Integrated Circuit). In the 1970s when semiconductor and other communication technologies were developed, VLSI came into the market. In the early stages, an electronic circuit would consist of a CPU, ROM, RAM and logic functions, VLSI integrated those all into a single chip. The applications of a VLSI IC are image and video processing, communication, high-performance computing, which are rising rapidly. This is as predicted by Intel co-founder Gordon Moore, that the number of transistors in a chip doubles every year, which in turn modified to 18 months instead of a year.

1.1 VLSI Design Flow and HLS

Once the specification is ready, the design goes through the whole HLS flows. Figure 1.1 shows a complete typical flow of a VLSI design. One of the problematic and important stages are HDL coding and RTL verification. HDL coding is being done in verilog or VHDL, which has the functionality and timing information. It is quite difficult to exploit the design in HDL. In the early stage of every project, a behavioral model has been generated to verify that, is it functionally implementable and specifications are correct or not? Then it goes through a phase of RTL coding. Let's take a minute here and think, won't this be great if we can directly implement the design in HDL language, right after it passes the preliminary verification phase with just a single click. HLS basically converts a particular software code of a design to HDL(verilog or VHDL). There are also some tools which generated testbench for them too. To make life easier and reduce time-to-market, many companies are adopting HLS.

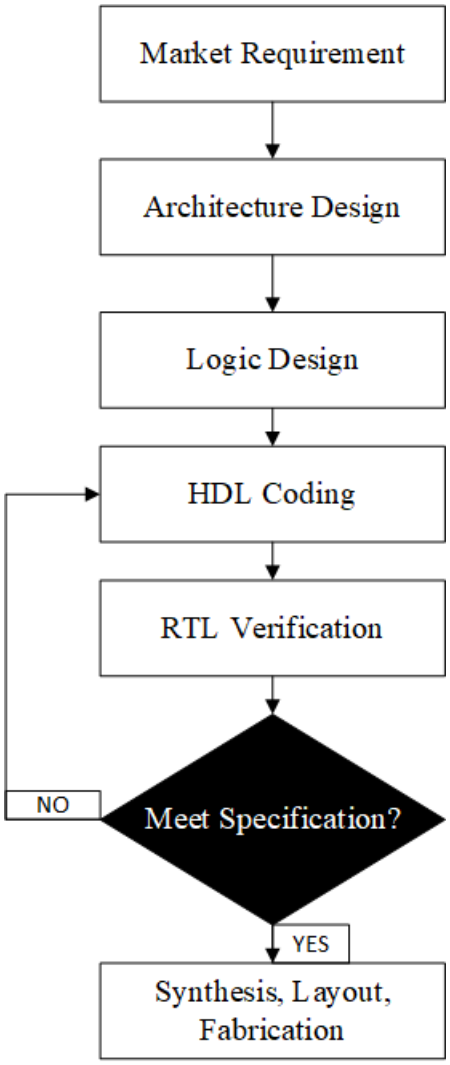


Figure 1.1: Typical VLSI Flow

Now, as HLS is commercially available and being implemented for many designs, we need to know which input language is better for HLS. Is It tool specific or design specific? If design specific, is it same input language for all the designs or different design will behave differently? We will discuss more details about this in the next three chapters. Unlike software, HDL will have architectural information in the coding, so it won't make much of a difference if we use VHDL or Verilog. But in HLS, the behavioral description doesnt have any information regarding the architecture of the design. So the architecture is different

for different input languages. In this thesis, we will design a flow which will give us the information that which input language is better than other for a specific design.

1.2 Thesis Flow

The thesis has been divided into three main chapters. In the second chapter, we will learn about High Level Synthesis how HLS has impacted the semiconductor industry and will impact in the future. In the third chapter we will discuss about why we need to protect BIP(Behavioral IP). And we will discuss about two well-known techniques for IP protection and compare both of them. In the fourth chapter we will explain in detail about a proposed flow, which will take input as a ANSI-C code and do some specific processing and at the end, it will tell us which input language is better than other. In the end, we will discuss the experimental results of the designs with the flow.

CHAPTER 2

HIGH LEVEL SYNTHESIS

To fully understand the benefits and potential of High Level synthesis, it is necessary to put things in the perspective of Hardware Design Flow. Nowadays all the projects start from a required specification. In most of the cases it is a written document, but very often an executable/working model has been created. The model will be in ANSI-C/C++ or SystemC. In this early stage of the design, a very little or no hardware implementation details are decided. Its primarily for testing and verifying the functionality. Once it is tested, the architecture will be defined for how this design is going to work. The functionality determines What the design perform, and the architecture defines How its going to perform the task. After all the architecture have been defined, the design team write the hardware codes(Verilog/VHDL) by using the architectural decision.

This is the most difficult task. The important problem is the manual nature of the whole process. Finding a suitable architecture is not easy, and finding an optimal one is even more difficult. In this process, the hand-coded Verilog/VHDL design will be tested and validated, all the bugs will be reported. And a whole lot of time spent to fix those bugs and move on to find a new bug. There is a good chance that it will be an endless process if it didnt have to end at some point before the deadlines. Now, if we think about a bigger design, the applications will be much more complicated, which again increase the probability of having huge bugs, and much complicated to solve as the design grows. To give an overview: the device we were using 15-20 years ago, and the same device now, the change is so gigantic it is self-explanatory how complicated it has become. Everything has changed to become more sophisticated and complex. Everything is being dramatically changed but the RTL creation process. It is not out of the blue that verification is the bottleneck of every ASIC project.

As the complexity of designs have gone so far, it also demands faster time-to-market delivery, which in turn requires even sophisticated methodologies. The assembly language

was first introduced in the 1950s, till that time binary code was being written to program a computer. To improve the productivity later, we developed high-level languages and respective compilers. Then the gate level simulation started in the late 1970s. After around ten years, cycle based simulation became available. Then Hardware description languages were adopted widely by multiple simulation tools. Then High Level Synthesis(HLS) came into the market.

High level synthesis finds the root cause of the bugs because it gives an error-free path from higher abstraction to RTL. When we work on high-level abstractions, decidedly fewer details of the architecture is needed. We dont need to worry about the underlying process, clock, hierarchy, etc., just need to concentrate on the behavior of the design. Example: Once the design engineer decides behavior and constraints, High level synthesis takes care of the allocation, scheduling, and binding. So radically the verification burden is reduced. [11] describes a formal flow of HLS.

2.1 Steps in High Level Synthesis

The HLS tool takes the behavioral description, constraint file and a technology library file as inputs. Figure 2.1 shows the steps of HLS. The first step is to convert the high-level language (C/C++/SystemC) to a CDFG(Control Data Flow Graph).

In this step basically, it checks for syntax and parses it. Then it optimizes the internal file, mostly dead code elimination, removes redundant operations. This step is critical because a poor optimizer may lead to adding an extra overhead in area or delay. A small example is given below:

```
Void example(c)
{A = c - c;}
```

In the above line, the value of A is always 0 irrespective of the value of c. If the optimizer does not optimize the above expression, it will tend to add two more registers(for storing

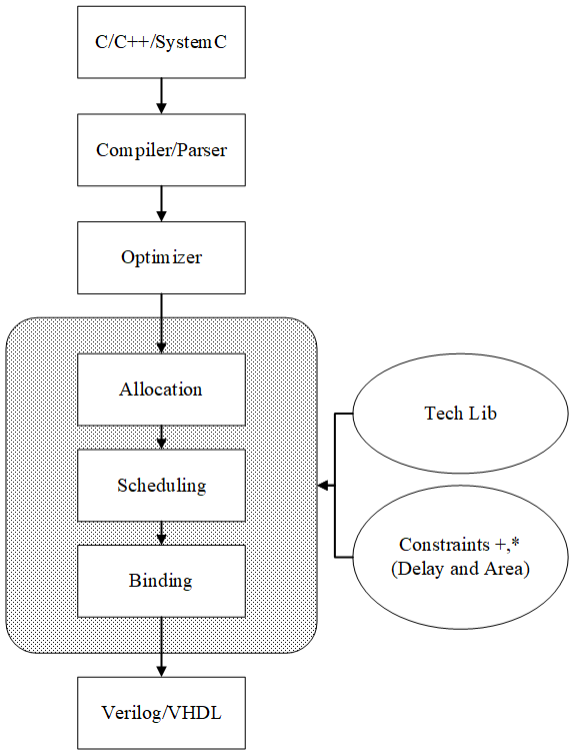


Figure 2.1: Essential steps in HLS

c) and one extra adder. What if this small function is called many times, each time it will require unnecessary delay and area, which is extremely redundant and not a feasible optimizer. The Control Data Flow Graph(CDFG), will be taken as input to three main HLS steps, i.e., Allocation, Scheduling, Binding.

2.1.1 Allocation

This step takes the input from the optimizer(foo.IFF), it defines the maximum number of functional units needed for the application/design. In this step, a constraint file has been generated which would contain the functional unit details as number and type of the units required for the design. Frequently, to adopt the maximum level of parallelism, the HLS tool will use the maximum number of functional units. Then the user has the flexibility to modify the number of functional units that the synthesizer can instantiate, but this can only be done after the allocation process. Below is an example which will explain in detail:

```c
int main(){

   /*   input   */
int a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11

   /*   output   */
int o1,o2,o3;

   /*   Variable   */
int x, y, z;


x  = a1 + a2;
o1 = (x - a3) * 5;
o2 = (a4 + a5 + a6;
y  =  a7 * a8;
z  =  y + a9 + a10;
o3 =  a11 * 9 * z;
```

(a)

Adder	8
Multiplier	2

Adder	1
Multiplier	1

(b)

Figure 2.2: Number of Functional Unit used for the snippet. (a) ANSI-C code (b) Functional unit allocated because of maximum operator count and minimum operator count.

The example C-code in figure 2.2a, makes use of multiple additions and multiplication operations. 2.2b shows what is the maximum number of adders and multipliers are required. It basically maps every operation to a single functional unit to use the highest level of parallelism. In this above scenario, It would need eight 32-bit signed adders(data type is int which is 32 bit long) and two 32-bit signed multipliers. But in 2.2b it also says, in the second table, what is the minimum number of functional units are required to execute the above code in hardware. This gives us two scenarios, in the first case it will end up making a larger circuit but with better performance due to less latency. In the second case, the area has been reduced as we are trying to use a minimum number of functional units, but it comes with a cost of performance. To be exact, the clock cycle will go from 1 to 6, and the resource will go from 1 to 5. In some FPGA targets, this won't be the best design, because of clock period constraint and also due to the cost of the multiplexers will it require to share the functional units.

2.1.2 Scheduling

Scheduling follows the allocation step. Usually, the behavioral description language is written in an HDL(Hardware Description Language) like Verilog, VHDL. Since there are numbers of languages can be treated as input language, so a canonical representation is necessary. The most common canonical representation is a CDFG(Control Data Flow Graph). Two main parts of CDFG are essential to describe an RTL(Register Transfer Level). I.e., FSM(Finite State Machine) and Datapath. The datapath consists the necessary FU(Functional Units) and storage, where the FSM includes the number of states, transition of states and actions need to be taken care in each transition. This scheduling process basically follows the CDFG.

Let X is the total number of operations that need to be scheduled, which are obtained from the HDL code. Let $Y_j \subseteq$ X is an operation which depends on $Z_j \subseteq$ X, Z_j needs to finish its operation before Y_j begins. In HLS there are multiple this kind of scenarios which will come into the picture. There is a module in HLS which contains operations like Adder, multipliers, etc., which also has information about the area, delay, and power. Many scheduling algorithms have been purposed in the past. Most of them are heuristics as it has been shown that resource constraint scheduling is an NPhard problem [12, 2, 3].The algorithms which can take care of scheduling can be divided into two categories, i.e., heuristic and deductive. Heuristic depends on experience and histories like resource constraint, time constraint, resource-time constraint, and un-constraint. Deductive is accurate unlike based on history, such as linear programming which generates the most optimal solution but it takes a humongous amount of time so that deductive is not feasible for all type applications. However, there are algorithms which generate results very close to the optimal solution in less amount of time. To name a few: ASAP(As Soon As Possible), ALAP(As Late As Possible), FDS(Forced Direct Scheduling) are a time constraint, and LS(List Scheduling) is resource constraint. Let's have a brief overview of each.

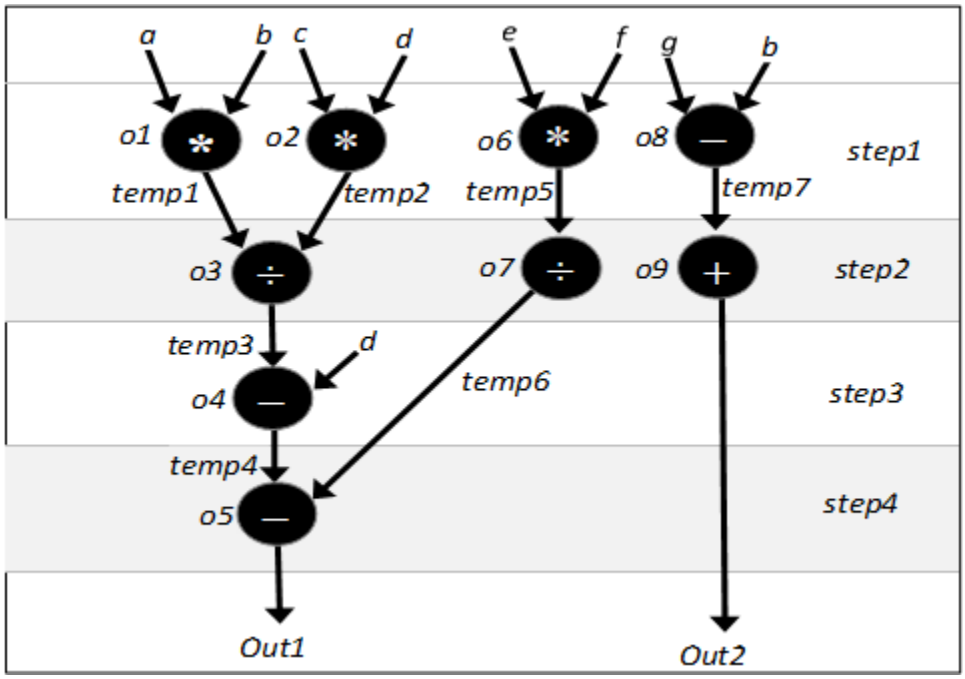


Figure 2.3: As-Soon-As-Possible flow example

2.1.2.1 ASAP

As-Soon-As-Possible [12] is the most common type of scheduling algorithm. It doesnt consider the resource constraint. It first finds out the maximum number of control steps, then tries to schedule each operation at a time into the earliest control step. The scheduling is again subject to successful completion of predecessor operation, i.e., the operation will only be scheduled, if and only if all the above operations, on which the current operation is dependent on, have been executed already. ASAP will be successful if all the operations can be scheduled in the given control steps.

Lets consider an example: Out1 = ((ab)/(cd))-a-((ef)/b) and Out2 = (g-b)+f and the maximum nuber of control steps, S = 4. Figure 2.3 explains ASAP scheduling.

Algorithm:

Do:

If x has no immediate predecessor(operation which computes from primary input)

Steps	S1	S2	S3	S4
Adder	0	1	0	0
Subtractor	1	0	NIL	NIL
Multiplier	3	0	0	0
Divider	0	2	0	0

Figure 2.4: As-Soon-As-Possible resource requirement

Control_step(x) = 1(earliest control_step)

Else control_step(x)= Max(control_step(Y_1, Y_2, Y_3))+ 1, yi= immediate predecessors of x.

End if the control_step of the latest operation M $--> $ Success

As we can see, o1,o2,o6,o8 are completely independent operations, i.e., they directly depend on input values. So these operations will go to control_step-1. However, O_3, O_7, and O_9 inputs are the direct output of O_1, O_2, O_6 and O_8. So based on the ASAP algorithm:

i.e. control_step(O_3) = maximum(control_step(O_1), control_step(O_2)) + 1 = 1 + 1 = 2

As we can see, O_1, O_2, O_6, O_8 are completely independent operations, i.e., they directly depend on input values. So these operations will go to control_step-1. However, O_3, O_7, and O_9 inputs are the direct output of O_1, O_2, O_6 and O_8. So based on the ASAP algorithm:

i.e. control_step(O_3) = maximum(control_step(O_1), control_step(O_2))+1 = 1 + 1 = 2

Based on the above explanation the control step of o4 is step-3 and o5 is step-4. This operation can be done in four steps, making it successful. As discussed earlier, this does not consider resource constraint, followed the scheduling the resource requirement would be determined. Figure 2.4 is the resource requirement for each step.

2.1.2.2 ALAP

As-Late-As-Possible [12] scheduling is pretty much similar to ASAP, instead of assigning the resources sooner it will assign it to the latest possible time step. First of all, the number

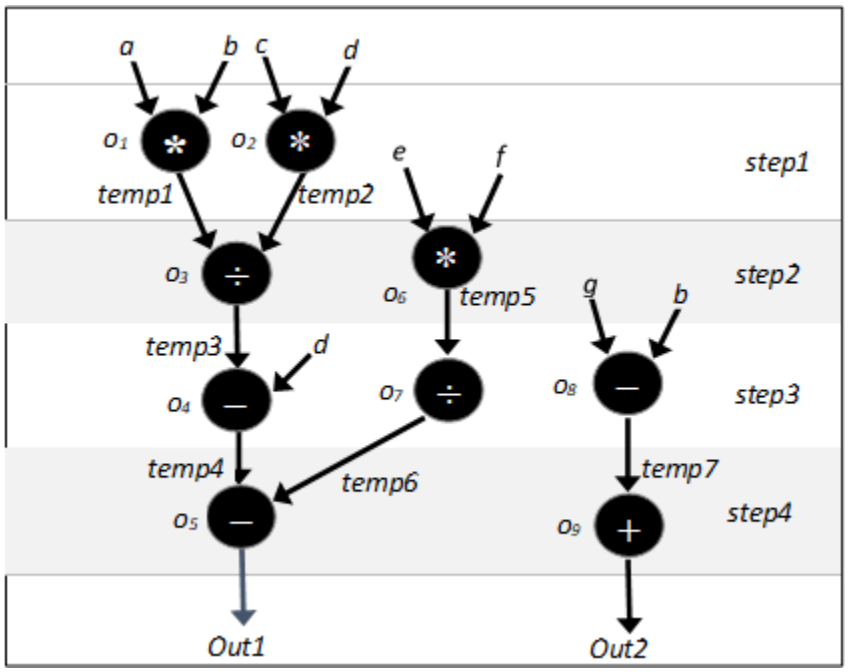


Figure 2.5: As-Late-As-Possible flow example

of control_steps will be decided. Then the ALAP will put the latest operation in the latest time slot. An operation will be scheduled if and only if the successors are scheduled in later time slots. If all the operations can be scheduled(moving backward) in the given time slots, then the scheduling will be successful.

Algorithm:

Do:

If x has no immediate successor(Final operation to generate output)

Control_step(x) = M(Latest control_step)

Else control_step(x) = control_step(y) 1: y = immediate successor of x.

End if all the operations are completed within step1 $--> $ Success

Lets consider the same example as ASAP: Out1 = ((ab)/(cd))-a-((ef)/b) and Out2 = (g-b)+f and the maximum number of control steps, S = 4. Figure 2.5 is the example of ALAP scheduling.

Steps	S1	S2	S3	S4
Adder	0	1	0	0
Subtractor	1	0	NIL	NIL
Multiplier	3	0	0	0
Divider	0	2	0	0

Figure 2.6: As-Late-As-Possible resource requirement

In ALAP algorithm as we see, O5 and O9 dont have any other successor, that means they directly generate the output. So these two operations can be in the control_step 4(the latest one). The immediate predecessor on O_5 is O_4 and O_7. So as per the algorithm discussed above, the control_step(O_4 and O_7) = M(4) 1 = 3, the same applies to O_8 too. Like this the control_step of O_3, O_6, and O_1, O_2 is step2 and step1 respectively.

As all the operations are finished within step1, this is considered as successful scheduling in ALAP. Figure 2.6 is the resource requirement table for the above operation.

As we see, the multiplier, divider, and subtractor in step-2, 3 and 4 are NIL because the those can be reused from step-one, two and three respectively.

2.1.2.3 Limitations to ASAP and ALAP

As we say in ASAP three multipliers, two dividers, one adder, and one subtractor are needed. And in ALAP two multiplier, one divider, two subtractors, and one adder are required. So in ASAP we saved one multiplier and one divider, but have to bear the penalty of one more subtracter. We can make it better if we can combine both ASAP and ALAP. Figure 2.7 is the combination of ASAP and ALAP, which is an example of the optimal solution. Now we can conclude that ASAP, and ALAP is heuristic solutions which wont always give the optimal solutions.

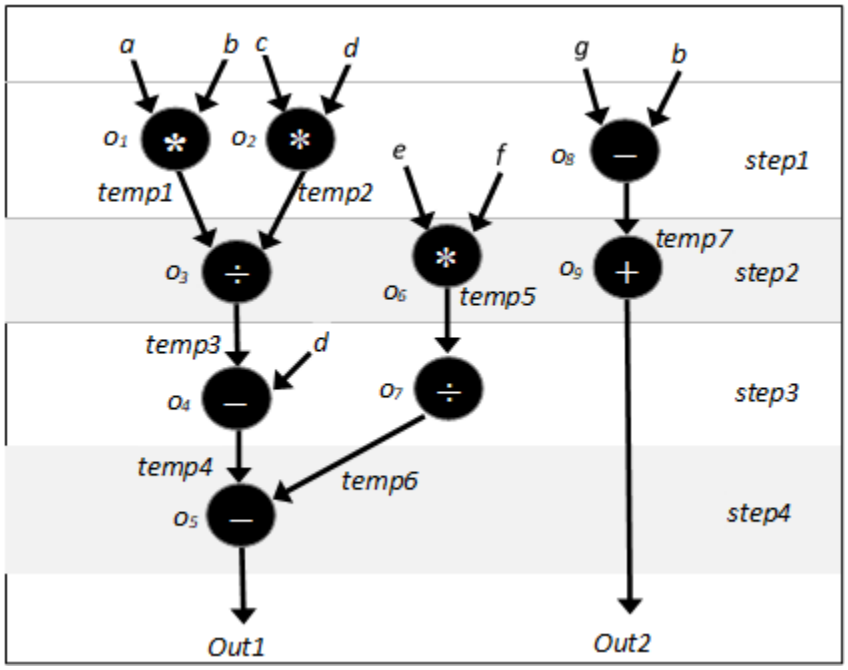


Figure 2.7: FDS flow example

2.1.2.4 FDS

Forced-Direct-Scheduling [4, 5] takes advantage of both ASAP and ALAP, where resources get uniformly distributed across the time constraint schedule. Advantages of ADS is higher functional unit utilization and minimizes the number of units used. FDS first finds the scheduling driven by ASAP and ALAP. However, if the schedules of both ASAL and ALAP are same, I.e., same control step is assigned by both ASAP and ALAP then FDS doesnt consider this. Because there is no flexibility to change the current scheduling.

But in another hand, FDS will find the flexibility range of such operation whose ASAP and ALAP does not match, i.e., control_step assigned by ASAP to control_step assigned by ALAP. Now the operation will be scheduled in each of the flexibility range so that the resource count will be minimum. To accomplish this, operations of each type are considered one by one. For a given type of operation, we find out the number of operators required by analyzing all the combinations of placing the operators in individual places. Once, one type of operation is done, we move further to another type. Figure 2.7 is the example of FDS.

2.1.2.5 LS

All scheduling algorithms we discussed till now are the time constant, in terms of, number of control steps, which is heuristic based. Now we will review another algorithm, called List Scheduling [6] which is again heuristic based, but it will be resource constraint, unlike ASAP, ALAP and FDS.

FDS tries to put a maximum number of operations into one control step, subject to resource constraint and data dependency. In the first step, it creates a ready list to keep track of the data-ready operations. I.e., the operations which are ready to be scheduled, or all the predecessors have already been scheduled. As long as there are operations in the ready list which meet the resource constraint, are moved from in-ready to current control step. If there is a case where multiple resources need to be scheduled in the same time slot, the choice is taken as per priority function. The functional unit having less flexibility, meaning the actual range, FU can be scheduled($ASAP_i$ - $ALAP_i$, smaller mobility), will be given higher priority as it has less place to go. Because delaying them will unnecessarily increase the time length.

This comes to an end of the Scheduling. Now we will discuss the third most crucial step of HLS, i.e., Binding.

2.1.3 Binding

Binding is the last step of HLS, where each operation is mapped to a functional unit and each variable to a register. After scheduling, each operation has to bind to at least one available resource. As the number of resources is limited, the binding algorithm has to be optimal in order to bind multiple operators together to a single available resource. The basic condition is, the maximum number of the same type of operators can be bind together in a single time slot, if and only if the available number of resources for the particular type of operation is equal or more than the total number of operators. The area and latency are the

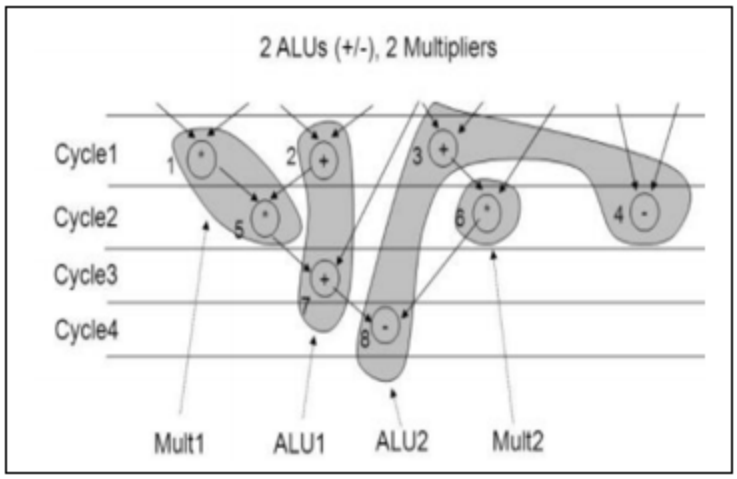


Figure 2.8: Binding Example

two important parameters we need to take care in any VLSI circuits. Figure 2.8 shows an example of binding.

In a resource dominant circuit, the area is Total area of resources bound in a given operation; latency is the time between start and end time of sink and source respectively. But in a non-resource dominant circuit, the area is determined by a logic circuit, registers, wiring and control and latency are determined by wiring and logic design. But there is a way to optimize is to provide a detailed architecture at the time of allocation. Below are some popular binding algorithms:

- Clique Partitioning
- Left edge algorithm
- Iterative algorithm

Let's consider the same example we followed in the scheduling(ASAP) process. Let's assume the number of ALU(adder, subtracter) available is one, and the number of multiplier/divider available is two. Figure 2.9 shows the binding of resources.

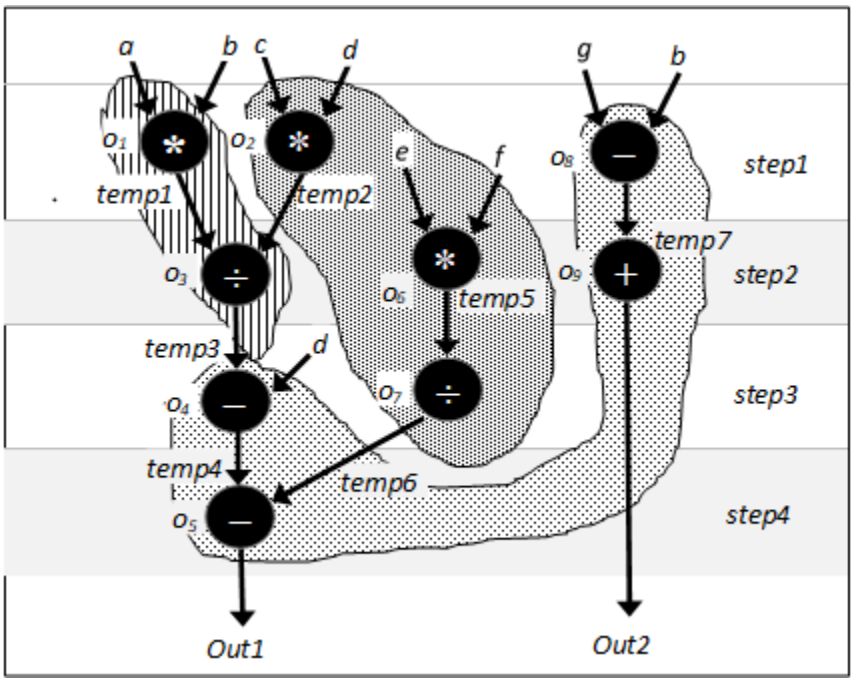


Figure 2.9: Binding of the equation explained above

As seen in figure 2.9, operations O_2, O_6, O_7 are in three different time slots and can be bound to one available resource(multiplier/divider) MD-1. Similarly, O_1 and O_3 can be bound to second multiplier/divider. Alternatively, O_7 can be added to the group O_1 and O_3 too. O_8, O_9, O_4, and O_5 can be bound to the one available ALU.

This brings us to conclude the three important steps of HLS. This step can either be sequentially or simultaneously performed. But it becomes a lot complicated when executed parallelly. The basic solution is to follow a timing constraint approach for a resource constraint problem and to follow a resource constraint approach for a timing constraint problem. Figure 2.10 explains the three steps in a single overview.

2.2 Advantages of HLS

The code can be easily converted to hardware from software in high-level synthesis. There are many compilers available which can compile C code to generate RTL. Then the RTL

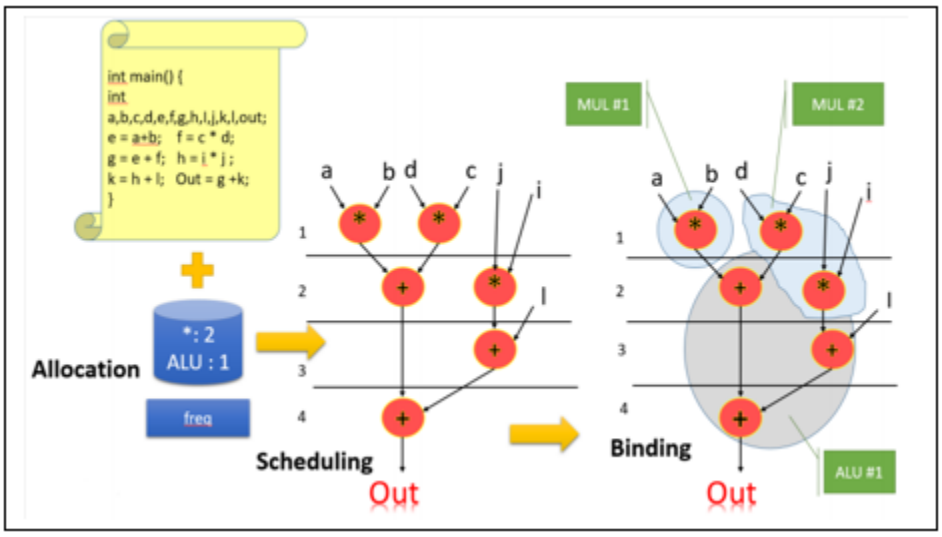


Figure 2.10: Allocation, Scheduling and Binding in HLS

can be synthesized by any FPGA vendor tools, in-order to implement in gate arrays. But, simply compiling software code for FPGA is seldom sufficient. It is often the case where the algorithm needs to be modified until we get the optimized one. However, rather than performing this step manually, HLS takes care of all this and the restructuring can be performed at a higher level.

HLS tool analyzes the structure of the algorithm and creates the control path. In contrast with this, the implementation of RTL when done manually requires explicit design of both control and data path. For a complex design, the effort to make control path is as much as designing the data path. The environments which has co-processor meaning both hardware and software in FPGA and CPU respectively, HLS equips it to work efficiently for both hardware and software. This also makes it easier to rapidly swap between hardware and software.

Nowadays, synthesis tools can utilize parallelism in the main three ways:

- Data dependencies can be analyzed by examining the data flow. From this pipelined can be achieved.

- Also, by analyzing the looping conditions in which there are limited data dependencies, can be made in a pipelined architecture, where the next iteration can be started before the previous iteration ends.

- By unrolling loops. Multiple parallel blocks are built to make the iterations parallelized.

It is not a surprise that verification is the biggest challenge in any hardware design. HLS makes it quite easy by doing the verification at a higher level. However, in the end, the RTL needs to be verified to check the final algorithm results before it gets into the implementation phase. In some HLS tool, it generate RTL testbenches from high-level verification code.

2.3 Disadvantages of HLS

Unfortunately, HLS is not as easy as it sounds. Every aspect of the design is in software. We have decidedly less or no access to get in touch with the hardware. With all the tools, the algorithms have to be written in a specific way in order to equip the synthesis tool to utilize parallelism and optimize it. But most of the times we code design in a software style, as a software design thinking, which leads to a very inefficient and poor performance circuit. The main factoris is that FPGA is a hardware-based design, not software. The algorithm design and execution is pretty stable in software, but the analyzation in hardware is quite complicated.

The major difficulty is pointer based algorithms, HLS seems to not able to work very well for pointers. Pointers pose two main challenges when implementing in an FPGA. First, FPGA has its own distributed memory with an address assigned, but its too small and independent block. Second, often the variables are stored in registers. Another devil is recursion, recursion based software design is very inefficient for HLS. FPGA implements each function as a hardware block, and stores the local variables in a register which is mutual in each invocation. So recursive algorithm uses its identical block instead. It ends

up adding a lot of extra overheads. At last, the RTL generated by HLS tools are not quite readable and very complicated to modify the code, which makes it challenging to do any minute changes in RTL.

2.4 Commercial HLS tools

There are many numbers of HLS tools are in the market. CWB is one of the lead in the run. Table 2.1 are some of the tool names:

Table 2.1: Commercial HLS tools with companies names

Tool Names	**Company**	**Input Language**
Vivado HLS	Xilinx Inc.	C, C++, SystemC
CatapaultC	Mentor Graphics	C++, SystemC
Stratus	Cadence Inc.	C, C++, SyatemC
CyberWorkBench (CWB)	NEC Corp.	C(BDL), SystemC
Synopsys	Synphony C compiler	C, SystemC
Intel HLS compiler	Altera	C, C++

CWB is the tool that has been used for the thesis work, which seems to work excellent.

2.5 CWB : CyberWorkBench

CyberWorkBench is an HLS tool. It also generates testbenches for DUTs(Design Under Test), makes it work for verification as well. It accepts SystemC and BDL(extension to C) as input language.BDL is an extension to C language Hardware C. It is ANSI-C based. Some of the features are described in table 2.2.

Table 2.2: Features in CWB

Feature	**Syntax**	**Description**
Variable type (Physical type)	ter/reg/var/mem /reset/clock	Type declarations used to represent hardware (registers, memory, etc.)
Bit width	(Start bit: bit width) (Start bit .. end bit)	Arbitrary bit specification
IO type	in/out/inout	Input/output type declaration
External module spec	outside/shared	Specification of external module for memory /registers/ter/var
Process declaration	Process	Specification of the execution start function
Data transfer type	::=	Assignment operator for describing hardware structure
Operators	::(Concatenation operator) &>, | >, ^>, & >, | >, ^> (Reduction operator)	An operator that concatenates (links) two variables. An operator that reduces the bit width of a variable to one bit
Timing Description	$ (Cycle boundary)	Specification of a clock cycle
Control Statement	wait/watch dmux /mux/allstates	Special control statement used for hardware operations
Special Constant	HiZ	Special constant for hardware

2.6 Summary

This chapter discussed the benefits of HLS and the primary steps associated with it. Widespread use of HLS has triggered many vendors to make BIPs using HLS, not for virtual prototyping but also for real-time designs.

CHAPTER 3

OBFUSCATION

Presently, globalization of IC design creates severe issues and concerns about reliability and trustworthiness. Keeping the time-to-market in consideration, it is impossible to design the whole SoC by in-house tools and IPs. Therefore, companies usually purchase the IPs from 3rd party organizations. Like Apple is buying Intel processors for MacBook. In order to meet the time-to-market constraint, 3rd party IPs (3PIPs) have been used frequently.

Furthermore, companies now adopting HLS to reduce the time-to-market constraint even more. One of the biggest challenge faced by the BIP providers that are affecting the expansion of their business is illegal use of the same IPs. A study conducted by [13], that the BIP providers face appx. $4 billion loss each year because of IP contravention.

Often, BIPs come with a price range just like iTunes, i.e., the BIP consumers must pay for the amount of disclosures of the IP. However, the consumer also can buy the whole BIP. Usually, the second type is 10-100X costlier than the first one. If the consumer purchases the complete source code, then the service by the provider is no longer needed because the consumer has the BIP with full visibility. Another issue that needs to be discussed is, when a consumer interested in buying a BIP, often they buy a trial version to analyze and examine that the performance meets their expectation or not. During this process, the IP cannot be made accessible and visible to the consumers else the consumer need not end up buying the BIP.

To address the above issue, BIP providers need to protect the IP before selling it. Later this chapter, we will discuss about two types of IP protection technique in detail.

3.1 Motivational Example

Figure 3.2 shows how the level of obfuscation affects the performance of the designed circuit when two different parsers of the same HLS tool has been used. The results are shown below,

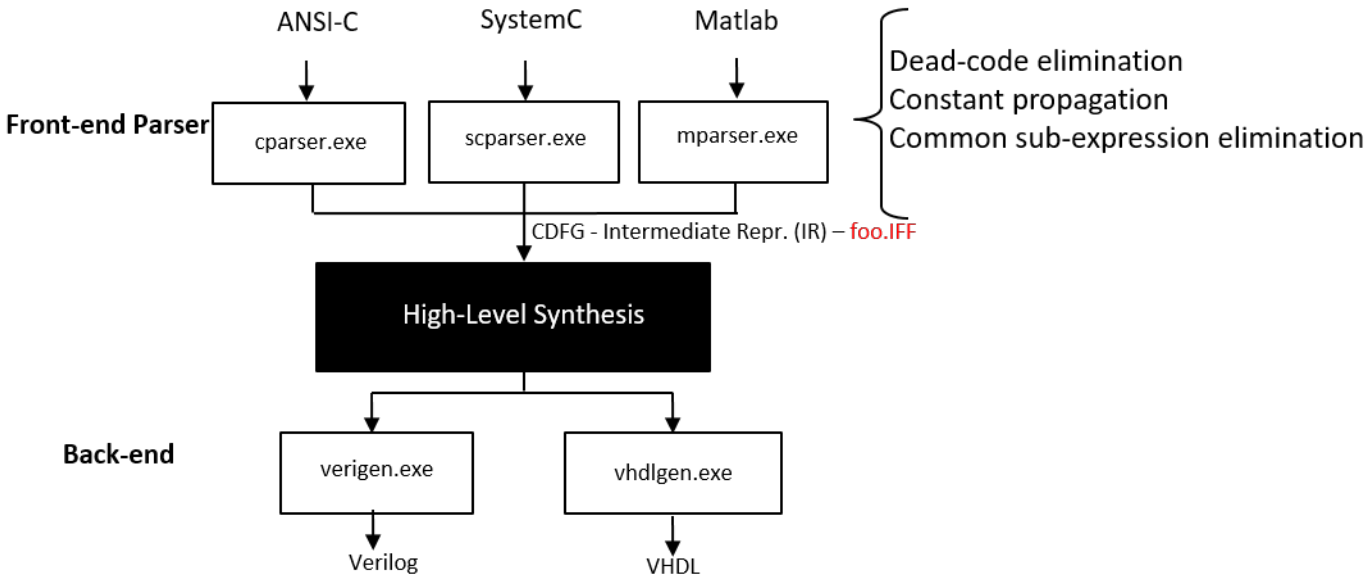
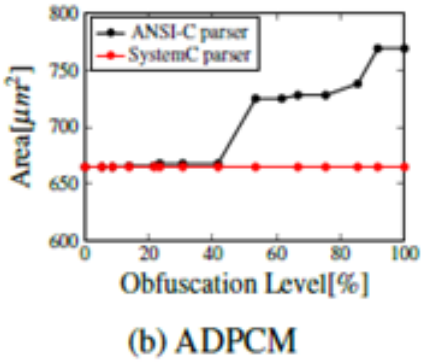


Figure 3.1: Typical HLS flow overview.

taken from S2Cbench [7] benchmark suite. Because CWB supports ANSI-C and SystemC, so the benchmark suite has been converted to C, in order to show the comparison. As shown in the figure 3.2 below the area is increasing monotonically with the increase in obfuscation level, in case of ANSI-C. However, the area remains almost unchanged in case of SystemC parser, because the SystemC parser is a third party professional parser unlike ANSI-C parser, which is a CWB in-house parser. Figure 3.1 shows the the traditional flow of HLS, and it clearly defines that the HLS depends on the front-end parser.

This clearly shows that the different parsers with different level of compiler optimizations lead to different synthesis results. Now we need to rely on the input language of the designs. The same design in BDL and SystemC behave differently with and without obfuscation(IP Protection) because of the parser inefficiency. It is, therefore essential to generate a process, which will do the processing and analysis of a design with two input languages (ANSI-C, SystemC) and provide which one is best for the particular design. Now we will briefly discuss different techniques to protect the BIP.



(b) ADPCM

Figure 3.2: Area degradation of different benchmarks with the increase in level of obfuscation [7, 8]

3.2 CWB Encryption

BIPs are called CyberWares by CWB. In CWB we can use pragmas to protect a BIP. pragma is a unique line which is specific to CWB and can be used to define a different implementation of the same code. Like, if a user is using a for loop and wants to exploit loop unrolling then below is the example with a pragma.

Without pragma:
For(i=0;i<=6;i++)
{*Some operation...*}

With Pragma:
/ Cyber unroll_times = all */*
For(i=0;i<=6;i++)
{Some operation...}

The pragma in the above code is giving instruction to CWB to unroll the loop to a specific number of times or all. The same way CWB can encrypt the source code to protect it from misuses. Figure 3.3 is an example of encryption.

The code in between the pragma start and end will be encoded, and the output is random numbers. But for each encryption, it needs a key and a developer Id. The technique is unique to CWB. The consumer needs to contact CWB to get the encryption utility (command line windows or Linux). The sole purpose of the developer id is to guarantee that the key is unique. The encryption tool is a command line tool, and CWB cannot decrypt the encrypted BIP. The BIP developer will contact CWB department to get the encryption key and the

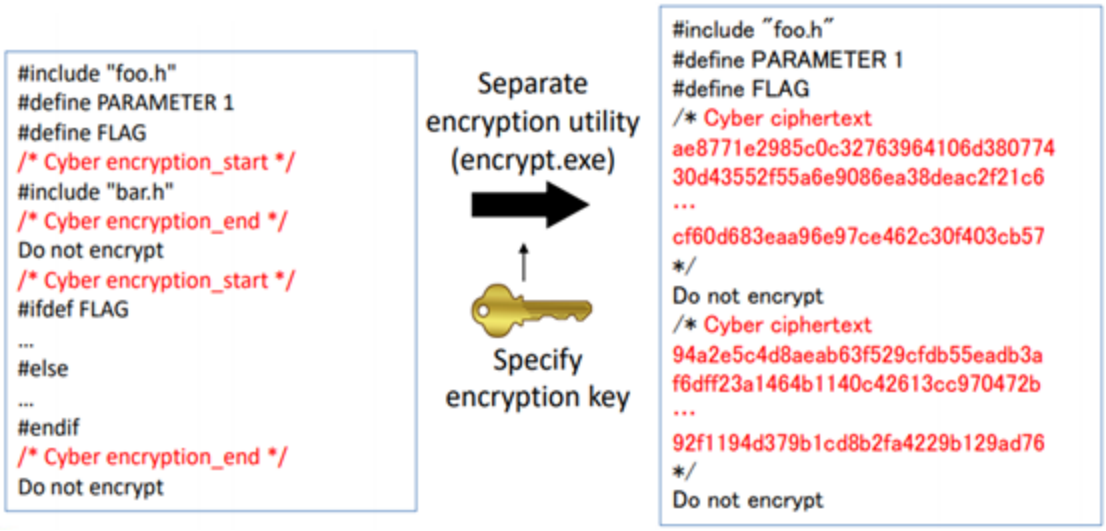


Figure 3.3: Encryption Example in CWB using pragma

developer id. The input to the encryption tool is the source code(foo.c), the Developer id (given by CWB to the developer) and the individual user key (provided to individual users). The output is the encrypted source code(foo.c.encrypt) and license file(cwb_ip_license.dat). Below is a snippet of an example.

The above process works very efficiently. However, there is a major limitation, which creates second thought about it. The whole encryption is CWB specific. The pragmas are only recognizable to CWB, which makes it less efficient. Therefore, we need to adopt some sort of protecting technique, which is more generic and does not depend on any HLS tool. To handle this issue, Obfuscation is the most reliable and inexpensive way of protecting the BIPs.

3.3 Obfuscation

Obfuscation can be described as an intentional act of preserving the functionality of a product to protect the intellectual property by making it human non-readable. Formally:

Definition: Let X is an obfuscator which can transfer a program Y into an obfuscated version X(Y) while still preserving the functionality Y as Z.

$$\{Y\} = F\{X(Y)\} = Z$$

Now its incomprehensible to reverse engineer and recover Y from X(Y). Software obfuscation is entirely different than hardware obfuscation. In software, the goal is to change the structure of the code, which makes it very difficult to reverse engineer and understand by attackers. Obfuscation is the most popular alternative to encryption as it doesnt require any inverse function or encryption key.

In hardware obfuscation, the goal is to protect the functionality by modifying the microarchitecture of the design, such that it cant be reverse engineered. The classification of hardware obfuscation technique is at different levels, i.e., RTL, Gate level, Layout as stated in [106(paper)]. It doesnt include behavioral description because at this stage the microarchitecture has not been defined and obfuscation in behavioral level is almost similar to software obfuscation. Below shows an example of obfuscated BIP snippet of the average of eight numbers using commercially available obfuscation tool [9]. As seen in Figure 3.4 the obfuscation version is exceptionally complicated to reverse engineer.

3.3.1 Typical Obfuscation Process

In this section, we will discuss the obfuscation process that has been used by obfuscation tools like [9].

3.3.1.1 Mangling Mathematical Expressions and Integers

The more we mangle the expression, the more the system is protected and harder for the attacker to understand the design. As we see in figure 3.3, the integer 8 in the last but one line has been modified to 0x235-492-0x41 and the expression sum =sum + buffer[i]; changed to k795f772c7c = k795f772c7c + z7929401884 [0x3ADF-0x2FED-2801+ zddd43c876a]+ 0xEFCD52364-0x2341; where k795f772c7c, z7929401884 and zddd43c876a defines sum, buffer and i respectively. The sum of 0x3ADF-0x2FED-2801 is zero. Though two lines look different, the functionality remains the same, and the mangled version is quite difficult to understand.

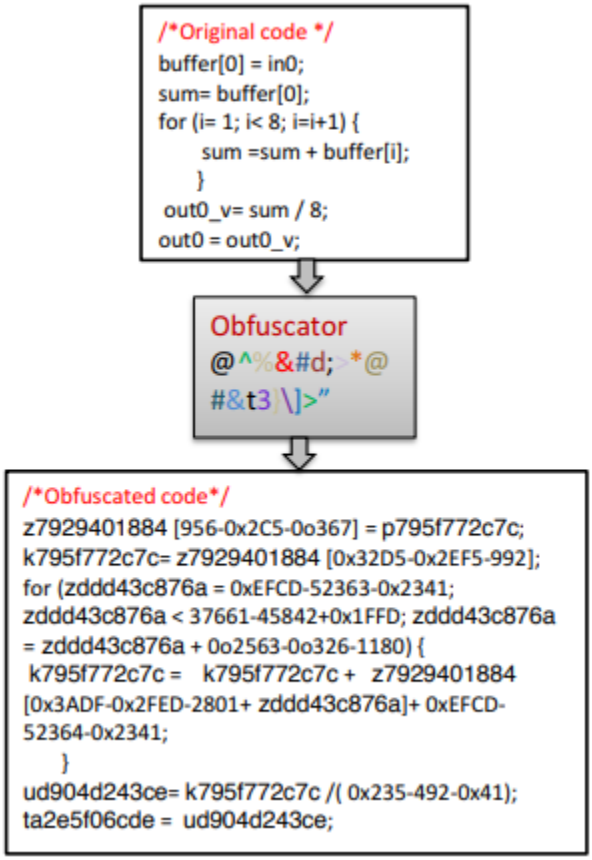


Figure 3.4: Behavioral IP obfuscation example

3.3.1.2 Stripping the Space

Writing code is an art. Frequently, software or hardware program has been written in a legitimate way to make it understandable for the reader to understand and modify in future if there is any need for that. Trimming the extra white spaces will cause the attacker to face a hard time decoding the design.

3.3.1.3 Replacing Identifiers and Signals

This step is one of the necessary and essential phases of most of the obfuscation tools.

- Combinations of characters from a user-specified set: replace all the identifiers with a combination of I and l(capital i and small L) or O and 0.

- Using mixcase: ANSYS and SystemC are case sensitive, randomly choosing some letters of some words and randomly making it mix type helps.
- Prefixing symbol: prefixing all or selected identifiers with a specified or random text.
- Replacing numeric constants with expressions: e.g. replacing 232 with (0x14b6+2119-0x1c15)
- Joining all lines in the code
- Md5 sum with user-specified "salt": sample replacement is zd7b6a02ec5. The salt, number of characters from hexified version of md5 sum and a prefix ("z" in the sample above) are the options of this mangler;
- Comments of code can be removed completely, which again will make it more difficult for the hacker to understand the design.

Obfuscation tools usually mix all these types up and make it difficult to understand. The above techniques work extremely well with the compilers (e.g., gcc and g++) as these are based on sturdy compilers which are being developed for decades. So the binary code of with and without obfuscation is similar.

As discussed in the motivational example section, the parser of the different EDA tools could lead to different results with obfuscation. Although, the functionality will be protected, but the parser will end up adding some extra area and delay overhead because HLS parsers are not as good as the software parsers.

3.4 The Reason Behind the Overhead

The output of the parser will generate similar functionality design. Now its the responsibility of compiler to optimize the code and generate the same as of non-obfuscated design. For

example the line in the 3.4 "sum =sum + buffer[i];" has been obfuscated to "k795f772c7c = k795f772c7c + z7929401884 [0x3ADF0x2FED-2801+ zddd43c876a]+ **0xEFCD-52364-0x2341**;". The highlighted part is a dead code added by obfuscation, meaning the final value of the expression is zero and doesnt affect anything if we remove that. If the compiler can not optimize the circuit by removing the dead code, unnecessarily extra circuitry will be added after synthesis. For this example, the synthesizer may generate some extra adders too to incorporate with the output of the un-optimized expression which, in turn, will lead to extra area and delay.

CHAPTER 4

PROPOSED BDL-SYSTEMC FLOW

As of 2010, 30 companies had adopted HLS for IP design [10]. The number has crossed 100 by 2018. As the value and viability of High-Level Synthesis have been established, now the most significant decision to determine is which Input language is best for HLS. It will be limited to C and SystemC as CWB only accepts C and SystemC as input language. Is it always SystemC is better as stated in [10], or it depends on the design...? The answer is quite tough because it depends on many factors that companies take into consideration while adopting the new flow like the parser, which converts the input language to an internal file and how the synthesizer has been implemented. The capacity of HLS has gone so far than the productivity of an engineer designing 10X complex logic in a single click. Reducing the Time-to-market for delivering the product with most complex SoC is now a huge challenge.

4.1 Designed Parser Flow

To come up with a solution, which input language is better for HLS, we first need to know how does the flow work.

A general HLS flow takes the input as C(foo.c) or SystemC(foo.sc, foo.h) then it goes through a parser which converts it to a data flow graph which is called CDFG(Control Data Flow Graph) for behavioral description. There are many parsers exist, ex. Lang1, lang2, langN. These tools convert behavioral description to CDFG. The output of these parsers are not similar even though the input DUT(Design Under Test) is same. As CDFG goes as an input to the synthesizer; it behaves differently because the input to the synthesizer differs from each other. Because of this parser output mismatch, the synthesizer gives different QoR(Quality of Results) for the same application.

The proposed flow in the thesis will give a better prediction for a particular application which input language will be better, again we need to trade-off between area and delay and

find out the input language and then go-ahead with our design. First of all, it is a redundant effort to design an application in different languages. Lets say we have a requirement of a particular design and it is not yet known which input language will be better. To address the above issue, a parser has been designed which takes ANSI-C as input and output is a header file and foo.sc. The parser is the first step of the whole flow.

4.1.1 Description of BDL-SystemC Flow

There are mainly three steps in the parser.

- It takes the BDL code then divide it into multiple parts and then starts analyzing it step by step.
- It first creates a header file for the SystemC. – foo.h
  - It creates a Module.
  - Define reset, clock and main function.
  - It creates a Module.
  - Define reset, clock and main function.
  - It defines the input and output variable with bit lengths Ex: sc_in ¡sc_uint¡8¿¿ in; in is the input to the design with, which is 8 bit in length. Type: Unsigned Integer
  - Then it calls the clock thread function(Refer to chapter-2 for details).
  - Then it defines if there are any other subfunctions which are being called by the clock thread function.
  - It also takes care, if there is any global variable which needs to be defined and assigned with a value.
  - Then it creates a constructor of the Module.

- It creates a Thread, as the benchmarks contain complicated designs and multicycle algorithms, CTHREAD has been chosen.
- It assigns the sensitivity edge and reset signal for the design.
- At last, it put a destructor of the Module.

- It creates the SystemC file. – foo.sc
  - Includes the header files
  - Declares the variables which are global in C.
  - Then It first creates a thread of the module and parses the corresponding C code with SystemC formatting.

    Ex: In C the input will be read with a = sign.

    C: temp = Inp

    SystemC: temp = Inp.read
  - Then It creates a while loop for the whole thread and wait until it finishes, then repeat itself.
  - Then it parses other functions, keeping arguments and return types preserved.

Above are the necessary steps of the parser flow. This first step basically saves us time by not writing the whole SystemC code.

4.2 BDL/SystemC to CDFG Parser

In the second step, The BDL and the systemC code passes through a yet another parser which takes the BDL/SystemC code as input and converts it to CDFG(foo.IFF) format. The systemC and BDL will be converted to a binary file(CDFG) format which in turn will be needed for synthesis. This IFF file is company specific and only understandable to in-house synthesizers. It is basically a binary file which is entirely non-human readable.

4.3 Synthesizing Flow

The third step is the synthesis step: There are a couple of synthesis modes that can be specified.

It takes a couple of files as input.

- CDFG(foo.IFF)

  It takes input the binary format of the CDFG, then based on the data it generates QoR.

- b) Functional Unit Constraint file (LMT)

  This file specifies the information and constraints regarding functional units used in the synthesis.

- Memory Constraint file (MLMT)

  This file specifies the information and constraints regarding memory used in synthesis.

- Port Constraint file (PLMT)

  This file specifies the information and constraints related to input-output ports of a module that is used in synthesis.

- Port Relation file (PREL)

  This file specifies the correspondence between I/O ports of the module that are used in the synthesis and I/O variables allocated to these ports.

- Basic Library file(BLIB)

  BLIB file contains the circuit (delay and area) information of the constant table; decoder enabled multiplexer bit operation, multiplexer, decoder, and register.

- Functional Unit Library file(FLIB)

  This file specifies the functional unit information that is used in synthesis. Functional

unit library file has two types of format for basic functional unit and arithmetic macro functional unit

- Functional Unit Count file(FCNT)

  This file specifies the constraint for all functional units used in synthesis.

Once all the above files are ready, we can start the behavioral synthesis process. It takes clock frequency, the CDFG, number of specific FUs and the information regarding the FUs i.e. area, delay etc. Below is an example of synthesis command specific to CWB.

Ex: Synthesis_tool_command -c1000(# of clock cycles it should run) foo.IFF -If1 foo.FLIB -Lfc foo.FCNT

The output of bdltran is foo_E.IFF, foo_C.IFF and synthesis analysis files. Foo_C/E.IFF file goes as input to veriloggen.

4.4 HDL Generation Flow

The fourth step is the generation of HDL file. In this step the input is foo_E.IFF, which is a binary file containing combined information of library delay, area based on the design with synthesis results and analysis. The output is foo.v/foo.vhdl, which is ready to be implemented in an FPGA/ASIC design.

Above are the four significant steps of the whole flow. As per the milestone, the thesis aims to decide which input language is best for a particular design.

4.5 Proposed Flow

As we have noticed early in the chapter, we need to have a design automation which will gives us a detail view of which input language is better for a particular design. Figure 4.1 flow, saves a lot of time, by not writing the code of the same design in SystemC. As stated, the parser converts the BDL to SystemC.

4.5.1 Flow Diagram

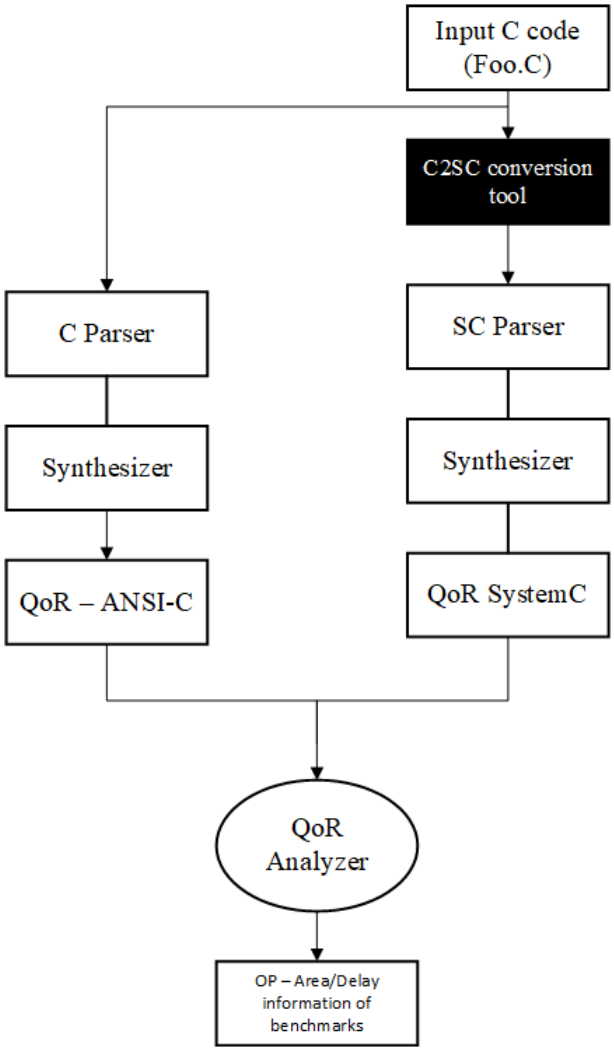


Figure 4.1: Proposed flow(to know which input language is better)

4.5.2 Description of the Flow

Step by step description of the flow given in figure 4.1

- A BDL input is passed through the designed parser, which will take C as input and give SystemC as output with the header file.
- Now both the designs will be passed with CWB as below:

- bdlpars or scparse for BDL or SystemC respectively.
- Output foo.IFF, as input to synthesizer, to get the HDL format file, which will be needed to generate verilog/VHDL code.
- HDL format file (foo_E.IFF) goes as an input to HDL generator(veriloggen) and output is foo.v.
- This synthesis also gives a QoR(Quality of result)

- Once the above step is done successfully, the QoR of both language designs will go to an analyzer, which will analyze the result and give us the details. In the Analyzer step we basically get the QoR of all the input languages, by considering the requirement, we can make an optimized design.

4.5.3 Experimental Results

Table 4.1: QoR(area and critical path delay) output of C and SystemC

Benchmark	ANSI-C		SystemC	
	AREA	Delay-ns	Area	Delay-ns
FIR	2119	3.06	2167	3.06
SOBEL	966	2.01	1014	2.01
ADPCM	732	6.01	3890	13.11
AES	109824	8.39	110592	8.39
AVE8	710	1.34	758	1.34

As stated in the results 4.1, the output is not the same in C and systemC, and it has a significant variation in this benchmarks, which are small designs, it will have a massive impact in larger designs, which will in-tern impact the efficiency. However, by following the below flow we can get the efficiency of the both BDL and SystemC, then we can decide which one to take into consideration. It will save much time because the parser will directly give the SystemC files as output. Below is the flow chart which describes the proposed flow.

4.5.4 Limitation to the Proposed Flow

So far we can get a good design with the best efficiency using HLS. However, what we did not consider, is protecting the IP. As an Industry, Nobody wants to sell an IP without protection, as discussed in chapter-3. One of them: Often companies make a contract to use a BIP(Behavioral IP) for a trial purpose, to evaluate the quality(delay, area, etc.). This BIPs cannot be made accessible and visible during the evaluation period as the consumer will not need to end up buying it. Also because of some other data protection reasons, we need to protect the IP. As we discussed in chapter-3, Obfuscation is the easiest and most inexpensive way to protect the BIPs.

4.6 Modified Flow with Obfuscation

When obfuscating, a same functional input file has been generated, which is almost impossible to understand to human-eyes and extremely hard to reverse engineer. As we have seen [8], it literally removes comments, renames intermediate variables and adds unnecessary steps to make it human non-readable format. All commercial HSL tools often use dedicated in-house parsers which are heavily dependent on input languages. Different parsers have parsed different input languages into an Internal Format File(foo.IFF). This IFF file is unique and only understandable to different in-house synthesizers. The preliminary step is responsible for traditional technology independent compiler optimizations, e.g., dead-code elimination, common sub-expression elimination and constant propagation. The synthesizer then read the IFF file and carry out HLS steps(resource allocation, scheduling, and binding) then it, in turn, generates either Verilog or VDHL. This HDL generation flow is quite popular and the biggest secret of every company. Being able to take the CDFG(binary format) as input makes it most efficient, by which it can take any number of input languages by just adding a front-end parser. The main downside is, the EDA companies who are not expert in this

fields made in-house parsers, which leads to inefficient design. This is extremely important in case of obfuscation as this step tries to insert multiple redundant operations to make it as less human-readable as possible, which in turn leads to impact the area, delay, and latency. This work defines the impact due to obfuscation and how to get a better design without changing the input language, basically without using any extra efforts and resources.

4.6.1 Modified Flow Diagram

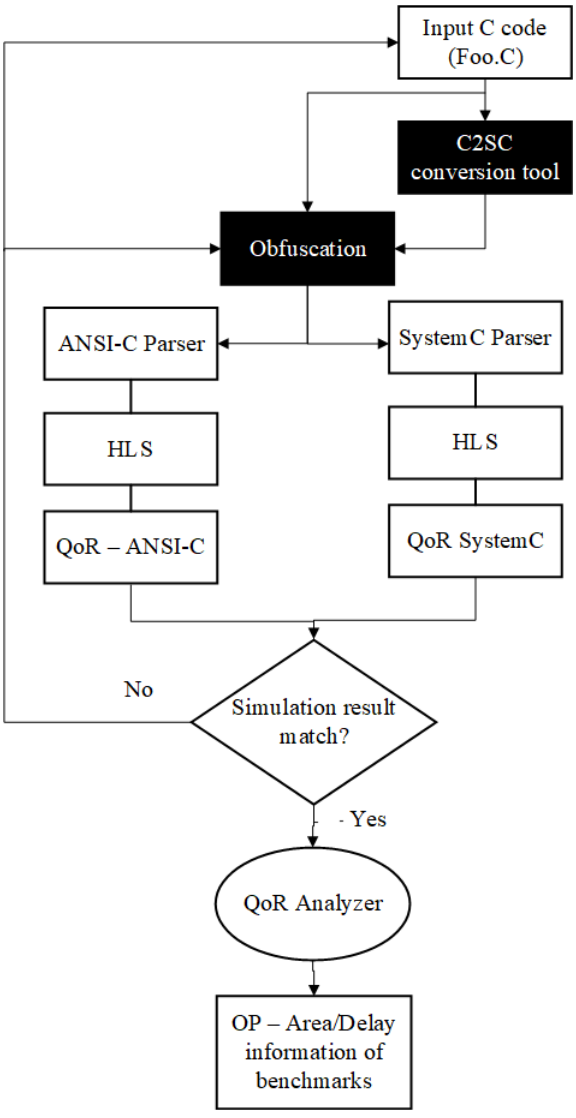


Figure 4.2: Modified flow diagram with obfuscation

4.6.2 Description of the Flow

This flow 4.2 is an addition to the flow that has been described earlier in this chapter which is without obfuscation. Before the parser parses the input(BDL) language and generates the systemC version, then the HLS flow starts. However, In this modified flow, the input(BDL) will be parsed to get the SystemC version. Then both BDL and SystemC codes will be obfuscated to get the Non-human-readable version of the design; then the HLS flow starts which will first parse the obfuscated code to get the foo.IFF, then it passes through the synthesis process and at last it goes to the HDL generation flow. In the end, The QoR will be generated, by seeing the QoR, one can quickly decide which input language is better for the design.

4.6.3 Impact of Obfuscation

The Obfuscation process in the flow is a combination of a couple of sub-processes. Stunix C/C++ tool has been used for obfuscation as a preliminary flow. Then a couple of manual redundancies have been inserted in order to consider the worst case scenario. As there are many numbers of obfuscation tools available in the market and every obfuscator behaves differently, so to reciprocate the effect of worst-case scenario manual obfuscation has also been added with the output of the stunix too. An example has been given below to make it more clear.

As we see in table 4.2, The first row shows the raw C code. Second row shows how stunix obfuscates the variables. And the last row shows how the final obfuscated code will look like. Some front-end parsers are unable to optimize this redundant behaviour which leads to unintended area overhead.

Table 4.2: Effect of obfuscation of the input

Original Input line:	for(i=0,i¡=12,i++) {op = x+3;}
Stunnix Ofuscator output:	for(0x98AA=0, 0x98AA <=12, 0x98AA++) {0x13B4 = 0x00B3+0x0011;}
Final op after manual obfuscation:	for(0x98AA=0,0x98AA<=0x10AA-0x10A2, 0x98AA++) {0x13B4 = 0x00B3+0x0011;}

4.6.4 Experimental Results

As the results stated in 4.3, the QoR of the obfuscated and non-obfuscated version of BDL and SystemC are entirely different.

Table 4.3: Area and Delay results with and without obfuscation

Benchmark	ANSI-C		SystemC		Obf. ANSI-C		Obf. SystemC	
	AREA	Delay-ns	Area	Delay-ns	AREA	Delay-ns	Area	Delay-ns
FIR	2119	3.0634	3205	3.7809	2167	3.0634	2167	3.7194
SOBEL	966	2.009	1354	3.0165	1014	2.009	1014	2.2009
ADPCM	732	6.01	759	5.35	3890	13.1105	3890	13.1105
AES	109824	8.39	110592	8.49	110592	8.39	110592	8.23
AVE8	710	1.335	1532	2.542	758	1.335	738	0.9675

The area and critical path delay, which are the most sensitive data in a Hardware design, are majorly impacted by obfuscation. As we stated earlier, the obfuscation usually adds overhead area or delay. However, if we take a closer look into AES, the obfuscated SystemC critical path delay(8.23ns) is lesser than non-obfuscated SystemC critical path delay(8.39ns). It proves that the HLS process heavily depends on the input parser(bdlpars,scpars). Because of the different binary output(CDFG-foo.IFF) of the parsers, the input to synthesizer differs from one another.

4.6.4.1 Experimental Setup

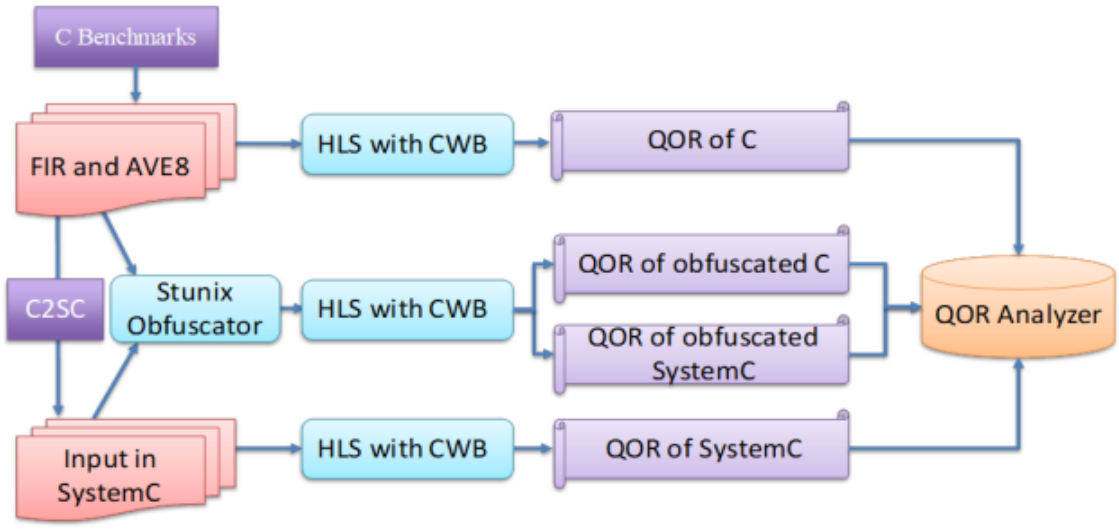


Figure 4.3: Experimental Setup of the Proposed flow

Figure 4.3 shows how the experimental results have been set up and below shows the tools, benchmarks and constraints, which have been used as a part of the research.

- Benchmark: FIR, SOBEL, ADPCM, AES, AVE8 from S2CBench suite.(Converted to ANSI-C to show the comparison)
- C2SC C to SystemC conversion tool(has been designed as a part of the thesis)
- HLS: CWB(Operating Freq.: 20MHz)
- QoR analyzer: input: different QoRs, output as extraction of area and delay of input QoRs (has been designed as a part of the thesis)

The first column of table 4.4 shows how the input language impacts the design. There are designs in which the impact is very less like FIR, SOBEL, AES. But as we see ADPCM performance is humongous. It has a tremendous impact on the design. 4.4 is the graph which shows how the area of the design distributed in different benchmarks for C and SystemC.

Table 4.4: Percentage change in area of C and SystemC with and without obfuscation

Benchmark	**C / SystemC % Change**	**C / Obf C % Change**	**SC / Obf SC % Chnange**	**Obf C/Obf SC % Change**
FIR	2.27	51.25	0.00	47.90
SOBEL	4.97	40.17	0.00	33.53
ADPCM	431.42	3.69	0.00	-80.49
AES	0.70	0.70	0.00	0.00
AVE8	6.76	115.77	-2.64	107.59

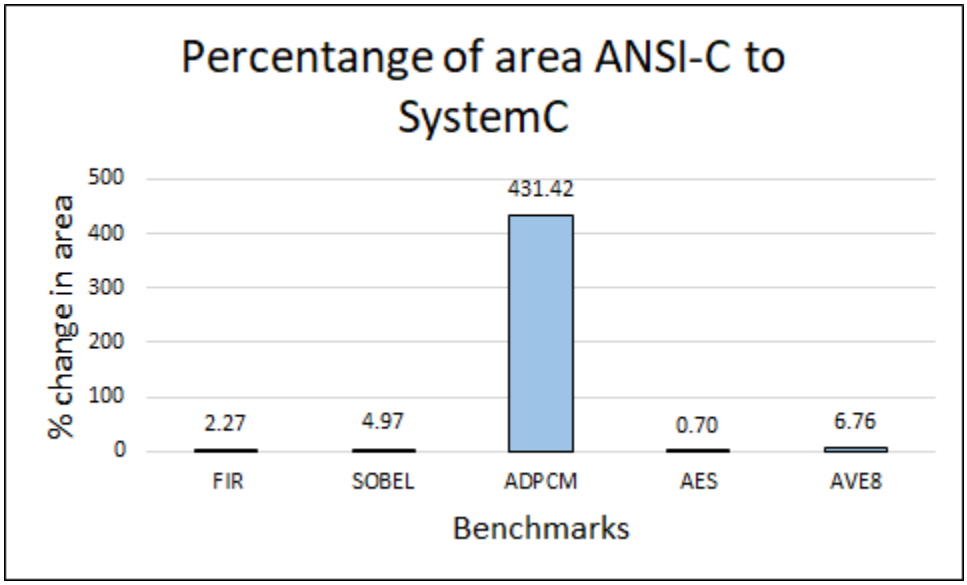


Figure 4.4: The percentage change of area from C to SystemC of benchmarks

Figure 4.4 describes the percentage change in the area of SystemC from BDL. As we see the area of ADPCM is 431 percent higher if the input language is SystemC. But for the other design, the area is very less impacted, i.e., 0 to 6% change. Except for ADPCM, other designs are somewhat acceptable for some applications. Let's assume a single SOBEL filter has been used in a design 100 times sequentially; then the area will increase by 50%. So from the above discussion and graph, we can conclude that BDL is better than SystemC for the above benchmarks. However, the IPs cant be sold like this without any protection. As discussed in Obfuscation chapter, Obfuscation is the only effective and inexpensive way

to protect the IP. Below, we will discuss how the obfuscation will impact the designs. The percentage change has been calculated by below formula.

$$X = \text{Original value} \qquad Y = \text{Increased/Decreased Value}$$

$$\%\ \text{Change} = \frac{X - Y}{X} \times 100$$

Now we will discus about the change in area of C to Obfuscated C. Figure 4.5 describes the percentage change of area from Obfuscated BDL to normal BDL. As we see, it varies from design to design. For FIR and SOBEL the change in the area is 40% - 50%. For ADPCM and AES the change is quite less, however, for AVE8 the difference is more than double. This proves that the design is heavily dependent on the input parser, which creates the CDFG and the synthesizer. Meaning, either the bdlparse not able to optimize the redundant behavior and the synthesizer not able to optimize the redundancy created by the parser.

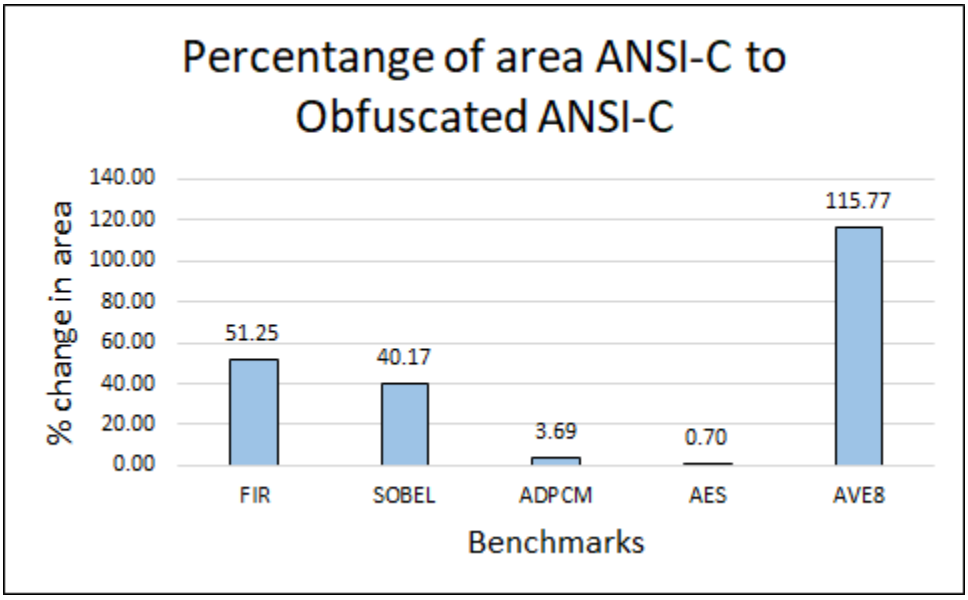


Figure 4.5: The percentage change of area from C to Obfuscated C of benchmarks

Same goes for the SystemC too. As we see in the figure 4.6, the percentage change is quite stable than BDL, except AVE8 which is preety small. The BDL parser which has been

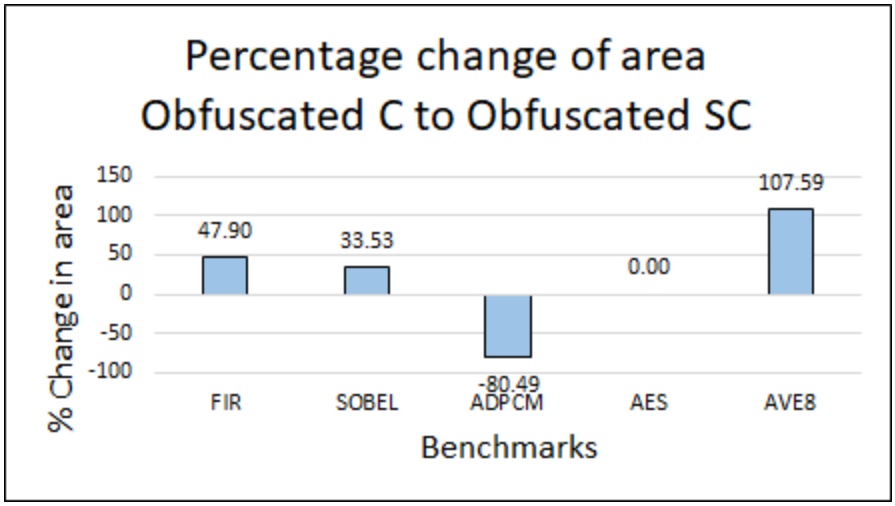


Figure 4.6: The percentage change of area from SystemC to Obfuscated SystemC of benchmarks

used for CWB is an in-house made parser, so the performance penalty huge. However, the SystemC parser is a third party professional parser, so it is much more stable than C parser. Now the question comes about the synthesizer. Even though the parser couldn't eliminate the deadcode(which will be there in CDFG binary file), the synthesizer should be able to optimize the CDFG and provide a more efficient design. Unfortunately, the synthesizer fails in case for C parser. The reasons behind the area overhead in the obfuscated SystemC of ave8 have been discussed below.

For AVE8: sum= buffer[0]; ——————————1

In the above equation, simply the first bit of the buffer is assigned to the variable sum. If we add a redundant behavior like below:

sum= buffer[0]+0x32D5-0x2EF5-992; ——————————-2

where 0x32D5-0x2EF5-992 = 0

So the functionality of the equation 1 and 2 are same, just a redundant behavior has been added to obfuscate the IP. Figure 4.7 shows the CDFG flow for AVE8 with and without change of the above equation.

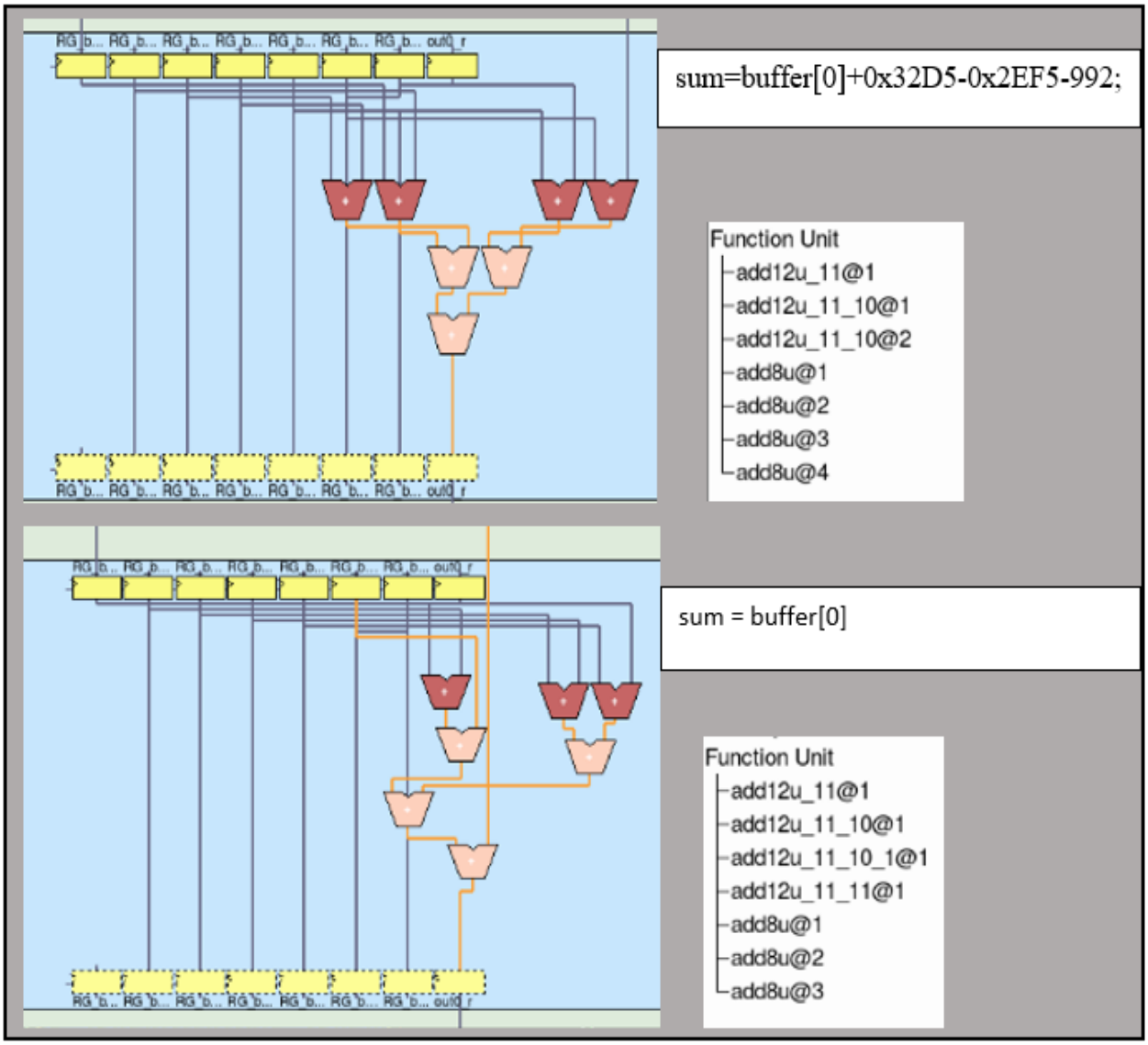


Figure 4.7: Penalty due to obfuscation

As we can observe from the figure 4.7, just a single line change with the same functionality leads to an increased area. Surprisingly, the obfuscated line gives better area. After adding the redundancy, the parser can optimize the redundant operation but in case of no redundancy the parser/synthesizer not able to perform as expected. It ends up adding an extra 12-bit adder, which leads to an increase in the area by 20units.

Below is an example which shows how the QoR affects the HLS. Below is the observation:

*sum = sum + data[i] * coeff[i] ; —- without redundancy —- 1*

*sum = sum + ((data[i]+**0x32D5-0x2EF5-992**) * (coeff[i]+**0x32D5-0x2EF5-992**)) +**0x32D5-0x2EF5-992**; —— with redundancy —— 2*

***0x32D5-0x2EF5-992 = 0*

After adding the redundancy in equation 2, the area is increased by 42%, which is non-avoidable. However, if we add a bracket in the redundancy in the equation, there is no penalty in the area. Below is the example:

*sum=sum+((data[i]+ **(0x32D5-0x2EF5-992)**)*(coeff[i]+ (0x32D5-0x2EF5-992)* *)) +0x32D5-0x2EF5-992 ;*

Looking at the above results, its crystal clear that the design performance and efficiency heavily depend on the parser. So its necessary to decide which input language should be considered for HLS. And its pretty random based on the design. Below are the comparison and percentage change of the Obfuscated C and Obfuscated SystemC.

Now we will discus which input language to choose. Figure 4.8 shows the percentage change of QoR(area) of obfuscated C to obfuscated SystemC.

The figure 4.8 is pretty much self-explanatory. For FIR, SOBEL, and AVE8 BDL has appx. 4%, 34% and 108% extra overhead in area than SystemC. For these three designs, SystemC will be a wise choice. For AES its exactly same, so it doesnt matter which language has been chosen as input. But, for ADPCM, BDL will give 80% less in the area, which is humongous. For ADPCM, BDL will be a wise option to consider as an input language.

4.7 Automation of the Flow

The whole flow, which includes with and without obfuscation, BDL-SystemC parser, and HLS have been automated using a python script. The steps of the script described below.

- It invokes the parser to convert the BDL to SystemC.

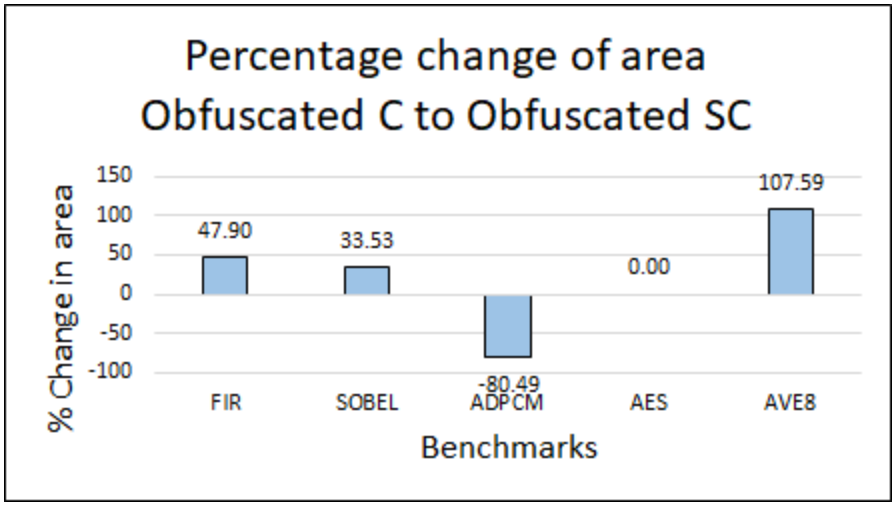


Figure 4.8: The percentage change of area from obfuscated C to Obfuscated SystemC of benchmarks

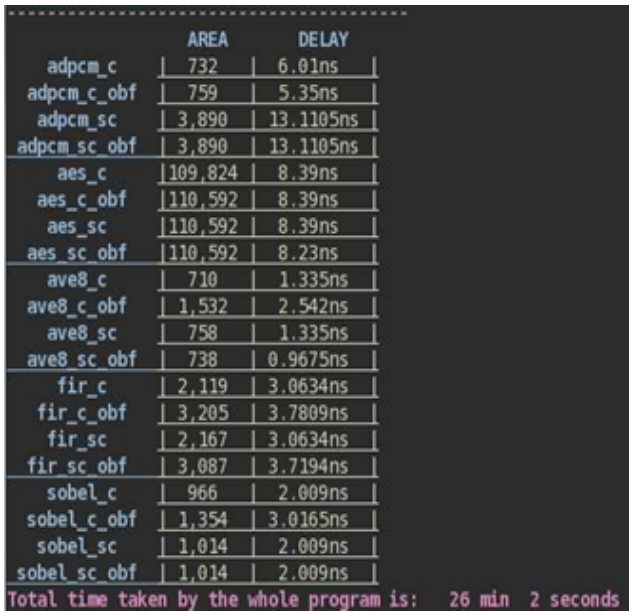
- Then both the BDL and SystemC passes through a phase of obfuscation which is being done by Stunix obfuscator, followed by manual obfuscation to show the worst case scenario.

- Then it passes through various stages of High-Level Synthesis. At last, it gives the output as foo.v.

- Then the analyzer considers the QoR and then generates a combination of output results, as shown below.

- Then by examining the results, we can decide which input language to consider for a particular design.

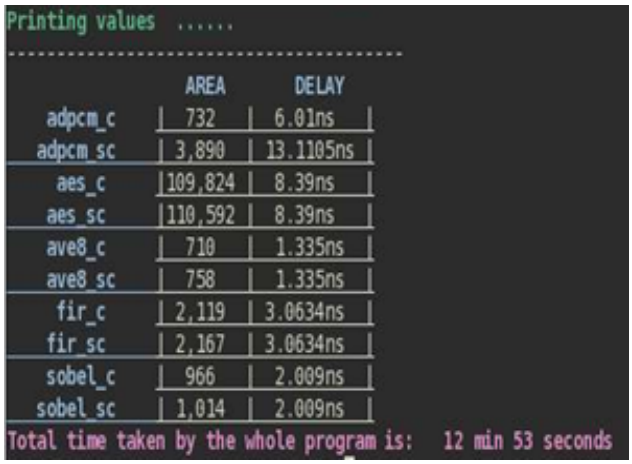
Figure 4.9a and 4.9b shows the QOR with and without obfuscation respectively. As stated earlier in this chapter, that obfuscation has higher impact in C input. After observing above output of the conversion tool, one can easily decide which input language to choose for the particular design



	AREA	DELAY
adpcm_c	732	6.01ns
adpcm_c_obf	759	5.35ns
adpcm_sc	3,890	13.1105ns
adpcm_sc_obf	3,890	13.1105ns
aes_c	109,824	8.39ns
aes_c_obf	110,592	8.39ns
aes_sc	110,592	8.39ns
aes_sc_obf	110,592	8.23ns
ave8_c	710	1.335ns
ave8_c_obf	1,532	2.542ns
ave8_sc	758	1.335ns
ave8_sc_obf	738	0.9675ns
fir_c	2,119	3.0634ns
fir_c_obf	3,205	3.7809ns
fir_sc	2,167	3.0634ns
fir_sc_obf	3,087	3.7194ns
sobel_c	966	2.009ns
sobel_c_obf	1,354	3.0165ns
sobel_sc	1,014	2.009ns
sobel_sc_obf	1,014	2.009ns

Total time taken by the whole program is: 26 min 2 seconds

(a)



Printing values .....

	AREA	DELAY
adpcm_c	732	6.01ns
adpcm_sc	3,890	13.1105ns
aes_c	109,824	8.39ns
aes_sc	110,592	8.39ns
ave8_c	710	1.335ns
ave8_sc	758	1.335ns
fir_c	2,119	3.0634ns
fir_sc	2,167	3.0634ns
sobel_c	966	2.009ns
sobel_sc	1,014	2.009ns

Total time taken by the whole program is: 12 min 53 seconds

(b)

Figure 4.9: Output of the modified proposed flow with and without obfuscation. (a) With Obfuscation (b) Without obfuscation

4.8 Summery

As explained in this chapter, knowing which input language is better, will lead to an efficient design. This chapter we got to know a new flow which will examine the QoR of the input languages and give us the detailed area and critical path delay of the designs, by seeing those we can come to a conclusion, which input language to use for the particular design.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we have proposed a new flow to know which input language is better for a particular design. There is very less performance penalty for this additional flow. But at the end, we will be able to reduce the resource utilization for the selection process of choosing the input language and more importantly the efficiency of the design. To convey in brief, the C code will be taken as an input to the conversion tool(C2SC), then it starts the processing(HLS) of the C code. Once the HLS of C is done, it converts the C to SystemC and does the same processing again. And at the end, it will describe the detailed area and critical path delay of the design. By looking at it, one can easily and quickly decide which input language to consider for the design.

5.2 Future Work

Another parser can be created as an extension to this thesis, which will convert SystemC to C. In that way the input language wont be constant to C. Then the input language can be written in any language(C/SystemC), having more flexibility. Another one is, to test this with many benchmarks to study the effect of obfuscation even more.

REFERENCES

[1] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, *"A formal approach to the scheduling problem in high level synthesis,"*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 10, no. 4, pp. 464475, 1991.

[2] S. Govindarajan, *"Scheduling algorithms for high-level synthesis,"*. Term paper ECE, vol. 834, 1995.

[3] Z. Baruch, *"Scheduling algorithms for high-level synthesis,"*. ACAM Scientific Journal, vol. 5, no. 1-2, pp. 4857, 1996.

[4] A. C. Parker, J. Pizarro, and M. Mlinar, *"MAHA: A Program for Datapath Synthesis,"*. in 23rd ACM/IEEE Design Automation Conference, June 1986, pp. 461466.

[5] R. Jain, A. Mujumdar, A. Sharma, and H. Wang, *"Empirical evaluation of some high-level synthesis scheduling heuristics,"*. in 28th ACM/IEEE Design Automation Conference, June 1991, pp. 686689.

[6] P. G. Paulin and J. P. Knight, *"Force-directed scheduling for the behavioral synthesis of ASICs,"*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, no. 6, pp. 661679, Jun 1989.

[7] B. Carrion Schafer and A. Mahapatra, *"S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis,"*. Embedded Systems Letters, IEEE, vol. 6, no. 3, pp. 5356, 2014.

[8] B. Carrion Schafer and N. Veeranna, *"Efficient Behavioral Intellectual Properties Source Code Obfuscation for High-Level Synthesis"*. 18th IEEE Latin American Test Symposium 2017

[9] *"Stunix c/c++ obfuscator,"*. http://stunnix.com/, accessed: 2018-11-15.

[10] Dan Gajski, *"What Input-Language is the Best Choice for High Level Synthesis (HLS)?"*. DAC 2010, June 13-18, 2010, Anaheim, California, USA

[11] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, *"An introduction to high-level synthesis,"*. IEEE Design & Test of Computers, vol. 26, no. 4, pp. 8-17, 2009.

[12] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, *"A formal approach to the scheduling problem in high level synthesis,"*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, no. 4, pp. 464475, 1991

[13] SEMI. (2008), *"Innovation is at risk as semiconductor equipment and materials industry loses up to $4 billion annually due to IP infringement"*. [Online]. Available: http://www.semi.org/en/Press/P043775

BIOGRAPHICAL SKETCH

Himanshu Patra was born in a remote city, Soro, on the outskits of Balasore District, Odisha, India on July 1st 1992. He completed his schooling in Satya Nanda Highschool in his home town, Soro. Growing up in a business family, he always wanted to roam the whole world and experience every culture. His only hobby in childhood was cricket. After completing his Bachelors degree at Institute of Technical Education & Research (ITER) in 2014, he joined Tata Consultancy Services as an assistant system engineer. He continued his employment until he joined The University of Texas at Dallas to pursue his Masters in Electrical Engineering in August 2016. He further joined the Design Automation and Reconfigurable Computing(DARC) lab on August 2017 under Dr. Benjamin Carrion Schaefer. Subsequently, he joined as a Graduate Technical Intern at Intel Corp., Folsom, CA. His interest in research lies in Machine Learning, Design Automation, Design Verification and Physical Design. Apart from his academics he loves to roam all places, try all kinds of cuisines.

CURRICULUM VITAE

Himanshu Patra

January 7, 2019

Contact Information:

Office: 1900 Prairie City Rd
Folsom, CA 95630, U.S.A.

Voice: (720) 257-3096
Email: himanshupatra.612@gmail.com

Educational History:

B-Tech, Electronics and Communication Engineering, Institute of Technical Education & Research, 2014
M.S., Electrical Engineering, The University of Texas at Dallas, 2019

Employment History:

Assistant System Engineer, Tata Consultancy Services(TCS), India, June 2014 – June 2016
Graduate Technical Intern, Intel Corp, Folsom, CA, May 2018 – December 2018

Papers Published:

An enhanced technique for color image water marking using HAAR transform, 2015 International Conference on Electrical, Electronics, Signals, Communication and Optimization (EESCO)