Chapter 2
Literature review

This chapter presents a literature review on HLS methods in VLSI design flows. The advantages and disadvantages of HLS flows are discussed. Subsequently, some of the optimization methods available in the literature for HLS flows are discussed which in turn surface the major research gaps and lay the foundation for the objectives of this thesis. Even though there are multiple HLS tools available in the community, Vivado HLS and MATLAB HDL coder have been considered for this study [2.1, 2.2]. These two platforms have been chosen because of their wide acceptance in research community and industry.

2.1 High-level synthesis in VLSI design

VLSI design starts with requirement specifications which get translated to functional specifications of the design. In some cases, requirements are a simple textually represented document about the features supported by the application, while in other cases, they are an executable high-level model, typically designed in C, C++, System C, or MATLAB. In this step of the design, the specification contains none or minimal hardware implementation details. Its primary goal is to verify and fine-tune the desired behavior. Once fully verified and reviewed, this model transforms into the actual hardware implementation. The first step in design implies the definition of optimal architecture that implements the expected functional model. The actual implementation involves converting these architectural decisions into RTL descriptions using hardware description languages such as Verilog, and VHDL.

Due to the manual process involved, it is almost impossible to find out the best possible optimal implementation from the desired functional specifications [2.3]. Moreover, the manual process of translation into RTL is error-prone and hence requires considerable time from verification engineers to ensure that the implementation is aligned with the intended architecture specification. So, the overall process, specification towards implementation, takes the shape of an iterative cycle, leading to a larger design cycle time. Hence, HLS is needed to optimize the overall design cycle time.

HLS is an automated design process that interprets an algorithmic description of the desired behavior. Then, it creates digital hardware which implements such behavior. The

synthesis starts with a problem's high-level specification, such that the behavior is usually decoupled from clock-level timing. The code is analyzed, architecturally constrained and scheduled to create an RTL description. In turn, such RTL description is frequently synthesized to the gate level using a logic synthesis tool.

As described in Figure 2.1, the intermediate representation describes the design in the form of control and data flows within the design, e.g., which data path to be enabled and when. In the next stage, the operations get mapped to functional units, variables get mapped to signals or storage, while the data transfers get mapped to buses, wires, or multiplexers. According to the control and data flow graph, these functional units, buses, wires, etc. get time allocation on when they would be exercised. In the last stage, a structured HDL code is obtained.

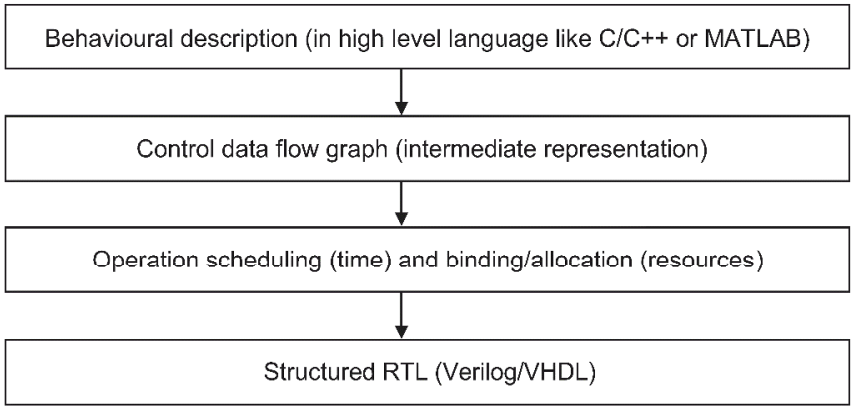


Figure 2.1. Steps involved in high-level synthesis.

HLS's main goal is to let hardware designers efficiently design and verify the hardware. HLS provides designers with an option to express the design at a higher level of abstraction, while the tool provides the corresponding RTL implementation in HDL. Another goal of HLS is to ensure continued verification of the design during the stages of the design process to reduce the functional verification time for the project.

Even though HDL languages existed in the late 80s and early 90s, most commercial HLS tools were introduced in the market in the 21st century. Some examples include Vivado HLS (Xilinx), Stratus (Cadence), Catapult-C (Mentor), Symphony (Synopsys), HDL coder (MathWorks) and BlueSpec (BlueSpec Inc.) [2.1, 2.4, 2.5, 2.6, 2.2]. Open-source and academic community tools have also been developed in the same domain [2.7, 2.8].

2.2 Advantages and limitations of HLS

2.2.1 Advantages of HLS

1. **Faster design and verification:** The engineers need not worry concerning implementation features like hierarchies, processes, clocks, or technology because the abstraction level is higher. Hence, it is easier to code. As the code is shorter and less complex precisely due to the higher abstraction level, it is easier to verify with better functional coverage. Further, HLS tools offer continued verification during multiple phases of design which makes it faster to achieve verification goals [2.9].
2. **Effective design reuse:** To address the challenge of the ever-increasing complexity of designs, the IP and "code" reuse across projects is very important. RTL implementations are typically describing design behavior in terms of clock cycles, pipelines, glitch filtering, etc., which make it very specific to technology and hence difficult to reuse across projects [2.10]. Moreover, introducing small changes to an IP for creating a derivative gets complex. With HLS, one needs to change the abstract source without worrying about adverse effects on an architecture or technology dependent feature to break. Hence, this process is much more seamless.
3. **Efficient investment for research and development resources:** The engineering resources spend fewer cycles on RTL coding and verification. Then, more time can be used on architectural exploration, algorithm development, system-level power, and area estimation when using HLS [2.11].
4. **Supportive electronic design automation ecosystem and natural evolution:** The movement to higher abstraction levels is a natural evolution for the EDA ecosystem. From the past few decades, the abstraction has increased from layouts to transistors, transistors to gates, gates to RTL. Hence, it is natural for the industry, in general, to move to an even higher level of abstraction [2.12]. The evolution of tools from open-source domains as well as companies like Cadence, Synopsys, and Mentor has aided thousands of successful tape-outs using HLS flows in the past few years.

2.2.2 Limitations of HLS

1. **Not a substitute for RTL designer:** Using HLS, one can describe the behavior at a higher abstraction level but having a deep knowledge about RTL implementation helps to tune the generated code for a targeted application. Hence, the designer still has to invest time to come up with an optimal high-level language model as per the desired RTL.

2. **No/Limited control over generated RTL:** HLS flows are very much dependent on EDA tools and algorithms to convert behavioral code into RTL. This means one has very limited control over the quality of code generated. As an example, if a user wants to implement a design using a specific combination of gates using a Boolean equation, he can do so using hand-coded RTL but there is no way to instruct the HLS compiler to perform the same.
3. **No standard high-level language:** When it comes to RTL, the two hardware description languages used in the design flow are VHDL and Verilog. But when migrating to HLS, different tools use varying behavioral description languages like C/C++, Python, or MATLAB. This nonstandardization makes it difficult for the community to adapt the flows due to the associated learning curve to understand a new platform.
4. **Absence of generic optimization methods:** Depending on the target application, it may be required to optimize the implementation for area, speed, or power. For example, in a smartphone application, there may be a need to optimize for power but for a server application there may be a requirement to optimize for faster frequency and performance. When using HLS flows, the optimization methods are very specific to the tool and application and not at all generic which makes it difficult to use them on different designs and wide variety of FPGAs available in commercial space.

2.3 Concepts of HLS optimization

Depending on the application of the VLSI design, one or more of the area, speed, and power optimization may be essential to achieve, e.g., in a computing application chip for a spacecraft trajectory, accuracy is critical while for a mobile phone computing environment, the reduction of power dissipation may be the most important target. Similarly, for a processing IC used in high-speed server design, the area might not be as important as the speed of operation. As the area, speed, and power in VLSI designs are orthogonal, there exist many methodologies which improve one or two of them at the cost of other(s) [2.13].

2.3.1 Speed improvement techniques

There are multiple speed improvement techniques used in synthesis. Some of these are explained below:

1. Retiming: In maintaining the same design functionality, the synthesis tool may choose to optimally distribute registers throughout the circuit to ease the critical path delay for achieving a better compile frequency. For example, in Figure 2.2, assuming a delay of N1 for multiplier and N2 for the adder, assuming N1>N2, the critical path is N1 units in Case 2

against N1+N2 units in Case 1 due to retiming.

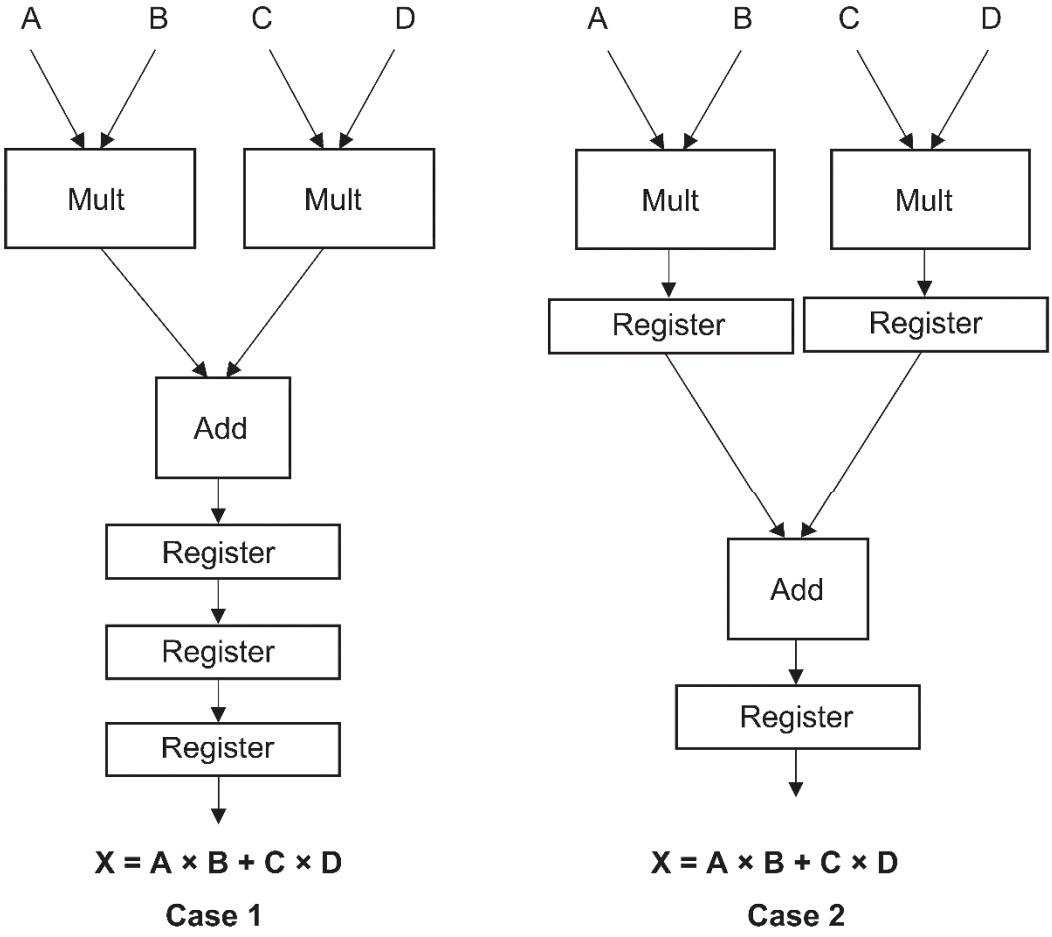


Figure 2.2. Retiming in designs using registers.

2. Module flattening: A hierarchical design (single or multiple modules instantiated within a module) is superior regarding readability and ease of debugging for verification purposes. However, from the performance improvement perspective, optimizations that could have taken place at module boundaries can happen in flattened designs leading to better retiming and hence lesser critical path.
3. Redundant/dead-logic elimination/logic reordering/constant folding: Removal of logic to reduce logic blocks functionally giving the same output can help in reducing the gate delay and hence achieve faster designs. For example, let us consider a high level code as seen in Figure 2.3. The else section of the code is never executed and hence is redundant or dead code leading to logic simplification to C = B-A. This leads to inference of a single subtractor in hardware contrary to 3.

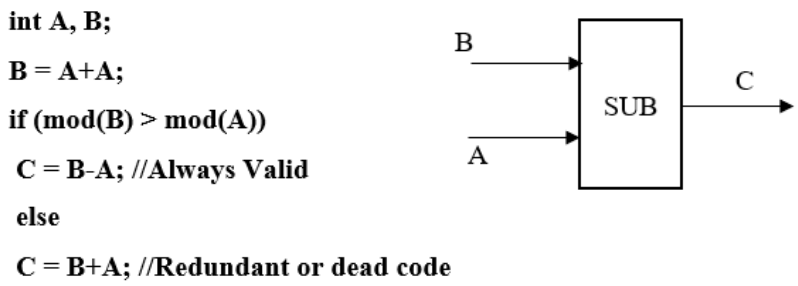


Figure 2.3. Dead code elimination in HLS.

In another example, we have to compute X = A+B+C+D. As shown in Figure 2.4, Case 1 would be faster than Case 2 though both the implementations are functionally equivalent. This is due to logic reordering. Another example is an inference of a 1-bit carry look-ahead adder in place of conventional 1-bit adders to reduce the total delay in the computation, which a synthesis tool may do under some specified constraints.

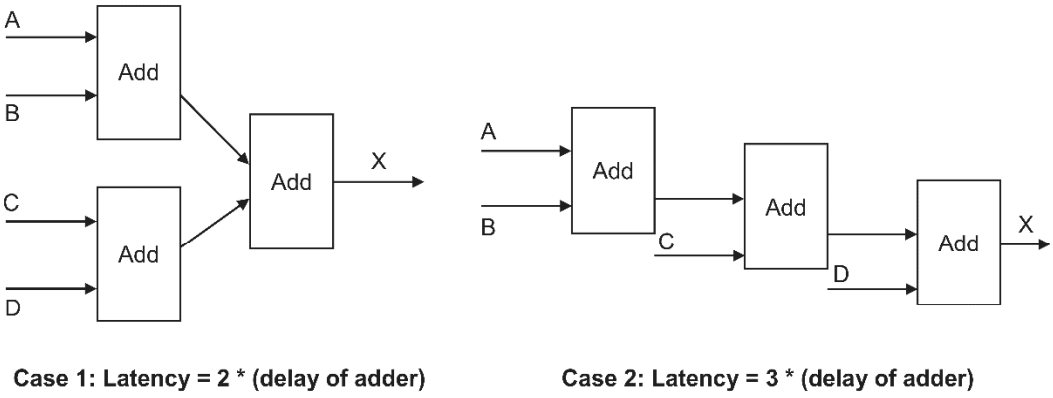


Figure 2.4. Logic reordering and effect on latency.

4. Synchronizing flops for Input/Outputs: Typically, any design would have general-purpose IOs which are modeled as “in” and “out” signals in the Verilog language. Synthesis tools can incorrectly infer unexpectedly long combinational paths due to pessimism involved in calculating path length through these IOs. When we specify constraints to infer retiming flops on input paths explicitly, the tool is bound to view them as either input or output paths, and hence the critical path length is shortened leading to better performance numbers.
5. Multicycle path specifications: In many cases, it may be possible for the designer to constrain a path as a multicycle path. Thus, leading the tool to choose the next critical path in the list

for performance bottleneck calculations, e.g., a tool by being timing pessimistic may report such a path as critical whose outputs are not to be sampled in a single clock cycle or the path is clocked at much smaller frequency. Such paths are typical candidates for marking as multicycle paths and hence if constrained intelligently, may lead to better performance numbers [2.14].

6. False path specifications: In some cases, a tool may pessimistically consider some paths for timing calculations that are never to be exercised as functional [2.14]. Such "false" paths, if constrained, could result in much better synthesis frequencies. An example of a false path from Flop A to Flop B is presented in Figure 2.5. As can be seen in Figure 2.5, synthesis tool may pick highlighted path (passing through both combinational logics) as a single critical path but in actual functional operation, both of these logic blocks will never be active at same time as both multiplexers have common select line.

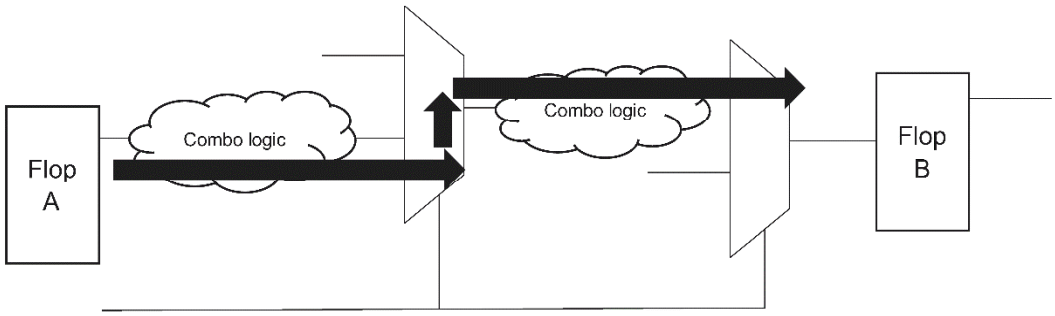


Figure 2.5. False timing path between two flip-flops.

7. Reducing clock tree complexity: In many designs, specially where clocks are multiplexed or pass through a complex combinational logic, it may not be easy for a synthesis tool to judge if it's an actual clock net or a simple data net, i.e., the synthesis tool may infer a standard "data" net as a clock net and hence may incorrectly consider it while doing clock tree synthesis. This may lead to false clock trees and may throttle the design performance. If the tool is constrained to infer the known data nets as no clock nets, this may help in reducing clock tree complexity leading to better design performance.
8. Breaking combinational loops: There may be unintentional combinational loops in the design which the synthesis tools may not be able to break. For example, in such a case, this may lead to an inference of unexpectedly long combinational paths. Most synthesis tools allow users to specify constraints to break these 'false' combinational loops and hence reduce the length of critical path thus improving design frequency.

2.3.2 Area optimization techniques

There are multiple areas of improvement for techniques used in FPGA synthesis. Some of these are explained below:

1. Redundant or dead-logic optimization: This strategy aims at remodeling the design and maintaining the same functionality such that a lesser number of logical units are used. As seen in Figure 2.3, final hardware implementation consumes a lesser area footprint than what would have been without any dead logic optimization.
2. Intelligent logic selection for constant operations: This methodology aims at the selection of logical units to implement the same functional behavior with a lesser number of units, e.g., if there is a variable register that gets multiplied by a constant (whose value is never changing, say, for example, 2), the same multiplication can be inferred by constant left shift instead of inferring a register to store the constant and along with a hardware multiplier.
3. Sharing of hardware resources: At times, it is possible to forgo performance for better area usage by sharing hardware resources like adders or multipliers using multiple logic paths [2.15]. An example is the sharing of memory ports for read and write by different masters accessing the memory in a time-multiplexed manner. Another example is shown in Figure 2.4. On one hand, in Case 2, as addition happens sequentially, then a single adder could be shared with the result being stored in different registers. On the other hand, in Case 1, as two additions are happening in parallel, there is no scope of sharing.
4. Data flow-based transformations: Most HLS compilers do data flow-based transformations [2.16]. Some examples are as below:

   a. Constant propagation (e.g., $A + B \times (x - x)$ can be reduced to A)

   b. Replacing $a = x^3$ by $a = x \times x \times x$

   c. Code motion: Moving a nondependent computation outside the loop.

   As shown below, Case 2 is optimal than Case 1 as sin(x) does not contribute to the critical path as it is deemed as a constant inside the loop.

```
input x;
For (int =0; int <= 100; int++)
{A = sin(x)
Y = cos(A*int)}
```

Case 1

```
input x;
A = sin(x)
for (int =0; int<=100; int++)
{Y = cos(A*int)}
```

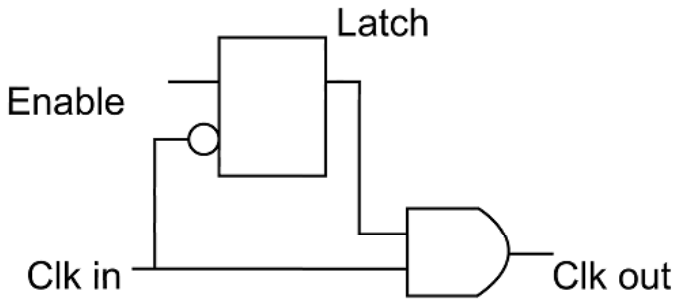
Case 2

2.3.3 Power reduction techniques

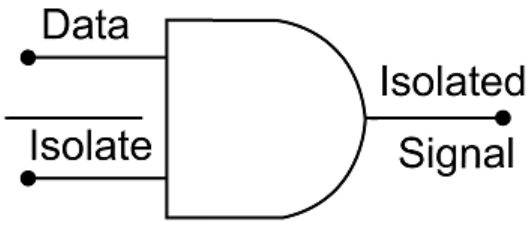
There are multiple power reduction techniques used in FPGA synthesis. Some of these are explained below:

1. Clock gating: Slowing down or switching off the clocks for portions of the design which are not required to be active during a particular operation being done by another part of the design is called clock gating [2.17]. For example, a multiplier may not be utilized during a branch instruction of a CPU. Hence, input registers to the multiplier can be retained at their previous values to save switching activity power. An example of a latch-based clock gating buffer is shown in Figure 2.6 (A).

2. Power gating: Shutting down the power of a part of the design when it is not in use is known as power gating [2.18]. The part of the design which is shut-off is known as the power domain and it is typically isolated from the rest of the design using isolation cells so that it does not corrupt the other logic. A typical isolation cell is shown in Figure 2.6 (B).



(A)



(B)

Figure 2.6. (A) Latch based clock gating (B). Isolation cell.

3. Path balancing: In conventional combinational logic circuits, spurious transitions account for 10% to 40% of switching activity power. As a result, the delays of the paths that converge at each gate should be made roughly equal to reduce the same. This is typically accomplished by adding unit delay buffers to inputs of the gates on faster paths.

Apart from the above, there are various other techniques like state encoding for minimum

toggle, sequential implementations and precomputations. In that last technique, the output value computation can be done one clock cycle in advance to reduce internal switching activity in the succeeding clock cycle.

2.4 Prior works in the area of HLS optimization

Today researchers in the area of HLS have come up with multiple methods which help to optimize multiple application designs. We classify the research in HLS optimization into two categories.

2.4.1 Proving the efficacy of HLS in VLSI design and optimization methods

Liang et al. demonstrated that performance of HLS based designs can be up to 40 times higher than the performance of traditional RTL design for the same algorithm [2.19]. In this study, a high-definition stereo matching application is selected as a benchmark to illustrate the performance gap between the two different design procedures. Ziegler et al. proposed a method of compiler analyses that could guide to map sequential C programs into an FPGA's pipelined implementation [2.20]. Meanwhile, Liu et al. introduced a novel customized optimization based on index set splitting to decrease pipelined loops' initiation overhead to reduce total latency [2.21]. Cong et al. have demonstrated how the quality of generated RTL design depends on source-level and intermediate-level optimizations [2.22]. They have implemented 56 different optimization techniques and show that some of them have a significant impact on the hardware quality. Huang et al. have studied the effects of various compiler optimization techniques on circuits generated using HLS [2.23]. According to the study of Huang et al., there are two important factors: the optimization methods and the order to enhance the generated circuits' performance. Six different optimization methods were implemented on benchmarks in this work, and a performance improvement of approximately 30% was achieved. Recently, there have been multiple works carried out which demonstrate the use of optimization methods like multicycle paths, false paths, etc. higher up in abstraction level for HLS designs.

2.4.2 Creating optimized VLSI application using HLS methods

Josh Monson et al. have published optimization techniques for HLS implementation of a Sobel edge filter design using Xilinx tools [2.24]. In their research, they discuss the fact how HLS tools bridge the gap between software programmers and hardware implementation platforms like

FPGAs. Further, they have demonstrated the use of directives and code restructuring steps such as loop unrolling, using FIFO-based input-outputs, and optimizing memory ports, which cause the variation of performance from 10.9 frames per second to 388 frames per second.

Similarly, Jason Cong et al. have published research that demonstrates high-performance designs on FPGAs using a fusion of two HLS platforms: AutoESL and Xilinx HLS [2.22]. Further, they have proved the methodology on a sphere decoder design by showing an 11% to 31% reduction in FPGA resource utilization using techniques such as dead code elimination, strength reduction, arithmetic simplification, function in-lining, and memory reuse.

V. P. Korakoppa et al., have proposed an adaptive threshold method for area-efficient implementation of moving object and face detection algorithms [2.25]. Umesharaddy et al. have proposed optimization techniques while implementing QPSK MODEM on FPGA using Vedic multipliers and carry look-ahead adders [2.26].

We have used Vivado HLS and MATLAB HDL coder as part of this study. So, the next two sections summarize the HLS optimization directives offered by these platforms.

2.5 HLS optimization directives in MATLAB HDL coder

The HDL coder is the HLS flow offered by MathWorks and it is part of the MATLAB and Simulink ecosystem [2.27]. MATLAB is script-based while Simulink is a block-based platform used for system design and verification. This tool enables the user to produce an HDL description from a system-level model hence offering multiple advantages. Simulink models are relatively more accessible than HDL languages to engineers who are not proficient in hardware design. Additionally, it is easy to reuse a script or a block in the same or different design, a typical advantage offered by HLS tools. Models are composed of fundamental blocks provided in the standard library as well as blocks from a collection of library elements also known as toolboxes provided by MathWorks.

When generating HDL code from a high-level model, a variety of configuration options are available to the user. Either Verilog or VHDL code may be generated, and it is possible to set optimization flags to optimize LUTs, flip-flops (area), clock speed, etc. It is also possible for users to have control over the generated code, such as the name of ports, synchronous or asynchronous reset etc. Individual blocks from the HDL coder library often also have such options and allow the user to sacrifice numerical integrity for additional speed. The tool will attempt to intelligently pipeline such that all signal wires have matched delays and the throughput will be maximized (subject to user-specified priorities). Some of the commonly used options offered by HDL coder are listed below:

1) Distributed pipelining and delay balancing: This introduces retiming flip-flops on long combinational paths in the model. This is a scheme used to break long critical combinational paths and hence improve the target frequency. This concept is illustrated in Figure 2.2. In addition to this, to compensate for the delay introduced on one path, there are additional flops introduced on parallel signal paths to keep functionality intact.
2) Multicycle path specifications: From the model, one can generate a register to register path information file or enable-based constraints. When this option is enabled in the HDL coder, it offers an option to the designer to specify a constraint about treating a flop to flop path as a multicycle path and ease out the timing calculation.
3) Resource sharing: This option identifies blocks in the design that have multiple inputs and replaces them with single blocks. In that way, two or more paths share a common block and consequently, the speed gets slower but the target area gets optimized. One can use a sharing factor to define the number of functionally equivalent resources being mapped into a single shared resource.
4) Random access memory (RAM) mapping: This option maps the pipeline registers in the generated HDL code to RAM blocks. This helps to save area. This is helpful only if the RAM size is greater than the RAM mapping threshold.
5) Loop unrolling: It creates copies of a single loop to improve performance with area tradeoff.

In addition to this, the tool offers multiple options to customize the generated RTL code which the designer can exercise. Some of these include: implementing synchronous or asynchronous reset, rising or falling edge choice, scalarizing vectors, minimizing intermediate signals (tradeoff between debuggability and target area), encoding scheme for enumeration data types, etc. The details of the same are available under the optimization guide for MATLAB and Simulink HDL coder [2.28].

2.6 HLS optimization directives in Vivado HLS

Vivado HLS is a platform from Xilinx Inc. that translates C, C++, or System C code into synthesizable RTL. Vivado HLS performs the code translation in three basic steps as mentioned below:

A) Scheduling: Determining which operations take place in which clock cycle. In case of a longer clock period or faster FPGA, more operations might complete in a single clock cycle, otherwise, the HLS schedules the operation over multiple clock cycles.

B) Binding: Identification of hardware resources that implement a scheduled operation. This uses the information from the target FPGA device.

C) Control logic extraction: It performs the extraction of control logic to construct a finite state machine and implement, in the RTL design, the sequence of operations. When using Vivado HLS, top-level C function arguments are implemented as IO ports of the design and internal functions are implemented as separate modules which get instantiated on the top-level parent module to create the hierarchical design.

Even though Vivado HLS offers multiple directives to optimize the generated RTL for the area, speed, and power optimization, some of the commonly used directives are discussed below:

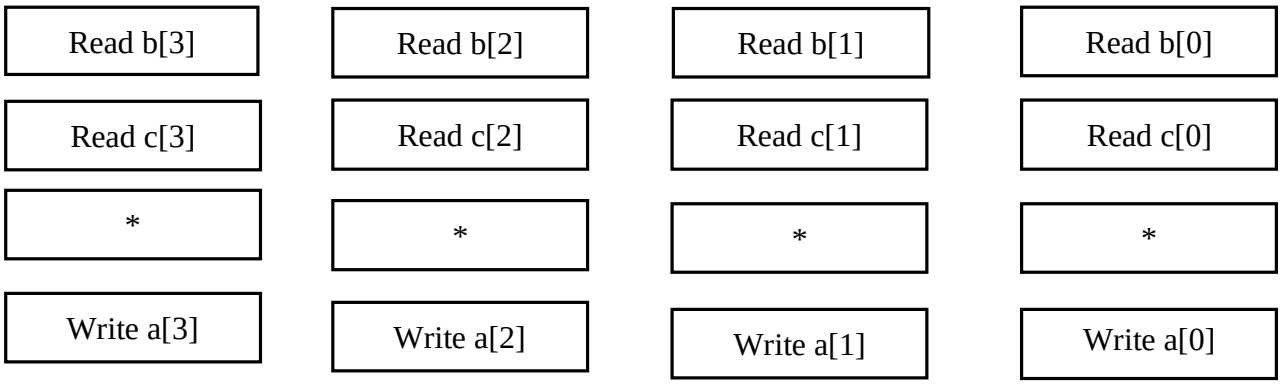
1. Loop unrolling

By default, loops in C language are rolled. This means every operation in a loop is implemented utilizing the same hardware resources for its iterations. The tool provides utility to partially or fully unroll "for" loops using "UNROLL" directive [2.29]. As shown in Figure 2.7, for a single "for" loop, there may be rolled, partially unrolled, and fully unrolled implementations.

C Function input to HLS

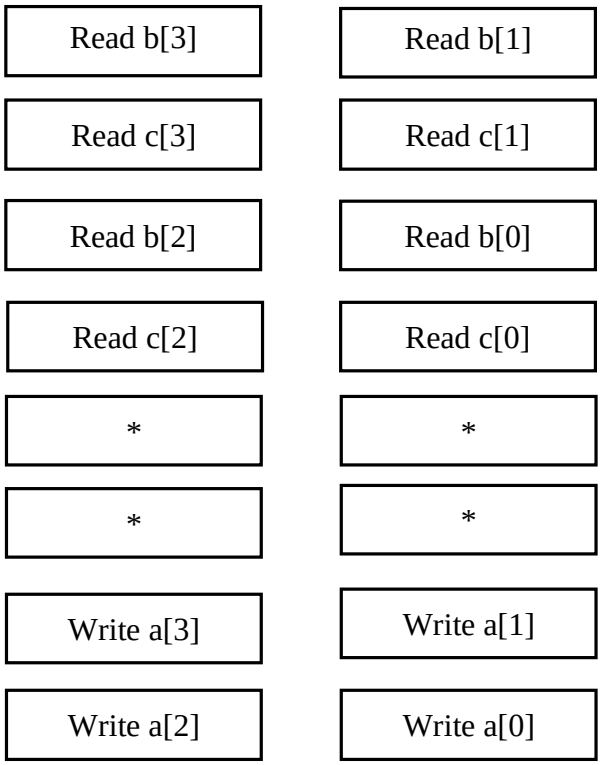
```
void top (...)
{
...
for_mult: for (i=3;i>0;i--)
{ a[i] = b[i] * c[i]; }
...
 }
```

Without unrolling (Time = 4 Clock cycles):



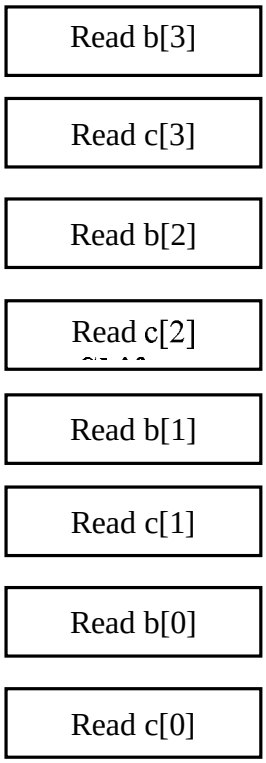
(A)

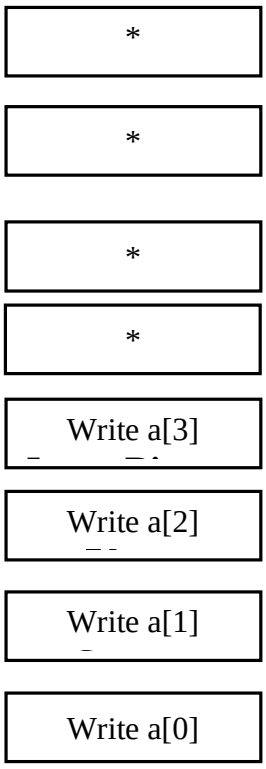
Partial unrolling (Time = 2 Clock Cycles):



(B)

Full unrolling (Time = 1 clock cycle):





(C)

Figure 2.7. (A) Without (B) Partial (C) Full unrolling optimization

In the case of a rolled loop, the iterations take four clock cycles while in the case of partial unrolling (factor of 2), the time is reduced to two clock cycles. When the loop is fully unrolled, the whole scheduling happens in a single clock cycle but at the cost of increased capacity, as multiple hardware elements need to operate in parallel. This example scenario assumes that a[i], b[i], and c[i] are mapped to block RAMs.

2. Pipelining

Tasks, functions, and loops inside a C code can be pipelined. This results in a greater level of concurrency and the highest level of performance. Therefore, the initiation interval is reduced by enabling, inside a loop or function, the operations' concurrent execution. If a function or a loop contains further loops, they are automatically unrolled when trying to pipeline parent functions [2.29].

3. Streaming

If one uses a "DATAFLOW" directive and the compiler cannot resolve whether the design tasks are streaming data, then the memory channel is implemented by default using block RAMs. One of the directives which save the target FPGA area under such a scenario is known as

"Streaming." This enables arrays to be implemented as FIFOs using registers for which the designer can specify a depth [2.29]. Such a directive is not suitable for scenarios where input data size is not known.

2.7 Gaps in existing research

Multiple authors have proposed hand-coded RTL implementations of different application designs in the literature like digital signal processing filters, Down converters, Image processing filters like Sobel edge or Harris Corner Detector, CPU cores and cryptography algorithms like AES [2.30, 2.31, 2.32, 2.33, 2.34, 2.35, 2.36, 2.37, 2.38, 2.39]. Even though HLS is the preferred design approach, as listed in Chapter 1, none or very few of these applications have been targeted using HLS flows in the literature. So, an attempt is made to demonstrate the usage of HLS tools for these 'benchmark designs' in this work. This aligns with the first objective of this work.

There have been some works in the literature aimed at proposing optimization methods for FPGA synthesis tools. In sections 2.5 and 2.6, we have discussed multiple optimization directives offered by Vivado HLS and MATLAB HDL Coder. As an example, a streaming directive may be applied in a C++ code using pragmas [2.29]. The choice of directives to be used based on the kind of design, given area, speed and power budgets is an important design decision which is not readily addressed by past studies in HLS. This aligns with the second objective of this work.

As it is evident, these directives are very specific to the tool and design being used. For example, directives for one HLS tool may not directly apply to other tools. Similarly a directive which helps optimize one type of application may not help another. Hence, there is a definite need for tool and application independent HLS optimization technique which is missing in the literature. This work presents a generic methodology for optimizing designs which applies to multiple HLS tools and applications. This aligns with the third objective of the work.

Most of the studies done in the literature have presented FPGA implementation of designs using HLS flows. To attempt the usage of HLS tools for ASIC flows in future, design community needs to have a confidence that there is no adverse effect of HLS optimization techniques on design functionality. Since most researchers have not compared the functional results of their implementations against ASIC results, this is a definite research gap. An attempt is made through this work by comparing the synthesis results and functionality with an already taped-out SoC model. This aligns with final objective of the work.

2.8 Concluding remarks

This chapter aimed at studying large number of optimized designs presented in the literature by multiple researchers in multiple application areas. HLS flows have multiple advantages as well as some limitations. Even though they are not a substitute for hand-coded RTL based, they are getting popular in VLSI design flows. One has an option to have a 'guided' HLS workflow in order to optimize the application being designed for area, speed and power. This can be done by making intelligent use of HLS directives presented in this chapter and Appendix. We studied some of the commonly used directives in Vivado HLS and MATLAB HDL coder which are the two core tools used as part of this thesis.

Many researchers have also presented HLS implementations for designs catering to benchmark applications. Multiple clear research gaps identified in past studies forms a basis for identifying the objectives of this thesis.