14 Spotlight on High-Level Synthesis

One of the significant trends in digital systems design is the move to accelerate development cycles, crucially without compromising on verification. This is inevitably driven by commercial factors, and in particular the need to take new products to market as quickly as possible. Of course, any strategies that reduce the design effort are also attractive due to the implied development cost savings.

The concepts underpinning design acceleration strategies are (i) *design reuse*, and (ii) *raising of the abstraction level*. Both of these themes have been introduced already in the book, and here we focus on the latter in particular, by placing the spotlight on Xilinx *Vivado High Level Synthesis* (or *Vivado HLS*). This is a tool for synthesis of digital hardware directly from a high level description developed in C, C++, or SystemC, i.e. removing the need to manually create a version of the design targeted to hardware, such as a VHDL or Verilog description. Rather, the HLS tool undertakes this task. The defining aspect of HLS is that the designed functionality and its hardware implementation are kept separate — the C-based description does not implicitly fix the hardware architecture, as is inherently true in RTL-level design — and this provides great flexibility.

We will see that working in high level, C-based languages has the potential to significantly reduce design time, due to the raised abstraction level. Additionally, the HLS process provides an integrated mechanism for generating and assessing variations on the hardware implementation, making it easy and convenient to find the best architecture.

This chapter will continue by defining and motivating the need for HLS, while also briefly reviewing the historical context and input languages used. The Vivado HLS tool will be introduced, and the design flow and processes of HLS reviewed from a high level perspective. This will be followed by a more in-depth look at Vivado HLS in Chapter 15.

14.1. High-Level Synthesis Concepts

Before proceeding to a discussion of the Vivado HLS tool and practical design methods in Chapter 15, it is important to establish some of the underpinning concepts. We will start with the most basic question!

14.1.1. What is High-Level Synthesis (HLS)?

It is worth beginning with a clear definition of *high-level synthesis*, and to do this we must first review the concept of abstraction with reference to digital design for FPGAs.

As in many other contexts, abstraction means 'taking away'. This is effectively the hiding of detail — the higher the level of abstraction, the more detail is hidden. There are various models for abstraction presented in the literature, many of which derive from the Gajsku and Kuhn 'Y-chart' for Very Large Scale Integration (VLSI) design [9], published as long ago as 1983! However, here we will focus our discussion on current FPGA design practices, and consider that there are four levels of abstraction, in ascending order: *structural*, *RTL*, *behavioural*, and *high-level*, as depicted in Figure 14.1.

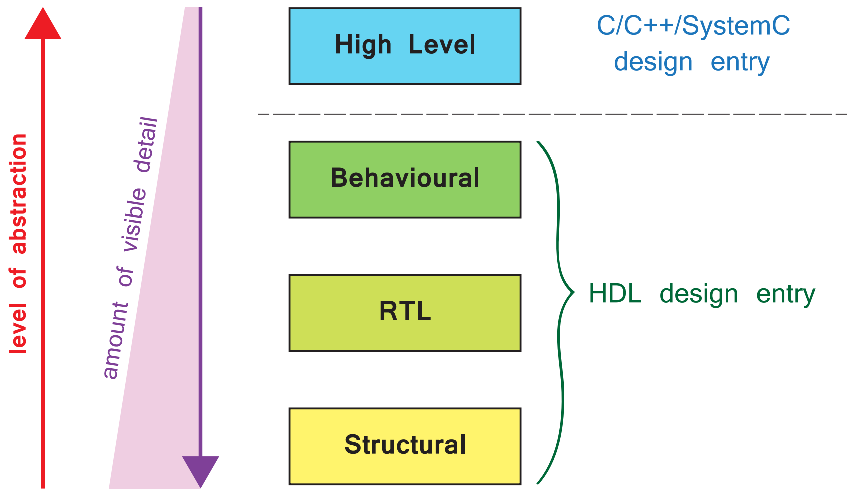


Figure 14.1: Levels of abstraction in FPGA designs

In terms of design entry using HDLs, such as VHDL or Verilog, the lowest level of abstraction (*structural*) involves explicitly instantiating, configuring and connecting each hardware element that comprises the design. This may even extend down to the level of LUTs and FFs. This style represents the designer taking explicit control over all of the design details, which reduces the potential for the synthesis process to optimise the design. More often, designers work at the '*Register Transfer Level*' *(RTL)* level. This level of abstraction hides the technology-level details, but still represents a description style from which registers, and the operations occurring between registers, can be interpreted. Logic synthesis tools are designed to work at this level of abstraction, i.e. to translate RTL code to hardware. *Behavioural* HDL descriptions are at a higher level of abstraction than RTL, and constitute an algorithmic description of the circuit (i.e. how it 'behaves'), rather than an expression of individual operations with respect to registers. The use of behavioural HDL, therefore, places a greater emphasis on the ability of the synthesis tool to infer hardware from the description, and as such the designer concedes some control over the final implementation, but with the implied benefit that designs can be created more rapidly. Finally, the *high-level* design entry method discussed in this chapter is not actually rooted in HDL, but uses a set of languages suited to expressing designs at an algorithmic level of abstraction, namely C, C++, and SystemC. These languages are, in fact, often used to develop System Level Models (SLMs) as a first, high-level step of modelling a system [22].

In the above discussion, we have used the term 'synthesis' in a slightly different context from HLS. *High-level synthesis* is distinct from other types of synthesis. In FPGA design, 'synthesis' usually refers to logic synthesis, i.e. the process of analysing and interpreting HDL code and forming a corresponding netlist. High-level synthesis actually means synthesising the high-level C, C++ or SystemC code into an HDL description, which would thereafter undergo logic synthesis to obtain a netlist. In other words, high-level synthesis and logic synthesis are both applied (one after the other) when taking a Vivado HLS design towards a hardware implementation, as shown in Figure 14.2.

Furthermore, *physical* synthesis refers to the conversion of HDL to a netlist with explicit knowledge of the target FPGA, and hence the optimisation of the design against the physical characteristics and available resources of the device.

14.1.2. Motivations for High-Level Synthesis

With a high-level representation abstracting low-level detail, the implication is that the description of the circuit becomes simpler. This approach does not actually remove the need to specify aspects such as wordlengths, parallelism and sharing, but to a large extent, the designer *directs* the process and the tools implement the details. The result is that designs can be generated much more rapidly than using more traditional methods, which

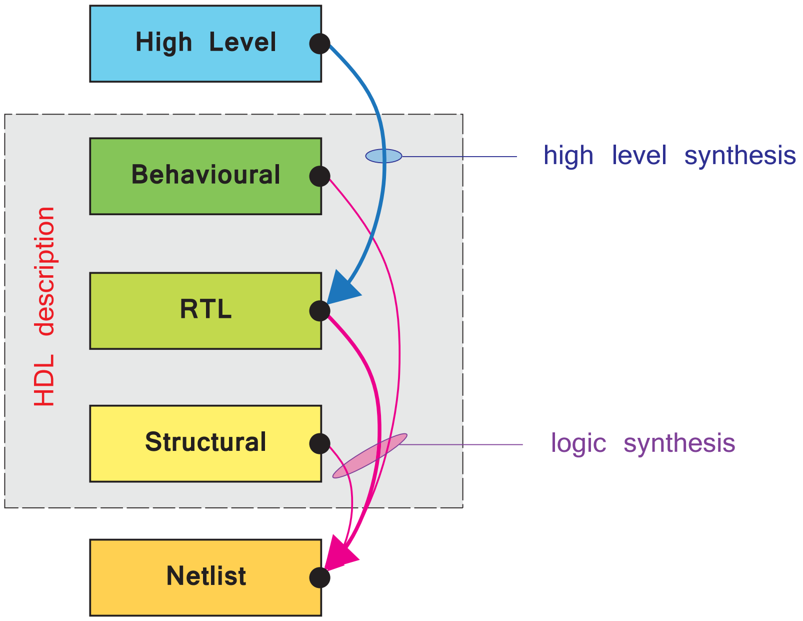


Figure 14.2: High level synthesis and logic synthesis

require the designer to explicitly describe all aspects. Importantly, the separation of functionality and implementation implied in HLS means that the source code does not fix the architecture. Variations on the architecture can be created quickly by applying appropriate directives to the HLS process, rather than having to fundamentally rework the source code, as would be necessary in RTL-level design.

Of course, the implication of giving up full control relies upon the designer trusting the HLS tools to implement lower-level functionality correctly and efficiently, and understanding how to exert influence over this process. There is however a clear motivation to do so, as the potential to accelerate the design process in this way is considerable.

On a more practical level, high-level synthesis from software languages is convenient for many designers, as these languages are widely adopted for developing algorithms and system-level descriptions. It would therefore be useful to leverage the experience of the many engineers comfortable with C, C++ and other programming languages for the purposes of FPGA design. The facility to rapidly convert from software to hardware design languages (crucially, without the need to create and validate an equivalent hardware-targeted design) therefore constitutes a simplification of the design process [22]. The transition from software description to hardware design would normally enforce the distinct design step of developing HDL and validating it against the software-based reference design. With this approach, the two processes can be combined into one. HLS is also beneficial in terms of systems development and software/hardware partitioning, as there is a common language for targeting both elements of the system; this makes it easy to adapt and retarget parts of the design as it is iterated to completion.

As we will see in detail later, the automation present in high-level synthesis tools permits the designer to rapidly produce alternative versions of a given design. For instance, variations on the same design could be produced with different levels of parallelism, allowing the design space to be explored, trade-offs evaluated, and the best solution identified.

In summary, if the motivation for HLS had to be expressed in one word, it would be 'productivity'!

14.1.3. Design Metrics and Hardware Architectures

An important point highlighted earlier in this chapter is that the functional description and the implementation are separate in HLS. Thus, the designer has the opportunity to evaluate possible architectures generated by the HLS process, and optimise according to his or her requirements. This statement does however beg the question, "*What is the designer trying to optimise?*".

In simple terms, hardware design has two fundamental metrics:

- ***Area***, or ***resource cost*** — the amount of hardware required to realise the desired functionality; and
- ***Speed***, or specifically ***throughput*** — the rate at which the circuit can process data.

There are also other factors, and in some cases subtle relationships between them (to be reviewed further in Section 14.4.4), but these are the two most important ones.

The designer is always faced with a trade-off between area (resource cost) and throughput, and this is the key aspect evaluated and optimised during HLS. There may be a certain minimum throughput or maximum area that the system can accommodate, and therefore the task is to decide on the best solution for their particular problem. HLS helps the designer to accomplish this quickly.

This point can be expanded upon using the analogy of decorating a house, and let us assume that the whole house needs decorated. The work could be undertaken by a single painter, and it would take him (or her) 6 weeks. On the other hand, 6 painters could finish the job in 1 week, or 3 painters would require 2 weeks, and so on. Various 'solutions' are possible in the trade-off between *resources* (number of painters) and *throughput* (the reciprocal of the time take to complete the job). A badly organised job might involve 5 painters taking 4 weeks to complete the decoration of the house, and this would of course represent a poor solution!

The realisation of a hardware architecture is similar. At a high level of abstraction, functionality is implemented using units of processing resource. Using more units allows processing to complete more quickly (high cost and high throughput), whereas at the opposite end of the scale, a single processing resource could be deployed and reused over time (low cost and low throughput). Several other points in the trade-off could also be attained. Of course, as with the decorating example, it would also be possible to come up with a poor design that achieves low throughput at high cost.

A simple perspective on the cost/throughput trade-off is provided in Figure 14.3.

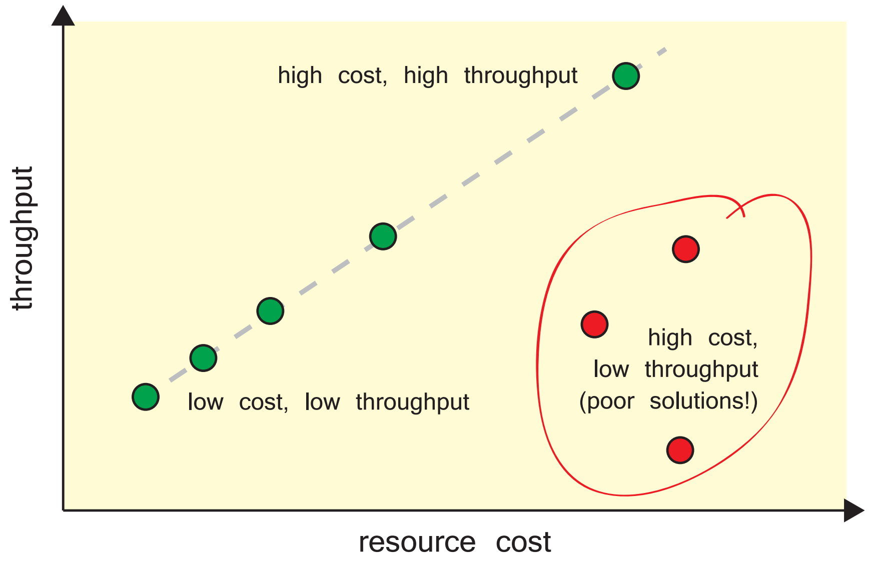


Figure 14.3: A conceptual illustration of the design trade-offs explored via HLS

14.2. Development of HLS Tools

High-level synthesis is not a new concept and, although it has received considerable research interest and found a level of adoption in industry, it has not yet become a widespread favourite among FPGA designers [12], [34]. However, with recent improvements in software design tools for HLS, and bearing in mind the clear motivations for HLS described in earlier sections, it is not surprising that interest in HLS is growing [21].

It is worth noting that contemporary HLS is very similar, in terms of motivations, conceptual framework and terminology, to the early methods reviewed in [20]; in particular, the *scheduling* and *binding* of operations (to be explained in detail later), and the ability to spin out multiple design variations from a single description. At the time the above referenced paper was written (the early 90's), there were still a variety of unresolved issues, including those of verification, human intervention/guidance, handling of complex constraints, and the integration of HLS-generated designs. Design entry methods included programming languages such as Pascal and Ada [29]. The C language subsequently gained prominence, and in fact C-based languages are still the most popular for HLS.

Later, interest grew in object-oriented programming languages for conversion to hardware descriptions, such as C++ and, to a lesser extent, Java [8], [19], [28]. Object-oriented software provides the abstraction mechanisms necessary to represent more sophisticated hardware designs. Further language features specific for hardware design are included in SystemC, namely the ability to model hierarchy, bit accuracy, and the concurrent execution behaviour of hardware [10], [25]. SystemC has gained prominence as a language for modelling complex designs at the system-level.

A number of other languages have enjoyed some level of adoption for C-based design, including *Handel-C*, which originated from the UK-based company, Celoxica, and was later purchased by Mentor Graphics [24]; Cadence's *C-to-Silicon Compiler* [7]; and *Impulse-C*, developed by Impulse Accelerated Technologies [15]. Although all of these represent methods of high-level design entry, the degree to which they fit our definition of HLS (i.e. the separation of function and architecture), varies. All of them represent specialist tools, additional to the algorithm and FPGA system design tools required for FPGA and Zynq development.

Meanwhile, MATLAB-based design entry was the subject of several academic papers in the early 2000's [2], [13], and has since featured in the *AccelDSP* tool (part of the Xilinx ISE suite, now deprecated) which was able to synthesise a subset of high-level MATLAB DSP functions [31]. Now, synthesis from MATLAB code and Simulink models prominently features in MathWorks' own *HDL Coder* product, providing a much more extensive MATLAB-to-RTL capability [23]. The attraction of synthesis from MATLAB code is similar to the motivation for synthesis from C code: high-level design methods can be used for design entry, abstracting many of the implementation details. This is particularly true as, given its status as an underpinning platform for Xilinx System Generator, many FPGA designers may already be familiar with MATLAB programming and its environment.

To bring the discussion back to the topic of this chapter, Xilinx' Vivado HLS tool was originally AutoPilot, developed by AutoESL Design Technologies Inc., a spin-out of the University of California, Los Angeles. Having surveyed the market for the best HLS technology available, Xilinx acquired AutoESL Design Technologies in early 2011 [3], [4], [30]. Xilinx subsequently integrated AutoESL into the ISE Design Suite, and later the Vivado Design Suite, at which time the tool was rebranded *Vivado HLS* [27].

The importance of these developments is significant, as they represent the graduation of HLS from niche tools to mainstream development.

14.3. HLS Source Languages

In the next section we will move on to discuss the Vivado HLS tool in detail, but it is useful to first review source languages for HLS in the general sense. We begin by providing some background on the three languages of Vivado HLS, namely C, C++, and SystemC, and then briefly summarise other languages supported by third party software development tools.

14.3.1. C

The popular, general-purpose C language is a procedural programming language which was initially developed by Bell Labs and has been in use since the 1970s [18], [26]. It was standardised by the American National Standards Institute (ANSI) in 1989, and subsequently adopted by the International Organization for Standardization (ISO) as standard ISO 9899. Since then, several revisions have been made to incorporate new features into the language, the most recent of which was in 2011 [16]. The standardised versions of C may be referred to as 'ANSI-C', 'ISO-C', or simply 'Standard C'.

Historically, C represented a raising in the abstraction level of software programming: it allowed programs to be written not in assembly language, but using higher-level constructs and commands. This had the significant advantages of simplifying the programming task: fewer lines of code were required, with less potential for error, enabling faster debugging and verification processes. It also made the code more portable, because it could be used on different platforms.

Of course, in order for C to be successful, it also had to be efficient enough to justify its use over assembly language, which gave the designer full and explicit control, but with greater design effort. In a sense, the adoption of C is analogous to the concept of HLS for hardware — in both cases designs are developed at a higher level of abstraction, with several benefits including a reduction in development time and effort, ease of verification, and enhanced portability.

Aside from Standard C, which defines the core functionality of the language, C can be extended by including application-specific and target-specific libraries. For instance, when using C to develop software for running on Zynq's ARM, additional libraries are included in the SDK to support floating and fixed point arithmetic, and to target the facilities of the NEON processor.

C has numerous relationships with other programming languages, the most significant of which is with C++.

14.3.2. C++

C++ is an object-oriented language based on C; it extends C with the concepts of classes, templates, polymorphism, and virtual functions, amongst several other features. The level of abstraction of C++ is normally higher than C, because C++ has features that hide detail and permit more sophisticated and flexible code to be developed. On the other hand, the language features and programming styles of C are compatible with C++, and consequently C++ can be considered a superset of C. In summary, C++ represents a higher level language than C, but it also retains support for low-level C operations.

Like C, C++ was also developed at Bell Labs. The standard for C++ was initially published in 1998 as ISO 14882, and has undergone revision to produce subsequent versions, most recently in 2011 [17].

14.3.3. SystemC

Although here we tend to treat SystemC as a language in its own right, strictly speaking it is an extension to C++, i.e. a specific class library which is included in development projects as the header file, 'systemc.h'. SystemC permits C++ style code to be written using many of the hardware-centric principles of HDLs, such as hierarchy, concurrency, and cycle accuracy, which do not form part of Standard C++.

SystemC is commonly used for Transaction Level Modelling (TLM) and Electronic System-Level (ESL) design, in the context of SoC [5]. TLM is the development of very high-level descriptions that model the interactions between system components, while ESL refers to the design of a system at a very high leve of abstraction, i.e. its functionality and basic architecture. As such, both of these represent methods for developing and refining the high-level framework for the SoC, before starting work on the functional implementation of individual components. In TLM, various levels of abstraction are possible in terms of, for example, the implementation target for individual components, and the timing fidelity associated with system transactions [6]. The language can also be used for ESL verification as an integrated part of the development process [11].

Early work on SystemC was supported by the Open SystemC Initiative consortium (which in 2011 merged with the Electronic Design Automation (EDA) standards organisation, Accellera, to form the Accellera Systems Initiative [1]). This represented the initial development phase of SystemC; the SystemC standard was ratified by the IEEE in 2005 as IEEE 1666, and later revised as IEEE 1666-2011 [14]. It is also closely related to *SystemC AMS*, which has analogue and mixed signal support.

Existing users of SystemC for TLM and systems engineering are in a particularly good position to adopt the language for HLS, but even without this background, designers should find that the hardware-oriented features of SystemC make the language very suitable for HLS, particularly for developing complex systems.

The scope of this chapter precludes in-depth introduction of SystemC, and we will use the C language for the majority of our illustrative examples. Readers are referred to [5] and [14] for further information on SystemC.

14.3.4. Other Languages for High-Level Synthesis

As reviewed in Section 14.2, the languages used for HLS are predominantly C-based, including C, C++ and SystemC as supported in Vivado HLS. We also noted that third party HLS design flows include other C-based languages such as HandelC, and also MATLAB code. Research interest in HLS continues, and it is possible that further high-level languages will be supported for FPGA synthesis in the future.

14.4. Introducing *Vivado HLS*

In this section, we will start by defining what Vivado HLS does and the steps involved, before considering its role in the design flow for Zynq. Later, Chapter 15 will cover use of the tool on a practical level, along with further discussions of algorithm and interface synthesis, and the processes involved in creating solutions and evaluating them.

14.4.1. What Does *Vivado HLS* Do?

In short, Vivado HLS transforms a *C*, *C++* or *SystemC* design into an RTL implementation, which can then be synthesised and implemented onto the programmable logic of a Xilinx FPGA or Zynq device [33]. This represents the high-level synthesis step depicted in Figure 14.2 on page 258.

It is important to reiterate that all C-based designs in the context of HLS are for implementation in *programmable logic*; i.e. as distinct from software code intended to run on a processor (whether Zynq's ARM processor, or a soft processor such as MicroBlaze).

In performing HLS, the two primary aspects of the design are analysed:

- The ***interface*** of the design, i.e. its top-level connections, and
- The ***functionality*** of the design, i.e. the algorithm(s) that it implements.

In Vivado HLS design, the functionality is synthesised from the input code via the process of ***Algorithm Synthesis***. The interface is created using one of two alternatives: it can either be (i) manually specified, or (ii) inferred from the code (***Interface Synthesis***). A simple conceptual diagram is provided in Figure 14.4 (note that this depicts only a subset of interface types).

In the specific case of the SystemC input language, interfaces must be manually specified, with two exceptions which will be mentioned later [33].

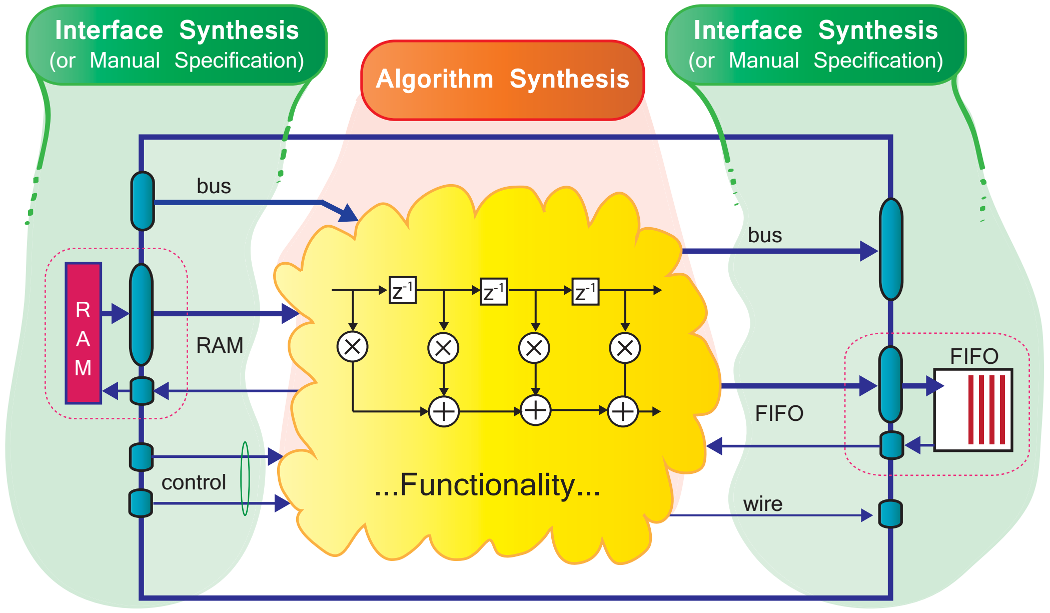


Figure 14.4: Clarification of the algorithm and interface, and showing a subset of interface types

Algorithm Synthesis

Algorithm Synthesis is concerned with the functionality of the design. The desired behaviour is interpreted from the C, C++, or SystemC code that forms the input to the process (the input to the HLS process is a function written in the chosen programming language). Operations are inferred, and these are translated into a set of RTL statements, which are usually executed over several clock cycles.

As will be discussed later, the designer can exert control over the algorithm synthesis process via Vivado HLS *directives*, resulting in variations on the RTL output from HLS being produced. For example, Solution 1 might require 500 slices to implement and 20 clock cycles to execute, whereas Solution 2 might require 1200 slices and 10 clock cycles. The designer is able to experiment and thus achieve a favourable result in terms of their priority implementation metric(s).

Interfaces I: Interface Synthesis

As its name suggests, ***Interface Synthesis*** refers to the interface of the HLS design, and this includes both the *ports* and the *protocols* used. The details of all ports (in terms of their types, dimensions, and directions) are inferred from the top-level function arguments and return values of the source C/C++ file; protocols are inferred from the behaviour of the ports. For example, the simplest interface would be a simple 1-bit wire, while for more complex interfaces, a bus or RAM interface may be used. Naturally the synthesised interface facilitates communicate with other modules in the system.

Interfaces that can be inferred from interface synthesis include: wires, registers, one-way and two-way handshakes, FIFOs, memories, and buses [33]. Further options are available relating to interface synthesis, specifically to permit ports to be inferred from global variables, and to include a global clock enable. Interface synthesis will be discussed in detail in Chapter 15.

Interfaces II: Manual Specification

Although C and C++ designs are fully supported for interface synthesis, SystemC designs are not, and as such, the interfaces of SystemC designs must be manually specified and their behaviour fully described. This corresponds with the hardware-specific features of the SystemC language, and an example will be provided in Chapter 15 which illustrates the similarity between SystemC-based interface specification, and the equivalent in VHDL.

Notably there are two exceptions to the general rule stated above: memory and bus interface types can both be inferred by interface synthesis of SystemC code.

Manual interface specification is also supported for C and C++ designs and may be used if desired; this means that there is the option to explicitly define interfaces, rather than have the interface synthesis process infer them.

14.4.2. Vivado HLS Design Flow

In the previous section, our discussion was limited to the primary processes and outputs, i.e. the execution of HLS algorithms and the production of RTL code. However, the full design flow for HLS comprises further stages, including elements of verification in particular. The full design flow is shown in Figure 14.5, and then summarised.

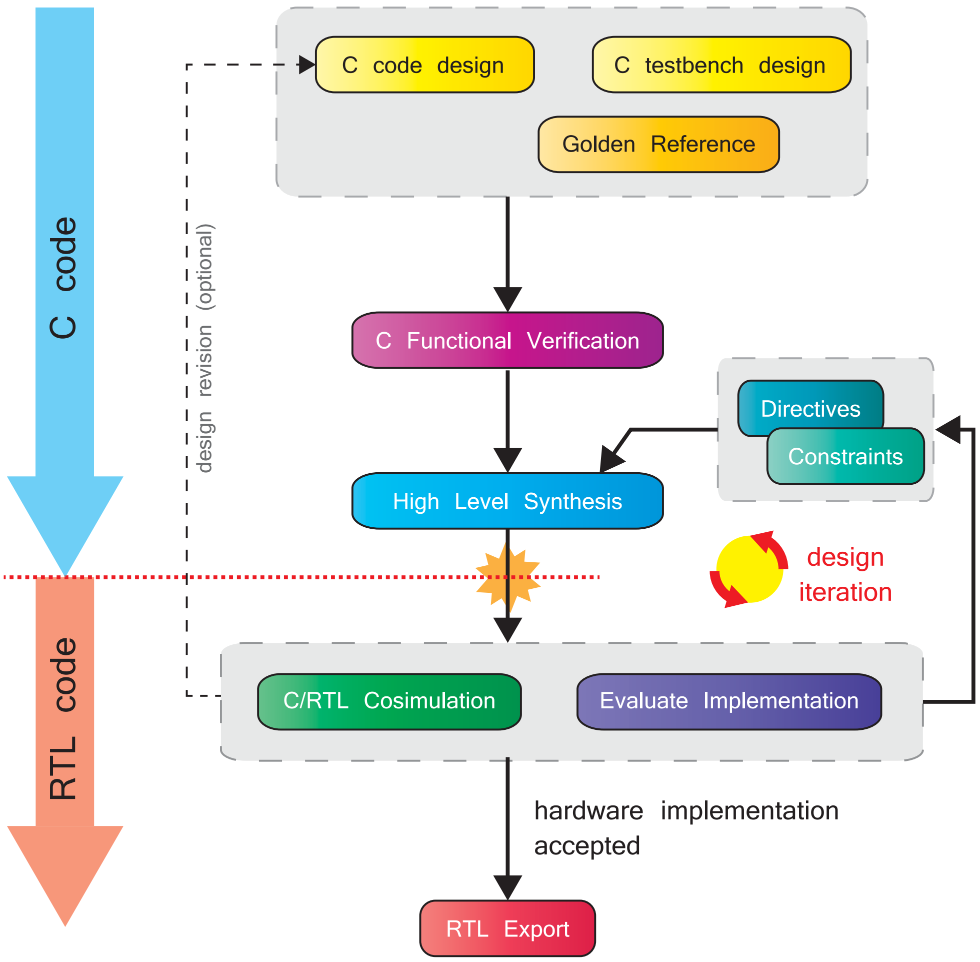


Figure 14.5: An overview of the Vivado HLS design flow

Inputs to the HLS Process

The primary input to the HLS process is a C/C++/SystemC function, along with a C-based testbench which has been developed to exercise the function and verify correct operation. This will involve a 'golden reference' against which to test the outputs produced by the function intended for synthesis; the golden reference may take the form of a prepared set of output values, or it may form part of the testbench itself.

Functional Verification

Firstly, it is necessary to verify the functional integrity of the C/C++/SystemC code that forms the input to HLS, before beginning the process of synthesising it into RTL code. This can be achieved by writing a testbench in the same high-level language, and checking the results produced against some form of 'golden reference'; for instance, this might be a prepared set of output test vectors which are known to be correct.

High-Level Synthesis

The next step is to undertake the HLS process itself, which involves analysis and processing of the C-based code, together with user-supplied directives and constraints, to create an RTL description of the circuit. Once the HLS process is complete, a set of output files is produced, including design files in the desired RTL language. Various other log and report files, testbenches, scripts, etc. are also created.

C/RTL Cosimulation

Once HLS has been performed and the equivalent RTL model produced, it can be checked against the original C/C++/SystemC code via the process of *C/RTL cosimulation* in Vivado HLS. This process re-uses the original, C-based testbench to supply inputs to the RTL version generated by HLS, and check the outputs it produces against expected values. Importantly, this saves the effort of generating a new RTL testbench. The SystemC output is particularly useful here, as it provides a mechanism for verifying the design in environments where an HDL simulator is not available. A good example covering both C functional verification and C/RTL cosimulation is available in [32].

Evaluation of Implementation

Along with verifying the integrity of the design, it is also necessary to evaluate the RTL output in terms of its implementation and performance. For example, the numbers of resources it requires in the PL, the latency of the design, maximum supported clock frequency, and so on. We will outline these metrics in Section 14.4.4, and discuss them in further detail in Chapter 15. For now, it is sufficient to note that, as the designer, we can influence the implementation via the constraints and directives applied to the HLS process.

Design Iterations

Noting the above, as part of the design flow, the implementation of the RTL is evaluated and, as necessary, the constraints and directives are refined; each revision corresponds to a new 'solution' in Vivado HLS terminology (more to follow on *solutions* in Section 14.4.6). It is also possible that evaluation of the design will prompt more fundamental review and refinement of the original algorithm, as designed in C code and input to the HLS process.

Figure 14.5 shows the steps undertaken from C design, to the creation of outputs for RTL synthesis. Note that multiple HLS iterations may be undertaken using modified directives and constraints, in order to find a 'best' solution; this corresponds to the feedback path shown on the right hand side of the diagram.

Should the designer be prompted to change the input C code, a more significant step backwards in the design process is involved, and this is indicated by the arrow at the left hand side of the diagram. Any changes to the C code require functional re-verification, before the subsequent HLS, C/RTL verification, and implementation evaluation processes are again performed and iterated as needed.

RTL Export

Once the design has been validated, and the implementation iterated to the point of achieving the intended design goals, it will be intended for integration into a larger system. This can be achieved directly using the RTL files automatically created by the HLS process (i.e. VHDL or Verilog code), however it may be more convenient to use the facilities of Vivado HLS for packaging IP. Packaging the outputs produced by Vivado HLS means that HLS designs can be easily introduced into other Xilinx tools, namely IP Integrator within Vivado IDE, XPS (for the ISE design flow), and System Generator.

SystemC outputs from the HLS process are produced to enable verification of the hardware produced, but are not themselves synthesisable.

14.4.3. C Functional Verification and C/RTL Cosimulation

Given the importance of verification, it is useful to further detail the *C functional verification* and *C/RTL cosimulation* processes. These are described graphically in Figure 14.6.

Depicted on the left hand side, a C-based testbench has been designed to create and supply input test vectors to the functional C module. The same test vectors are passed through a 'known good' golden reference design, or alternatively read from a prepared file, to give golden reference output test vectors. These are compared with the outputs from the C module, and the testbench reports a pass if the two set of results match, or failure otherwise. The testbench may also be designed to report the total number of errors, or to provide other automated feedback on the results.

As part of the Vivado HLS C/RTL cosimulation process, an equivalent testbench configuration is automatically created by Vivado HLS (shown on the right of Figure 14.6). The testbench verifies the RTL version of the original C module, i.e. the primary output of HLS, against the golden reference, and reports success or failure as before.

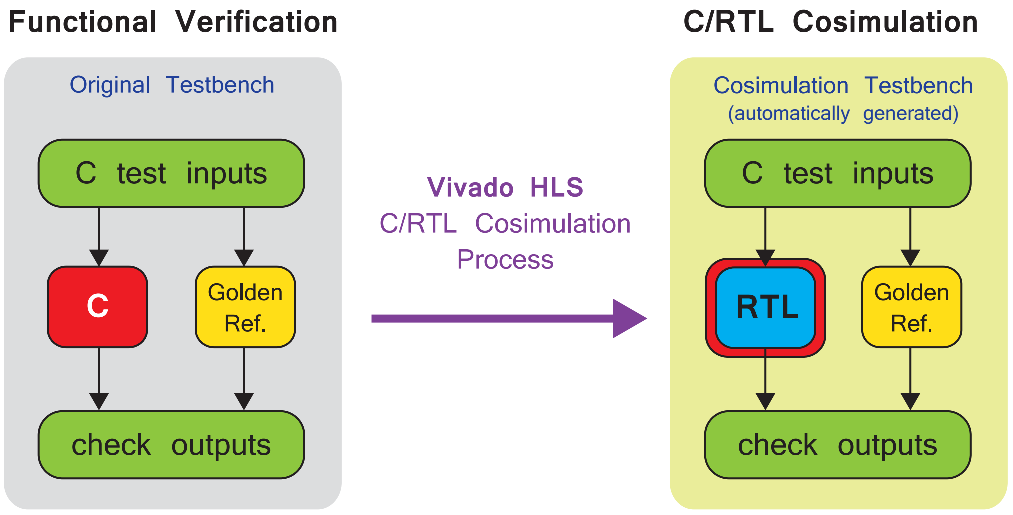


Figure 14.6: C functional verification and C/RTL cosimulation in Vivado HLS

All of the files required for C/RTL cosimulation are created automatically by Vivado HLS, which removes the need for manual RTL testbench creation. The generated testbench includes the necessary translations of data passing between the C-based testbench and the RTL module being tested.

Vivado HLS is also capable of creating a bit-true, cycle accurate System-C model of the generated hardware, and this can be co-simulated in circumstances where an RTL simulation is not available.

Verification is clearly an important aspect of the design process, and the availability of this tool support for RTL-level testing enables increased productivity. In particular, the designer does not need to spend time creating an equivalent testbench for RTL simulation, and the potential for introducing errors by doing so are eliminated. However, it is important to note that RTL simulation is a functional simulation, which does not model realistic timing or bus protocol behaviours. Therefore, it cannot completely verify the correct operation of the module in non-ideal conditions.

14.4.4. Implementation Metrics and Considerations

At this stage, it is useful to define the implementation metrics and related considerations which form part of the design process. We do so here in a simple and informal style; these issues will be returned to and expanded upon later, in Chapter 15.

- ***Resources / area*** — How many resources are needed to implement my design, and how does this compare to the amount available on my target FPGA / Zynq device?
- ***Throughput*** — At what rate can I pass data through the design? Does this meet the needs of my application?
- ***Clock frequency*** — What is the maximum clock frequency that I can run my design at? Is this compatible with the rest of my system?
- ***Latency*** — How many clock cycles does it take for my design to produce an output? Is this delay acceptable in the context of the system in general?
- ***Power consumption*** — How much power does my design consume when it is operating? Is this part of a system sensitive to power consumption?
- ***I/O requirements*** — How complex are the interfaces of my design? Are they compatible with other components of the system?

Any or all of the above factors may be constrained in some way, often with certain factors being prioritised over others. This normally depends on the requirements of the application. For example, a system targeted at a low cost application might prioritise resource minimisation, in order to utilise a smaller device; whereas, on the other hand, a system requiring to adapt quickly to changing inputs may seek to minimise latency and maximise throughput, at the expense of greater resource utilisation.

It is useful to describe briefly the term *clock uncertainty*. The HLS process aims to achieve the target clock period minus the clock uncertainty, which provides a margin for other delays that cannot be modelled at the HLS stage, e.g. RTL synthesis and place and route delays. HLS naturally treats the module in isolation, whereas routing delays are only fully exposed at the system integration stage when the system is populated. The clock uncertainty is user-specified, with a default value of 12.5% [33].

14.4.5. Overview of the High-Level Synthesis Process

Although we have discussed the design flow in general terms, it is useful to consider the process of HLS is more detail. This section will therefore summarise the steps involved in the HLS of C/C++/SystemC design files to achieve an RTL equivalent. For the purpose of providing practical examples, C will be used as the design language.

Recall from Section 14.4.1 that the HLS process performs (i) algorithm synthesis and (ii) interface synthesis (if the interface is not explicitly specified). Here we focus on the former.

Aside from the design files themselves, other inputs to this process are the specification of a particular target device, and the directives and constraints supplied by the designer. As will be discussed in Section 14.4.6, these directly influence the implementation produced by HLS. For the purposes of our current review, we will consider that the target device is fixed, and the constraints and directives applied are constant.

Algorithm synthesis comprises three primary stages, which occur in the following order:

1. Extraction of data path and control;
2. Scheduling and binding; and
3. Optimisations

Each of these will now be briefly explained in turn.

Extraction of Datapath and Control

The first stage of HLS is to analyse the C/C++/SystemC code and interpret the required functionality. This may, for instance, include: logical and arithmetic operations; conditional statements and branching; array operations; and loops.

The implementation will have a *datapath* component, and normally there will also be a *control* component. For clarification, here 'datapath' processing refers to operations performed on the data samples, whereas 'control' is the circuitry required to co-ordinate dataflow processing. The nature of the algorithm fundamentally defines both the datapath and control components but, as we will see in Chapter 15, the designer can take steps during HLS to minimise the complexity of the control component in particular.

Scheduling and Binding

HLS is comprised of two main processes: *scheduling* and *binding*. These are undertaken on an iterative basis, as shown in Figure 14.7, as one affects the other. The operations performed in the two processes are summarised below.

- ***Scheduling*** is the translation of the RTL statements interpreted from the C code into a set of operations, each with an associated duration in terms of clock cycles. The decisions made at this stage are affected by the clock frequency and uncertainty, the target device technology, and any directives applied by the user.

- ***Binding*** is the process of associating the scheduled operations with the physical resources of the target device. The functional and timing characteristics of these resources may affect scheduling, and therefore binding information is fed back into the scheduling process. For instance, the use of DSP48x resources implies a shorter critical path than an equivalent operator built from logic slices.

For example, if the synthesised algorithm requires that a set of arithmetic operations are performed, the HLS process must decide how to *schedule* the operations (how many clock

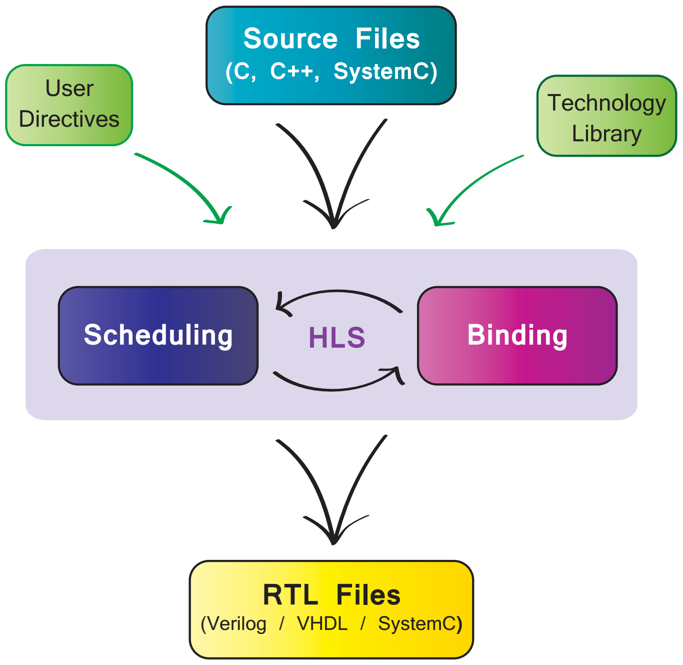


Figure 14.7: Vivado HLS scheduling and binding processes

cycles to allocate to their completion), and how to *bind* the operations (i.e. how to map them to the computational resources on the PL), bearing in mind the target clock frequency and uncertainty. Recall that the hardware architecture is not conveyed or specified by the C source code, but rather different architectural variations can be generated from the source code by applying directives.

The resulting implementation has a set of characteristics, principally in terms of (i) latency, (ii) throughput, and (iii) the resources used.

To illustrate, let us assume that our C algorithm involves calculating the average of an array input, consisting of ten numbers. The implied operations are:

- 9 addition operations to find the total; followed by
- A multiplication by 0.1 to calculate the average.

There are a few different options in terms of scheduling and binding these operations. One possibility is to operate serially over several clock cycles, using a single adder and a single multiplier. Alternatively, the critical path may permit multiple operations to take place within one clock cycle at the targeted frequency, resulting in an implementation with lower latency and higher throughput.

Three variations are depicted in Figure 14.8. Consider in particular the differences in implementation characteristics between them.

1. The first implementation uses the fewest resources (1 adder and 1 multiplier, both constructed from logic fabric), and has a latency of 11 clock cycles. This design has a low throughput — one 11th of the clock rate — because a new operation cannot start until the last one has finished (we assume that pipelining is not used here — to be explained later in Chapter 15).
2. Based on the target technology, the HLS process determines that 3 addition operations can be scheduled per clock cycle, while meeting the timing constraints. This results in an implementation using more resources on the device, but with a shorter latency and higher throughput.
3. Finally, it is determined that, if DSP48x slices are used in place of fabric resources, all operations can take place within one clock cycle. This corresponds to the most costly implementation, requiring 9 DSP48x slices in total (one each for the first 8 additions, with the last addition and multiplication combined into a single DSP48x slice), but it has a much reduced latency (1 clock cycle), and a throughput equal to the clock rate. This style of implementation would of course only be applicable if

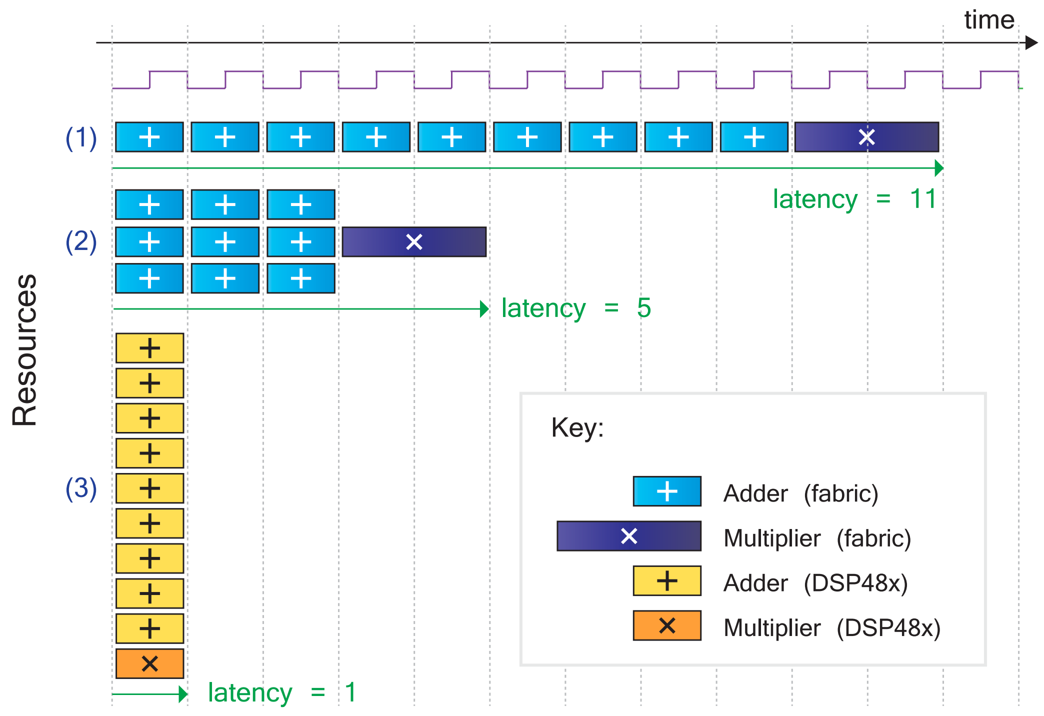


Figure 14.8: Comparison of three possible outcomes from HLS for an example function

timing constraints were met — otherwise Vivado HLS may insert pipelining registers to meet timing.

By default, the HLS process will optimise area, i.e. it will adopt the first strategy outlined above, which consumes the fewest resources. The disadvantages of this implementation are its long latency and low throughput, which may not meet the requirements of the application. However, the designer can exert influence by constraining and directing the HLS processes of scheduling and binding, and thus optimising in a different way.

Optimisations

As mentioned above, the designer has mechanisms available to drive the high-level synthesis process towards his or her implementation goals. There are two methods which can be used to dictate the behaviour of the HLS process, and hence influence the results:

- ***Constraints*** — The designer places a limit on some aspect of the design. For instance, the minimum clock period may be specified. This makes it easy to ensure that the implementation meets the requirements of the system into which it will be integrated. Similarly, the designer may choose to constrain resource utilisation or other criteria, with the aim of optimising the design for the application in hand.

- ***Directives*** — The designer can exert more specific influence over aspects of the RTL implementation via directives. There are various types of directive available which map to certain features of the code, enabling the designer to dictate, for example, how the HLS engine treats loops or arrays identified in the C code, or the latency of particular operations. This can yield significant changes to the RTL output. Therefore, with knowledge of the available directives, the designer can optimise according to application requirements.

A Note on the RTL Output

The user has the option to specify the RTL language for the generated output files, and can choose from VHDL, Verilog, and SystemC. It is, of course, notable that SystemC is also one of the available input languages for HLS, and therefore it might initially seem curious that it is also listed as an output type. However, the SystemC file produced as an output of HLS is an RTL description of the hardware, i.e. a lower-level representation than the input to HLS. In the same way as for the other HLS output languages (VHDL and Verilog), the implementation is based on the chosen target device. This output can be particularly useful in verifying the design where an RTL simulator is not available.

14.4.6. Solutions: Exploring the Design Space

A Vivado HLS project is comprised of a set of design files, a set of testbench files, and project settings, The project can have multiple *solutions* — this is an important term because it relates to the theme of "exploring the design space", i.e. generating a selection of possible implementations which can be compared, and the most suitable version chosen.

It is important to recognise that each solution is a different implementation of *the same* C/C++/SystemC source code. The differences in the synthesised RTL solution are determined by four factors:

- The target technology and part (e.g. Virtex-7, Kintex-7, Zynq-7000...)
- The target clock frequency
- The implementation constraints applied
- User defined synthesis directives

For a given application, the first two of these (target technology and part) are likely to remain constant, and the designer exerts influence over the generated solutions by varying the implementation constraints and synthesis directives, as discussed in the previous section. These will be defined in more detail as they relate to interface and algorithm synthesis in Chapter 15. Once created, each solution contains information about the target, the applied directives and constraints, and the results obtained.

As will be discussed in detail in the next chapter, the results obtained for a set of solutions can differ according to resource cost, throughput, and latency. These can be explored, and further refined, according to the requirements of the task.

14.4.7. Vivado HLS Library Support

It is worth noting that Vivado HLS includes support for arithmetic and mathematical functions, as well as linear algebra, video processing, DSP and others. Full details of library support is available in [33].

14.5. HLS in the Design Flow for Zynq

In Section 14.4.2, the design flow for Vivado HLS was reviewed as an independent process. However, we must also set it in context of the design flow for Zynq.

HLS is a design method for creating functional modules, or IP blocks, for inclusion in a Zynq-based system. IP blocks designed using HLS are for implementation in the PL part of the target Zynq device. There may be multiple such modules in a particular Zynq system design, and part of the task involves appropriately interfacing them to the rest of the design (using for example AXI connections). Referring back to Figure 3.2 on page 53, this corresponds to the stage marked ‘Vivado HLS / HDL / System Generator’ — modules are created using Vivado HLS for integration into the design as IP blocks.

While Vivado HLS can be used to describe commonly used functional modules such as FIR filters and FFTs from first principles, the designer should be aware that dedicated and optimised support for these operations already exists, and is accessible via the IP Catalog of IP Integrator, or the Xilinx BlockSet of the System Generator tool. The same functionality can be introduced into the Vivado HLS design by calling a function (e.g. an FFT or FIR function) from a supplied library — details of these functions can be obtained from the *Vivado Design Suite User Guide: High-Level Synthesis* [33]. HLS is also a very powerful method for developing custom functionality that does not have a direct equivalent in the existing catalogue of IP.

14.6. Chapter Review

This chapter has reviewed the concept of high-level synthesis, and explained its growing significance as a design method, particularly in the context of Zynq and SoC design. HLS permits an algorithm to be defined at a high level of abstraction using a software language, and it is subsequently converted into an RTL description with the aid of a high-level synthesis tool. This design method has the potential to greatly enhance productivity in terms of both design entry and verification effort.

The Vivado HLS tool was introduced, and its design flow described. It was noted in particular that, in addition to synthesis itself, the Vivado HLS flow also integrates facilities for streamlined verification at a functional level, and also subsequently for validation of the generated RTL code.

Towards the end of the chapter, we reviewed the processes underpinning HLS, and the mechanisms available to the designer to exert influence over the behaviour of these processes, and hence the results obtained. Standard implementation metrics and considerations were outlined, and it was noted that multiple 'solutions' can be generated from the same set of Vivado HLS input files. This permits detailed exploration of the implementation possibilities, evaluation in terms of key metrics, and optimisation with respect to the designer's priority criteria.

In the next chapter, we will build upon this conceptual framework by taking a detailed look at the creation of designs using Vivado HLS.

14.7. References

Note: All URLs last accessed June 2014.

[1] Accellera Systems Initiative website,
available: http://www.accellera.org/

[2] P. Banerjee, "Overview of a compiler for synthesisizing MATLAB programs onto FPGAs", *IEEE Transactions on VLSI Systems*, Vol. 12, Issue 3, March 2004, pp. 312-324.

[3] Berkeley Design Technology, Inc., "An Independent Evaluation of: High-Level Synthesis Tools for Xilinx FPGAs", consultation paper, 2010.
Available: http://www.xilinx.com/technology/dsp/BDTI_techpaper.pdf

[4] J. Bier and J. E. White, "BDTI Study Certifies High-Level Synthesis Flows for DSP-Centric FPGA Design", *Xilinx Xcell Journal*, Issue 71, second quarter 2010, pp. 12 - 17.
Available: http://www.xilinx.com/publications/archives/xcell/Xcell71.pdf

[5] D. C. Black, J. Donovan, B. Bunton and A. Keist, *SystemC: From the Ground Up*, 2nd Edition, Springer, 2009.

[6] M. Burton, J. Aldis, R. Günzel and W. Klingauf, "Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified", *Forum on Design Languages*, 2007, pp. 92-97.

[7] Cadence, *C-to-Silicon Compiler* webpage,
Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx

[8] J. M. P. Cardoso and H. C. Neto, "Towards an Automatic Path from Java Bytecodes to Hardware Through High-Level Synthesis", *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems*, 1998, vol. 1, pp. 85-88.

[9] D. Gajski and R. Kuhn, "Guest Editors' Introduction: New VLSI Tools", *Computer*, vol. 16, no.12, pp.11 - 14, December 1983.

[10] D. Gadski, T. Austin and S. Svoboda, "What Input-Language is the Best Choice for High Level Synthesis (HLS)?", *panel session, Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC)*, pp. 857-858, June 2010.

[11] D. Große and R. Drechsler, *Quality-Driven SystemC Design*, Springer, 2010.

[12] R. Gupta and F. Brewer, "High-Level Synthesis: A Retrospective" in *High-Level Synthesis: From Algorithm to Digital Circuit*, edited by P. Coussy and A. Morawiec, Springer, 2008.

[13] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, "FPGA Hardware Synthesis from MATLAB", *Proceedings of the 14th International Conference on VLSI Design*, January 2001, pp. 299-304.

[14] IEEE Computer Society, "IEEE Standard for Standard SystemC Language Reference Manual", *IEEE Std 1666-2011*, January 2012.

[15] Impulse Accelerated Technologies, *Impulse CoDeveloper C-to-FPGA Tools* product webpage.
Available: http://www.impulseaccelerated.com/products_universal.htm

[16] International Organization for Standardization, "ISO/IEC 9899:2011: Information technology - Programming languages - C", 2011.

[17] International Organization for Standardization, "ISO/IEC 14882:2011: Information technology - Programming languages - C++", 2011.

[18] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[19] T. Kuhn and W. Rosenstiel, "Java Based Object Oriented Hardware Specification and Synthesis", *Proceedings of the Asia and South Pacific Design Automation Conference*, 2000, pp. 579-581.

[20] M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, vol. 78, no.2, pp 301 - 318, Feb 1990.

[21] G. Martin and G. Smith, "High Level Synthesis: Past, Present and Future", *IEEE Design and Test of Computers*, Vol. 26, Issue 4, July/August 2009, pp. 18 - 24.

[22] A. Mathur, E. Clarke, M Fujita, and R. Urard, "Functional Equivalence Verification Tools in High-Level Synthesis Flows", *IEEE Design & Test of Computers*, July/August 2009, pp. 88 - 95.

[23] Mathworks, *HDL Coder* product webpage.
Available: http://www.mathworks.com/products/hdl-coder/

[24] Mentor Graphics, *Handel-C Synthesis Methodology* webpage.
Available: http://www.mentor.com/products/fpga/handel-c/

[25] M. Meredith and S. Svoboda, "The Next IC Design Methodology Transition is Long Overdue", Open SystemC Initiative, February 2010.
Available: http://www.accellera.org/resources/articles/icdesigntrans/community/articles/icdesigntrans/ic_design_transition_feb2010.pdf

[26] D. M. Ritchie, "The Development of the C Language", *Proceedings of the 2nd History of Programming Languages Conference*, Cambridge, Massachusetts, April 1993.

[27] M. Santarini, "Xilinx Unveils Vivado Design Suite for the Next Decade of 'All Programmable' Devices", *Xilinx Xcell Journal*, Issue 79, second quarter 2012, pp. 8 - 13.
Available: http://www.xilinx.com/publications/archives/xcell/Xcell79.pdf

[28] R. Thomson, V. Chouliaras and D. Mulvaney, "The Hardware Synthesis of a Java Subset", *Proceedings of the Norchip Conference*, 2006, pp. 217-220.

[29] H. Trickey, "Flamel: A High-Level Hardware Compiler", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 2, March 1987, pp. 259 - 269.

[30] Xilinx, Inc., Press Release: "Xilinx Acquires AutoESL to Enable Designer Productivity and Innovation with FPGAs and Extensible Processing Platform", 30th January, 2011.
Available: http://press.xilinx.com/2011-01-30-Xilinx-Acquires-AutoESL-to-Enable-Designer-Productivity-and-Innovation-With-FPGAs-and-Extensible-Processing-Platform

[31] Xilinx, Inc., "UG634 - AccelDSP Synthesis Tool User Guide", v11.4, December 2009.
Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/acceldsp_user.pdf

[32] Xilinx, Inc., "UG871 - Vivado Design Suite Tutorial: High Level Synthesis", v2014.1, May 2014.
Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug871-vivado-high-level-synthesis-tutorial.pdf

[33] Xilinx, Inc, "UG902 - Vivado Design Suite User Guide: High-Level Synthesis", v2014.1, May 2014.
Available: http://www.xilinx.com/support/documentation/sw_manuals_j/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[34] J. Yi and H. Kwon, "Samsung's Viewpoints for High-Level Synthesis" in *High-Level Synthesis: From Algorithm to Digital Circuit*, edited by P. Coussy and A. Morawiec, Springer, 2008.

15

Vivado HLS: A Closer Look

One of the most significant developments in the Xilinx design methodology has been the introduction of a capable tool for high-level synthesis: *Vivado HLS*. Having established the motivation for HLS in the previous chapter, we now take a detailed look at the methods used to design with Vivado HLS.

In doing so, this chapter will cover several topics, including the specification of data types and implications for circuit synthesis, the creation of port and block-level interfaces, and aspects of algorithm synthesis. The use of directives and constraints to influence the solutions produced by HLS will also be demonstrated.

It must be noted that the facilities of HLS are so broad and varied that there is simply not enough space in this chapter to cover everything, so instead we aim for an appealing introduction, and direct the reader to [17], [18], and [19] for more detailed review and tutorial material. This chapter will, however, present a case study focussing on loops, which acts as a good basis to demonstrate a some of Vivado HLS's features and optimisations, and thus to provide the reader with a taste of what can be achieved with HLS.

As with other components of the Vivado Design Suite, there is an emphasis on integration and design reuse, and therefore Vivado HLS includes facilities for packaging IP for convenient integration into system designs. We will briefly review this aspect of the tool towards the end of the chapter.

15.1. Anatomy of a Vivado HLS Project

It is useful to begin by with a high-level model of the Vivado HLS ***synthesis*** process. (Note that simulation and verification are also undertaken using the C testbench (and an automatically generated RTL version of the C testbench), as was previously described in Section 14.4.3 on page 269.)

The inputs to the process are:

- ***C, C++ or SystemC files*** — These contain the functions to be synthesised. In a simple design, there may be a single file containing a single function, or in a more complex case, a hierarchy of sub-functions and multiple files.
- ***C testbench files*** — The C testbench files form the basis for verifying both the C code and the RTL code generated by the HLS process.
- ***Constraints*** — The designer supplies a timing constraint (desired clock period), along with a clock uncertainty figure and details of the target device. Together, these influence the synthesis process along with *directives*.
- ***Directives*** — Directives applied by the designer influence the style of implementation generated from the high-level description (input C code), for instance in terms of pipelining and parallelism.

The outputs produced are as listed below. The designer is able to choose which of these are created.

- ***SystemC model*** — This is an RTL-level model of the output from the HLS process, i.e. a different style of description that an input SystemC file. This SystemC output is not intended for synthesis, but only for RTL simulation.
- ***VHDL or Verilog files*** — The Vivado HLS process generates an RTL-level output in the VHDL or Verilog language, depending on user preference. This is synthesisable code that could be integrated into a project and used to generate a bitstream (*.bit file) for programming an FPGA or Zynq device.
- ***Packaged IP for Vivado, System Generator, or XPS*** — The packaged outputs are convenient for direct inclusion into an IP Integrator project, XPS project, or System Generator design.

The various files outlined above form the basis of a Vivado HLS project. Next, we will move on to introduce the Vivado HLS development environment.

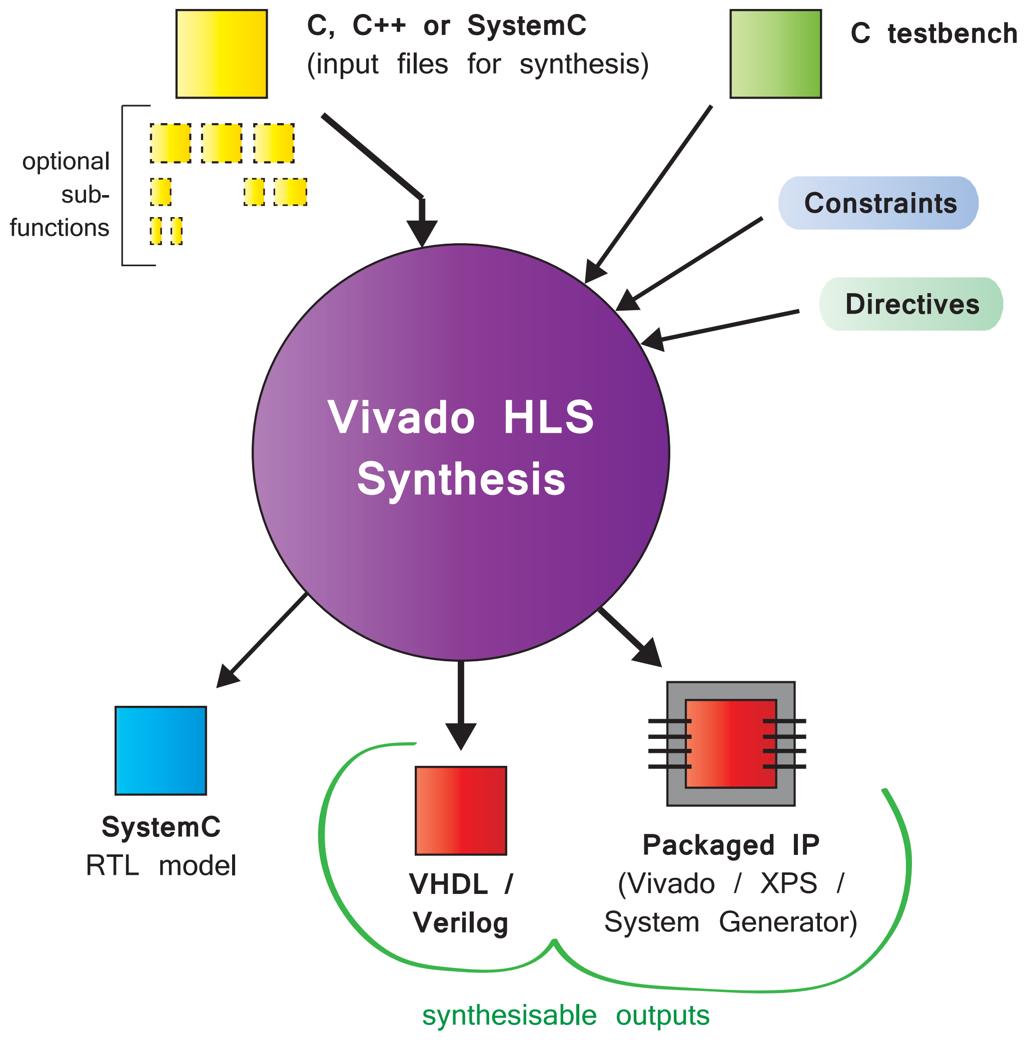


Figure 15.1: An overview of the Vivado HLS synthesis process

15.2. Vivado HLS User Interfaces

The Vivado HLS tool provides both a Graphical User Interface (GUI) and a Command Line Interface (CLI), which may be used separately or in conjunction with each other according to individual preference. Both methods provide access to the same set of functionality, and offer different advantages from a user perspective.

15.2.1. Graphical User Interface

The Vivado HLS GUI is similar to software development environments, providing facilities for managing projects, code editing, and debugging. Additionally, there HLS-specific features for directing the HLS process and evaluating the synthesised hardware. It is useful to highlight these, and the sections following provide a brief overview of each.

Figure 15.2 provides clarification of how these specific components relate to the Vivado HLS GUI. The GUI actually provides three different *perspectives*: Debug, Synthesis and Analysis, which by default lay out the GUI with relevant information panes. The facilities highlighted here relate to the Synthesis and Analysis perspectives.

Synthesis Perspective: Project Organisation

The iterative nature of Vivado HLS development is based on the concept of ‘solutions’ — alternative implementations of the same source code, generated on the basis of different guidance from the user. The structure of Vivado HLS projects reflects this: there are separate folders for the source code and testbench, together with any included header files, and a set of solution folders. Each solution represents a different implementation, and contains files, reports and results relating to that implementation; the user can generate as many solutions as required, and then evaluate and compare them. At any one time, only one of the solutions is said to be ‘active’, and the subject of directives and analysis.

The project structure of an example project is shown in Figure 15.2, where the active project is shown in bold. There are also folders for the HLS output files and reports.

Synthesis Perspective: Directives Pane

Vivado HLS also features a pane for setting up and managing directives, which influence how the HLS process behaves. The *Directives* pane reflects the ‘active’ solution only, and is only visible when the source code is open in the main window.

Note that directives can be either:

- Kept separate from the source code, as TCL commands within a directives file; or
- Integrated into the source file as pragmas.

The choice of inserting directives into the code, or applying them in a separate file, affects how the directives appear in the Directives pane. As indicated in Figure 15.2, a hash (#) symbol indicates a pragma (incorporated into the source code), while a percentage sign (%) indicates a directive located in the dedicated directives file.

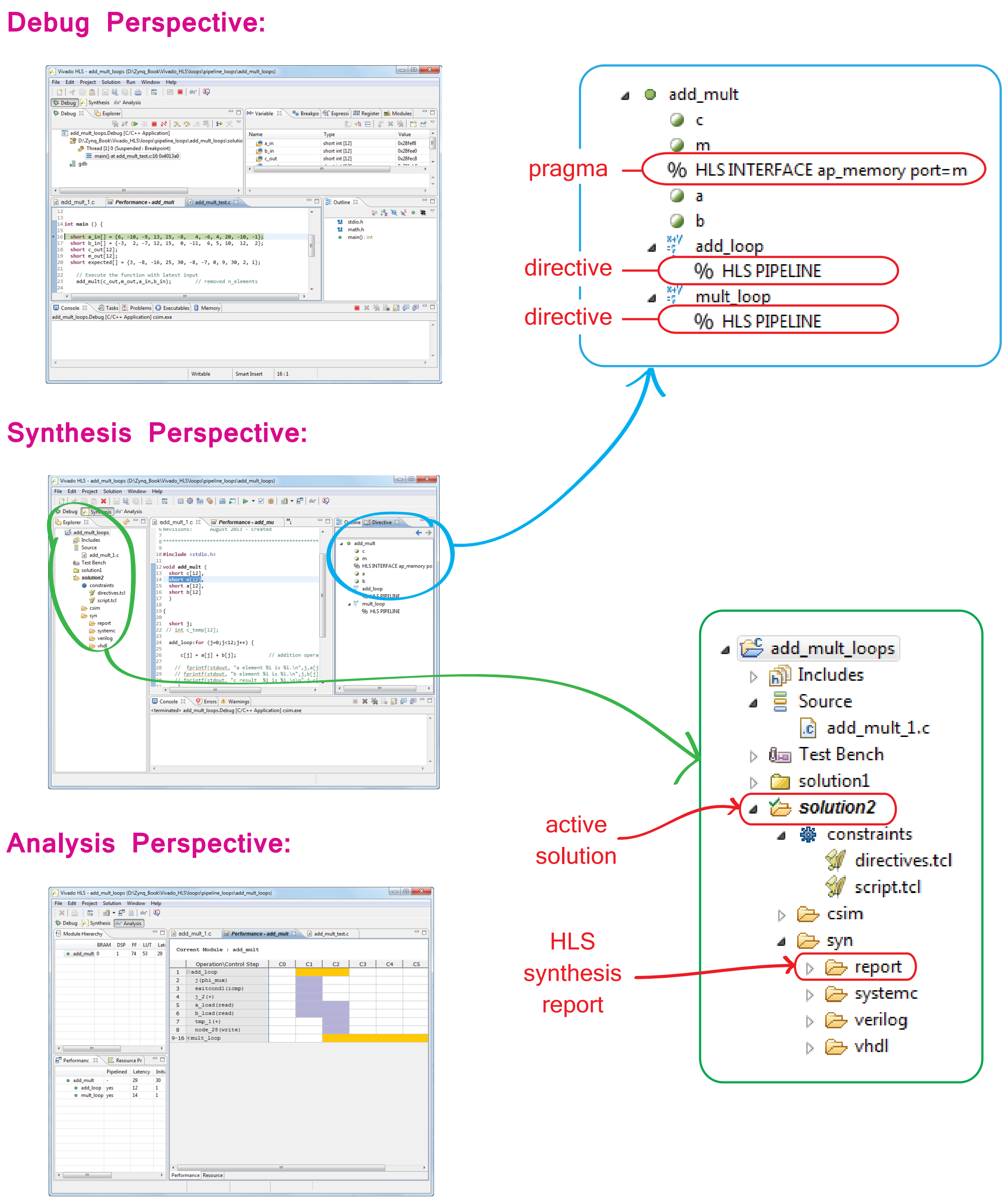


Figure 15.2: Overview of Vivado HLS GUI perspectives

There are advantages to both approaches, and these will be discussed in Section 15.4.6, in the context of interfaces.

Synthesis Perspective: Synthesis Report

Vivado HLS produces a synthesis report for each solution, which represents a consolidated set of statistics relating to that particular implementation. The details include:

- Clock information, and comparison with constraints;
- Latency statistics;
- Details of loops identified within the code (e.g. trip count, latency per loop);
- The estimated cost of the implementation, in terms of different PL resources;
- A list of the synthesised RTL interface ports, including their directions, dimensions and associated protocols.

A separate comparison report can be generated for a specified set of solutions. This permits direct comparison of key implementation metrics across solutions, and it is a very useful facility for iterating towards the optimum solution.

Analysis Perspective

Aside from the synthesis results, there is also an analysis perspective available in the GUI which presents a graphical visualisation of the synthesised design, in terms of operations and controls steps. This information is linked to resources, and also cross-references to the C/C++ code from which it was originally synthesised.

The analysis perspective is useful because it permits the designer a deeper appreciation of how the design has been synthesised, and thus informs further iterations. Viewing the details of the synthesised design helps the designer to identify operations which cause bottlenecks, and which may benefit from further optimisation.

The Analysis perspective will be discussed further in Section 15.6.

15.2.2. Command Line Interface (CLI)

The command line interface / TCL scripting method is well suited when performing repetitive or prescribed tasks, because the required steps can be performed in an automated fashion, thus saving time and ensuring reproducible results.

The method of entering commands is via the TCL language. This is an open source scripting language which is widely used in ASIC and FPGA development. When working with Vivado HLS, TCL can be used to run basic tasks such as setting up projects and running simulations, through to driving extensive sets of tests using defined sets of parameters and directives. In terms of usage, normally it is convenient to prepare a script in advance (e.g. 'my_hls_script.tcl', to include all of the required settings and commands), and execute it once. It is also possible to run commands individually by typing them directly at the prompt.

A comprehensive guide to all TCL commands for Vivado HLS is available, and this is the key resource for developing scripts to drive the tool [18]. Examples of using scripts are also incorporated into the Xilinx tutorials for Vivado HLS [17], and some general guidance is available in a general TCL guide for Vivado Design Suite [16]. You can also gain experience with this method by following one of the tutorials that accompanies this book — see Chapter 16 for more details.

15.3. Data Types

When using traditional design methods for FPGAs, the specification of data type is important as it has a direct effect both on the integrity of the design, and the key metrics of the implementation (namely resource utilisation, timing performance, and power consumption). In the case of numerical data types, underspecifying the wordlength compromises accuracy, while overspecifying can lead to increased resources, inflated power consumption, and a suboptimal maximum clock frequency. It is therefore necessary to specify data types carefully.

This aspect is equally important in Vivado HLS as compared to other methods such as HDL development or block-based design, even if the data types at the point of design entry are different. Understanding the available C, C++ and SystemC data types, and their synthesis, is fundamental to developing effective and efficient designs. To that end, this section will be devoted to reviewing the available types, and explaining how these are translated into an RTL design and hence into hardware. We will start by considering the native data types of the C and C++ languages, followed by arbitrary precision types.

15.3.1. C and C++ Native Data Types

The C and C++ languages have several built-in data types which derive from four basic numeric types: `char`, `int`, `float` and `double`, as summarised in Table 15.1.

In the cases of `char`, `int`, and derived types, these are signed by default, but can also be specified as `unsigned` (or `signed` to prevent ambiguity). Notably the standard `int` type, and the `short`, `long`, and `long long` versions of the `int` type, equate to minimum sizes. Here, typical values are chosen [5], [9].

Table 15.1: Native data types in the C language

Type	Description	Number of Bits[a]	Range[b]
`char`	Representation of the basic character set.	8	-128 to 127
`signed char`		8	-128 to 127
`unsigned char`		8	0 to 255
`short int`	A reduced precision version of `int`, requiring less storage.	16	-32,768 to 32,767
`unsigned short int`		16	0 to 65,535
`int`	The basic integer data type.	32	-2,147,483,648 to 2,147,483,647
`unsigned int`		32	0 to 4,294,967,295
`long int`	In many cases the `long int` type will be the same length as `int`, i.e. 32 bits.	32	-2,147,483,648 to 2,147,483,647
`unsigned long int`		32	0 to 4,294,967,295
`long long int`	An extended precision integer type.	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long int`		64	0 to 18,446,744,073,709,551,615
`float`	Single precision floating point (IEEE 754)	32	$-3.403e^{+38}$ to $3.403e^{+38}$
`double`	Double precision floating point (IEEE 754)	64	$-1.798e^{+308}$ to $1.798e^{+308}$

a. The numbers of bits for the given types are not fixed according the C language definition, but depend on implementation. A set of representative values are shown here.
b. The ranges given are based on the above representative lengths of each type in bits.

There are also some further types of interest, which are briefly summarised below:

- A boolean type, `bool`, is available on inclusion of the header file 'stdbool.h', with the standard values of {true, false}.
- Support for complex numbers is provided via the 'complex.h' library header file. This relates to the floating point types.
- An extended precision floating point type is defined, `long double`, although it may in practice have the same format as the `double` type.

As may be noted from Table 15.1, the native C / C++ data types lie along 8-bit boundaries (8-bit, 16-bit, 32-bit and 64-bit), which arises from the fact that software code is usually targeted at processors of these dimensions. However, such restrictions are not ideal from the perspective of generating efficient hardware architectures.

For optimal hardware implementations, no more bits are used than are necessary, due to the extra implied hardware cost. Arbitrary wordlengths are needed in order to realise circuits requiring arbitrary levels of precision. In fact, the effect of restricting wordlengths to those lying on 8-bit boundaries may be exacerbated when targeting certain dedicated resources of the PL. For example, the multiplication of two 18-bit numbers, *A* and *B*, to give the 36-bit result, *S*, would require *A* and *B* to be represented using 32-bits, and *S* using 64-bits. This would lead to an inefficient multiplier implementation, using four DSP48x slices instead of one — a 300% overhead! (If necessary, refer back to Section 2.2.2 on page 25 for a review of the DSP48x architecture). Furthermore, any registers or other operations on *A*, *B*, and *S*, would be over-sized.

Clearly, then, there is motivation to support arbitrary precision data types in a similar manner to other design entry methods such as HDL and System Generator. Vivado HLS therefore provides specific support for arbitrary precision C / C++ data types. Additionally, SystemC features its own arbitrary precision data types as an integral part of the language, and these are fully supported by Vivado HLS.

15.3.2. Vivado HLS Arbitrary Precision Data Types for C and C++

Having established the need for arbitrary precision arithmetic, i.e. to enable efficient hardware implementations, the direct outcome is an arbitrary precision integer type. However, this does not fully satisfy the requirements of most hardware designers, who normally prefer fixed point arithmetic for certain applications. Therefore, Vivado HLS also provides an arbitrary precision fixed point type, for use in C++ only.

Arbitrary Precision Integer Types

Support for arbitrary precision integer types is achieved using different types and associated libraries for each of the C and C++ input languages, as detailed in Table 15.2. In both cases, wordlengths between 1 bit and 1024 bits are permitted, i.e. $1 \le N \le 1024$.

Table 15.2: Arbitrary precision integer data types for use in C and C++ Vivado HLS designs

Language	Integer Data Type	Description	Required Header
C	int*N* (e.g. `int7`)	signed integer of *N* bits precision	`#include “ap_cint.h”`
	uint*N* (e.g. `uint7`)	unsigned integer of *N* bits precision	
C++	ap_int<*N*> (e.g. `ap_int<7>`)	signed integer of *N* bits precision	`#include “ap_int.h”`
	ap_uint<*N*> (e.g. `ap_uint<7>`)	unsigned integer of *N* bits precision	

Note that a different compiler (*apcc* as opposed to *gcc*) must be used when working with arbitrary precision integer types in C, as detailed in [18]. This does not apply to C++.

Figure 15.3 shows equivalent snippets of code demonstrating the use of arbitrary precision integer types in C and C++; notice the slightly different syntax used.

```
// C code example
#include “ap_cint.h”

void top_level_function (..)
{
      // declarations
        int6 small_signed;
        uint10 big_unsigned;
        int22 vbig_signed;
        ...
}
```

```
// C++ code example
#include “ap_int.h”

void top_level_function (..)
{
      // declarations
        ap_int<6> small_signed;
        ap_uint<10> big_unsigned;
        ap_int<22> vbig_signed;
        ...
}
```

Figure 15.3: The use of arbitrary precision integer types in C (left) and C++ (right)

Arbitrary Precision Fixed Point Types

Fixed point arithmetic has the generic word format shown in Figure 15.4, with a specified numbers of integer bits to the left of the binary point, and fractional bits to the right of the binary point. As for integer binary numbers, the MSB has a positive weighting for unsigned numbers, and a negative weighting for signed numbers.

For consistency with Xilinx Vivado HLS documentation [17], [18], in our discussions the overall wordlength is denoted as *W*, the number of integer bits as *I*, and the number of fractional bits as *B*, i.e. $W = I+B$. In the example of Figure 15.4, $I = 5$, $B = 7$, and $W = 12$.

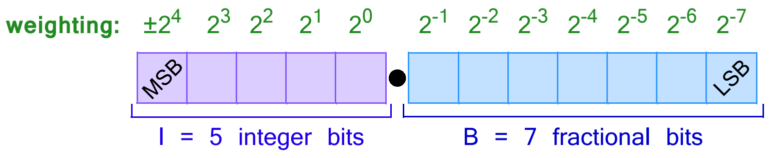


Figure 15.4: An example of the fixed point word format

Vivado HLS fixed point formats for C++ are defined as in Table 15.3; note that the C language is not supported. Here, *W* and *I* are as defined above; *Q* is a string specifying the quantisation mode; *O* gives the overflow mode; and *N* specifies the number of saturation bits in overflow wrap modes (i.e. the *N* most significant bits are set to 1). Details of these options can be found in [18]. The latter three are optional arguments; if unspecified, the quantisation mode, *Q*, defaults to **truncation to zero**, and the overflow mode, *O*, defaults to **wraparound**.

Table 15.3: Arbitrary precision fixed point data types for Vivado HLS designs

Language	Fixed Point Data Type	Description	Required Header
C++	ap_fixed<*W*,*I*,*Q*,*O*,*N*>	Signed fixed point number of *I* integer bits and *W-I* fractional bits.	`#include “ap_fixed.h”`
	ap_ufixed<*W*,*I*,*Q*,*O*,*N*>	Unsigned fixed point number of *I* integer bits and *W-I* fractional bits.	

In a similar manner to the previous section on integer data types, it is useful to confirm the syntax for declaring variables using the general type definitions given in Table 15.3. Figure 15.5 provides a code example in which a selection of variables are created, each with different integer and fractional wordlengths. The use of quantisation and overflow modes is also demonstrated.

Note that a set of strings are defined for *Q* (the quantisation mode) and *O* (the overflow mode), as listed in Table 15.4 [18], where the defaults are given in bold type.

Table 15.4: Quantisation and overflow modes for the C++ ap_fixed and ap_ufixed types (defaults in bold type)

Parameter	String	Description
Q (quantisation)	AP_RND	Rounding to positive infinity
	AP_RND_ZERO	Rounding to zero
	AP_RND_MIN_INF	Rounding to negative infinity
	AP_RND_INF	Rounding to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to negative infinity
	AP_TRN_ZERO	**Truncation to zero**
O (overflow)	AP_SAT	Saturation
	AP_SAT_ZERO	Saturation to zero
	AP_SAT_SYM	Symmetrical saturation
	AP_WRAP	**Wraparound**
	AP_WRAP_SM	Sign magnitude wrap around

15.3.3. Arbitrary Precision Types for SystemC

As mentioned earlier, SystemC features in-built support for integer and fixed point types. Use of these types follows a very similar style to C++, as summarised by Table 15.5. Note, however, that SystemC provides two different data types to cater for small (up to 64 bit) and large (up to 512 bit) integer wordlengths, whereas C and C++ use a single data type for up to 1024-bit words.

Code examples for SystemC, and an equivalent set of mode specifiers to Table 15.4, can additionally be found in [18].

```
// C++ code example
#include “ap_fixed.h”

void top_level_function (..)
{
     // declarations
       ap_ufixed<8,3> small_unsigned; // 3 int, 5 fract, defaults
       ap_fixed<10,4,AP_RND> big_signed; // round to + inf.
       ap_ufixed<10,4,AP_RND_ZERO> big_unsigned; // round to zero
       ap_fixed<21,10,AP_TRN,AP_SAT> vbig_signed; // trunc., satur.
       ap_ufixed<21,10,AP_RND_CONV> vbig_unsigned; // conv. round.
       ...
}
```

Figure 15.5: Example C++ code showing declaration of fixed point variables

Table 15.5: Summary of SystemC data types

SystemC Data Type	Description	Required Preamble
sc_int<*W*> sc_bigint<*W*>	signed integer: (up to 64 bits) (up to 512 bits)	`#include “systemc.h”`
sc_uint<*W*> sc_ubigint<*W*>	unsigned integer: (up to 64 bits) (up to 512 bits)	

sc_fixed<*W,I,Q,O,N*>	signed fixed point	`#define SC_INCLUDE_FX`
		`[#define SC_FX_EXCLUDE_OTHER]`
		`#include “systemc.h”`

sc_ufixed<*W,I,Q,O,N*>	unsigned fixed point	

15.3.4. Floating Point Data Types and Operators

Vivado HLS supports the use of floating point data types and operations, in so far as these map to available cores from the Xilinx technology library. For instance, standard arithmetic operations such as addition, subtraction, multiplication and division all have corresponding Xilinx cores, and these can be inferred by Vivado HLS. Further mathematical functions can be also be used on inclusion of the appropriate header files [18]. However, not all floating point operations are supported by HLS, and code should be written with these restrictions in mind.

To provide an example of compatible floating point operations for HLS, the code provided in Figure 15.6, which represents multiplication and addition operations on (single precision) floating point variables, would be successfully synthesised by Vivado HLS using instances of the floating point adder and multiplier cores. This can be confirmed by inspecting the reports produced by Vivado HLS.

15.3.5. Validation of Arbitrary Precision Models

It is possible to compare and validate a function based on arbitrary precision arithmetic against the equivalent function implemented using the native C / C++ data types, the latter usually being the starting point when developing a Vivado HLS design. This is a rapid and useful method of tuning the arithmetic wordlengths, i.e. achieving a level of arithmetic precision that is sufficiently accurate for the purpose, but which does not imply superfluous hardware resources.

Models can be developed with two type specifications for a given variable (or variables): (i) the original, with which the function was initially validated; and (ii) a reduced precision version, i.e. the candidate for synthesis. Only one of these definitions is adopted within a simulation at any given time, with the switch being made via a C macro [18]. This means

```
// C++ code example
void floating_arith (float *s, float a, float b, float d)
{
  float ab;

  ab = a * b;
  *s = ab + d;
}
```

Figure 15.6: Example code demonstrating the use of the 'float' type

that functional simulations can be conveniently executed for each of the variable type definitions, and the results quickly compared, while retaining all parameters from the original reference design, rather than fundamentally removing or changing them.

If the results indicate that the reduced precision version produces the required results, then naturally it is preferred for synthesis due to the reduced hardware cost implied.

15.4. Interface Specification and Synthesis

As mentioned in Section 14.4.1, the mechanism for specifying the interface of an HLS function differs according to the HLS input language: C and C++ support synthesis of the interface from a high-level description, whereas SystemC requires explicit, manual specification (with exceptions). Here we will focus our discussion on synthesis from C and C++.

In Vivado HLS, the input arguments and return value of the designed top-level C/C++ function are synthesised into RTL data ports, each with an associated protocol. Together, the ports and protocol form a port interface. Port interfaces are used to communicate with other subsystems and, if applicable, the processor in the system. In addition to the port interfaces inferred from the C function arguments, block-level protocols and associated ports are used to coordinate the exchanges of data between subsystems.

Over the next few pages, we will first consider the top-level C/C++ function definition from which interfaces are inferred. The synthesis of ports and protocols from this definition will then be explained, the functionality of block-level protocols will be introduced, and finally, the use of directives to influence the interface synthesis process will be reviewed.

15.4.1. C/C++ Function Definition

The functional part of a Vivado HLS design is a C/C++ function, which may contain other sub-functions in a hierarchical manner. The top-level function, i.e. the highest level of hierarchy, forms the basis of the interface synthesis process.

To provide an example, the code listing in Figure 15.7 represents a simple C design with the top level function, `find_average_of_best_X()`. The detailed internal workings of the function are not of consequence, although the reading/writing operations on each argument determines the direction of the synthesised ports, as will be covered in Section 15.4.2.

```
void find_average_of_best_X (int *average, int samples[8], int X)

{
     // body of function (statements, sub-function calls, etc.)

}
```

Figure 15.7: An example top level function for HLS

The function definition contains three arguments, and it should be interpreted that the 8-element array `samples` is an input, as is the integer `X`, while `average` is an output of the function. At a simple level, these three function parameters are therefore converted by HLS into two input interfaces, and one output interface, as shown in Figure 15.8.

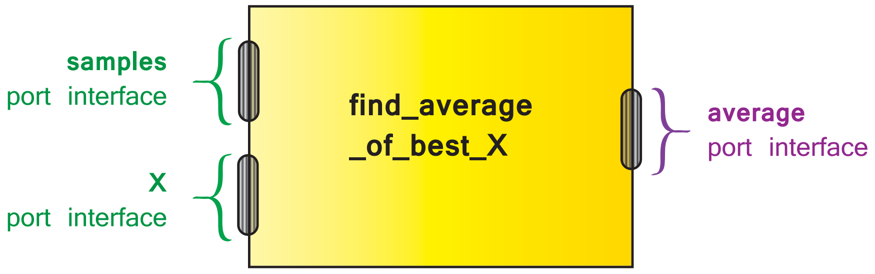


Figure 15.8: Simplified interface diagram for the example function 'find_average_of_best_X()'

Notably, these interfaces may comprise control inputs or outputs in addition to the data port itself, depending on the protocol adopted. We will return to, and update, this interface diagram later in the chapter, after introducing the subject of port protocols.

15.4.2. Synthesis of Port-Level Interfaces

Following our intuitive example of interface synthesis in Section 15.4.1, it is useful to present more formally the conventions adopted when synthesising an RTL port-level interface from C/C++ code. (Note that block-level interfaces will be covered in Section 15.4.5, and these are distinct from port-level interfaces.)

The RTL-level description of a port includes the following aspects:

- The name of the port;
- Its direction (*input*, *output* or *inout*);
- The data type and dimension.

Therefore, when designing using Vivado HLS, all of these properties must be synthesised from the high-level C/C++ code. We will consider each of them during the remainder of this section.

Port Name

The name of the port is drawn from the name of the corresponding function parameter. For example, in the function of Figure 15.7, 'samples' is an array type input parameter of the function, and therefore the name 'samples' would also be used for the array data port. An exception is any port synthesised from a function return statement, which is given the name 'ap_return' (and there is no return statement present in our working example).

In some cases, additional control signals and associated ports are synthesised along with a data port, and this depends on the protocol adopted. We will cover protocols in more detail in Section 15.4.3, but for now, it is useful to note that any control signals associated with a synthesised data port will adopt the same name, with the relevant suffixes for the control types.

Port Direction

The interpretation of port direction follows a set of rules as summarised in Table 15.6. For example, an argument of the C/C++ function that is only ever read from by the function, and never written to, will be synthesised into an RTL *input* port. Likewise, an argument that is always written to, and never read from, will be translated into an *output* port.

Table 15.6: Synthesis of port directions

C/C++ Function Argument	RTL Port Type
An argument which is read from and never written to	in
An argument which is written to and never read from	out
A value output by the function return statement	out
An argument which is both written to and read from	inout (bidirectional)

Data Type and Dimension

The data types and dimensions of ports synthesised from C/C++ function arguments follow the same conventions as for data types in general, as reviewed in Section 15.3. Some interface protocols (to be reviewed in the next section) require additional control ports to be generated, and these are generally 1-bit signals, with some exceptions.

15.4.3. Port Interface Protocol Types

In addition to the port itself, an associated protocol defines the style of interchanges taking place via that port. Vivado HLS specifies a set of available protocols, ranging in sophistication from 'none' (i.e. no explicit protocol) to 'hs' (a hand shaking protocol), 'ack' (an acknowledgement protocol), and even AXI protocols. A full list of the available protocols is included below, where the names used in the Vivado HLS tool are given, along with a brief explanation of each [18].

- ***ap_none*** — This is the simplest protocol type, with no explicit interface protocol, no additional control signals, and no associated hardware overhead. However, there is an implication that timing of input and output operations is independently and correctly handled.

- ***ap_stable*** — This is a similar protocol to *ap_none*, in that it does not involve additional control signals or related hardware. The difference is that *ap_stable* is intended for inputs (only) that change infrequently, i.e. that are generally stable apart from at reset, such as configuration data. The inputs are not constants, but neither do they require to be registered.

- ***ap_ack*** — This protocol behaves differently for input and output ports. For inputs, an output acknowledge port is added, and held high on the same clock cycle as the input is read. For outputs ports, an input acknowledge port is added. After every write to the output port, the design must wait for the input acknowledge to be asserted before it may resume operation.

- ***ap_vld*** — An additional port is provided to validate data. For input ports, a *valid* input control port is added, which qualifies input data as valid. For output ports, a *valid* output port is added, and asserted on clock cycles when output data is valid.

- ***ap_ovld*** — This protocol is the same as *ap_vld*, but can only be implemented on output ports, or the output portion of an inout (bidirectional) port.

- ***ap_hs*** — The _hs suffix of this protocol stands for 'handshaking', and it is a superset of *ap_ack*, *ap_vld*, and *ap_ovld*. The *ap_hs* protocol can be used for both input and output ports, and facilitates a two-way handshaking process between the producer and consumer of data, including both validation and acknowledgement transactions. As such, it requires two control ports and associated overhead. It is, however, a robust method of passing data, with no need to ensure timing externally.

- ***ap_memory*** — This memory-based protocol supports random access transactions with a memory, and can be used for both input, output, and bidirectional ports. The only argument type compatible with this protocol is the array type, which corresponds with the structure of a memory. The *ap_memory* protocol requires control signals for clock and write enables, as well as an address port.

- ***bram*** — The same as *ap_memory*, except that when bundled using IP Integrator, the ports are not shown as individual ports, but grouped together into a single port.

- ***ap_fifo*** — The FIFO protocol is also compatible with array arguments, provided that they are accessed sequentially rather than in random order. It does not require any address information to be generated, and therefore is simpler in implementation than the *ap_memory* interface. The *ap_fifo* protocol can be used for input and output ports, but not bidirectional ports. The associated control ports indicate the fullness or emptiness of the FIFO, depending on the port direction, and ensure that processing is stalled to prevent overrun or underrun.

- ***ap_bus*** — The *ap_bus* protocol is a generic bus interface that is not tied to a specific bus standard, and may be used to communicate with a bus bridge, which can then arbitrate with a system bus. The *ap_bus* protocol supports single read operations, single write operations, and burst transfers, and these are coordinated using a set of control signals. In addition to this generic bus interface, specific support for AXI bus interfaces can be integrated at a later stage, using an interface synthesis directive.

- ***axis*** — This specifies the interface as AXI stream.

- ***s_axilite*** — This specifies the interface as AXI Slave Lite.

- ***m_axi*** — This specifies the interface the AXI Master protocol.

It is beyond the scope of this chapter to explain in detail the mechanics of each of these protocols; however, extensive further information can be found in [18], including detailed timing diagrams, and there are also a number of relevant practical examples in [17].

15.4.4. Synthesis of Port Interface Protocols

Having defined the available protocols, we will now concentrate on the synthesis of specific protocols from the high-level description, and related restrictions.

The designer has the option to choose a protocol for each individual port by supplying an appropriate directive. The set of supported protocols is restricted according to (i) the implied port direction, and (ii) the type of the C/C++ function argument, as shown in Table 15.7 [18]. If the protocol is not explicitly defined via a directive, or if an unsupported protocol is mistakenly selected, then Vivado HLS will apply the default protocol. The defaults are also a function of (i) and (ii) above, as indicated in Table 15.7.

Table 15.7: Protocol synthesis: supported types and defaults (S = supported; D = default) [18]

Argument Type	**Variable**			**Pointer Variable**			**Array**			**Reference Variable**		
	pass-by-value			**pass-by-reference**			**pass-by-reference**			**pass-by-reference**		
Interface Type[a]	**I**	**IO**	**O**	**I**	**IO**	**O**	**I**	**IO**	**O**	**I**	**IO**	**O**
ap_none	**D**	-	-	**D**	S	S	-	-	-	**D**	S	S
ap_stable	S	-	-	S	S	-	-	-	-	S	S	-
ap_ack	S	-	-	S	S	S	-	-	-	S	S	S
ap_vld	S	-	-	S	S	**D**	-	-	-	S	S	**D**
ap_ovld	-	-	-	-	**D**	S	-	-	-	-	**D**	S
ap_hs	S	-	-	S	S	S	S	-	S	S	S	S
ap_memory	-	-	-	-	-	-	**D**	**D**	**D**	-	-	-
bram	-	-	-	-	-	-	S	S	S	-	-	-
ap_fifo	-	-	-	S	-	S	S	-	S	S	-	S
ap_bus	-	-	-	S	S	S	S	S	S	S	S	S
axis	S	-	-	S	-	S	S	-	S	S	-	S
s_axilite	S	-	S	S	S	S	-	-	-	S	S	S
m_axi	-	-	-	S	S	S	S	S	S	S	S	S

a. Reading along the row: **I** = input port; **IO** = inout (bidirectional) port; **O** = output port.

Given the dependencies between protocol, port type and direction, it is important to consider the *type* of the C/C++ function arguments when developing the high-level C/C++ description. As given by the column headings in Table 15.7, values can be passed into and out of C/C++ functions using four different argument types, i.e. as: (i) variables; (ii) pointers; (iii) arrays; and (iv) references. Therefore, a particular argument type corresponds to a limited set of available protocols. For example, passing an array argument as an input restricts the set of available protocols to: *ap_hs*, *ap_memory*, *bram*, *ap_fifo*, *ap_bus*, *axis* and *m_axi*, with *ap_memory* as the default.

To link this to the practical issue of design entry, you may wish to refer back to the function defined in Figure 15.7, and confirm that function has three arguments:

- ***samples*** — an array input;
- ***X*** — in integer (scalar) input, passed by value;
- ***average*** — an output pointer variable.

Then, with reference to Table 15.7, it may be noted that the default protocol for ***samples*** is *ap_memory*, the default for ***X*** is *ap_none*, and the default for ***average*** is *ap_vld*. Taking into account the additional control ports required for these protocols, we can update the synthesised RTL interface originally given in Figure 15.8, to that shown in Figure 15.9. Notably the data ports are 32 bit, as a result of using the C `int` data type.

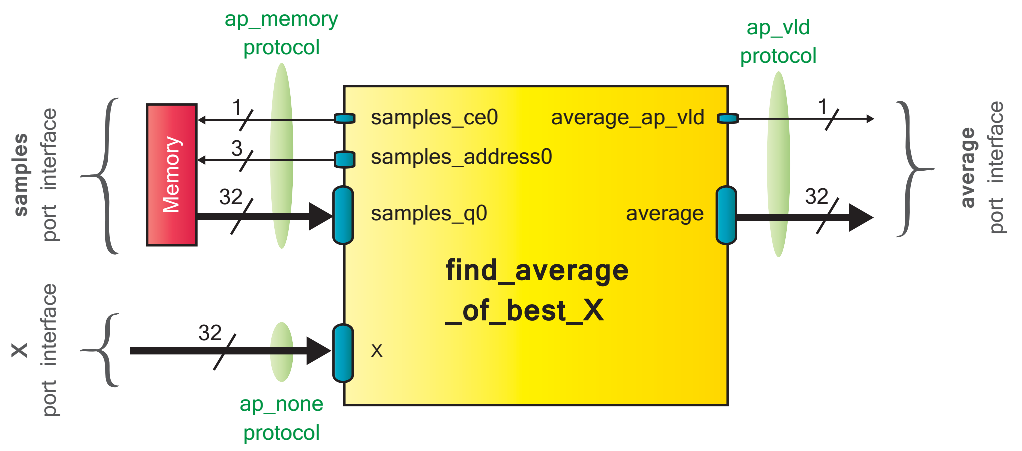


Figure 15.9: RTL interface diagram for the 'find_average_of_best_X()' function, showing default port level ports and protocols

Note also that this diagram shows ports and port-level protocols only — it does not include any block-level protocols, an aspect to be covered in the next section. The clock signal will be added later as part of the block-level protocol.

15.4.5. Block-Level Interface Ports and Protocols

In addition to the logical ports inferred from the arguments of the C/C++ function, and their associated protocols, it is also possible to add block-level protocols to the design (these may also be referred to as function-level protocols). This provides a mechanism to control the execution of the subsystem, and it is useful when integrating one or several Vivado HLS blocks into a system and managing the flow of data between them.

Before defining the protocols, it is worthwhile briefly confirming some terminology, which we will do with the aid of Figure 15.10. This diagram depicts five blocks in cascade, with the direction of data flow indicated; in practice there may also be FIFO buffers between blocks, which are not shown. (Please be aware that, in choosing this example, we cover one possible use model of Vivado HLS blocks — there is no intention to imply that a chain of blocks is the only, or indeed the typical, scenario.)

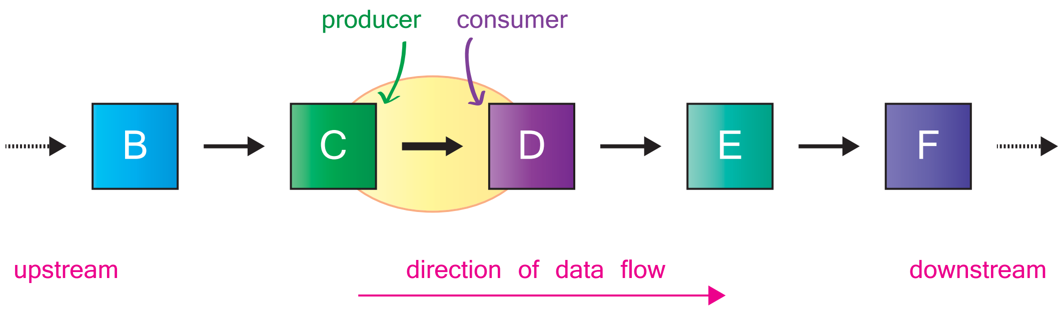


Figure 15.10: Data flow between Vivado HLS blocks

With reference to Block D, blocks to the left of this point (C, B...) would be referred to as *upstream*, while blocks to the right (E, F...) would be considered *downstream*. When considering the interface between any pair of blocks, the upstream block outputs data and passes it to the downstream block. For example, with respect to the interface between Blocks C and D, Block C is the *producer* of data, while Block D is the *consumer* of data.

In some cases it is desirable for a block to exert 'backpressure' on upstream blocks. In other words, the consumer block may wish to prevent the producer block from presenting more data until it is ready to receive it (if necessary, propagating as a ripple effect to further upstream blocks). This is one aspect of operation that can be implemented using block-level control.

There are three types of block-level protocol, and these are listed below according to the terms used in Vivado HLS:

- ***ap_ctrl_none*** — Choosing this option simply means that a block-level protocol is not added. Instead, control is exerted entirely at the port interface level, using port-level protocols.
- ***ap_ctrl_hs*** — A block-level control protocol with handshaking. An *ap_start* control input is asserted to prompt the block to begin operation, and the block produces three output control signals (*ap_ready*, *ap_idle*, and *ap_done*) to indicate its stage of operation. Specifically, the *ap_ready* signal indicates that the block is ready for new inputs, the *ap_idle* indicates when it is processing data, and *ap_done* is asserted when output data is available. To provide a usage example, the *ap_ctrl_hs* protocol is appropriate when a single HLS block is to be interfaced with the controlling processor.
- ***ap_ctrl_chain*** —This protocol is similar to *ap_ctrl_hs*, but has an additional input control signal, *ap_continue*, and is designed for chaining multiple Vivado HLS blocks together. The *ap_continue* input indicates the ability of the downstream block to accept new data, and therefore it can exert backpressure on upstream blocks if necessary. If *ap_continue* is de-asserted, the block will complete its current computation to the stage of presenting the results at the output, but will then stall until *ap_continue* is set high again.

If a block-level protocol is used, it operates independently of any port-level protocols that are adopted on individual ports. There are also two input signals applied to the block that are added regardless of the block-level protocol chosen: *ap_clk* and *ap_rst*. These are required because the internal operation of the block is synchronous, thus it requires a clock signal, and because the block must be capable of being externally reset.

As a general point, it is worth noting that an AXI4-Lite bus interface can be applied to the block-level interface protocol, thus enabling block-level control signals to be passed between the block and a controlling processor. In some circumstances, it is also possible to bundle the block-level control ports with the ports for the port-level interfaces, resulting in a consolidated AXI4-Lite interface [18].

To round off our discussion, let us reconsider our example function, augmented with the *ap_ctrl_hs* block-level control protocol (the default), as shown in Figure 15.11. In this case, six additional ports are added: *ap_clk* and *ap_reset* (as for all Vivado HLS designs); the *ap_start* control input; and the *ap_done*, *ap_ready*, and *ap_idle* control outputs. These new, block-level interface ports are shown towards the top of the diagram.

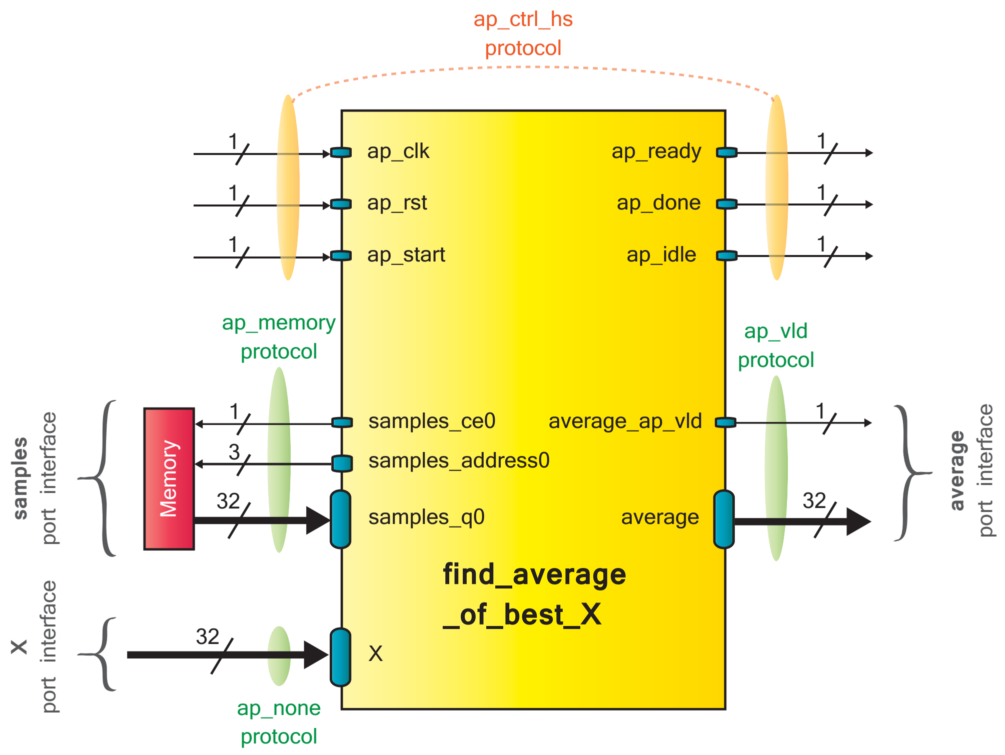


Figure 15.11: RTL interface diagram for the 'find_average_of_best_X()' function, showing both default port level ports and protocols, and default block level ports and protocol

15.4.6. Interface Synthesis Directives

Directives are the mechanism by which the designer can exert high-level control over the implementation of the designed C/C++/SystemC code. Of the HLS directives available, a subset relate specifically to interface synthesis, and these can be used for both port-level directives and block-level directives, reflecting the two types of interface as reviewed over the previous few pages. In addition to explicitly specifying protocols, other aspects of interfaces can also be specified, such as the form of array inputs, and the resources used to implement memories or FIFO buffers.

Directive Types

There are six types of directive which can be applied to both port-level and block-level interfaces, as reviewed below.

- ***Array Map*** — Combines several arrays to form one larger array, with the goal of using fewer FIFO or RAM resources and control ports to implement the interface.
- ***Array Partition*** — Separates array interfaces into several smaller sections, resulting in an expanded set of ports, control signals, and implementation resources, but with increased bandwidth.
- ***Array Reshape*** — In this case, an original array is partitioned into smaller arrays, which are then recombined to form an array with fewer elements, and wider data elements. This implies fewer memory locations and shorter addresses.
- ***Interface*** — This directive can be used to explicitly specify a port-level interface protocol as one of the available options (as listed in Table 15.7 on page 300), or the block-level protocol as *ap_none*, *ap_ctrl_block* or *ap_ctrl_chain*, as covered in Section 15.4.5.
- ***Resource*** — A particular resource can be chosen to implement the interface. For instance, a one or two port RAM can be specified for an *ap_memory* interface, or an *ap_fifo* interface can target a FIFO constructed from a Block RAM or LUTs.
- ***Stream*** — This directive specifies the interface as a streaming port, utilising FIFOs and permitting the depth of the FIFO to be explicitly chosen.

The specification of port-level interface type is made via the corresponding function argument, while the block-level interface is applied to the top-level function.

By applying the required directives, the design example shown in Figure 15.11 can be exported to an IP Integrator block as shown in Figure 15.12. The result of consolidating the port- and block-level interfaces into a single AXI4-Lite interface results is shown in Figure 15.13 (note that the array input `samples` is separate, as it requires a stream interface). This is typically done to enable control by software running on a processor or microcontroller.

Further Options

In addition to the directives detailed above, there is also the option to register individual ports, meaning that an additional clock cycle of delay will be added at the port interface. This brings a potential improvement to the timing performance of the design.

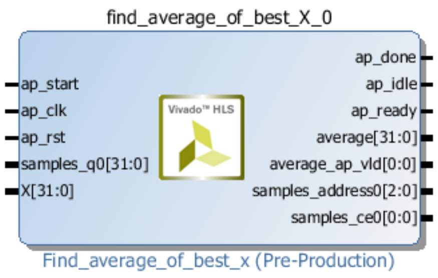


Figure 15.12: An IP Integrator block produced for the "find_average_of_best_X" function, with port- and block-level protocols as specified in Figure 15.11.

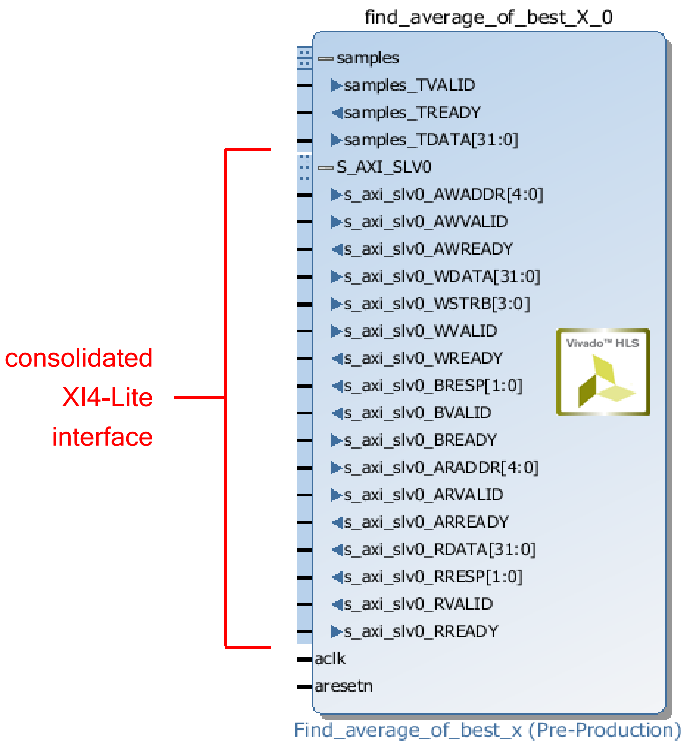


Figure 15.13: An IP Integrator block produced for the "find_average_of_best_X" function, with the block-level protocol ports, and some port-level protocol ports, combined into an AXI4-Lite interface.

Source File or Directive File?

Directives can be kept together in a separate file within the Vivado HLS project, or integrated into the source file as pragmas. An example showing the use of pragmas is given in Figure 15.14. These would be automatically inserted by Vivado HLS, directly into the top of the function body (as indicated in Figure 15.7 on page 296), thus there is no need to type these lines of code manually (although you can if you want to!). In both cases, it is useful to recognise how pragmas are formed.

In the example of Figure 15.14 there are three pragmas: the first two set interface directives, and the third sets a resource directive. Specifically, the first pragma specifies the *ap_memory* protocol for the `samples` input port, and likewise the second line sets the *ap_vld* protocol for the `X` input port. Notice also that a register is inserted on the `X` port. The third directive specifies that the `samples` input port is implemented using a specific hardware resource, in this case a single port ROM implemented with LUTs (i.e. using slice logic as opposed to Block RAM resources).

```
#pragma HLS INTERFACE ap_memory port=samples
#pragma HLS INTERFACE ap_vld register port=X
#pragma HLS RESOURCE variable=samples core=ROM_1P_LUTRAM
```

Figure 15.14: An example of pragmas inserted in the C/C++ source code for interface synthesis

As the interface is often a static aspect of the system, with the designer concentrating optimisation efforts on the functional implementation by creating different *solutions*, interface directives are commonly implemented as pragmas in the source file. This means that the interface settings can be easily ported between Vivado HLS solutions, without the need to re-specify them, or to take particular care when passing directives from one solution to the next. However, the designer may, for good reason, prefer to specify the directives via file, thus keeping the directives and source code separate — this is also a perfectly valid approach.

In a general sense, the insertion of pragmas makes code less portable, and so pragmas best suited to aspects that the designer intends to fix. For instance, it may be desirable to use pragmas to fix the configuration of the interfaces in the design.

Aspects that are changed between 'solutions' as the designer attempts to optimise his or her code, should be specified using a separate directives file, so as not to embed them within the code. Keeping the code and directives separate enables a greater degree of high-level design flexibility.

15.4.7. Manual Interface Specification

Up until this point, the discussion has focussed on interface synthesis from C and C++ functions, but recall that interfaces can also be explicitly specified. This is compulsory when coding in SystemC (with the exception of the *ap_bus* and *ap_memory* interface types, which can be synthesised), and optional when coding in C or C++.

Interface Specification in SystemC

In SystemC, any designed component is represented by a C++ class inherited from the base class, SC_MODULE. This is used to define the interface and functionality of the component [1].

Specification of interfaces in SystemC is similar to an HDL-level description, and includes an explicit specification of the type, direction, and dimension of each port. An example is provided in Figure 15.15 for a simple counter, and this shows the first section of the `my_counter` module (class) definition, where the ports are declared. You may notice the similarities between this code extract and a VHDL entity declaration.

As a result of all port information being manually specified, no synthesis of the interface is required in this case.

```
SC_MODULE (my_counter)  {

      // top level ports
       sc_in<bool> clk;
       sc_in<bool> ce;
       sc_in<bool> reset;
       sc_out<int> count;

      // the rest of the module body definition...
}
```

Figure 15.15: Manual definition of the ports of a counter design using SystemC

Interface Specification in C/C++

In some circumstances it may be desirable to define an interface which has a different set of control ports and associated protocol from those available in Vivado HLS. This may be accomplished in C or C++ by adding an extra block of code to specify the control signals and the required series of transactions on these control ports (recall that these would normally *not* be user-specified, but rather the user would specify data ports, with control ports being included by the HLS process). In a Vivado HLS pragma implementation, the ordering of transactions is specified with the aid of an `ap_wait()` function, which instructs Vivado HLS to insert clock cycles between IO operations within a specifically labelled section of code (a 'protocol region').

A code example demonstrating this manual interface specification technique can be found in [18].

15.5. Algorithm Synthesis

The theme of this section is the synthesis of functional hardware from designed C/C++/SystemC code. The limited scope of this chapter precludes a detailed treatment of all aspects, and rather we focus on a few topics that are of key interest, and representative of the design methods necessary to successfully develop systems using Vivado HLS. In other words, the content that follows in this section should be considered a 'taster' of the processes and possibilities of HLS algorithm synthesis, and not a comprehensive guide. Extensive further information can be found in the Xilinx User Guide 902, "*Vivado Design Suite User Guide: High-Level Synthesis*" [18].

One particular aim of this section is to highlight the control that the designer can exert over the eventual, synthesised hardware implementation through the use of directives, and the opportunities for readily generating and comparing alternatives without the need to significantly change the source code. We accomplish this through a feature case study on the synthesis of loops.

The remainder of this chapter will consider algorithm synthesis from select different styles of C designs, with illustrative examples. However, before doing so, it is important to recognise the metrics used to evaluate and constrain Vivado HLS designs in general.

15.5.1. Implementation Metrics and Constraints

As HLS translates the algorithm expressed by a C/C++/SystemC function into hardware, quantities are required to measure the characteristics of the implementation. This is especially useful when comparing implementation variations, i.e. the set of Vivado HLS 'solutions' generated by applying different directives to the HLS process.

Limits based on implementation metrics can be input by the user, and act as design constraints. These influence the behaviour of Vivado HLS in executing high-level synthesis: the tool attempts to meet the targets set where possible and, if it cannot meet them, it instead produces a 'best effort' RTL design. The rest of this section is devoted to defining the various metrics that may be used to both benchmark and constrain an HLS design.

Area / Resources

The most intuitive measure of an implementation is the hardware cost of building the circuit, in terms of the resources (or equivalently 'area') on the FPGA or PL. With resources inherently fixed for a particular target device, there may be motivation to minimise the cost of a particular Vivado HLS component, depending on the requirements and circumstances of the system as a whole.

By default, Vivado HLS seeks to minimise area, which implies time-sharing of hardware. This generally leads to increased *latency* and reduced *throughput* (both defined below).

Clock Period, Clock Rate, and Clock Uncertainty

The clock period metric expresses the minimum period and inversely the maximum clock frequency that can be supported by a design. It is a function of the physical characteristics of the target device, as well as the *critical path* of the RTL design synthesised from HLS. The critical path is defined as "*the longest combinatorial logic path between two clocked elements*", and it directly limits the maximum clock frequency. Critical path is normally managed in hardware designs via pipelining techniques (i.e. the strategic insertion of registers) and, in similar fashion, can be influenced by the use of pipelining in Vivado HLS.

Vivado HLS prompts the user to specify a target ***Clock Period***, together with a ***Clock Uncertainty***, and together these act as timing constraints. Where possible, the tool creates a design to meet the target minus the uncertainty value, based on Xilinx technology library information. The uncertainty figure is included to cover other factors not known at the HLS stage, in particular RTL synthesis, placement, and routing delays [18].

Latency

In Vivado HLS, the term 'latency' adopts its usual definition as the number of clock cycles between applying an input, and achieving the corresponding output. Latency can be examined at different levels of hierarchy, from the top-level function down to sub-functions, loops or specific sections of code. In the context of loops, latency refers to the completion of all iterations of the loop; the term *iteration latency* is used when referring to a single iteration. The total latency is equal to the iteration latency, multiplied by the number of iterations of the loop (the 'trip count').

Latency can also be specified as a design constraint, i.e. the maximum acceptable latency is defined by the user, and the Vivado HLS tool optimises the design (where possible) to meet the requirement.

Initiation Interval and Throughput

The Iteration Interval (II) is the number of clock cycles that separate the acceptance of inputs to the Vivado HLS design. Without applying directives, the initiation interval and latency may be the same, because the default behaviour of Vivado HLS is to optimise for area, resulting in a serial design. However, the strategic use of pipelining can reduce the iteration interval to much less than the latency of the design. On the other hand, this may increase the area of the design, so there is a trade-off involved.

Initiation interval corresponds directly to throughput, and is analogous to the relationship between clock period and clock frequency. Throughput expresses the rate at which data can be passed through the system. The best possible initiation interval is 1, meaning that new input samples can be accepted on every clock cycle, in which case the throughput is equivalent to the clock rate. Higher levels of throughput can sometimes be achieved through use of partial loop unrolling, or by replicating a synthesised function.

15.5.2. Data Types

As for interface synthesis, the choice of data types in a high-level algorithm has a fundamental impact on the synthesised hardware. The use of wordlengths that are longer than required can lead to a needlessly expensive implementation, in terms of the amount of PL resources required to create the design. It also has the potential to impact on other implementation metrics, such as latency, maximum clock frequency, and initiation interval.

The use of arbitrary precision data types, as reviewed in Section 15.3, is an effective mechanism for specifying appropriate wordlengths and hence contributing to an efficient overall implementation.

15.5.3. Pipelining

The term 'pipelining' is widely used in the context of hardware design to refer to the insertion of registers into a circuit, usually for the purpose of minimising the critical path (i.e. the longest combinatorial logic path between clocked elements), and hence maximising the achievable clock frequency. Consequently, we can think of data samples moving along the processing path in a regular, synchronous fashion, with storage of intermediate samples in the pipeline registers. Alternatively, it could be said that the samples are "in a pipeline".

When the term is used to refer to processor operation, it means that a task is broken down into a regular structure of subtasks that can be performed simultaneously by different 'pipeline stages' of the processor. The number of pipeline stages, and the operations performed by each of them, are fixed features of the processor architecture. For instance, a 5-stage pipeline might allow the processor to simultaneously: (i) read an instruction; (ii) decode an instruction; (iii) read data; (iv) execute; and (v) write a result, all on the same clock cycle, using a dedicated pipeline stage for each. Individual pipeline stages process consecutive samples, meaning that a new output can be computed on each clock cycle.

In HLS, pipelining has a meaning which is related to both of the above, but distinct. In particular, we can define the concept of pipelining as referring to the partitioning into substages of an arbitrary set of dependent operations, whether these are combinatorial or clocked.

To define HLS pipelining further, we must abstract the detail of low-level operations and consider that pipelining relates to the segmentation of *logical processing stages*. Unlike the hardware case, these are not necessarily physical, combinatorial processing elements (although they could be). Pipelining is incorporated to permit the overlapping of operations, similar to the concept of pipelining in processors. However, unlike processors with fixed architectures, there are no restrictions on the nature of the operations themselves, because the FPGA fabric or Zynq PL provides a blank canvas with which to implement any arbitrary functionality.

The motivation for pipelining is to provide an opportunity for parallel processing, and thus to increase the throughput supported by the design. Pipelining can be applied as a directive in Vivado HLS, at the level of functions and loops. Over the next few pages, we will consider the pipelining of operations, while the pipelining of loops will be covered in Section 15.5.5.

Algorithm Execution and Data Dependency

It may be assumed that any algorithm expressed in software includes a set of functional steps, or operations. Each step depends on a certain set of data samples being available as inputs, and in some cases these data samples may not be available until a previous step has been completed. Consequently, there is an implied data dependency between the steps forming the algorithm, and a required order of execution.

A direct synthesis of the algorithm may therefore lead to a set of operations which must logically occur at the same time, due to the data dependency between them. In other words, all operations belong to the same processing *stage*, which must be entirely completed before new inputs can be processed. Whether the execution of the stage corresponds to a single clock cycle or multiple clock cycles, the important point is that there is no opportunity to begin computing the next output until the previous one has been completed.

To give a simple example, consider a function given in Figure 15.16, which comprises three processing steps, Op1, Op2, and Op3. The second step, Op2, relies on the outputs of the first step, Op1, while Op3 depends on the outputs from Op2. As such, and with no memory to store intermediate results, there is a dependency between the operations: the final output is generated only after all operations have been completed (in order). Therefore we can designate the group of Op1, Op2, and Op3 as a processing *stage*.

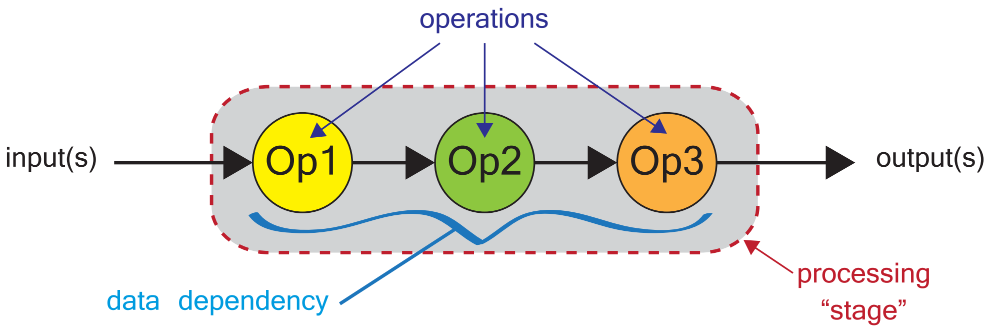


Figure 15.16: Example function with data dependencies between operations

In terms of processing data, the data dependency within the stage means that only one sample output can be processed at a time; the three operations must run sequentially on each set of input samples, and the next set of inputs cannot be accepted until the current output is ready. Thus the *data sample period* is equivalent to the total processing time of Op1, Op2, and Op3, and this directly governs the data throughput supported by the design. The other important metric, input-to-output *latency*, is equivalent to the individual latencies of the three operations. Both of these are indicated on the waveforms shown in Figure 15.17.

It may be noted that a clock signal is not drawn in Figure 15.17, and this is deliberate. At an abstract level, Op1, Op2, and Op3 may represent logic in a combinatorial path *or* a set of clocked operations. In fact, as pipelining has the effect of separating the stages, it will result in increased throughput irrespective of whether the circuit is clocked or combinatorial.

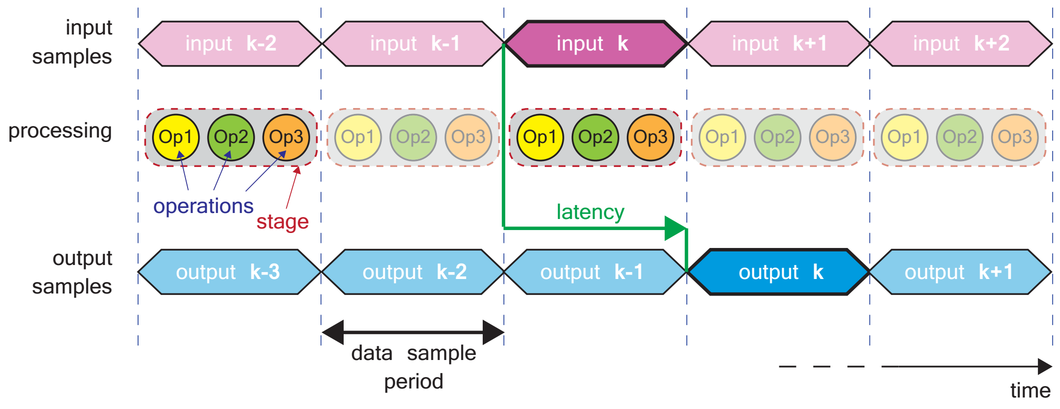


Figure 15.17: Waveform showing throughput and latency, before pipelining

Pipelining an Algorithm

Pipelining means that the processing stage is split up into smaller stages that can each process different data samples simultaneously. In other words, the data dependencies that caused a set of operations to be grouped together are broken up, and this allows the operators to execute in parallel.

In hardware terms, the method used to achieve this separation of stages is to insert registers between the new, smaller stages, which allows data samples to be held in memory. A direct consequence is that the overall latency from input to output increases with respect to the sample period (a disadvantage), however the sample period can be reduced because the stages are shorter. The latter has a direct impact on throughput, which is normally considered the more important performance metric. A further benefit is that, where the original stages represent a combinatorial logic path, the insertion of pipeline registers may also cause an increase in the maximum supported clock frequency.

From a hardware-oriented perspective, the pipelined function is equivalent to the signal flow graph shown in Figure 15.18. As before, the final outputs depend on signals that have rippled through Op1, Op2 and Op3, but now there is memory to hold the intermediate results. The outcome is that the three operators are now free to operate simultaneously, each acting upon consecutive input data samples as shown in Figure 15.19. This removes the data dependency between operations, and they can now be considered as each representing a separate processing stage.

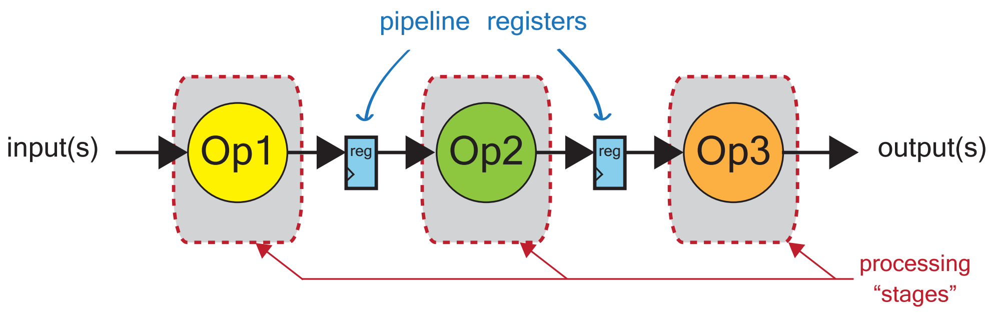


Figure 15.18: Partitioning of operations into separate stages via pipelining

The updated waveform diagram reflects the smaller stages (each effectively a single operation) and confirms a corresponding shortening of the data sample period. This means that the throughput has increased by a factor of three. It is also apparent that the operators can simultaneously compute results for consecutive samples; for instance, Op1 can process sample *k*+1 as soon as it has finished with sample *k*, and passed it to Op2. The latency is now greater with respect to the duration of a single stage, as the inserted pipeline registers add to the delay between input and output, but this is an acceptable overhead.

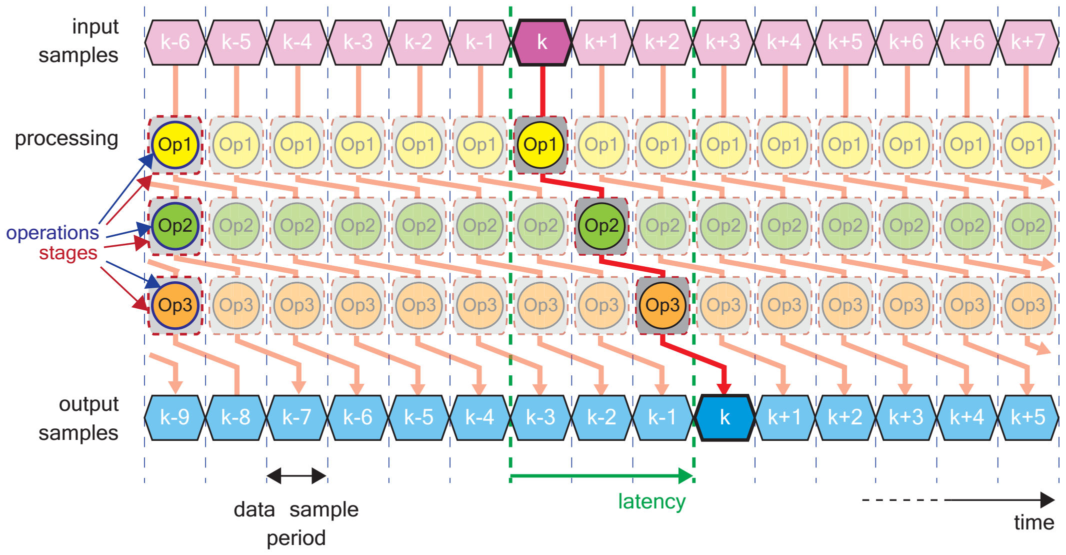


Figure 15.19: Waveform showing throughput and latency, after pipelining

15.5.4. Dataflow

As discussed in the previous section, pipelining is a method of increasing the concurrency of the hardware produced from a software description, and thus improving the throughput. Pipelining can be performed at the level of operations within a function, or on loops.

Dataflow pipelining (or simply 'dataflow') optimisation is a similar concept, but is actually applied at a higher level in the design hierarchy: it acts to improve the concurrency between functions. For example, a top-level design may include four functions, F1, F2, F3 and F4, with data dependencies between them. A direct synthesis of the top-level function would result in the sub-functions executing in a sequential, non-overlapping manner, analogous to the grouping together of operations depicted in Figure 15.16 (which is labelled a processing *stage*). With dataflow pipelining, the equivalent optimisation to pipelining is performed, in the sense that registers are inserted between functions to partition them into separate stages that can execute concurrently.

Given that functions are more complex than simple operations, (which was discussed in the context of pipelining), dataflow optimisation can actually go a step further, by analysing the content of functions and the dependencies between them. This might allow a further degree of function 'overlapping', with the effect of shortening overall execution latency as well as increasing throughput. For instance, it might be possible to start function F2 once F1 is 50% complete, rather than waiting until it has entirely finished. This is most easily illustrated by example, and here we will attempt to relate dataflow optimisation it to an everyday scenario.

Suppose that you went to a coffee shop and it was manned by a single person, whom we shall name Penelope. Penelope would have to perform multiple functions to serve you. She would: (i) greet you and take your order; (ii) assemble your requested food; (iii) make your coffee; and (iv) take payment. Penelope would only be able to serve one person at a time, by performing all of the above tasks in order: in other words, the latency (i.e. the total time to serve a customer) and the throughput (the rate of serving customers) would be limited.

For the purposes of illustration, we will assume that the following functions are defined:

• Function F1:	take order	3 time units
• Function F2:	prepare food	2 time units
• Function F3:	prepare coffee	4 time units
• Function F4:	accept payment	3 time units

Therefore, it will take Penelope a total of 12 time units to serve each customer. She can only serve one customer at once, so the customer throughput is 1 every 12 time units, and each customer must wait for 12 time units to complete their purchase.

Matters could be improved by adding more staff. Suppose that Penelope now has the assistance of Cameron, Hamish, and Isla. Cameron will greet customers and take their orders, Hamish will assemble food orders, Isla will take payments, and Penelope will make the coffees. As well as increasing customer throughput, latency can be reduced, because knowledge of the operations that comprise the functions allows them to be overlapped. For example, Cameron can ask for drinks orders first (to enable Penelope to start serving them) and then food orders (to enable Hamish to start assembling them), such that Penelope's function can begin before Cameron's has completed. Likewise, Isla can commence her function of taking payment at the till as soon as the complete order is known, and while Penelope and Hamish are still completing their serving functions.

These two variations on the coffee shop scenario are depicted in Figures 15.20 and 15.21. Originally, as shown in Figure 15.20, only one customer could be served every 12 time units (analogous to clock cycles), and customers also had to wait for 12 time units to complete their transactions. This is because Penelope had to do everything by herself, and could only complete one task (or 'operation') at a time.

Notice that, as a result of adding more staff (as in Figure 15.21), one customer can be served every four time units, rather than one in every 12. There is also a reduction in the time each customer has to wait, from 12 time units to 4. The improvement in throughput arises from the increased concurrency (different operations can now take place at the same

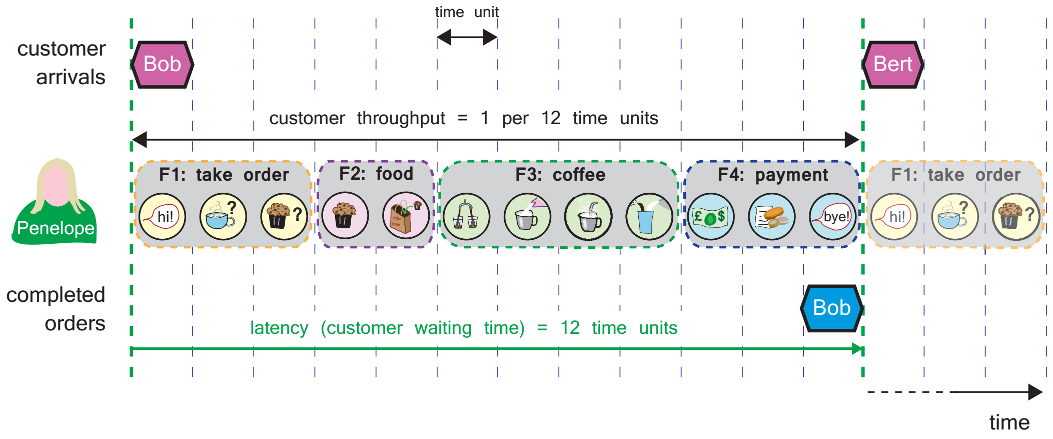


*Figure 15.20: Coffee shop example **without** dataflow optimisation*

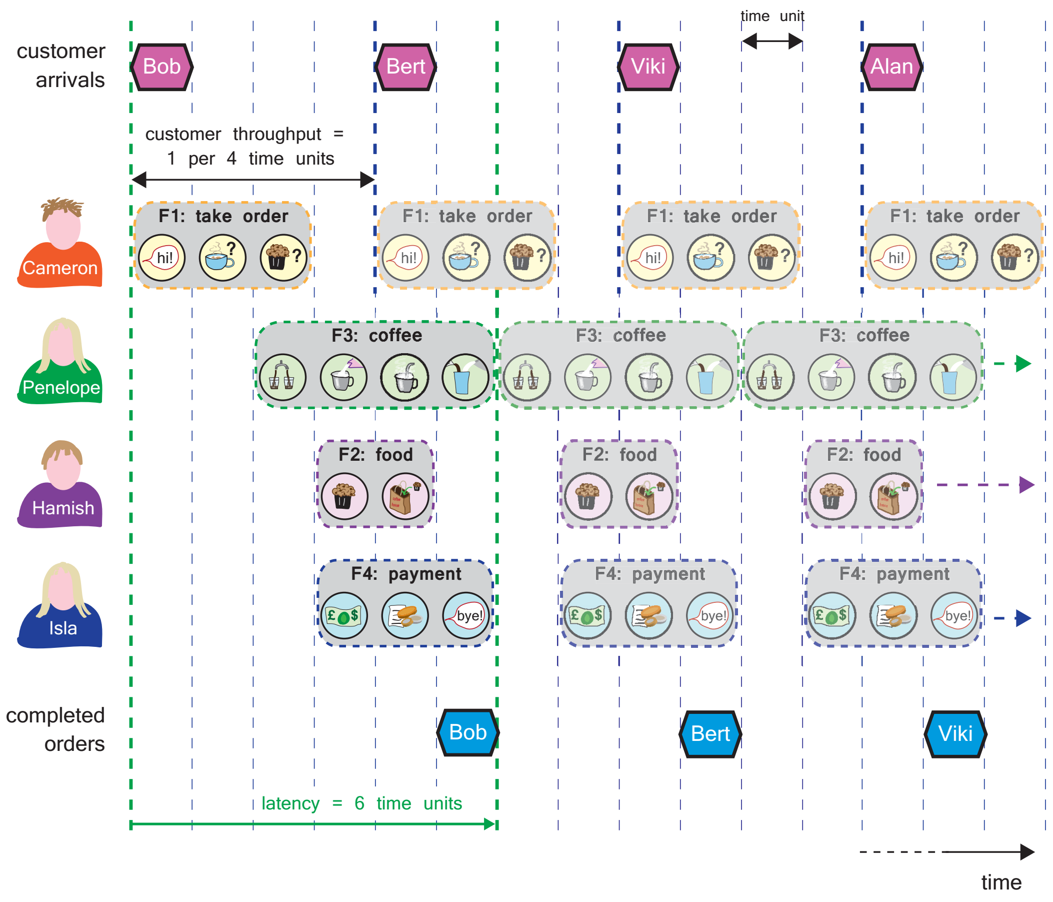


Figure 15.21: Coffee shop example ***with*** *dataflow optimisation*

time), while the reduction of latency (time a customer has to wait) is attributable to the overlapping of functions. Simply adding more staff, but having each of them undertake all functions in order (as Penelope did originally), would still leave Bob, Bert and friends waiting 12 time units to receive their orders!

We note in Figure 15.21 that the throughput is limited by Penelope's coffee making function, which takes 4 time units, whereas all of the others require less time to complete. If it were possible to pipeline *within* this function, e.g. to use one or more additional baristas to prepare the espresso, froth the milk, etc. concurrently, then further improvements could be achieved.

Similarly to the design of a hardware circuit, adding more concurrency (i.e. employing more staff) has the effect of improving performance. This is not 'free' as obviously the staff must be paid(!), but it may be desirable in order to obtain performance improvements.

The use of dataflow optimisation can be specified by directive in a similar fashion to pipelining. As we have seen through this example, the motivation for dataflow optimisation is to increase the obtainable throughput as a result of concurrent processing, and also to reduce latency by overlapping functions.

15.5.5. Algorithm Case Study: Loops

Loops are used extensively in software programming, and constitute a very succinct and natural method of expressing operations that are repetitive in some way. They are also used in a similar manner in HDLs, for example to iteratively instantiate and connect circuit components. However, an important difference is that the designer can prompt the loop(s) to be synthesised in different ways via the mechanism of directives in Vivado HLS. This contrasts with the use of loops in HDL, where code expressing loops is directly converted into hardware, usually resulting in prescribed and fixed architectures.

As an important software construct, loops are very well supported by Vivado HLS for hardware synthesis. Several loop optimisations can be made using directives, allowing the architecture of the resulting implementation to be altered with few or no changes required to the software code. During the remainder of this section, we will consider the default synthesis of loops, and architectural variations that can be achieved through the use of directives. Simple code examples are chosen in order to clearly make the necessary HLS-related points.

Default Loop Synthesis

By default, Vivado HLS seeks to optimise area, and therefore unless the designer directs otherwise, loops are automatically 'rolled', meaning that they time-share a minimal set of hardware. This means that the repetitive operations described by the loop are realised by a single piece of hardware implementing the body of the loop. To take a simple illustrative example, if a loop was designed to add the individual elements of two, 12 element arrays, then conceptually the implementation would involve a single adder (the body of the loop), shared 12 times according the number of loop iterations.

There is some latency associated with each iteration of the loop, and in this case the latency is affected by interactions with the memory interfaces at the inputs and output of the function. Memory interfaces are inferred due to the use of array arguments, according to the default interface protocols as discussed in Sections 15.4.3 and 15.4.4. Additional clock cycles are also required for entering and exiting the loop.

Code for this example is provided in Figure 15.22. Analysis of HLS synthesis with default settings shows that the overall latency is 26 clock cycles: 2 cycles each for 12 iterations (including reading the inputs from memory, adding, and writing the output to memory), and a further two clock cycles for entering and exiting the loop. Execution is illustrated in Figure 15.23.

```
void add_array (short c[12], short a[12], short b[12])
{
  short j;                          // loop variable

  add_loop: for (j=0;j<12;j++) {    // loop through elements (x12)
          c[j] = a[j] + b[j];      // addition operation
      }
}
```

Figure 15.22: Example code for a loop adding the elements of two arrays

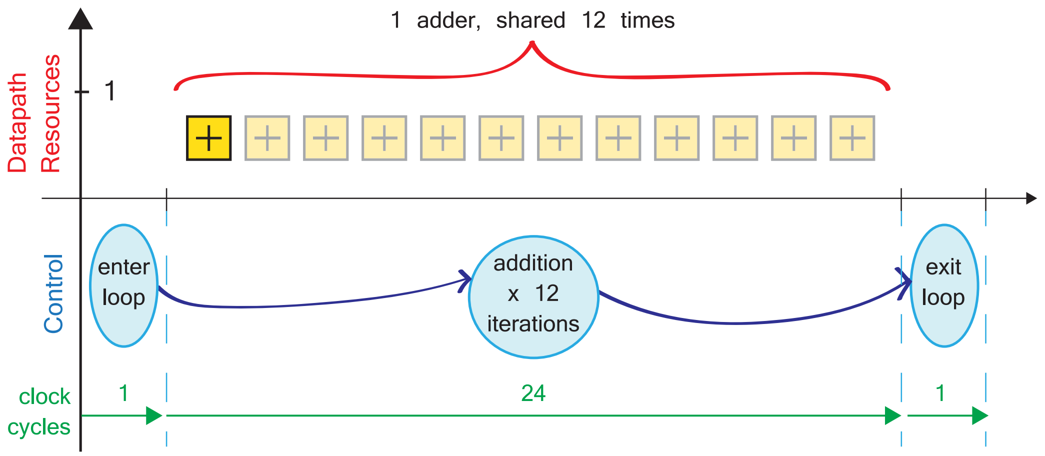


Figure 15.23: Extraction of addition loop into datapath and control logic

Simple Loop Architecture Variations

The default, rolled loop implementation may not always be desirable, but there are alternatives. Directives can be used to specify the architecture as summarised below.

- ***Unrolled*** — In a rolled implementation, a single instance of the hardware is inferred from the loop body and shared to the maximum extent. *Unrolling* a loop means that instead the hardware inferred from the loop body is created up to *N* times, where *N* is the number of loop iterations. In practice, the number of instances may be lower than *N* if other limiting factors are identified in the design, such as memory operations. The clear disadvantage of the unrolled version is that it consumes much larger area on the device than the rolled design, but the advantage is increased throughput.

- ***Partially unrolled*** — This constitutes a compromise between rolled and unrolled, and is typically used when a rolled implementation does not provide high enough throughput. If a rolled architecture represents minimal hardware cost but maximal time sharing (lowest throughput), and an unrolled architecture represents maximal hardware cost but minimal sharing (highest throughput), then we may try to find a different balance between the two. Control is exerted by the use of directives, and a number of different positions in the trade-off may be possible.

With reference to the upper section of Figure 15.23, which depicts a rolled architecture, fully or partially unrolling the loop would cause the number of datapath resources (adders) to increase, but to be shared to a lesser extent. Meanwhile, in the lower section of the diagram, the large, central state constituting the addition of array elements would require fewer clock cycles to complete. When fully unrolled, the implementation effectively does not contain a loop, and as a result the loop entry and exit clock cycles are saved, too.

With these observations in mind, the decision to select a rolled, unrolled, or partially unrolled implementation for the loop will be based on the specific requirements of the application, particularly in terms of the target throughput and any constraint on area utilisation that may apply.

Optimisation: Merging Loops

In some cases there might be two loops occurring one after the other in the code. For instance, the addition loop in the example of Figure 15.22 might be followed by a similar loop which multiplies the elements of the two arrays. Assuming that the loops are both rolled (according to the default mode), a possible optimisation in this case would be to merge the two loops, such that there is only one loop, with both the addition and multiplication operations being conducted within the single loop body.

The advantage of merging loops may not be immediately obvious, but in fact it relates to the control aspect of the design (recall from Section 14.4.5 that the C source code is analysed and decomposed into datapath and control components as part of the HLS process). Control is realised in the form of a Finite State Machine (FSM), with each loop corresponding to at least one state; thus the FSM can be simplified due to the merging of loops, as this results in fewer loops overall, and thus fewer FSM states. This is demonstrated by the code examples in Figures 15.24 and 15.25. The first example shows two separate loops, one each for addition and multiplication of the arrays, while the second shows the effect of merging the loops to create a single loop.

The `add_loop` represents 12 iterations of an addition operation (which takes 2 clock cycles), while the `mult_loop` represents 12 iterations of a multiplication operation (which takes 4 clock cycles). Therefore, the overall latencies of the two loops are 24 and 48 clock cycles, respectively. Merging the loops has the effect that the latency of the new, combined loop reduces to the longer of the original two loops, i.e. 48 clock cycles. One further clock cycle is saved due to the removed loop transition, i.e. the 'exit/enter' state in Figure 15.24.

The merging of loops can be controlled automatically using a Vivado HLS directive, and consequently there is no need to make an explicit change to the source code. The code provided in Figure 15.25 is for illustrative purposes only, i.e. to show the effect of applying the 'merge' directive.

```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;


  add_loop: for (j=0;j<12;j++) {
        c[j] = a[j] + b[j];
    }


  mult_loop: for (j=0;j<12;j++) {
        m[j] = a[j] * b[j];
    }
}
```

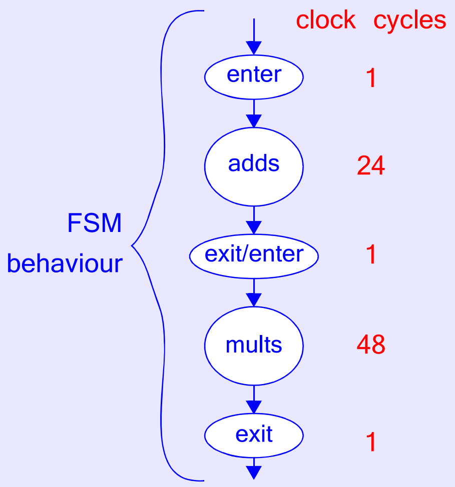
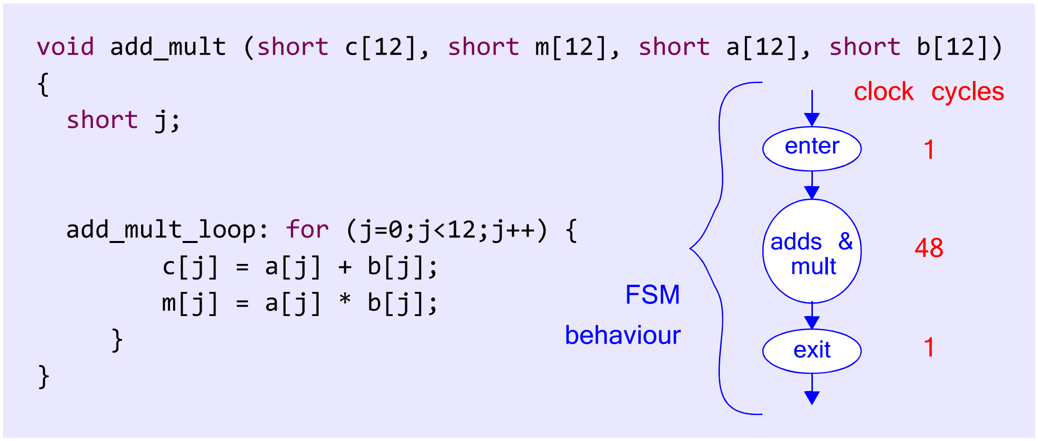


Figure 15.24: Consecutive loops for addition and multiplication within a function



```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;


  add_mult_loop: for (j=0;j<12;j++) {
        c[j] = a[j] + b[j];
        m[j] = a[j] * b[j];
     }
}
```

Figure 15.25: Merged addition and multiplication loops

Note that there are some practical restrictions on loop merging, with respect to compatibility and the limits of the loops to be merged. In our simple example, the limits were equal, but this may not always be the case; guidelines on this issue can be found in [18].

Nested Loops

Another common configuration is to nest loops, i.e. place one loop inside another. There may even be multiple levels of nesting. To give an example of a 2-level nested loop, suppose we extend our array addition example from linear arrays to 2-dimensional arrays. Mathematically, this is equivalent to adding two matrices, as shown in Equation (1).

$$\begin{bmatrix} f_{00} & f_{01} & f_{02} & f_{03} \\ f_{10} & f_{11} & f_{12} & f_{13} \\ f_{20} & f_{21} & f_{22} & f_{23} \end{bmatrix} = \begin{bmatrix} d_{00} & d_{01} & d_{02} & d_{03} \\ d_{10} & d_{11} & d_{12} & d_{13} \\ d_{20} & d_{21} & d_{22} & d_{23} \end{bmatrix} + \begin{bmatrix} e_{00} & e_{01} & e_{02} & e_{03} \\ e_{10} & e_{11} & e_{12} & e_{13} \\ e_{20} & e_{21} & e_{22} & e_{23} \end{bmatrix} \tag{1}$$

Now, in order to add the arrays, we must iterate through the rows, and for each row, iterate through the columns, adding together the two values for each array element. Coding the matrix addition operation requires an outer and an inner loop to cycle through the rows and columns, respectively, as shown by the code example in Figure 15.26. According to (1), there are 3 rows and 4 columns, and this determines the limits of the nested loops (note that indexing starts as zero, as per the usual convention).

Extending this idea, it would be possible to work with three dimensional arrays, or even higher dimensions, by increasing the levels of nesting in the loop structure.

```
void add_matrix (short f[3][4], short c[3][4], short d[3][4])
{
  short j,k;                                // loop variable

  row_loop: for (j=0;j<3;j++) {             // iterate through rows
       column_loop : for (k=0;k<4;k++) {  // iterate through columns
          f[j][k] = c[j][k] + d[j][k];     // addition operation
     }
  }
}
```

Figure 15.26: Example code for nested loops adding the elements of two dimensional arrays

Optimisation: Flattening Loops

In the case of nested loops, we have the option to perform 'flattening'. This means that the loop hierarchy is effectively removed during high-level synthesis, while preserving the algorithm, i.e. all of the operations performed by the loop(s). The advantage of flattening is similar to merging: the additional clock cycles associated with transitioning into or out of a loop are avoided, meaning that the overall duration of algorithm execution reduces, thus improving the achievable throughput.

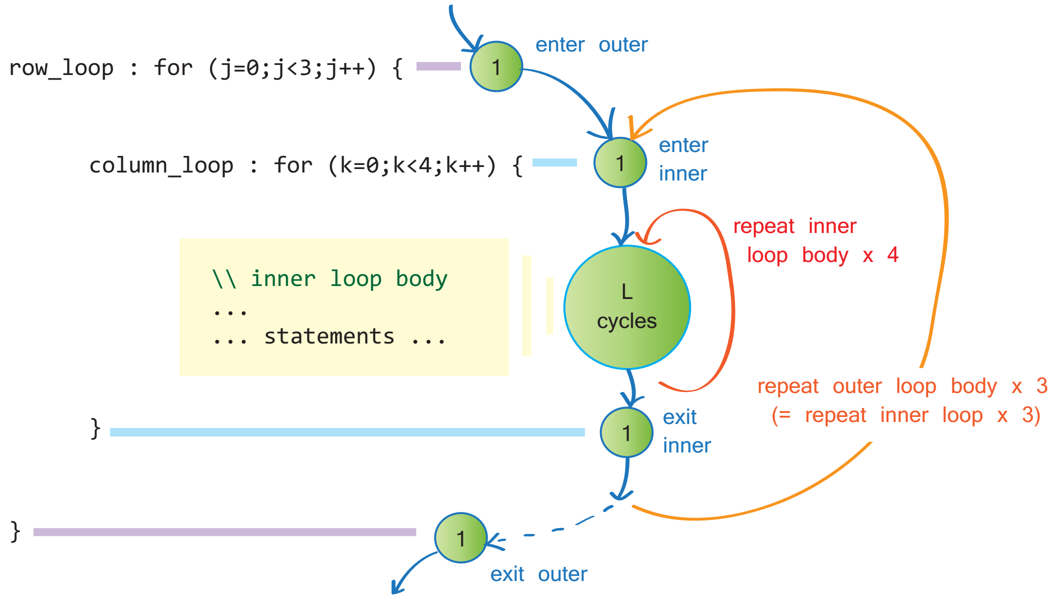
In order to explain flattening in further detail, it is necessary to clarify the terms *loop* and *loop body*. By *loop*, we refer to an entire code structure of a set of statements repeated a defined number of times. The statements inside the loop, i.e. the statements that are repeated, are the *loop body*. For instance, `column_loop` is a *loop*, and the statements contained within `column_loop` correspond to the *loop body*.

When loops are nested, and again taking the example of a 2-level nested structure, the outer loop body contains another loop, i.e. the inner loop. The outer loop body (including the inner loop) is executed a certain number of times; for instance, `row_loop` has 3 repetitions in the example of Figure 15.26, and hence there are 3 executions of the inner loop, `column_loop`. Each execution of the inner loop involves repeating the inner loop body a specified number of times, as well: in our example, a statement to calculate the matrix element `f[j][k]` is executed 4 times, where `j` is the row index and `k` is the column index.

The overhead of loop transitioning means that two additional clock cycles are required each time the inner loop is executed, i.e. one to enter the inner loop, and one to exit from it.

To clarify this point, a diagram depicting the control flow for our matrix addition example, and associated clock cycles, is provided in Figure 15.27. This represents the original loop structure. The process of flattening 'unrolls' the inner loop, and as a consequence, reduces the number of clock cycles associated with transitioning into and out of loops; specifically, the 'enter_inner' and 'exit_inner' states in Figure 15.27 are removed. These would have been repeated 3 times, hence 6 clock cycles are saved in total in this case.

In our simple 3 x 4 matrix addition example, the saving equates to 6 clock cycles, but in other examples this could be considerably higher (in particular where the outer loop iterates many times, or there are several layers of nesting), and thus there is a clear motivation to flatten loops. Similar to merging of loops, flattening can be achieved by a directive and does not involve explicit unrolling of the loop by manually changing the code. However, depending on its initial form, some manual restructuring may also be required in order to achieve a loop structure optimal for flattening [18].



```
row_loop : for (j=0;j<3;j++) {

    column_loop : for (k=0;k<4;k++) {

        \\ inner loop body
        ...
        ... statements ...

    }

}
```

1
enter outer
1
enter inner
repeat inner loop body x 4
L cycles
repeat outer loop body x 3 (= repeat inner loop x 3)
1
exit inner
1
exit outer

Figure 15.27: Control flow through the matrix addition, without flattening (clock cycles in circles)

Pipelining Loops

The direct interpretation of a loop written in C code is that executions of the loop occur consecutively, i.e. each loop iteration cannot begin until the previous one has completed. In hardware terms, this translates to a single set of hardware (as inferred from the loop body) that is capable of executing the operations of only one loop iteration at any particular instant, and which is shared over time according to the number of loop iterations.

This relates to our previous discussion on pipelining, i.e. as covered in Section 15.5.3. We observed that throughput is limited when a set of operations are grouped together into a processing stage. In the case of loops, the loop body (i.e. the set of repeated operations) forms such a stage, and without pipelining this would result in all stages operating in a sequential manner, and within them, all operations executing in a sequential manner. In effect, all operations from all iterations of the loop body would occur one after the other, similar to Figure 15.17 on page 314. The total number of clock cycles to complete the execution of a loop, N_{loop}, would therefore be:

$$N_{loop} = (J \times N_{body}) + N_{control} \qquad (2)$$

where J is the number of loop iterations, N_{body} is the number of clock cycles to execute all operations in the loop body, and $N_{control}$ represents the overhead of transitioning into and out of the loop.

The insertion of pipelining into the loop means that registers separate the implemented operators within the loop body. Given that the loop body is repeated several times, this carries the implication that operations within loop iteration j+1 can commence before those of the loop iteration j have completed. In fact, at any instant, operations corresponding to several different iterations of the loop may be active.

As a consequence of pipelining the loop, the hardware required to implement the loop body is more fully utilised, and loop performance is improved in terms of both throughput and latency. The effect of adding a pipelining directive can therefore be considerable, especially where there are multiple operations within the loop body, and many iterations of the loop are performed.

When working with nested loops, it is useful to consider at which level of hierarchy pipelining should be applied. Pipelining at a certain level of hierarchy will cause all levels below (i.e. all nested loops) to be unrolled, which may produce a more costly implementation than intended. Therefore, a good balance between performance and resource utilisation may be achieved by pipelining only at the level of the innermost loop (for instance, `column_loop` in Figure 15.27).

15.5.6. Arrays

In Vivado HLS, array types normally represent storage, and hence arrays are usually synthesised into memories.

The memories inferred during the HLS process are mapped to physical resources on the PL — whether Block RAM, or distributed RAM constructed from logic slices — and therefore it is important to be aware of the size and shape of synthesised memories, and how these map to the resources available on the device. In fact, often it is desirable to exert influence over the synthesised memory to enhance its mapping to physical memory resources, and designers may achieve this through the use of directives.

A number of array optimisations are available, and these can be specified via directive as detailed below. Note that these are similar to the interface directives of the same name, but in this case they relate to elements within the circuit.

- ***Resource*** — The designer can choose to map an array in the C-based HLS source code to a specific memory resource.

- ***Array Map*** — Allows several small arrays to be combined into a single, larger array. This can bring the advantage of reducing the total amount of memory resources required (e.g. it may be possible to use a single Block RAM to implement all memories combined, rather than one Block RAM each for four discrete memories). Mapping can be either *horizontal* (arrays are concatenated to form an array with more elements), or *vertical* (array elements are combined, resulting in an array with longer words).

- ***Array Partition*** — This directive could be considered the opposite to the Array Map directive, in the sense that it allows the designer to determine the subdivision of a large array into a set of smaller ones. The motivation for partitioning is generally to improve the aggregate rate at which memory transactions can take place, e.g. a large dual port RAM can accommodate two transactions per clock cycle, whereas four smaller dual port RAMs can accommodate a total of eight transactions per clock cycle (two each). At the extreme, array partitioning can prompt the subdivision of an array into individual register elements.

- ***Array Reshape*** — This directive allows an array with many elements, each with short words, to be reshaped into an array with fewer, longer words. The motivation for applying this directive is to reduce the number of required memory accesses.

- ***Stream*** — Applying the stream directive prompts an array to be synthesised into a FIFO, rather than a RAM.

Where combination of memories is based on arrays with differing dimensions, this is accommodated by the tool as described in [18].

As is evident from the selection of array directives reviewed here, the designer can mould arrays in various different ways, according to his or her needs. In some circumstances, it is desirable to combine arrays in order to optimise resource usage. On other occasions, it may be more important to optimise memory bandwidth, and this can be accomplished by separating arrays into smaller memories, such that there are more memory access ports available.

Array manipulation can therefore be considered a flexible and powerful technique available to the HLS designer, whether the principal implementation aim is to minimum resource utilisation, or maximise performance.

15.6. Design Evaluation and Optimisation

As mentioned extensively throughout this chapter, designs developed in Vivado HLS are based on a structure wherein the source code is constant, and a series of variations ('solutions') are generated by supplying different constraints and directives. The designer can explore possibilities and iterate towards the optimum solution by tuning these parameters.

In this section, we consolidate some of these points, and consider in particular the steps a designer might take to refine a design via the mechanisms provided in Vivado HLS.

15.6.1. Design Constraints

Certain constraints can be applied within the design process to limit some aspect of the produced solution. The most commonly applied type of constraint is a timing constraint, which usually places an upper limit on the clock period (although other aspects of timing can also be defined). Another possibility is to constrain the latency of a design, meaning that the number of clock cycles between applying an input, and observing the corresponding output, is given an upper or lower limit. Aside from aspects of timing, the designer can also constrain the resources used to implement the desired functionality.

The HLS process may produce different results depending on the applied constraints. For example, if a maximum latency is prescribed, then the generated design may use more resources to implement the required algorithm; on the other hand, if resource utilisation is limited, then the resulting implementation is likely to employ time sharing, and thus to exhibit higher latency.

15.6.2. Synthesis Directives

We have seen through the discussion in this chapter that there are several types of directive available, which permit the designer to influence certain aspects of the synthesised hardware. For instance, we noted that interface constraints can be applied to prompt a particular type of protocol to be used, and that pipelining directives influence concurrency, latency and throughput.

Directives can be applied as TCL commands and consolidated in dedicated file, or inserted inline in the C/C++/SystemC source code, as pragmas. These methods may each be preferred for different reasons. To provide a common example, often the interface of a design is defined first, and fixed; therefore applying interface directives as pragmas means that these choices are consistent across all solutions. On the other hand, while actively exploring the algorithm synthesis design space, it is appropriate to keep the directives separate from the source code, as this makes it easier to manage the application of directives, and create 'fresh' solutions where desired.

15.6.3. Statistics and Reports

With each solution produced by Vivado HLS, an accompanying report will be generated to encapsulate its various statistics, including estimates of timing (clock) performance, latency, and resource utilisation. (Note that these are estimates, bearing in mind that full detail is not available until the more time consuming stages of RTL synthesis and implementation are undertaken.) Full details of the synthesised interface are also provided. Where applicable, the report will also contain details of individual loops within the design, in terms of their trip count (number of iterations), latency and initiation interval.

A further option is to generate a report consolidating the statistics from a set of solutions. Vivado HLS prompts the user to make a selection from all solutions in the project, and a report is prepared accordingly. This summary is a helpful method of comparing statistics across a set of solutions, and thus identifying the solution providing the best fit with requirements, or the trend associated with applying a certain directive.

15.6.4. Design Iterations and Optimisation

It should be apparent from previous discussion that the designer can exert considerable control over the results of HLS via the use of directives and constraints. The reports produced for individual solutions will help to identify issues such as memory bottlenecks, excessive loop latency, and over-utilisation of resources. He or she can then refine the existing directives, or augment them to better direct the synthesis process towards the optimum solution.

15.7. Exporting from Vivado HLS

Designs can be exported from Vivado HLS to form a selection of different outputs. These are provided to permit easy integration of Vivado HLS IP with other development tools in the Vivado and ISE design suites.

At the stage of exporting, there is an opportunity to 'Evaluate' the design, meaning that the stages of RTL synthesis and implementation are performed, and this produces a further report confirming actual values for resource utilisation and timing performance. This can be undertaken using VHDL or Verilog as the RTL language.

15.7.1. Vivado IP Catalog (IP-XACT Format)

The primary option is to export from HLS to the IP-XACT format, which allows the module to be integrated into a Vivado IP Integrator design. An option is presented allowing the designer to label the IP package, and to insert author and version information. The result is a zip folder residing in the 'impl\ip' subfolder of the relevant solution, and this represents the IP Catalog package.

Once in IP-XACT format, the IP produced from Vivado HLS can be easily shared and distributed.

15.7.2. System Generator for DSP

A further option is to export the HLS design as an IP block for use in System Generator. When doing so, the user is prompted to specify either ISE or Vivado, depending on their tool of choice for performing logic synthesis and implementation. That is to say, if the final System Generator system (including the IP originating from HLS) is intended to be synthesised using ISE, then ISE should be chosen when exporting from Vivado HLS.

15.7.3. Pcore for XPS

Users working with the XPS tool for embedded systems design have the option to export Vivado HLS IP as a *pcore*, which can be easily integrated into an XPS-based system.

15.8. Chapter Review

This chapter has provided a detailed look at the Vivado HLS development tool, which provides the facility to rapidly develop hardware designs from C-based software descriptions.

Although there is insufficient scope in a single chapter to review all aspects of the tool, we have covered the Vivado HLS environment, the use of data types (including the facilities for working with arbitrary precision formats), and various aspects of interface and algorithm synthesis. Several conceptual and code-based examples have been presented to illustrate the points made.

A running theme throughout has been the ability of the designer to influence the synthesis from input C code, via the use of directives and constraints, and thus to produce different 'solutions'. As part of this discussion, key performance and implementation metrics were reviewed, and examples were presented to illustrate the designer's ability to control these using directives.

Finally, it was noted that designs produced in Vivado HLS can be readily exported for integration into larger system projects, whether in IP Integrator, XPS, or System Generator.

15.9. References

Note: All URLs last accessed June 2014.

[1] D. C. Black, J. Donovan, B. Bunton and A. Keist, *SystemC: From the Ground Up*, 2nd Edition, Springer, 2009.

[2] M. Burton, J. Aldis, R. Günzel and W. Klingauf, "Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified", *Forum on Design Languages*, 2007, pp. 92-97.

[3] D. Gajski and R. Kuhn, "Guest Editors' Introduction: New VLSI Tools", *Computer*, vol. 16, no.12, pp.11 - 14, December 1983.

[4] D. Gadski, T. Austin and S. Svoboda, "What Input-Language is the Best Choice for High Level Synthesis (HLS)?", *panel session, Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC)*, pp. 857-858, June 2010.

[5] GNU, *The GNU C Reference Manual.*
Available: http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

[6] D. Große and R. Drechsler, *Quality-Driven SystemC Design*, Springer, 2010.

[7] IEEE Computer Society, "IEEE Standard for Standard SystemC Language Reference Manual", *IEEE Std 1666-2011*, January 2012.

[8] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[9] S. G. Kochan, *Programming in C: A Complete Introduction to the C Programming Language*, 3rd Edition, Sams Publishing, 2005.

[10] M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, vol. 78, no.2, pp 301 - 318, Feb 1990.

[11] G. Martin and G. Smith, "High Level Synthesis: Past, Present and Future", *IEEE Design and Test of Computers*, Vol. 26, Issue 4, July/August 2009, pp. 18 - 24.

[12] A. Mathur, E. Clarke, M Fujita, and R. Urard, "Functional Equivalence Verification Tools in High-Level Synthesis Flows", *IEEE Design & Test of Computers*, July/August 2009, pp. 88 - 95.

[13] M. Meredith and S. Svoboda, "The Next IC Design Methodology Transition is Long Overdue", Open SystemC Initiative, February 2010.
Available: http://www.accellera.org/resources/articles/icdesigntrans/community/articles/icdesigntrans/ic_design_transition_feb2010.pdf

[14] D. M. Ritchie, "The Development of the C Language", *Proceedings of the 2nd History of Programming Languages Conference*, Cambridge, Massachusetts, April 1993.

[15] Xilinx, Inc., "UG634 - AccelDSP Synthesis Tool User Guide", v11.4, December 2009.
Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/acceldsp_user.pdf

[16] Xilinx, Inc., "UG835 - Vivado Design Suite Tcl Command Reference Guide", v2014.1, April 2014.
Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug835-vivado-tcl-commands.pdf

[17] Xilinx, Inc., "UG871 - Vivado Design Suite Tutorial: High Level Synthesis", v2014.1, May 2014.
Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug871-vivado-high-level-synthesis-tutorial.pdf

[18] Xilinx, Inc, "UG902 - Vivado Design Suite User Guide: High-Level Synthesis", v2014.1, May 2014.
Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[19] Xilinx, Inc., "UG998 - Introduction to FPGA Design with Vivado High-Level Synthesis", v1.0, July, 2013.
Available: http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf