

Parallel Programming for FPGAs

Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer

2018-05-11

Contents

Preface	7
Acknowledgements	9
1 Introduction	11
1.1 High-level Synthesis (HLS)	11
1.2 FPGA Architecture	13
1.3 FPGA Design Process	19
1.4 Design Optimization	20
1.4.1 Performance Characterization	20
1.4.2 Area/Throughput Tradeoffs	22
1.4.3 Restrictions on Processing Rate	26
1.4.4 Coding Style	27
1.5 Restructured Code	28
1.6 Book Organization	28
2 Finite Impulse Response (FIR) Filters	31
2.1 Overview	31
2.2 Background	32
2.3 Base FIR Architecture	33
2.4 Calculating Performance	35
2.5 Operation Chaining	36
2.6 Code Hoisting	38
2.7 Loop Fission	39
2.8 Loop Unrolling	40
2.9 Loop Pipelining	43
2.10 Bitwidth Optimization	47
2.11 Complex FIR Filter	49
2.12 Conclusion	52
3 CORDIC	55
3.1 Overview	55
3.2 Background	56
3.3 Calculating Sine and Cosine	61
3.4 Cartesian to Polar Conversion	63
3.5 Number Representation	66
3.5.1 Binary and Hexadecimal Numbers	66
3.5.2 Negative numbers	69

3.5.3	Overflow, Underflow, and Rounding	70
3.5.4	Binary arithmetic	72
3.5.5	Representing Arbitrary Precision Integers in C and C++	73
3.5.6	Floating Point	75
3.6	Further Optimizations	75
3.7	Conclusion	76
4	Discrete Fourier Transform	77
4.1	Fourier Series	77
4.2	Discrete Fourier Transform (DFT) Background	79
4.3	Matrix-Vector Multiplication Optimizations	82
4.4	Pipelining and Parallelism	83
4.5	Storage Tradeoffs and Array Partitioning	87
4.6	Baseline Implementation	92
4.7	DFT optimization	94
4.8	Conclusion	97
5	Fast Fourier Transform	99
5.1	Background	99
5.2	Baseline Implementation	108
5.3	Bit Reversal	111
5.4	Task Pipelining	113
5.5	Conclusion	116
6	Sparse Matrix Vector Multiplication	117
6.1	Background	117
6.2	Baseline Implementation	119
6.3	Testbench	120
6.4	Specifying Loop Properties	122
6.5	C/RTL Cosimulation	123
6.6	Loop Optimizations and Array Partitioning	124
6.7	Conclusion	128
7	Matrix Multiplication	129
7.1	Background	129
7.2	Complete Matrix Multiplication	130
7.3	Block Matrix Multiplication	133
7.4	Conclusion	144
8	Prefix Sum and Histogram	145
8.1	Prefix Sum	145
8.2	Histogram	148
8.3	Histogram Optimization and False Dependencies	149
8.4	Increasing Histogram Performance	152
8.5	Conclusion	156

9	Video Systems	157
9.1	Background	157
9.1.1	Representing Video Pixels	158
9.1.2	Digital Video Formats	158
9.1.3	Video Processing System Architectures	161
9.2	Implementation	164
9.2.1	Line Buffers and Frame Buffers	164
9.2.2	Causal Filters	166
9.2.3	Boundary Conditions	168
9.3	Conclusion	171
10	Sorting Algorithms	173
10.1	Introduction	173
10.2	Insertion Sort	174
10.2.1	Basic Insertion Sort Implementation	175
10.2.2	Parallelising Insertion Sort	178
10.2.3	Explicit Systolic Array For Insertion Sort	180
10.3	Merge Sort	185
10.3.1	Basic Merge Sort	186
10.3.2	Restructured Merge Sort	189
10.4	Conclusion	192
11	Huffman Encoding	195
11.1	Background	195
11.2	Implementation	199
11.2.1	Filter	202
11.2.2	Sort	203
11.2.3	Create Tree	207
11.2.4	Compute Bit Length	211
11.2.5	Truncate Tree	213
11.2.6	Canonize Tree	215
11.2.7	Create Codeword	217
11.2.8	Testbench	220
11.3	Conclusion	223
	Bibliography	225
	Glossary	231
	Acronyms	235

Chapter 1

Introduction

1.1 High-level Synthesis (HLS)

The hardware design process has evolved significantly over the years. When the circuits were small, hardware designers could more easily specify every transistor, how they were wired together, and their physical layout. Everything was done manually. As our ability to manufacture more transistors increased, hardware designers began to rely on automated design tools to help them in the process of creating the circuits. These tools gradually become more and more sophisticated and allowed hardware designers to work at higher levels of abstraction and thus become more efficient. Rather than specify the layout of every transistor, a hardware designer could instead specify digital circuits and have [electronic design automation \(EDA\)](#) tools automatically translate these more abstract specifications into a physical layout.

The Mead and Conway approach [\[50\]](#) of using a programming language (e.g., Verilog or VHDL) that compiles a design into physical chips took hold in the 1980s. Since that time, the hardware complexity has continued to increase at an exponential rate, which forced hardware designers to move to even more abstract hardware programming languages. [register-transfer level \(RTL\)](#) was one step in abstraction, enabling a designer to simply specify the registers and the operations performed on those registers, without considering how the registers and operations are eventually implemented. [EDA](#) tools can translate [RTL](#) specifications into a digital circuit model and then subsequently into the detailed specification for a device that implements the digital circuit. This specification might be the files necessary to manufacture a custom device or might be the files necessary to program an off-the-shelf device, such as an [field-programmable gate array \(FPGA\)](#). Ultimately, the combination of these abstractions enables designers to build extraordinarily complex systems without getting lost in the details of how they are implemented. A non-technical perspective on the value of these abstractions can be found in [\[42\]](#).

[high-level synthesis \(HLS\)](#) is yet another step in abstraction that enables a designer to focus on larger architectural questions rather than individual registers and cycle-to-cycle operations. Instead a designer captures behavior in a program that does not include specific registers or cycles and an [HLS](#) tool creates the detailed [RTL](#) micro-architecture. One of the first tools to implement such a flow was based on behavioral Verilog and generated an [RTL](#)-level architecture also captured in Verilog [\[35\]](#). Many commercial tools now use C/C++ as the input language. For the most part the language is unimportant, assuming that you have a tool that accepts the program you want to synthesize!

Fundamentally, algorithmic HLS does several things automatically that an RTL designer does manually:

- **HLS** analyzes and exploits the concurrency in an algorithm.
- **HLS** inserts registers as necessary to limit critical paths and achieve a desired clock frequency.
- **HLS** generates control logic that directs the data path.
- **HLS** implements interfaces to connect to the rest of the system.
- **HLS** maps data onto storage elements to balance resource usage and bandwidth.
- **HLS** maps computation onto logic elements performing user specified and automatic optimizations to achieve the most efficient implementation.

Generally, the goal of **HLS** is to make these decisions automatically based upon user-provided input specification and design constraints. However, **HLS** tools greatly differ in their ability to do this effectively. Fortunately, there exist many mature HLS tools (e.g., Xilinx Vivado[®] HLS, LegUp [13], and Mentor Catapult HLS) that can make these decisions automatically for a wide range of applications. We will use Vivado[®] HLS as an exemplar for this book; however, the general techniques are broadly applicable to most HLS tools though likely with some changes in input language syntax/semantics.

In general, the designer is expected to supply the **HLS** tool a functional specification, describe the interface, provide a target computational device, and give optimization directives. More specifically, Vivado[®] HLS requires the following inputs:

- A function specified in C, C++, or SystemC
- A design testbench that calls the function and verifies its correctness by checking the results.
- A target FPGA device
- The desired clock period
- Directives guiding the implementation process

In general, **HLS** tools can not handle any arbitrary software code. Many concepts that are common in software programming are difficult to implement in hardware. Yet, a hardware description offers much more flexibility in terms of how to implement the computation. It typically requires additional information to be added by the designers (suggestions or **#pragmas**) that provide hints to the tool about how to create the most efficient design. Thus, **HLS** tools simultaneously limit and enhance the expressiveness of the input language. For example, it is common to not be able to handle dynamic memory allocation. There is often limited support for standard libraries. System calls are typically avoided in hardware to reduce complexity. The ability to perform recursion is often limited. On the other hand, **HLS** tools can deal with a variety of different interfaces (direct memory access, streaming, on-chip memories). And these tools can perform advanced optimizations (pipelining, memory partitioning, bitwidth optimization) to create an efficient hardware implementation.

We make the following assumptions about the input function specification, which generally adheres to the guidelines of the Vivado[®] HLS tool:

- No dynamic memory allocation (no operators like `malloc()`, `free()`, `new`, and `delete()`)
- Limited use of pointers-to-pointers (e.g., may not appear at the interface)

- System calls are not supported (e.g., `abort()`, `exit()`, `printf()`, etc. They can be used in the code, e.g., in the testbench, but they are ignored (removed) during synthesis.
- Limited use of other standard libraries (e.g., common `math.h` functions are supported, but uncommon ones are not)
- Limited use of function pointers and virtual functions in C++ classes (function calls must be compile-time determined by the compiler).
- No recursive function calls.
- The interface must be precisely defined.

The primary output of an `HLS` tool is a `RTL` hardware design that is capable of being synthesized through the rest of the hardware design flow. Additionally, the tool may output testbenches to aid in the verification process. Finally, the tool will provide some estimates on resource usage and performance. Vivado[®] HLS generates the following outputs:

- Synthesizable Verilog and VHDL
- RTL simulations based on the design testbench
- Static analysis of performance and resource usage
- Metadata at the boundaries of a design, making it easier to integrate into a system.

Once an `RTL`-level design is available, other tools are usually used in a standard `RTL` design flow. In the Xilinx Vivado[®] Design Suite, `logic synthesis` is performed, translating the `RTL`-level design into a `netlist` of primitive `FPGA` logical elements. The `netlist` (consisting of logical elements and the connections between them) is then associated with specific resources in a target device, a process called `place and route (PAR)`. The resulting configuration of the `FPGA` resources is captured in a `bitstream`, which can be loaded onto the `FPGA` to program its functionality. The `bitstream` contains a binary representation of the configuration of each `FPGA` resource, including logic elements, wire connections, and on-chip memories. A large Xilinx UltraScale `FPGAs` will have over 1 billion configuration bits and even the “smaller” devices have hundreds of millions of bits [64].

1.2 FPGA Architecture

It is important to understand the modern `FPGA` architectures since many of the `HLS` optimizations specifically target these features. Over the decades, `FPGAs` have gone from small arrays of programmable logic and interconnect to massive arrays of programmable logic and interconnect with on-chip memories, custom data paths, high speed I/O, and microprocessor cores all co-located on the same chip. In this section, we discuss the architectural features that are relevant to `HLS`. It is not our intention (nor do we think it is necessary) to provide substantial details of the `FPGA` architecture. Rather we aim to give the reader enough information to understand the `HLS` reports and successfully use and leverage the `HLS` directives, many of which very specifically target modern `FPGA` architectural features.

`FPGAs` are an array of programmable logic blocks and memory elements that are connected together using programmable interconnect. Typically these logic blocks are implemented as a `lookup table (LUT)` – a memory where the address signal are the inputs and the outputs are stored

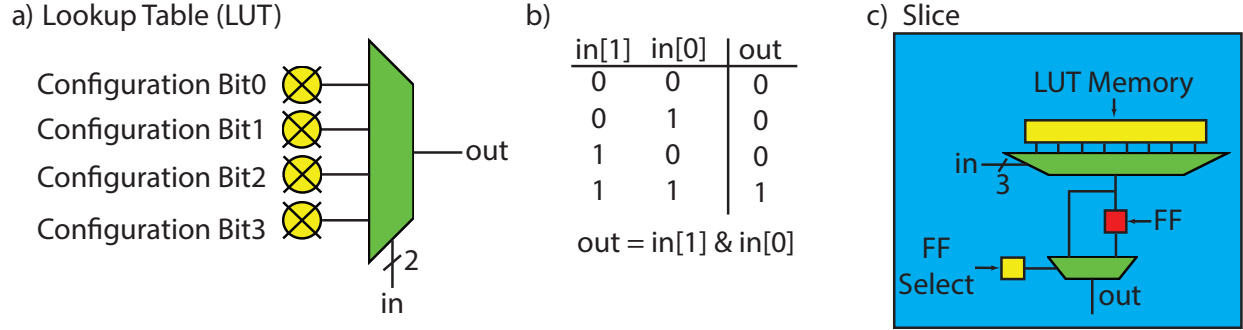


Figure 1.1: Part a) shows a 2 input **LUT**, i.e., a 2-LUT. Each of the four configuration bits can be programmed to change the function of the 2-LUT making it a fully programmable 2 input logic gate. Part b) provides a sample programming to implement an AND gate. The values in the “out” column from top to bottom correspond directly to configuration bits 0 through 3. Part c) shows a simple **slice** that contains a slightly more complex 3-LUT with the possibility of storing the output into a **flip-flop (FF)**. Note that there are nine configuration bits: eight to program the 3-LUT and one to decide whether the output should be direct from the 3-LUT or the one stored in the **FF**. More generally, a **slice** is defined as a small number of **LUTs** and **FFs** combined with routing logic (multiplexers) to move inputs, outputs, and internal values between the **LUTs** and **FFs**.

in the memory entries. An n -bit **LUT** can be programmed to compute any n -input Boolean function by using the function’s truth table as the values of the **LUT** memory.

Figure 1.1 a) shows a 2 input **LUT**. It has $2^2 = 4$ configuration bits. These bits are the ones that are programmed to determine the functionality of the **LUT**. Figure 1.1 b) shows the truth table for a 2 input AND gate. By using the four values in the “out” column for configuration bits 0-3, we can program this 2 input LUT to be a 2 input AND gate. The functionality is easily changed by reprogramming the configuration bits. This is a flexible and fairly efficient method for encoding smaller Boolean logic functions. Most **FPGAs** use a **LUTs** with 4-6 input bits as their base element for computation. Larger **FPGAs** can have millions of these programmable logic elements.

How would you program the 2-LUT from Figure 1.1 to implement an XOR gate? An OR gate? How many programming bits does an n input (n -LUT) require?

How many unique functions can a 2-LUT be programmed to implement? How many unique functions can a n input (n -LUT) implement?

The **FF** is the basic memory element for the **FPGA**. They are typically co-located with a **LUTs**. **LUTs** can be replicated and combined with **FFs** and other specialized functions (e.g., a full adder) to create a more complex logic element called a configurable logic block (CLB), logic array block (LAB), or **slice** depending on the vendor or design tool. We use the term **slice** since it is the resource reported by the Vivado[®] HLS tool. A **slice** is a small number of **LUTs**, **FFs** and multiplexers combined to make a more powerful programmable logic element. The exact number and combination of **LUTs**, **FFs** and multiplexers varies by architecture, but generally a slice has only few of each of these elements. Figure 1.1 c) shows a very simple slice with one 3-LUT and

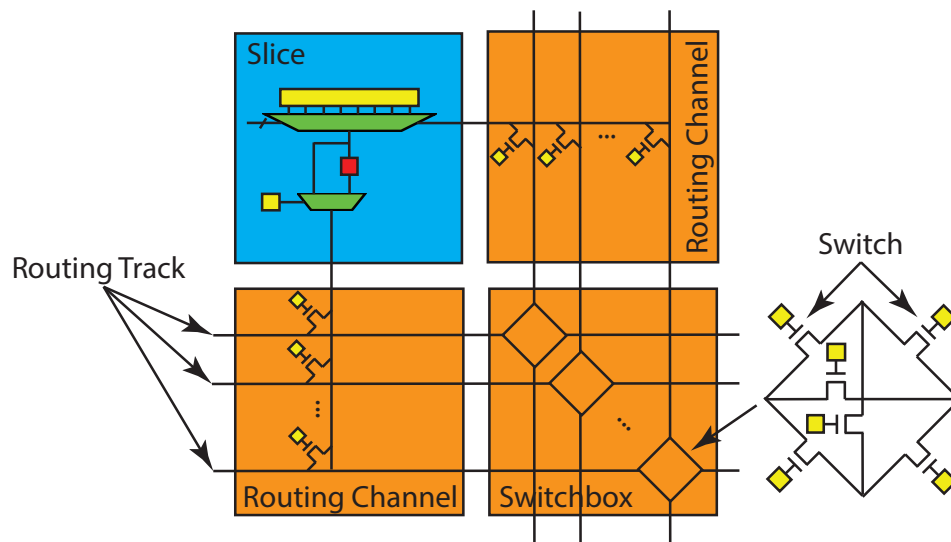


Figure 1.2: A **slice** contains a small number of **LUTs** and **FF**. We show a very simple **slice** with one **LUT** and one **FF** though generally these have two or more of each. Slices are connected to one another using a **routing channel** and **switchbox**. These two provide a programmable interconnect that provide the data movement between the programmable logic elements (**slices**). The **switchbox** has many switches (typically implemented as pass transistors) that allow for arbitrary wiring configurations between the different routing tracks in the routing tracks adjacent to the switchbox.

one FF. A **slice** may also use some more complex logic functions. For example, it is common to embed a full adder into a slice. This is an example of “hardening” the FPGA; this full adder is not programmable logic – it can only be used as a full adder, but it is common to use full adders (to make addition operations) and it is more efficient to use the custom full adder as opposed to implementing a full adder on the programmable logic (which is also an option). And thus, overall it is beneficial to have a hard full adder in the slice.

Programmable interconnect is the other key element of an FPGA. It provides a flexible network of wires to create connections between the slices. The inputs and outputs of the **slice** are connected to a **routing channel**. The **routing channel** contains a set configuration bits can be programmed to connect or disconnect the inputs/outputs of the **slice** to the programmable interconnect. Routing channels are connected to **switchboxes**. A **switchbox** is a collection of switches that are implemented as pass transistors. These provide the ability to connect the routing tracks from one **routing channels** to another.

Figure 1.2 provides an example of how a **slice**, **routing channel**, and **switchbox** are connected. Each input/output to the **slice** can be connected to one of many routing tracks that exist in a routing channel. You can think of routing tracks as single bit wires. The physical connections between the slice and the routing tracks in the routing channel are configured using a pass transistor that is programmed to perform a connect or disconnect from the input/output of the slice and the programmable interconnect.

The **switchboxes** provides a connection matrix between routing tracks in adjacent **routing channels**. Typically, an **FPGA** has a logical 2D representation. That is, the **FPGA** is designed in a manner that provides a 2D abstraction for computation. This is often called an “island-style” architecture where the **slices** represent “logic islands” that are connected using the **routing channels** and **switchboxes**. Here the **switchbox** connects to four **routing channels** to the north, east, south,

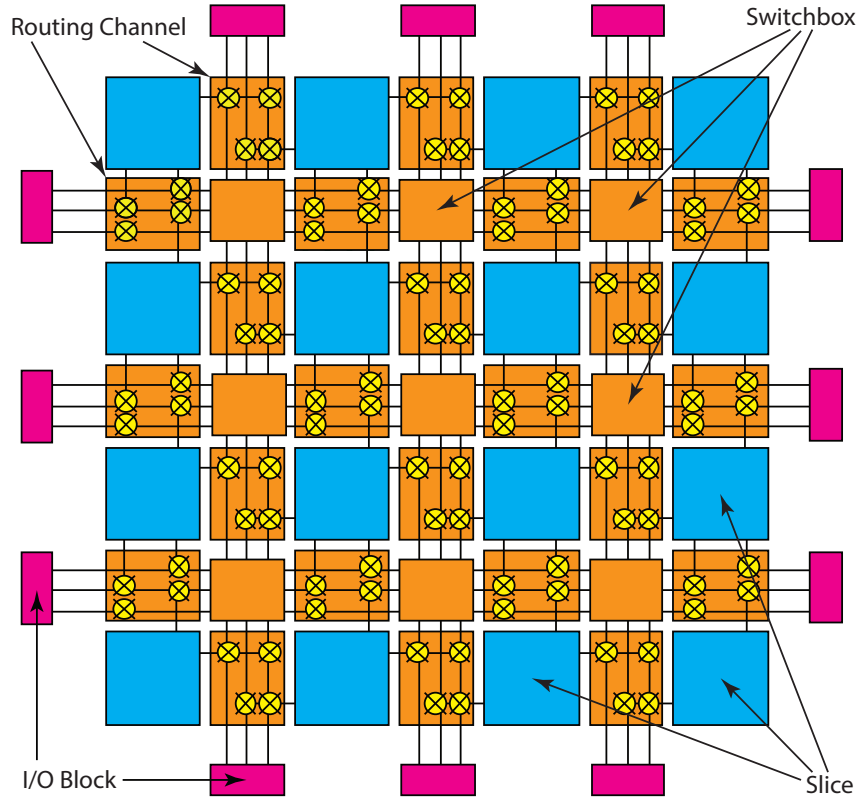


Figure 1.3: The two-dimensional structure of an **FPGA** showing an island style architecture. The logic and memory resources in the **slices** are interconnected using **routing channels** and **switchboxes**. The input/output (I/O) blocks provide an external interface, e.g., to a memory, microprocessor, sensor, and/or actuator. On some FPGAs, the I/O directly connects to the chip pins. Other FPGAs use the I/O to connect the programmable logic fabric to on-chip resources (e.g., a microprocessor bus or cache).

and west directions. The exact programming of the switches in the **routing channels** and **switchboxes** determines how the inputs and outputs of the programmable logic blocks are connected. The number of channels, the connectivity of the switchboxes, the structure of the slice, and other logic and circuit level FPGA architectural techniques are very well studied; we refer the interested reader to the following books and surveys on this topic for more information [12, 10, 30]. Generally, it is not necessary to understand all of the nitty-gritty details of the FPGA architecture in order to successfully use the **HLS** tools, rather it is more important to have a general understanding of the various FPGA resources and how the **HLS** optimizations effect the resource usage.

Figure 1.3 provides an even more abstract depiction of an **FPGA** programmable logic and interconnect. This provides a larger view of the 2D dimensional layout of the programmable logic (e.g., **slices**), **routing channels**, and **switchboxes**. The **FPGA** programmable logic uses **I/O blocks** to communicate with an external device. This may be a microcontroller (e.g., an on-chip ARM processor using an AXI interface), memory (e.g., an on-chip cache or an off-chip DRAM memory controller), a sensor (e.g., an antenna through an A/D interface), or an actuator (e.g., a motor through a D/A interface). More recently, **FPGAs** have integrated custom on-chip I/O handlers, e.g., memory controllers, transceivers, or analog-to-digital (and vice versa) controllers directly into

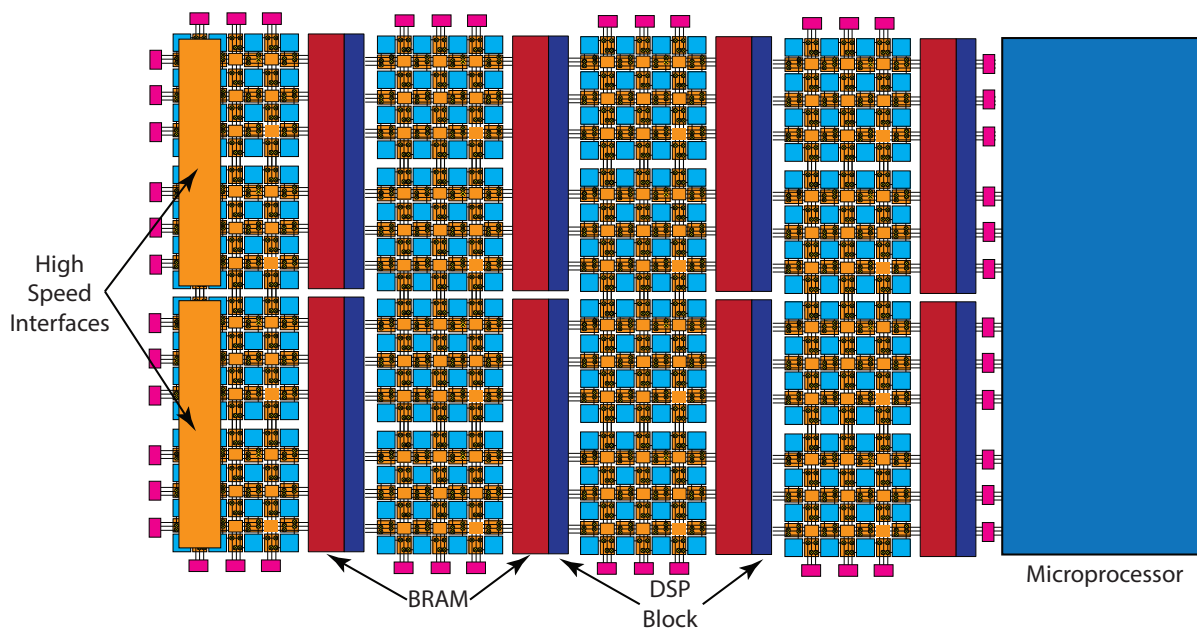


Figure 1.4: Modern **FPGAs** are becoming more heterogenous with a mix of programmable logic elements and “hardened” architectural elements like register files, custom datapaths, and high speed interconnect. The **FPGA** is often paired with one or more microprocessors, e.g., ARM or x86 cores, that coordinates the control of the system.

the fabric in order to increase performance.

As the number of transistors on the FPGA has continued to increase, FPGA architectures have begun to incorporate more and more “hard” resources. These are custom resources designed specifically to perform a task. For example, many applications make heavy use of addition and multiplication operations. Thus, the FPGA fabric added custom resources targeted at these operations. An example of this is the DSP48 custom datapaths, which efficiently implement a series of arithmetic operations including multiplication, addition, multiply-accumulate, and word level logical operations. These DSP48 blocks have some programmability, but are not as flexible as the programmable logic. Yet, implementing a multiply or MAC operation on these DSP48s is much more efficient than performing the same operation on the programmable logic. Thus, there is a fundamental tradeoff of efficiency versus flexibility. Modern FPGAs will have hundreds to thousands of these DSP48 distributed throughout the logic fabric as shown in Figure 1.4.

Compare the performance of a multiply accumulate operation using the programmable logic versus the DSP48s. What is the maximum frequency that one can obtain in both cases? How does the FPGA resource usage change?

A **block RAM (BRAM)** is another example of a hardened resource. **BRAMs** are configurable random access memory modules that support different memory layouts and interfaces. For example, they can be changed to have byte, half-word, word, and double word transfers and connected to a variety of different interfaces including local on-chip buses (for talking to the programmable fabric) and processor buses (to communicate with on-chip processors). Generally, these are used to transfer data between on-chip resources (e.g., the FPGA fabric and microprocessor) and store large data

	External Memory	BRAM	FFs
count	1-4	thousands	millions
size	GBytes	KBytes	Bits
total size	GBytes	MBytes	100s of KBytes
width	8-64	1-16	1
total bandwidth	GBytes/sec	TBytes/sec	100s of TBytes/sec

Figure 1.5: A comparison of three different on- and off-chip memory storage options. External memory provides the most density but has limited total bandwidth. Moving on-chip there are two options: **FFs** and **BRAMs**. **FFs** have the best total bandwidth but only a limited amount of total data storage capability. **BRAMs** provide an intermediate value between external memory and **FFs**.

sets on chip. We could choose to store the data set in the slices (using the **FFs**) but this would incur overheads in performance and resource usage.

A typical **BRAM** has around 32 Kbit of memory storage which can be configured as 32K x 1 bit, 16K x 2 bits, 8K x 4 bits, etc. They can be cascaded together to create larger memories. All of this configuration is done by the Vivado tools; this is a major advantage of Vivado® HLS: the designer does not need to worry about these low level details. The **BRAMs** are typically co-located next to the DSP48. For HLS, it may be beneficial to think of the **BRAMs** as configurable register files. These **BRAMs** can directly feed the custom datapaths (DSP48s), talk to on-chip microprocessors, and transfer data to custom datapaths implemented on the programmable logic.

Implement a design that stores a large (e.g., thousand item) array in BRAMs and programmable logic. How does the performance change? What about the resource usage?

Figure 1.5 provides a comparison between different on-chip and off-chip memory resources. There are millions of **FFs** on-chip and these provide hundreds of Kbytes of bit level storage. These can be read to and written to on every cycle and thus provide a tremendous amount of total bandwidth. Unfortunately, they do not provide the most efficient storage capacity. **BRAMs** provide a bit more storage density at the cost of limited total bandwidth. Only one or two entries of the **BRAMs** can be accessed during every cycle which is the major limiting factor for the bandwidth. Going even further in this direction, we can use very high density off-chip external memory, but the bandwidth is even further reduced. The decision about where to place your application's data is crucial and one that we will consider extensively throughout this book. The Vivado® HLS tool has many options that allow the designer to specify exactly where and how to store data.

As on-chip transistors have become more plentiful, we have the ability to consider integrating even more complex hardened resources. On-chip microprocessors are a prime example of this. High-end modern FPGAs can include four or more on-chip microprocessors (e.g., ARM cores). While only the largest FPGAs include four microprocessors, it is very common to see one microprocessor included in even the smaller FPGA devices. This provides the ability to run an operating system (e.g., Linux) and quickly leverage all of its facilities, e.g., to communicate with devices through drivers, run larger software packages like OpenCV, and use more common high level programming languages like Python to get the system up and running quickly. The microprocessor is often the controller for the system. It orchestrates data movement between off-chip memories, sensors, and actuators to on-chip resources like the **BRAMs**. And the microprocessor can coordinate between the custom IP cores developed with the **HLS** tools, third party IP cores, and the board level resources.

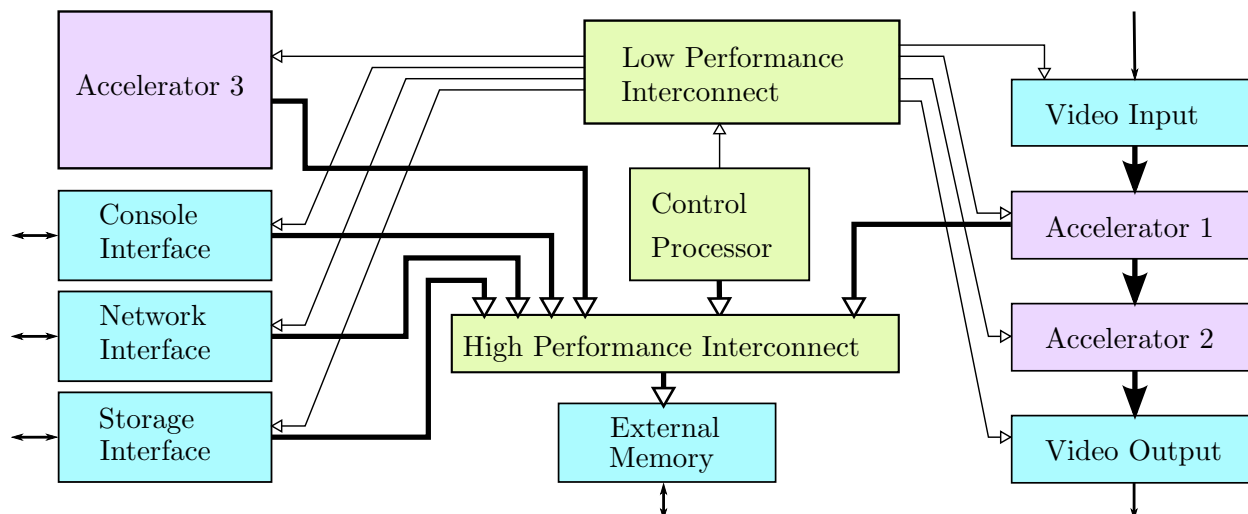


Figure 1.6: A block diagram showing a hypothetical embedded FPGA design, consisting of I/O interface cores (shown in blue), standard cores (shown in green), and application specific accelerator cores (shown in purple). Note that accelerator cores might have streaming interfaces (Accelerator 2), memory-mapped interfaces (Accelerator 3), or both (Accelerator 1).

1.3 FPGA Design Process

Given the complexity and size of modern FPGA devices, designers have looked to impose a higher-level structure on building designs. As a result, FPGA designs are often composed of components or **IP cores**, structured something like Figure 1.6. At the periphery of the design, close to the I/O pins, is a relatively small amount of logic that implements timing-critical I/O functions or protocols, such as a memory controller block, video interface core or analog-to-digital converter. This logic, which we will refer to as an *I/O interface core*, is typically implemented as structural RTL, often with additional timing constraints that describe the timing relationships between signals and the variability of these signals. These constraints must also take into account interference of signals propagating through traces in a circuit board and connectors outside of the FPGA. In order to implement high speed interfaces, these cores typically make use of dedicated logic in the FPGA architecture that is close to the I/O pins for serializing and deserializing data, recovering and distributing clock signals, and delaying signals with picosecond accuracy in order to repeatedly capture data in a register. The implementation of I/O interface cores is often highly customized for a particular FPGA architecture and hence are typically provided by FPGA vendors as reference designs or off-the-shelf components, so for the purposes of this book, we can ignore the detailed implementation of these blocks.

Away from I/O pins, FPGA designs often contain *standard cores*, such as processor cores, on-chip memories, and interconnect switches. Other standard cores include generic, fixed-function processing components, such as filters, FFTs, and codecs. Although instances of these cores are often parameterized and assembled in a wide variety of ways in a different designs, they are not typically the differentiating element in a customers design. Instead, they represent commodity, horizontal technology that can be leveraged in designs in many different application areas. As a result, they are often provided by an FPGA vendor or component provider and only rarely implemented by a system designer. Unlike interface I/O cores, standard cores are primarily synchronous circuits that require few constraints other than basic timing constraints specifying the clock period. As a

result, such cores are typically portable between FPGA families, although their circuit structure may still be highly optimized.

Lastly, FPGA designs typically contain customized, application-specific *accelerator cores*. As with standard cores, accelerator cores are primarily synchronous circuits that can be characterized by a clock period timing constraint. In contrast, however, they are almost inevitably constructed for a specific application by a system designer. These are often the “secret sauce” and used to differentiate your system from others. Ideally a designer can quickly and easily generate high-performance custom cores, perform a design space exploration about feasible designs, and integrate these into their systems in a short timeframe. This book will focus on the development of custom cores using HLS in a fast, efficient and high performance manner.

When integrating a design as in Figure 1.6, there are two common design methodologies. One methodology is to treat HLS-generated accelerator cores just like any other cores. After creating the appropriate accelerator cores using HLS, the cores are composed together (for instance, in a tool such as Vivado®IP Integrator) along with I/O interface cores and standard cores to form a complete design. This *core-based design methodology* is similar to the way that FPGA designs are developed without the use of HLS. A newer methodology focuses on standard design templates or platforms, which combine a stable, verified composition of standard cores and I/O interface cores targeting a particular board. This *platform-based design methodology* enables a high-level programmer to quickly integrate different algorithms or *roles* within the interfaces provided by a single platform or *shell*. It can also make it easier to port an accelerator from one platform to another as long as the shells support the same standardized interfaces.

1.4 Design Optimization

1.4.1 Performance Characterization

Before we can talk about optimizing a design, we need to discuss the key criterion that are used to characterize a design. The computation time is a particularly important metric for design quality. When describing synchronous circuits, one often will use the number of clock cycles as a measure of performance. However, this is not appropriate when comparing designs that have different clock rates, which is typically the case in HLS. For example, the clock frequency is specified as an input constraint to the Vivado® HLS, and it is feasible to generate different architectures for the same exact code by simply changing the target clock frequency. Thus, it is most appropriate to use seconds, which allows an apples-to-apples comparison between any HLS architecture. The Vivado® HLS tool reports the number of cycles and the clock frequency. These can be used to calculate the exact amount of time that some piece of code requires to compute.

It is possible to optimize the design by changing the clock frequency. The Vivado® HLS tool takes as input a target clock frequency, and changing this frequency target will likely result in the tool generating different implementations. We discuss this throughout the book. For example, Chapter 2.4 describes the constraints that are imposed on the Vivado® HLS tool depending on the clock period. Chapter 2.5 discusses how increasing the clock period can increase the throughput by employing operation chaining.

We use the term *task* to mean a fundamental atomic unit of behavior; this corresponds to a function invocation in Vivado® HLS. The *task latency* is the time between when a task starts and when it finishes. The *task interval* is the time between when one task starts and the next starts

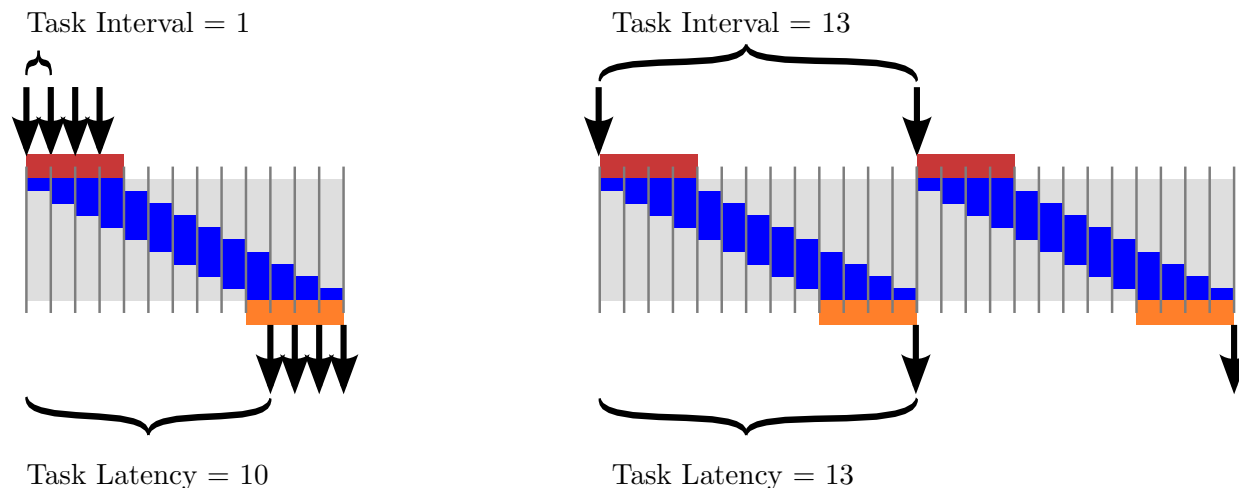


Figure 1.7: The task interval and task latency for two different designs. The left design is pipelined while the right one uses a more sequential implementation.

or the difference between the start times of two consecutive tasks. All task input, output, and computation is bounded by the task latency, but the start of a task may not coincide with the reading of inputs and the end of a task may not coincide with writing of outputs, particularly if the task has state that is passed from one task to the next. In many designs, **data rate** is a key design goal, and depends on both the **task interval** and the size of the arguments to the function.

Figure 1.7 shows two different designs of some hypothetical application. The horizontal axis represents time (increasing to the right) and the vertical axis represents different functional units in the design. Input operations are shaded red and output operations are shaded orange. Active operators are represented in dark blue and inactive operators are represented in light blue. Each incoming arrow represents the start of a task and each outgoing arrow represents the completion of a task. The diagram on the left represents four executions of an architecture that starts a new task every cycle. This corresponds to a ‘fully-pipelined’ implementation. On the right, there is task with a very different architecture, reading a block of four pieces of input data, processing it, and then producing a block of four output samples after some time. This architecture has the same latency and interval (13 cycles) and can only process one task at a time. This behavior is in contrast to the behavior of the pipelined design on the left, which has multiple tasks executing at any given instant of time. Pipelining in HLS is very similar to the concept of instruction pipelines in a microprocessor. However, instead of using a simple 5-stage pipeline where the results of an operation are usually written back to the register file before being executed on again, the Vivado® HLS tool constructs a circuit that is customized to run a particular program on a particular FPGA device. The tool optimizes the number of pipeline stages, the initiation interval (the time between successive data provided to the pipeline – similar to the task interval), the number and types of functional units and their interconnection based on a particular program and the device that is being targeted.

The Vivado® HLS tool counts cycles by determining the maximum number of registers between any input and output of a task. Thus, it is possible for a design to have a task latency of zero cycles, corresponding to a combinational circuit which has no registers in any path from input to output. Another convention is to count the input and/or output as a register and then find the maximum

registers on any path. This would result in a larger number of cycles. We use the Vivado® HLS convention throughout this book.

Note that many tools report the task interval as “throughput”. This terminology is somewhat counterintuitive since a longer task interval almost inevitably results in fewer tasks being completed in a given time and thus lower data rates at the interfaces. Similarly, many tools use “latency” to describe a relationship between reading inputs and writing outputs. Unfortunately, in designs with complex behavior and multiple interfaces, it is hard to characterize tasks solely in terms of inputs and outputs, e.g., a task may require multiple reads or writes from a memory interface.

1.4.2 Area/Throughput Tradeoffs

In order to better discuss some of the challenges that one faces when using an HLS tool, let’s consider a simple yet common hardware function – the **finite impulse response** (FIR) filter. An FIR performs a convolution on an input sequence with a fixed set of coefficients. An FIR is quite general – it can be used to perform different types of filter (high pass filter, low pass, band pass, etc.). Perhaps the most simple example of an FIR is a moving average filter. We will talk more background on FIR in Chapter 2 and describe many specific optimizations that can be done using HLS. But in the meantime just consider its implementation at a high level.

The C code in Figure 1.8 provides a functional or task description for HLS; this can be directly used as input to the Vivado® HLS tool, which will analyze it and produce a functionally equivalent RTL circuit. This is a complex process, and we will not get into much detail about this now, but think of it as a compiler like gcc. Yet instead of outputting assembly code, the HLS “compiler” creates an RTL hardware description. In both cases, it is not necessary to understand exactly how the compiler works. This is exactly why we have the compiler in the first place – to automate the design process and allow the programmer/designer to work at a higher level of abstraction. Yet at the same time, someone that knows more about how the compiler works will often be able to write more efficient code. This is particularly important for writing HLS code since there are many options for synthesizing the design that are not typically obvious to one that only knows the “software” flow. For example, ideas like custom memory layout, pipelining, and different I/O interfaces are important for HLS, but not for a “software compiler”. These are the concepts that we focus on in this book.

A key question to understand is: “What circuit is generated from this code?”. Depending on your assumptions and the capabilities of a particular HLS tool, the answer can vary widely. There are multiple ways that this could be synthesized by an HLS tool.

One possible circuit would execute the code sequentially, as would a simple RISC microprocessor. Figure 1.9 shows assembly code for the Xilinx Microblaze processor which implements the C code in Figure 1.8. Although this code has been optimized, many instructions must still be executed to compute array index expressions and loop control operations. If we assume that a new instruction can be issued each clock cycle, then this code will take approximately 49 clock cycles to compute one output sample of the filter. Without going into the details of how the code works, we can see that one important barrier to performance in this code is how many instructions can be executed in each clock cycle. Many improvements in computer architecture are fundamentally attempts to execute more complex instructions that do more useful work more often. One characteristic of HLS is that architectural tradeoffs can be made without needing to fit in the constraints of an instruction set architecture. It is common in HLS designs to generate architectures that issue


```
#define NUM_TAPS 4

void fir(int input, int *output, int taps[NUM_TAPS])
{
    static int delay_line[NUM_TAPS] = {};

    int result = 0;
    for (int i = NUM_TAPS - 1; i > 0; i--) {
        delay_line[i] = delay_line[i - 1];
    }
    delay_line[0] = input;

    for (int i = 0; i < NUM_TAPS; i++) {
        result += delay_line[i] * taps[i];
    }

    *output = result;
}
```

Figure 1.8: Code for a four tap FIR filter.

hundreds or thousands of RISC-equivalent instructions per clock with pipelines that are hundreds of cycles deep.

By default, the Vivado[®] HLS tool will generate an optimized, but largely sequential architecture. In a sequential architecture, loops and branches are transformed into control logic that enables the registers, functional units, and the rest of the data path. Conceptually, this is similar to the execution of a RISC processor, except that the program to be executed is converted into a finite state machine in the generated RTL rather than being fetched from the program memory. A sequential architecture tends to limit the number of functional units in a design with a focus on resource sharing over massive parallelism. Since such an implementation can be generated from most programs with minimal optimization or transformation, this makes it easy for users to get started with HLS. One disadvantage of a sequential architecture is that analyzing and understanding data rates is often difficult since the control logic can be complex. The control logic dictates the number of cycles for the task interval and task latencies. The control logic can be quite complex, making it difficult to analyze. In addition, the behavior of the control logic may also depend on the data being processed, resulting in performance that is data-dependent.

However, the Vivado[®] HLS tool can also generate higher performance pipelined and parallel architectures. One important class of architectures is called a *function pipeline*. A function pipeline architecture is derived by considering the code within the function to be entirely part of a computational data path, with little control logic. Loops and branches in the code are converted into unconditional constructs. As a result, such architectures are relatively simple to characterize, analyze, and understand and are often used for simple, high data rate designs where data is processed continuously. Function pipeline architectures are beneficial as components in a larger design since their simple behavior allows them to be resource shared like primitive functional units. One disadvantage of a function pipeline architecture is that not all code can be effectively parallelized.

The Vivado[®] HLS tool can be directed to generate a function pipeline by placing the

```

fir:
.frame r1,0,r15 # vars= 0, regs= 0, args= 0
.mask 0x00000000
addik r3,r0,delay_line.1450
lwi r4,r3,8 # Unrolled loop to shift the delay line
swi r4,r3,12
lwi r4,r3,4
swi r4,r3,8
lwi r4,r3,0
swi r4,r3,4
swi r5,r3,0 # Store the new input sample into the delay line
addik r5,r0,4 # Initialize the loop counter
addk r8,r0,r0 # Initialize accumulator to zero
addk r4,r8,r0 # Initialize index expression to zero
$L2:
muli r3,r4,4 # Compute a byte offset into the delay_line array
addik r9,r3,delay_line.1450
lw r3,r3,r7 # Load filter tap
lwi r9,r9,0 # Load value from delay line
mul r3,r3,r9 # Filter Multiply
addk r8,r8,r3 # Filter Accumulate
addik r5,r5,-1 # update the loop counter
bneid r5,$L2
addik r4,r4,1 # branch delay slot, update index expression

rtsd r15, 8
swi r8,r6,0 # branch delay slot, store the output
.end fir

```

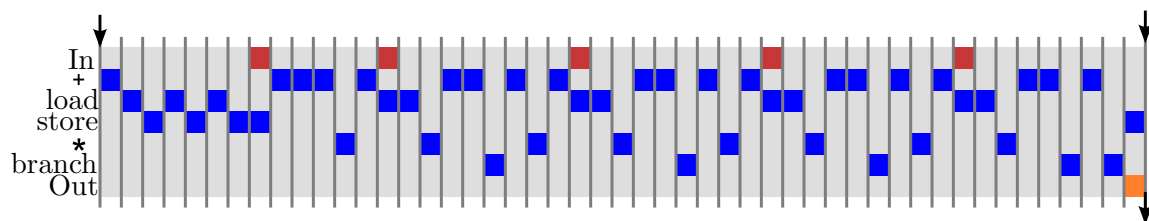


Figure 1.9: RISC-style assembly generated from the C code in Figure 1.8, targeting the Xilinx Microblaze processor. This code is generated using `microblazeel-xilinx-linux-gnu-gcc -O1 -mno-xl-soft-mul -S fir.c`

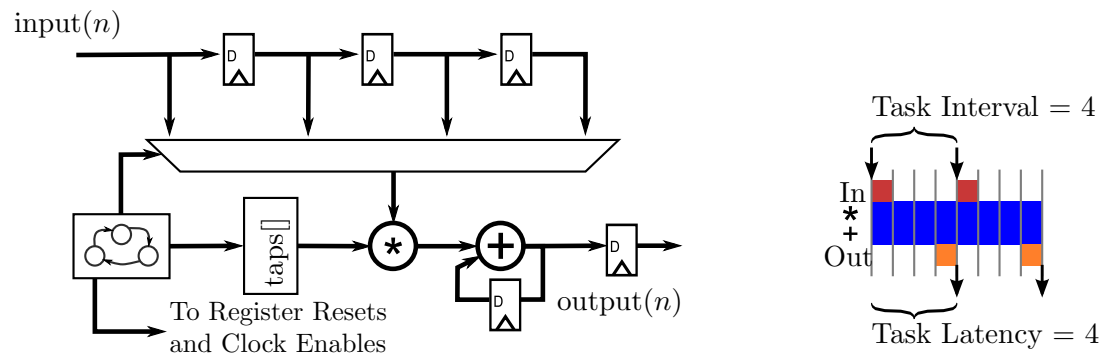


Figure 1.10: A “one tap per clock” architecture for an FIR filter. This architecture can be implemented from the code in Figure 1.8.

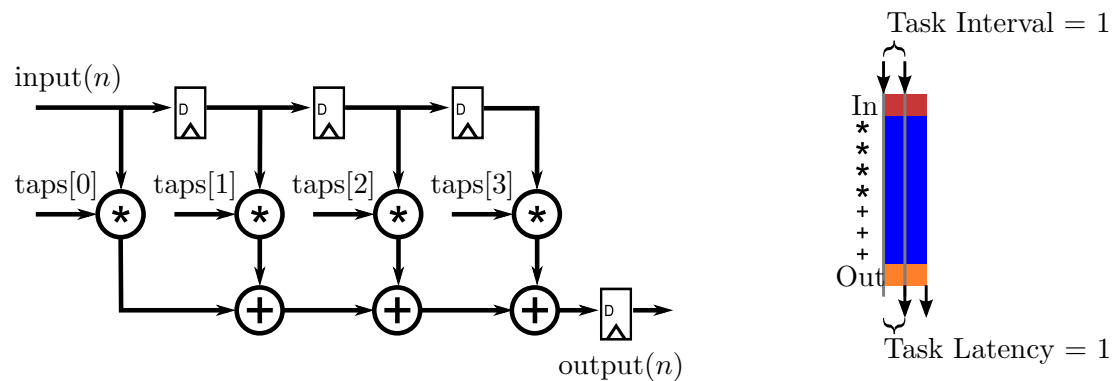


Figure 1.11: A “one sample per clock” architecture for an FIR filter. This architecture can be implemented from the code in Figure 1.8 using function pipeline.

`#pragma HLS pipeline` directive in the body of a function. This directive takes a parameter that can be used to specify the initiation interval of the the pipeline, which is the same as a task interval for a function pipeline. Figure 1.10 shows one potential design – a “one tap per clock” architecture, consisting of a single multiplier and single adder to compute the filter. This implementation has a task interval of 4 and a task latency of 4. This architecture can take a new sample to filter once every 4 cycles and will generate a new output 4 cycles after the input is consumed. The implementation in Figure 1.11 shows a “one sample per clock” architecture, consisting of 4 multipliers and 3 adders. This implementation has a task interval of 1 and a task latency of 1, which in this case means that the implementation accept a new input value every cycle. Other implementations are also possible, such as architectures with “two taps per clock” or “two samples per clock”, which might be more appropriate for a particular application. We discuss more ways to optimize the FIR function in depth in Chapter 2.

In practice, complex designs often include complicated tradeoffs between sequential architectures and parallel architectures, in order to achieve the best overall design. In Vivado[®] HLS, these tradeoffs are largely controlled by the user, through various tool options and code annotations, such as `#pragma` directive.

What would the task interval of a “two taps per clock” architecture for a 4 tap filter be? What about for a “two samples per clock” architecture?

1.4.3 Restrictions on Processing Rate

As we have seen, the task interval of a design can be changed by selecting different kinds of architectures, often resulting in a corresponding increase in the processing rate. However, it is important to realize that the task interval of any processing architecture is fundamentally limited in several ways. The most important limit arises from `recurrences` or feedback loops in a design. The other key limitation arises from *resource limits*.

A *recurrence* is any case where a computation by a component depends on a previous computation by the same component. A key concept is that *recurrences fundamentally limit the throughput of a design*, even in the presence of pipelining [56, 43]. As a result, analyzing recurrences in algorithms and generating hardware that is guaranteed to be correct is a key function of an HLS tool. Similarly, understanding algorithms and selecting those without tight recurrences is a important part of using HLS (and, in fact, parallel programming in general).

Recurrences can arrive in different coding constructs, such as static variables (Figure 1.8), sequential loops (Figure 1.10). Recurrences can also appear in a sequential architecture and disappear when pipelining is applied (as in Figures 1.10 and 1.11). In other cases, recurrences can exist without limiting the throughput of a sequential architecture, but can become problematic when the design is pipelined.

Another key factor that can limit processing rate are resource limitations. One form of resource limitation is associated with the wires at the boundary of a design, since a synchronous circuit can only capture or transmit one bit per wire per clock cycle. As a result, if a function with the signature `int32_t f(int32_t x);` is implemented as a single block at 100 MHz with a task interval of 1, then the most data that it can process is 3.2 Gbits/sec. Another form of resource limitation arises from memories since most memories only support a limited number of accesses per clock cycle. Yet another form of resource limitation comes from user constraints. If a designer limits the number of operators that can instantiated during synthesis, then this places a limit on the processing rate of the resulting design.

```
#define NUM_TAPS 4

void block_fir(int input[256], int output[256], int taps[NUM_TAPS],
int delay_line[NUM_TAPS]) {
    int i, j;
    for (j = 0; j < 256; j++) {
        int result = 0;
        for (i = NUM_TAPS - 1; i > 0; i--) {
            #pragma HLS unroll
            delay_line[i] = delay_line[i - 1];
        }
        delay_line[0] = input;

        for (i = 0; i < NUM_TAPS; i++) {
            #pragma HLS pipeline
            result += delay_line[i] * taps[i];
        }
        output[j] = result;
    }
}
```

Figure 1.12: Alternative code implementing an FIR filter.

1.4.4 Coding Style

Another key question you should ask yourself is, “Is this code the best way to capture the algorithm?”. In many cases, the goal is not only the highest quality of results, but maintainable and modifiable code. Although this is somewhat a stylistic preference, coding style can sometimes limit the architectures that a HLS tool can generate from a particular piece of code.

For instance, while a tool might be able to generate either architecture in Figure 1.11 or 1.10 from the code in Figure 1.8, adding additional directives as shown in Figure 1.12 would result in a specific architecture. In this case the delay line is explicitly unrolled, and the multiply-accumulate **for** loop is stated to be implemented in a pipelined manner. This would direct the HLS tool to produce an architecture that looks more like the pipelined one in Figure 1.11.

The chapter described how to build filters with a range of different processing rates, up to one sample per clock cycle. However, many designs may require processing data at a higher rate, perhaps several samples per clock cycle. How would you code such a design? Implement a 4 samples per clock cycle FIR filter. How many resources does this architecture require (e.g., number of multipliers and adders)? Which do you think will use more FPGA resources: the 4 samples per clock or the 1 sample per clock cycle design?

We look further into the optimization of an FIR function in more detail in Chapter 2. We discuss how to employ different optimizations (pipelining, unrolling, bitwidth optimization, etc.), and describe their effects on the performance and resource utilization of the resulting generated architectures.

1.5 Restructured Code

Writing highly optimized synthesizable HLS code is often not a straightforward process. It involves a deep understanding of the application at hand, the ability to change the code such that the Vivado® HLS tool creates optimized hardware structures and utilizes the directives in an effective manner.

Throughout the rest of the book, we walk through the synthesis of a number of different application domains – including digital signal processing, sorting, compression, matrix operations, and video processing. In order to get the most efficient architecture, it is important to have a deep understanding of the algorithm. This enables optimizations that require rewriting the code rather than just adding synthesis directives – a processing that we call code restructuring.

Restructured code maps well into hardware, and often represents the eccentricities of the tool chain, which requires deep understanding of micro-architectural constructs in addition to the algorithmic functionality. Standard, off-the-shelf code typically yields very poor quality of results that are orders of magnitude slower than CPU designs, even when using HLS directives such as pipelining, unrolling, and data partitioning. Thus, it is important to understand how to write code that the Vivado® HLS will synthesize in an optimal manner.

Restructured code typically differs substantially from a software implementation – even one that is highly optimized. A number of studies suggest that restructuring code is an essential step to generate an efficient FPGA design [46, 47, 15, 14, 39]. Thus, in order to get an efficient hardware design, the user must write restructured code with the underlying hardware architecture in mind. Writing restructured code requires significant hardware design expertise and domain specific knowledge.

Throughout the rest of this book, we go through a number of different applications, and show how to restructure the code for a more efficient hardware design. We present applications such as finite impulse response (FIR), discrete Fourier transform (DFT), fast Fourier transform (FFT), sparse matrix vector multiply (SpMV), matrix multiplication, sorting, and Huffman encoding. We discuss the impact of restructured code on the final hardware generated from high-level synthesis. And we propose a restructuring techniques based on best practices. In each chapter, we aim to:

1. Highlight the importance of restructuring code to obtain FPGA designs with good quality of result, i.e., a design that has high performance and low area usage;
2. Provide restructured code for common applications;
3. Discuss the impact of the restructuring on the underlying hardware; and
4. Perform the necessary HLS directives to achieve the best design

Throughout the book, we use example applications to show how to move from a baseline implementation and restructure the code to provide more efficient hardware design. We believe that the optimization process is best understood through example. Each chapter performs a different set of optimization directives including pipelining, dataflow, array partitioning, loop optimizations, and bitwidth optimization. Additionally, we provide insight on the skills and knowledge necessary to perform the code restructuring process.

1.6 Book Organization

We organized this book to teach by example. Each chapter presents an application (or application domain) and walks through its implementation using different HLS optimizations. Generally, each

chapter focuses on a limited subset of optimization techniques. And the application complexity generally increases in the later chapters. We start with a relatively simple to understand finite impulse response (FIR) filter in Chapter 2 and move on to implement complete video processing systems in Chapter 9.

There are of course benefits and drawbacks to this approach. The benefits are: 1) the reader can see how the optimizations are directly applicable to an application, 2) each application provides an exemplar of how to write HLS code, and 3) while simple to explain, toy examples can be hard to generalize and thus do not always make the best examples.

The drawbacks to the teach by example approach are: 1) most applications requires some background to give the reader a better understanding of the computation being performed. Truly understanding the computation often necessitates an extensive discussion on the mathematical background on the application. For example, implementing the best architecture for the fast Fourier transform (FFT) requires that the designer have deep understanding of the mathematical concepts behind a discrete Fourier transform (DFT) and FFT. Thus, some chapters, e.g., Chapter 4 (DFT) and Chapter 5 (FFT), start with a non-trivial amount of mathematical discussion. This may be off-putting to a reader simply looking to understand the basics of HLS, but we believe that such a deep understanding is necessary to understand the code restructuring that is necessary to achieve the best design. 2) some times a concept could be better explained by a toy example that abstracts away some of the non-important application details.

The organization for each chapter follows the same general pattern. A chapter begins by providing a background on the application(s) under consideration. In many cases, this is straightforward, e.g., it is not too difficult to explain matrix multiplication (as in Chapter 7) while the DFT requires a substantial amount of discussion (Chapter 4). Then, we provide a *baseline implementation* – a functionally correct but unoptimized implementation of the application using Vivado® HLS. After that, we perform a number of different optimizations. Some of the chapters focus on a small number of optimizations (e.g., Chapter 3 emphasizes bitwidth optimizations) while others look at a broad range of optimizations (e.g., Chapter 2 on FIR filters). The key optimizations and design methods are typically introduced in-depth in one chapter and then used repeatedly in the subsequent chapters.

The book is made to be read sequentially. For example, Chapter 2 introduces most of the optimizations and the later chapters provide more depth on using these optimizations. The applications generally get more complex throughout the book. However, each chapter is relatively self-contained. Thus, a more advanced HLS user can read an individual chapter if she only cares to understand a particular application domain. For example, a reader interested in generating a hardware accelerated sorting engine can look at Chapter 10 without necessarily have to read all of the previous chapters. This is another benefit of our teach by example approach.

Table 1.1 provides an overview of the types of optimization and the chapters where they are covered in at least some level of detail. Chapter 2 provides a gentle introduction the HLS design process. It overviews several different optimizations, and shows how they can be used in the optimization of a FIR filter. The later chapters go into much more detail on the benefits and usage of these optimizations.

The next set of chapters (Chapters 3 through 5) build digital signal processing blocks (CORDIC, DFT, and FFT). Each of these chapters generally focuses on one optimization: bitwidth optimization (Chapter 3), array optimizations (Chapter 4, and task pipelining (Chapter 5). For example, Chapter 3 gives an in-depth discussion on how to perform bitwidth optimizations on the CORDIC application. Chapter 4 provides an introduction to array optimizations, in particular, how to perform array partitioning in order to increase the on-chip memory bandwidth. This chapter also talks about loop unrolling and pipelining, and the need to perform these three optimizations in concert.

Table 1.1: A map describing the types of HLS optimizations and the chapters that discuss the concepts beyond them.

Chapter	FIR 2	CORDIC 3	DFT 4	FFT 5	SPMV 6	MatMul 7	Histogram 8	Video 9	Sorting 10	Huffman 11
Loop Unrolling	X		X	X	X		X		X	
Loop Pipelining	X		X	X	X		X	X	X	X
Bitwidth Optimization	X	X								X
Function Inlining	X									X
Hierarchy	X			X			X	X	X	X
Array Optimizations			X	X	X	X	X	X	X	X
Task Pipelining				X			X	X	X	X
Testbench					X	X			X	X
Co-simulation					X					
Streaming						X		X	X	
Interfacing								X		

Chapter [5](#) describes the optimizations of the FFT, which itself is a major code restructuring of the DFT. Thus, the chapter gives a background on how the FFT works. The FFT is a staged algorithm, and thus highly amenable to task pipelining. The final optimized FFT code requires a number of other optimizations including loop pipelining, unrolling, and array optimizations. Each of these chapters is paired with a project from the Appendix. These projects lead the design and optimization of the blocks, and the integration of these blocks into wireless communications systems.

Chapters [6](#) through [11](#) provide a discussion on the optimization of more applications. Chapter [6](#) describes how to use a testbench and how to perform RTL co-simulation. It also describes array and loop optimizations; these optimizations are common and thus are used in the optimization of many applications. Chapter [7](#) introduces the streaming optimization for data flow between tasks. Chapter [8](#) presents two applications (prefix sum and histogram) that are relatively simple, but requires careful code restructuring in order to create the optimal architecture. Chapter [9](#) talks extensively about how to perform different types interfacing, e.g., with a video stream using different bus and memory interfaces. As the name implies, the video streaming requires the use of the stream primitive, and extensive usage of loop and array optimizations. Chapter [10](#) goes through a couple of sorting algorithms. These require a large number of different optimizations. The final chapter creates a complex data compression architecture. It has a large number of complex blocks that work on a more complex data structure (trees).

Chapter 2

Finite Impulse Response (FIR) Filters

2.1 Overview

Finite Impulse Response (FIR) filters are commonplace in digital signal processing (DSP) applications – they are perhaps the most widely used operation in this domain. They are well suited for hardware implementation since they can be implemented as a highly optimized architecture. A key property is that they are a linear transform on contiguous elements of a signal. This maps well to data structures (e.g., FIFOs or tap delay lines) that can be implemented efficiently in hardware. In general, streaming applications tend to map well to FPGAs, e.g., most of the examples that we present throughout the book have some sort of streaming behavior.

Two fundamental uses for a filter are signal restoration and signal separation. Signal separation is perhaps the more common use case: here one tries to isolate the input signal into different parts. Typically, we think of these as different frequency ranges, e.g., we may want perform a low pass filter in order remove high frequencies that are not of interest. Or we may wish to perform a band pass filter to determine the presence of a particular frequency in order to demodulate it, e.g., for isolating tones during frequency shift keying demodulation. Signal restoration relates to removing noise and other common distortion artifacts that may have been introduced into the signal, e.g., as data is being transmitted across the wireless channel. This includes smoothing the signal and removing the DC component.

Digital FIR filters often deal with a discrete signal generated by sampling a continuous signal. The most familiar sampling is performed in time, i.e., the values from a signal are taken at discrete instances. These are most often sampled at regular intervals. For instance, we might sample the voltage across an antenna at a regular interval with an analog-to-digital converter. Alternatively we might sample the current created by a photo-diode to determine the light intensity. Alternatively, samples may be taken in space. For instance, we might sample the value of different locations in an image sensor consisting of an array of photo-diodes to create a digital image. More in-depth descriptions of signals and sampling can be found in [\[41\]](#).

The format of the data in a sample changes depending upon the application. Digital communications often uses complex numbers (in-phase and quadrature or I/Q values) to represent a sample. Later in this chapter we will describe how to design a complex FIR filter to handle such data. In image processing we often think of a pixel as a sample. A pixel can have multiple fields, e.g., red, green, and blue (RGB) color channels. We may wish to filter each of these channels in a different way again depending upon the application.

The goal of this chapter is to provide a basic understanding of the process of taking an algorithm and creating a good hardware design using high-level synthesis. The first step in this process

is always to have a deep understanding of the algorithm itself. This allows us to make design optimizations like code restructuring much more easily. The next section provides an understanding of the FIR filter theory and computation. The remainder of the chapter introduces various HLS optimizations on the FIR filter. These are meant to provide an overview of these optimizations. Each of them will be described in more depth in subsequent chapters.

2.2 Background

The output signal of a filter given an impulse input signal is its *impulse response*. The impulse response of a linear, time invariant filter contains the complete information about the filter. As the name implies, the impulse response of an FIR filter (a restricted type of linear, time invariant filter) is finite, i.e., it is always zero far away from zero. Given the impulse response of an FIR filter, we can compute the output signal for any input signal through the process of *convolution*. This process combines samples of the impulse response (also called *coefficients* or *taps*) with samples of the input signal to compute samples of the output signal. The output of filter can be computed in other ways (for instance, in the frequency domain), but for the purposes of this chapter we will focus on computing in the time domain.

The convolution of an N-tap FIR filter with coefficients $h[]$ with an input signal $x[]$ is described by the general difference equation:

$$y[i] = \sum_{j=0}^{N-1} h[j] \cdot x[i - j] \quad (2.1)$$

Note that to compute a single value of the output of an N-tap filter requires N multiplies and N-1 additions.

Moving average filters are a simple form of lowpass FIR filter where all the coefficients are identical and sum to one. For instance in the case of the three point moving filter, the coefficients are $h = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. It is also called a *box car filter* due to the shape of its convolution kernel. Alternatively, you can think of a moving average filter as taking the average of several adjacent samples of the input signal and averaging them together. We can see that this equivalence by substituting $1/N$ for $h[j]$ in the convolution equation above and rearranging to arrive at the familiar equation for an average of N elements:

$$y[i] = \frac{1}{N} \sum_{j=0}^N x[i - j] \quad (2.2)$$

Each sample in the output signal can be computed by the above equation using $N - 1$ additions and one final multiplication by $1/N$. Even the final multiplication can often be regrouped and merged with other operations. As a result, moving average filters are simpler to compute than a general FIR filter. Specifically, when $N = 3$ we perform this operation to calculate $y[12]$:

$$y[12] = \frac{1}{3} \cdot (x[12] + x[11] + x[10]) \quad (2.3)$$

This filter is *causal*, meaning that the output is a function of no future values of the input. It is possible and common to change this, for example, so that the average is centered on the current sample, i.e., $y[12] = \frac{1}{3} \cdot (x[11] + x[12] + x[13])$. While fundamentally causality is an important property for system analysis, it is less important for a hardware implementation as a finite non-causal filter can be made causal with buffering and/or reindexing of the data.

Moving average filters can be used to smooth out a signal, for example to remove random (mostly high frequency) noise. As the number of taps N gets larger, we average over a larger number of samples, and we correspondingly must perform more computations. For a moving average filter, larger values of N correspond to reducing the bandwidth of the output signal. In essence, it is acting like a low pass filter (though not a very optimal one). Intuitively, this should make sense. As we average over larger and larger number of samples, we are eliminating higher frequency variations in the input signal. That is, “smoothing” is equivalent to reducing higher frequencies. The moving average filter is optimal for reducing white noise while keeping the sharpest step response, i.e., it creates the lowest noise for a given edge sharpness.

Note that in general, filter coefficients can be crafted to create many different kinds of filters: low pass, high pass, band pass, etc.. In general, a larger value of number of taps provides more degrees of freedom when designing a filter, generally resulting in filters with better characteristics. There is substantial amount of literature devoted to generating filter coefficients with particular characteristics for a given application. When implementing a filter, the actual values of these coefficients are largely irrelevant and we can ignore how the coefficients themselves were arrived at. However, as we saw with the moving average filter, the structure of the filter, or the particular coefficients can have a large impact on the number of operations that need to be performed. For instance, symmetric filters have multiple taps with exactly the same value which can be grouped to reduce the number of multiplications. In other cases, it is possible to convert the multiplication by a known constant filter coefficient into shift and add operations [34]. In that case, the values of the coefficients can drastically change the performance and area of the filter implementation [52]. But we will ignore that for the time being, and focus on generating architectures that have constant coefficients, but do not take advantage of the values of the constants.

2.3 Base FIR Architecture

Consider the code for an 11 tap FIR filter in Figure 2.1. The function takes two arguments, an input sample x , and the output sample y . This function must be called multiple times to compute an entire output signal, since each time that we execute the function we provide one input sample and receive one output sample. This code is convenient for modeling a streaming architecture, since we can call it as many times as needed as more data becomes available.

The coefficients for the filter are stored in the `c[]` array declared inside of the function. These are statically defined constants. Note that the coefficients are symmetric. i.e., they are mirrored around the center value `c[5] = 500`. Many FIR filter have this type of symmetry. We could take advantage of it in order to reduce the amount of storage that is required for the `c[]` array.

The code uses **typedef** for the different variables. While this is not necessary, it is convenient for changing the types of data. As we discuss later, bit width optimization – specifically setting the number of integer and fraction bits for each variable – can provide significant benefits in terms of performance and area.

Rewrite the code so that it takes advantage of the symmetry found in the coefficients. That is, change `c[]` so that it has six elements (`c[0]` through `c[5]`). What changes are necessary in the rest of the code? How does this effect the number of resources? How does it change the performance?

The code is written as a streaming function. It receives one sample at a time, and therefore it must store the previous samples. Since this is an 11 tap filter, we must keep the previous 10

```
#define N 11
#include "ap_int.h"

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

void fir(data_t *y, data_t x) {
    coef_t c[N] = {
        53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53
    };
    static
        data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    Shift_Accum_Loop:
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

Figure 2.1: A functionally correct, but highly unoptimized, implementation of an 11 tap FIR filter.

samples. This is the purpose of the `shift_reg[]` array. This array is declared **static** since the data must be persistent across multiple calls to the function.

The **for** loop is doing two fundamental tasks in each iteration. First, it performs the multiply and accumulate operation on the input samples (the current input sample `x` and the previous input samples stored in `shift_reg[]`). Each iteration of the loop performs a multiplication of one of the constants with one of the sample, and stores the running sum in the variable `acc`. The loop is also shifting values through `shift_array`, which works as a FIFO. It stores the input sample `x` into `shift_array[0]`, and moves the previous elements “up” through the `shift_array`:

```
shift_array[10] = shift_array[9]
shift_array[9] = shift_array[8]
shift_array[8] = shift_array[7]
...
shift_array[2] = shift_array[1]
shift_array[1] = shift_array[0]
shift_array[0] = x
```

The label `Shift_Accum_Loop`: is not necessary. However it can be useful for debugging. The Vivado[®] HLS tool adds these labels into the views of the code.

After the `for` loop completes, the `acc` variable has the complete result of the convolution of the input samples with the FIR coefficient array. The final result is written into the function argument `y` which acts as the output port from this `fir` function. This completes the streaming process for computing one output value of an FIR.

This function does not provide an efficient implementation of a FIR filter. It is largely sequential, and employs a significant amount of unnecessary control logic. The following sections describe a number of different optimizations that improve its performance.

2.4 Calculating Performance

Before we get into the optimizations, it is necessary to define precise metrics. When deriving the performance of a design, it is important to carefully state the metric. For instance, there are many different ways of specifying how “fast” your design runs. For example, you could say that it operates at X bits/second. Or that it can perform Y operations/sec. Other common performance metrics specifically for FIR filters talk about the number of filter operations/second. Yet another metric is multiply accumulate operations: MACs/second. Each of these are related to one another, in some manner, but when comparing different implementations it is important to compare apples to apples. For example, directly comparing one design using bits/second to another using filter operations/second can be misleading; fully understanding the relative benefits of the designs requires that we compare them using the same metric. And this may require additional information, e.g., going from filter operations/second to bits/second requires information about the size of the input and output data.

All of the aforementioned metrics use seconds. High-level synthesis tools talk about the designs in terms of number of cycles, and the frequency of the clock. The frequency is inversely proportional to the time it takes to complete one clock cycle. Using them both gives us the amount of time in seconds to perform some operation. The number of cycles and the clock frequency are both

important: a design that takes one cycle, but with a very low frequency is not necessary better than another design that takes 10 clock cycles but operates at a much higher frequency.

The clock frequency is a complicated function that the Vivado® HLS tool attempts to optimize alongside the number of cycles. Note that it is possible to specify a target frequency to the Vivado® HLS tool. This is done using the `create_clock` tcl command. For example, the command `create_clock -period 5` directs the tool to target a clock period of 5 ns and equivalently a clock frequency of 200 MHz. Note that this is only a target clock frequency only and primarily affects how much operation chaining is performed by the tool. After generating RTL, the Vivado® HLS tool provides an initial timing estimate relative to this clock target. However, some uncertainty in the performance of the circuit remains which is only resolved once the design is fully place and routed.

While achieving higher frequencies are often critical for reaching higher performance, increasing the target clock frequency is not necessarily optimal in terms of an overall system. Lower frequencies give more leeway for the tool to combine multiple dependent operations in a single cycle, a process called *operation chaining*. This can sometimes allow higher performance by enabling improved logic synthesis optimizations and increasing the amount of code that can fit in a device. Improved operation chaining can also improve (i.e., lower) the initiation interval of pipelines with recurrences. In general providing a constrained, but not over constrained target clock latency is a good option. Something in the range of 5 – 10 ns is typically a good starting option. Once you optimize your design, you can vary the clock period and observe the results. We will describe operation chaining in more detail in the next section.

Because Vivado® HLS deals with clock frequency estimates, it does include some margin to account for the fact that there is some error in the estimate. The goal of this margin is to ensure enough timing slack in the design that the generated RTL can be successfully placed and routed. This margin can be directly controlled using the `set_clock_uncertainty` TCL command. Note that this command only affects the HLS generated RTL and is different from the concept of clock uncertainty in RTL-level timing constraints. Timing constraints generated by Vivado® HLS for the RTL implementation flow are solely based on the target clock period.

It is also necessary to put the task that you are performing in context with the performance metric that you are calculating. In our example, each execution of the `fir` function results in one output sample. But we are performing $N = 11$ multiply accumulate operations for each execution of `fir`. Therefore, if your metric of interest is MACs/second, you should calculate the task latency for `fir` in terms of seconds, and then divide this by 11 to get the time that it takes to perform the equivalent of one MAC operation.

Calculating performance becomes even more complicated as we perform pipelining and other optimizations. In this case, it is important to understand the difference between task interval and task latency. It is a good time to refresh your understanding of these two metrics of performance. This was discussed in Chapter [1.4](#). And we will continue to discuss how different optimizations effect different performance metrics.

2.5 Operation Chaining

Operation chaining is an important optimization that the Vivado® HLS performs in order to optimize the final design. It is not something that a designer has much control over, but it is important that the designer understands how this works especially with respect to performance. Consider the multiply accumulate operation that is done in a FIR filter tap. Assume that the `add` operation takes 2 ns to complete, and a `multiply` operation takes 3 ns. If we set the clock period to 1

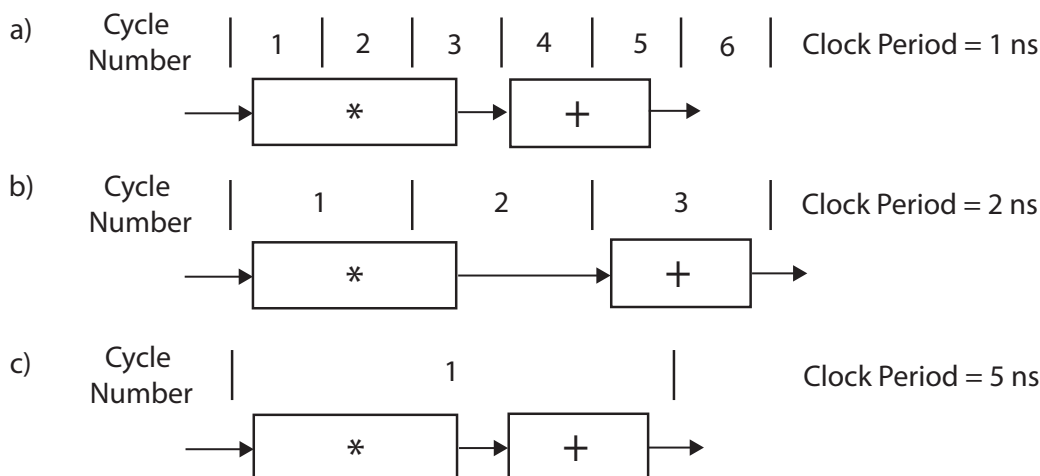


Figure 2.2: The performance of multiply accumulate operation changes depending upon the target clock period. Assume the multiply operation takes 3 ns and add operation takes 2 ns. Part a) has a clock period of 1 ns, and one MAC operation takes 5 cycles. Thus the performance is 200 million MACs/sec. Part b) has a clock period of 2 ns, and the MAC takes 3 cycles resulting in approximately 167 million MACs/sec. Part c) has a clock period of 5 ns. By using operation chaining, a MAC operation takes 1 cycle for a clock period of 200 million MACs/sec.

ns (or equivalently a clock frequency of 1 GHz), then it would take 5 cycles for the MAC operation to complete. This is depicted in Figure 2.2 a). The multiply operation is executed over 3 cycles, and the add operation is executed across 2 cycles. The total time for the MAC operation is 5 cycles \times 1 ns per cycle = 5 ns. Thus we can perform $1/5 \text{ ns} = 200$ million MACs/second.

If we increase the clock period to 2 ns, the multiply operation now spans over two cycles, and the add operation must wait until cycle 3 to start. It can complete in one cycle. Thus the MAC operation requires 3 cycles, so 6 ns total to complete. This allows us to perform approximately 167 million MACs/second. This result is lower than the previous result with a clock period of 1 ns. This can be explained by the “dead time” in cycle 2 where no operation is being performed.

However, it is not always true that increasing the clock period results in worse performance. For example, if we set the clock period to 5 ns, we can perform both the multiply and add operation in the same cycle using operation chaining. This is shown in Figure 2.2 c). Thus the MAC operation takes 1 cycle where each cycle is 5 ns, so we can perform 200 million MACs/second. This is the same performance as Figure 2.2 a) where the clock period is faster (1 ns).

So far we have performed chaining of only two operations in one cycle. It is possible to chain multiple operations in one cycle. For example, if the clock period is 10 ns, we could perform 5 add operations in a sequential manner. Or we could do two sequential MAC operations.

It should start to become apparent that the clock period plays an important role in how the Vivado[®] HLS tool optimizes the design. This becomes even more complicated with all of the other optimizations that the Vivado[®] HLS tool performs. It is not that important to fully understand the entire process of the Vivado[®] HLS tool. This is especially true since the tool is constantly being improved with each new release. However, it is important to have a good idea about how the tool may work. This will allow you to better comprehend the results, and even allow you to write more optimized code.

Although the Vivado[®] HLS can generate different different hardware for different target clock

```

Shift_Accum_Loop:
for (i = N - 1; i > 0; i--) {
    shift_reg[i] = shift_reg[i - 1];
    acc += shift_reg[i] * c[i];
}

acc += x * c[0];
shift_reg[0] = x;

```

Figure 2.3: Removing the conditional statement from the **for** loop creates a more efficient hardware implementation.

periods, overall performance optimization and determining the optimal target clock period still requires some creativity on the part of the user. For the most part, we advocate sticking within a small subset of clock periods. For example, in the projects we suggest that you set the clock period to 10 ns and focus on understanding how other optimizations, such as pipelining, can be used to create different architectures. This 100 MHz clock frequency is relatively easy to achieve, yet it provides a good first order result. It is certainly possible to create designs that run at a faster clock rate. 200 MHz and faster designs are possible but often require more careful balance between clock frequency targets and other optimization goals. You can change the target clock period and observe the differences in the performance. Unfortunately, there is no good rule to pick the optimal frequency.

Vary the clock period for the base FIR architecture (Figure 2.1) from 10 ns to 1 ns in increments of 1 ns. Which clock period provides the best performance? Which gives the best area? Why do you think this is the case? Do you see any trends?

2.6 Code Hoisting

The **if/else** statement inside of the **for** loop is inefficient. For every control structure in the code, the Vivado® HLS tool creates logical hardware that checks if the condition is met, which is executed in every iteration of the loop. Furthermore, this conditional structure limits the execution of the statements in either the **if** or **else** branches; these statements can only be executed after the **if** condition statement is resolved.

The **if** statement checks when $x == 0$, which happens only on the last iteration. Therefore, the statements within the **if** branch can be “hoisted” out of the loop. That is we can execute these statements after the loop ends, and then remove the **if/else** control flow in the loop. Finally, we must change the loop bounds from executing the “0th” iteration. This transform is shown in Figure 2.3. This shows just the changes that are required to the **for** loop.

The end results is a much more compact implementation that is ripe for further loop optimizations, e.g., unrolling and pipelining. We discuss those optimizations later.

Compare the implementations before and after the removal of the **if/else** condition done through loop hoisting. What is the difference in performance? How do the number of resources


```
TDL:
for (i = N - 1; i > 0; i--) {
    shift_reg[i] = shift_reg[i - 1];
}
shift_reg[0] = x;

acc = 0;
MAC:
for (i = N - 1; i >= 0; i--) {
    acc += shift_reg[i] * c[i];
}
```

Figure 2.4: A code snippet corresponding to splitting the **for** loop into two separate loops.

change?

2.7 Loop Fission

We are doing two fundamental operations within the **for** loop. The first part shifts the data through the `shift_reg` array. The second part performs the multiply and accumulate operations in order to calculate the output sample. *Loop fission* takes these two operations and implements each of them in their own loop. While it may not intuitively seem like a good idea, it allows us to perform optimizations separately on each loop. This can be advantageous especially in cases when the resulting optimizations on the split loops are different.

The code in Figure 2.4 shows the result of manual loop fission optimization. The code snippet splits the loop from Figure 2.3 into two loops. Note the label names for the two loops. The first is TDL and the second is MAC. *Tapped delay line (TDL)* is a common DSP term for the FIFO operation; MAC is short-hand for “multiply accumulate”.

Compare the implementations before and after loop fission. What is the difference in performance? How do the number of resources change?

Loop fission alone often does not provide a more efficient hardware implementation. However, it allows each of the loops to be optimized independently, which could lead to better results than optimizing the single, original **for** loop. The reverse is also true; merging two (or more) **for** loops into one **for** loop may yield the best results. This is highly dependent upon the application, which is true for most optimizations. In general, there is not a single ‘rule of thumb’ for how to optimize your code. There are many tricks of the trade, and your mileage may vary. Thus, it is important to have many tricks at your disposal, and even better, have a deep understanding of how the optimizations work. Only then will you be able to create the best hardware implementation. Let us continue to learn some additional tricks...

```

TDL:
for (i = N - 1; i > 1; i = i - 2) {
    shift_reg[i] = shift_reg[i - 1];
    shift_reg[i - 1] = shift_reg[i - 2];
}
if (i == 1) {
    shift_reg[1] = shift_reg[0];
}
shift_reg[0] = x;

```

Figure 2.5: Manually unrolling the TDL loop in the fir11 function.

2.8 Loop Unrolling

By default, the Vivado[®] HLS tool synthesizes **for** loops in a sequential manner. The tool creates a data path that implements one execution of the statements in the body of the loop. The data path executes sequentially for each iteration of the loop. This creates an area efficient architecture; however, it limits the ability to exploit parallelism that may be present across loop iterations.

Loop unrolling replicates the body of the loop by some number of times (called the *factor*). And it reduces the number of iterations of the loop by the same factor. In the best case, when none of the statements in the loop depend upon any of the data generated in the previous iterations, this can substantially increase the available parallelism, and thus enables an architecture that runs much faster.

The first **for** loop (with the label TDL) in Figure 2.4 shifts the values up through the `shift_reg` array. The loop iterates from largest value ($N-1$) to the smallest value ($i = 1$). By unrolling this loop, we can create a data path that executes a number of these shift operations in parallel.

Figure 2.5 shows the result of unrolling the loop by a factor of two. This code replicates the loop body twice. Each iteration of the loop now performs two shift operations. Correspondingly, we must perform half of the number of iterations.

Note that there is an additional **if** condition after the **for** loop. This is required in the case when the loop does not have an even number of iterations. In this case, we must perform the last “half” iteration by itself. The code in the **if** statement performs this last “half” iteration, i.e., moving the data from `shift_reg[0]` into `shift_reg[1]`.

Also note the effect of the loop unrolling on the **for** loop header. The decrement operation changes from `i--` to `i=i-2`. This is due to the fact that we are doing two times the “work” in each iteration, thus we should decrement by 2 instead of 1.

Finally, the condition for terminating the **for** loop changes from `i > 0` to `i > 1`. This is related to the fact that we should make sure that the “last” iteration can fully complete without causing an error. If the last iteration of the **for** loop executes when $i = 1$, then the second statement would try to read from `shift_reg[-1]`. Rather than perform this illegal operation, we do the final shift in the **if** statement after the **for** loop.

Write the code corresponding to manually unrolling this TDL **for** loop by a factor of three. How does this change the loop body? What changes are necessary to the loop header? Is the additional code in the **if** statement after the **for** loop still necessary? If so, how is it different?

Loop unrolling can increase the overall performance provided that we have the ability to execute some (or all) of the statements in parallel. In the unrolled code, each iteration requires that we read two values from the `shift_reg` array; and we write two values to the same array. Thus, if we wish to execute both statements in parallel, we must be able to perform two read operations and two write operations from the `shift_reg` array in the same cycle.

Assume that we store the `shift_reg` array in one BRAM, and that BRAM has two read ports and one write port. Thus we can perform two read operations in one cycle. But we must sequentialize the write operations across two consecutive cycles.

There are ways to execute these two statements in one cycle. For example, we could store all of the values of the `shift_reg` array in separate registers. It is possible to read and write to each individual register on every cycle. In this case, we can perform both of the statements in this unrolled **for** loop in one cycle. You can tell the Vivado[®] HLS tool to put all of the values in the `shift_reg` array into registers using the directive `#pragma HLS array_partition variable=shift_reg complete`. This is an important optimization, thus we discuss the **array_partition** directive in more detail later.

A user can tell the Vivado[®] HLS tool to automatically unroll the loop using the **unroll** directive. To automatically perform the unrolling done manually in Figure 2.5, we should put the directive `#pragma HLS unroll factor=2` into the body of the code, right after the **for** loop header. While we can always manually perform loop unrolling, it is much easier to allow the tool to do it for us. It makes the code easier to read; and it will result in fewer coding errors.

Unroll the TDL **for** loop automatically using the **unroll** directive. As you increase the unroll factor, how does this change the number of resources (FFs, LUTs, BRAMs, DSP48s, etc.)? How does it effect the throughput? What happens when you use the **array_partition** directive in conjunction with the **unroll** directive? What happens if you do not use the **unroll** directive?

Now, consider the second **for** loop (with the label MAC) in Figure 2.4. This loop multiplies a value from the array `c[]` with a value from the array `shift_array[]`. In each iteration it accesses the *i*th value from both arrays. And then it adds the result of that multiplication into the `acc` variable.

Each iteration of this loop performs one multiply and one add operation. Each iteration performs one read operation from array `shift_reg[]` and array `c[]`. The result of the multiplication of these two values is accumulated into the variable `acc`.

The load and multiplication operations are independent across all of the iterations of the **for** loop. The addition operation, depending on how it is implemented, may depend upon the values of the previous iterations. However, it is possible to unroll this loop and remove this dependency.

Figure 2.6 shows the code corresponding to unrolling the MAC **for** loop by a factor of four. The first **for** loop is the unrolled loop. The **for** loop header is modified in a similar manner to when we unrolled the TDL loop. The bound is changed to `i>=3`, and *i* is decremented by a factor of 4 for each iteration of the unrolled loop.

While there was loop carried dependency in the original, unrolled **for**, it is no longer present in the unrolled loop. The loop carried dependency came due to the `acc` variable; since the result of the multiply accumulate is written to this variable ever iteration, and we read from this register in every iteration (to perform the running sum), it creates a read-after-write (RAW) dependency across iterations. Note that there is not a dependency on the `acc` variable in the unrolled **for** loop due to the way this is written. Thus we are free to parallelize the four individual MAC operations in the unrolled **for** loop.

There is an additional **for** loop after the unrolled **for** loop. This is necessary to perform any partial iterations. Just like we required the **if** statement in the TDL, this performs any computations

```

acc = 0;
MAC:
for (i = N - 1; i >= 3; i -= 4) {
    acc += shift_reg[i] * c[i] + shift_reg[i - 1] * c[i - 1] +
    shift_reg[i - 2] * c[i - 2] + shift_reg[i - 3] * c[i - 3];
}

for (; i >= 0; i--) {
    acc += shift_reg[i] * c[i];
}

```

Figure 2.6: Manually unrolling the MAC loop in the `fir11` function by a factor of four.

on a potential last iteration. This occurs when the number of iterations in the original, unrolled **for** loop is not an even multiple of 4.

Once again, we can tell the Vivado[®] HLS tool to automatically unroll the loop by a factor of 4 by inserting the code `#pragma HLS unroll factor=4` into the MAC loop body.

By specifying the optional argument `skip_exit_check` in that directive, the Vivado[®] HLS tool will not add the final **for** loop to check for partial iterations. This is useful in the case when you know that the loop will never require these final partial iterations. Or perhaps performing this last few iterations does not have an (major) effect on the results, and thus it can be skipped. By using this option, the Vivado[®] HLS tool does not have to create that additional **for** loop. Thus the resulting hardware is simpler, and more area efficient.

The **for** loop is completely unrolled when no factor argument is specified. This is equivalent to unrolling by the maximum number of iterations; in this case a complete unrolling and unrolling by a factor of 11 is equivalent. In both cases, the loop body is replicated 11 times. And the loop header is unnecessary; there is no need to keep a counter or check if the loop exit condition is met. In order to perform a complete unrolling, the bounds of the loop must be statically determined, i.e., the Vivado[®] HLS tool must be able to know the number of iterations for the **for** loop at compile time.

Complete loop unrolling exposes a maximal amount of parallelism at the cost of creating an implementation that requires a significant amount of resources. Thus, it is ok to perform a complete loop unroll on “smaller” **for** loops. But completely unrolling a loop with a large number of iterations (e.g., one that iterates a million times) is typically infeasible. Often times, the Vivado[®] HLS tool will run for a very long time (and many times fail to complete after hours of synthesis) if the resulting loop unrolling creates code that is very large.

If your design does not synthesize in under 15 minutes, you should carefully consider the effect of your optimizations. It is certainly possible that large designs can take a significant amount of time for the Vivado[®] HLS tool to synthesize them. But as a beginning user, your designs should synthesize relatively quickly. If they take a long time, that most likely means that you used some directives that significantly expanded the code, perhaps in a way that you did not intend.

Synthesize a number of designs by varying the unroll factor for the MAC loop. How does the performance change? How does the unroll factor number affect the number of resources? Compare these results with the trends that you found by unrolling the TDL.

2.9 Loop Pipelining

By default, the Vivado[®] HLS tool synthesizes **for** loops in a sequential manner. For example, the **for** loop in Figure 2.1 will perform each iteration of the loop one after the other. That is, all of the statements in the second iteration happen only when all of the statements from the first iteration are complete; the same is true for the subsequent iterations. This happens even in cases when it is possible to perform statements from the iterations in parallel. In other cases, it is possible to start some of the statements in a later iteration before all of the statements in a former iteration are complete. This does not happen unless the designer specifically states that it should. This motivates the idea of *loop pipelining*, which allows for multiple iterations of the loop to execute concurrently.

Consider the MAC **for** loop from Figure 2.4. This performs one multiply accumulate (MAC) operation per iteration. This MAC **for** loop has four operations in the loop body:

- Read `c[]`: Load the specified data from the C array.
- Read `shift_reg[]`: Load the specified data from the `shift_reg` array.
- `*`: Multiply the values from the arrays `c[]` and `shift_reg[]`.
- `+`: Accumulate this multiplied result into the `acc` variable.

A schedule corresponding to one iteration of the MAC **for** loop is shown in Figure 2.7 a). The Read operations each require 2 cycles. This is due to the fact that the first cycle provides the address to the memory, and the data from the memory is delivered during the second cycle. These two Read operations can be done in parallel since there are no dependencies between them. The `*` operation can begin in Cycle 2; assume that it takes three cycles to complete, i.e., it is finished in Cycle 4. The `+` operation is chained to start and complete during Cycle 4. The entire body of the MAC **for** loop takes 4 cycles to complete.

There are a number of performance metrics associated with a **for** loop. The *iteration latency* is the number of cycles that it takes to perform one iteration of the loop body. The iteration latency for this MAC **for** loop is 4 cycles. The *for loop latency* is the number of cycles required to complete the entire execution of the loop. This includes time to calculate the initialization statement (e.g., `i = 0`), the condition statement (e.g., `i >= 0`), and the increment statement (e.g., `i++`). Assuming that these three header statements can be done in parallel with the loop body execution, the Vivado[®] HLS tool reports the latency of this MAC **for** loop as 44 cycles. This is the number of iterations (11) multiplied by the iteration latency (4 cycles) plus one additional cycle to determine that the loop should stop iterating. And then you subtract one. Perhaps the only strange thing here is the “subtract 1”. We will get to that in a second. But first, there is one additional cycle that is required at the beginning of the next iteration, which checks if the condition statement is satisfied (it is not) and then exits the loop. Now the “subtract 1”: Vivado[®] HLS determines the latency as the cycle in which the output data is ready. In this case, the final data is ready during Cycle 43. This would be written into a register at the end of Cycle 43 and correspondingly the

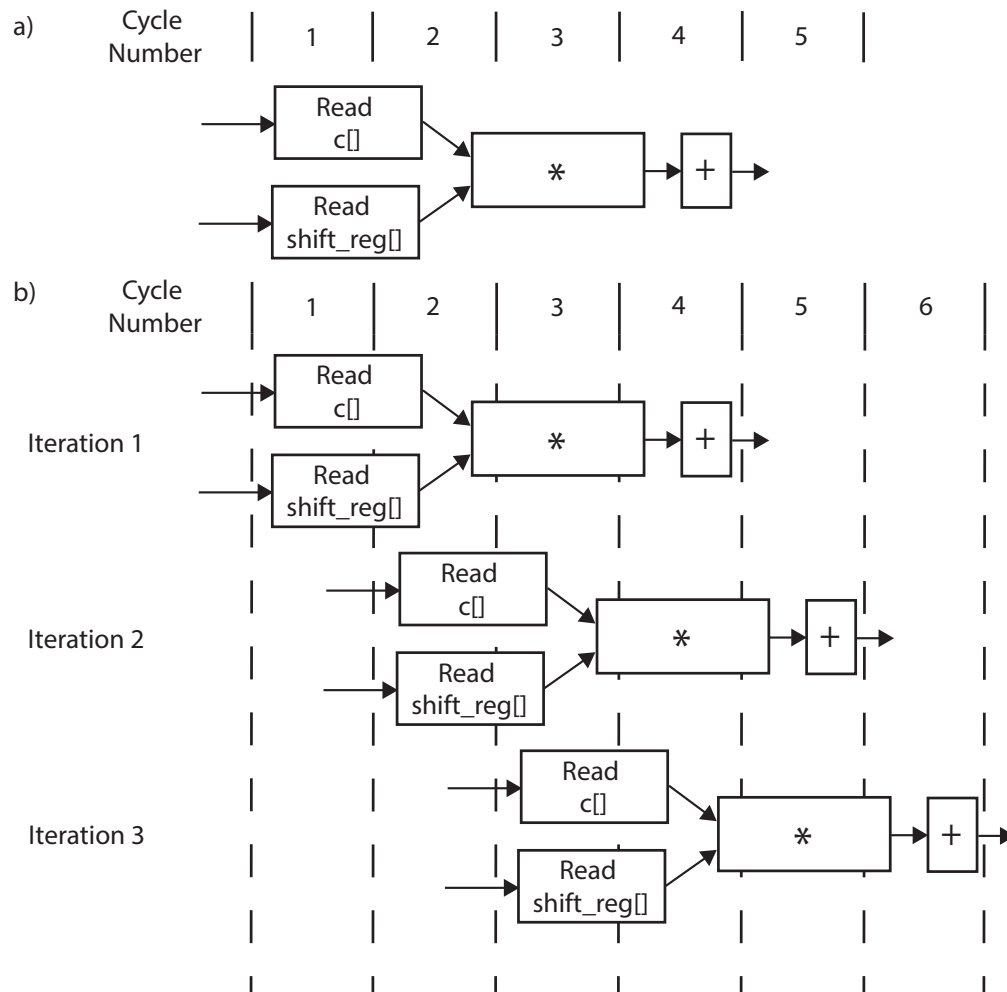


Figure 2.7: Part a) shows a schedule for the body of the MAC **for** loop. Part b) shows the schedule for three iterations of a pipelined version of the MAC **for** loop.

beginning of Cycle 44. Another way to think of this is that the latency is equal to the maximum number of registers between the input data and the output data.

Loop pipelining is an optimization that overlaps multiple iterations of a **for** loop. Figure 2.7 b) provides an example of pipelining for the MAC **for** loop. The figure shows three iterations of the **for** which are executed simultaneously. The first iteration is equivalent to the non-pipelined version as depicted in Figure 2.7 a). The difference is the start times of the subsequent iterations. In the non-pipelined version, the second iteration begins after the first iteration is completed, i.e., in Cycle 5. However, the pipelined version can start the subsequent iteration before the previous iterations complete. In the figure, Iteration 2 starts at Cycle 2, and Iteration 3 starts at Cycle 3. The remaining iterations start every consecutive cycle. Thus, the final iteration, Iteration 11, would start at Cycle 11 and it would complete during Cycle 14. Thus, the loop latency is 14.

The *loop initiation interval (II)* is another important performance metric. It is defined as the number of clock cycles until the next iteration of the loop can start. In our example, the loop II is 1, which means that we start a new iteration of the loop every cycle. This is graphically depicted in Figure 2.7 b). The II can be explicitly set using the directive. For example, the directive `#pragma HLS pipeline II=2` informs the Vivado[®] HLS tool to attempt to set the II=2. Note that this may not always be possible due to resource constraints and/or dependencies in the code. The output reports will tell you exact what the Vivado[®] HLS tool was able to achieve.

Explicitly set the loop initiation interval starting at 1 and increasing in increments of 1 cycle. How does increasing the II effect the loop latency? What are the trends? At some point setting the II to a larger value does not make sense. What is that value in this example? How do you describe that value for a general **for** loop?

Any **for** loop can be pipelined, so let us now consider the TDL **for** loop. This **for** loop has a similar header to the MAC **for** loop. The body of the loop performs an element by element shift of data through the array as described in Section 2.3. There are two operations: one Read and one Write to the `shift_reg` array. The iteration latency of this loop is 2 cycles. The Read operation takes two cycles, and the Write operation is performed at the end of Cycle 2. The **for** loop latency for this non-pipelined loop is 20 cycles.

We can pipeline this loop by inserting the directive `#pragma HLS pipeline II=1` after the loop header. The result of the synthesis is a loop initiation interval equal to 1 cycle. This means that we can start a loop iteration every cycle.

By modifying the example slightly, we can demonstrate a scenario where the resource constraints do not allow the Vivado[®] HLS tool to achieve an II=1. To do this, we explicitly set the type of memory for the `shift_reg` array. By not specifying the resource, we leave it up to the Vivado[®] HLS tool to decide. But we can specify the memory using a directive, e.g., the directive `#pragma HLS resource variable=shift_reg core=RAM_1P` forces the Vivado[®] HLS tool to use a single port RAM. When using this directive in conjunction with the loop pipelining objective, the Vivado[®] HLS tool will fail to pipeline this loop with an II=1. This is due to the fact that a pipelined version of this code requires both a Read and Write operation in the same cycle. This is not possible using a single port RAM. This is evident in Figure 2.8 b). Looking at Cycle 2, we require a Write operation to the array `shift_reg` in Iteration 1, and a Read operation to the same array in Iteration 2. We can modify the directive to allow HLS more scheduling freedom by removing the explicit request for II=1, e.g. `#pragma HLS pipeline`. In this case, HLS will automatically increase the initiation interval until it can find a feasible schedule.

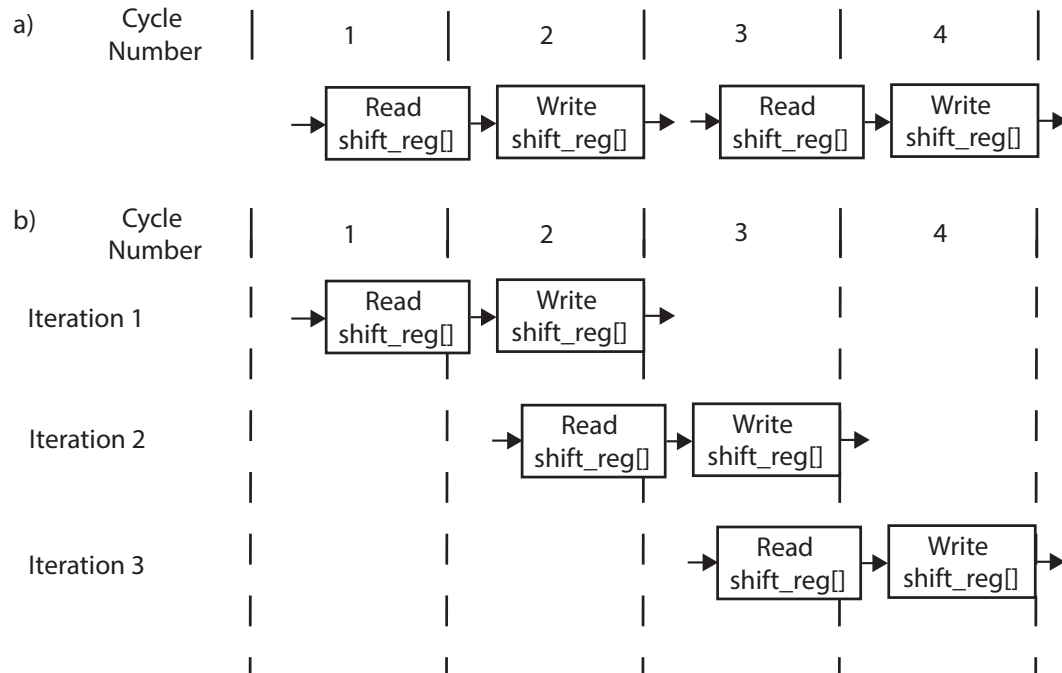


Figure 2.8: Part a) shows a schedule for two iterations of the body of the TDL **for** loop. Part b) shows the schedule for three iterations of a pipelined version of the TDL **for** loop with $II=1$.

The `RESOURCE` directive allows the user to force the Vivado[®] HLS tool to map an operation to a hardware core. This can be done on arrays (as shown above) and also for variables. Consider the code `a = b + c;`. We can use the `RESOURCE` directive `#pragma HLS RESOURCE variable=a core=AddSub_DSP` to tell the Vivado[®] HLS tool that the `add` operation is implemented using a DSP48. There are a wide variety of cores described in the Vivado[®] HLS documentation [63]. In general, it is advised to let the Vivado[®] HLS decide the resources. If these are not satisfactory, then the designer can use directives.

2.10 Bitwidth Optimization

The C language provides many different data types to describe different kinds of behavior. Up until this point, we have focused on the `int` type, which Vivado[®] HLS treats as a 32-bit signed integer. The C language also provides floating point data types, such as `float` and `double`, and integer data types, such as `char`, `short`, `long`, and `long long`. Integer datatypes may be **unsigned**. All of these data types have a size which is a power of 2.

The actual number of bits for these C data types may vary depending upon the processor architecture. For example, an `int` can be 16 bits on a micro controller and 32 bits on a general purpose processor. The C standard dictates minimum bit widths (e.g., an `int` is at least 16 bits) and relations between the types (e.g., a `long` is not smaller than an `int` which is not smaller than a `short`). The C99 language standard eliminated this ambiguity with types such as `int8_t`, `int16_t`, `int32_t`, and `int64_t`.

The primary benefits of using these different data types in software revolve around the amount of storage that the data type requires. For large arrays, using 8 bit values instead of 16 bit values can cut memory usage in half. The drawback is that the range of values that you can represent is reduced. An 8 bit signed value allows numbers in the range `[-128,127]` while the 16 bit signed data type has a range of `[-32,768, 32,767]`. Smaller operations may also require fewer clock cycles to execute, or may allow more instructions to be executed in parallel.

The same benefits are seen in an FPGA implementation, but they are even more pronounced. Since the Vivado[®] HLS tool generates a custom data path, it will create an implementation matched to the specified data types. For example, the statement `a = b * c` will have different latency and resource usage depending upon the data type. If all of the variables are 32 bits wide, then more primitive boolean operations need to be performed than if the variables are only 8 bits wide. As a result, more FPGA resources (or more complex resources) must be used. Additionally, more complex logic typically requires more pipelining in order to achieve the same frequency. A 32-bit multiplication might require 5 internal registers to meet the same frequency that an 8 bit multiplication can achieve with only one internal register. As a result, the latency of the operation will be larger (5 cycles instead of 1 cycle) and HLS must take this into account.

Create a simple design that implements the code `a = b * c`. Change the data type of the variables to `char`, `short`, `int`, `long`, and `long long`. How many cycles does the multiply operation take in each case? How many resources are used for the different data types?

What primitive boolean operations are needed to implement the multiplication of 8-bit numbers? How does this change when implementing multiplication of 32-bit numbers? Hint: How many primitive decimal operations are needed to implement multiplication of two 8 digit decimal numbers?

In many cases to implement optimized hardware, it is necessary to process data where the bitwidth is not a power of two. For example, analog to digital converters often output results in 10 bits, 12 bits, or 14 bits. We could map these to 16 bit values, but this would likely reduce performance and increase resource usage. To more accurately describe such values, Vivado® HLS provides *arbitrary precision data types* which allow for signed and unsigned data types of any bitwidth.

There are two separate classes for unsigned and signed data types:

- Unsigned: `ap_uint<width>`
- Signed: `ap_int<width>`

where the `width` variable is an integer between 1 and 1024¹. For example, `ap_int<8>` is an 8 bit signed value (same as `char`), and `ap_uint<32>` is a 32 bit unsigned value (same as `unsigned int`). This provides a more powerful data type since it can do any bitwidth, e.g., `ap_uint<4>` or `ap_int<537>`. To use these data types you must use C++ and include the file `ap_int.h`, i.e., add the code `#include "ap_int.h"` in your project and use a filename ending in `.cpp`².

Consider coefficients array `c[]` from the fir filter code in Figure 2.1. It is reprinted here for you convenience: `coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};`. The data type `coef_t` is defined as an `int` meaning that we have 32 bits of precision. This is unnecessary for these constants since they range from -91 to 500. Thus we could use a smaller data type for this. We will need a signed data type since we have positive and negative values. And the maximum absolute value for any of these 11 entries is 500, which requires $\lceil \log_2 500 \rceil = 9$ bits. Since we need negative numbers, we add an additional bit. Thus `coef_t` can be declared as `ap_int<10>`.

What is the appropriate data type for the variable `i` in the fir function (see Figure 2.1)?

We can also more accurately define the data types for the other variables in the fir function, e.g., `acc` and `shift_reg`. Consider the `shift_reg` array first. This is storing the last 11 values of the input variable `x`. So we know that the `shift_reg` values can safely have the same data type as `x`. By “safe”, we mean that there will be no loss in precision, i.e., if `shift_reg` had a data type with a smaller bitwidth, then some significant bits of `x` would need to be eliminated to fit them into a value in `shift_reg`. For example, if `x` was defined as 16 bits (`ap_uint<16>`) and `shift_reg` was defined as 12 bits (`ap_uint<12>`), then we would cut off the 4 most significant bits of `x` when we stored it into `shift_reg`.

Defining the appropriate data type for `acc` is a more difficult task. The `acc` variable stores the multiply and accumulated sum over the `shift_reg` and the coefficient array `c[]` i.e., the output value of the filter. If we wish to be safe, then we calculate the largest possible value that could be stored in `acc`, and set the bitwidth as that.

¹1024 is the default maximum value, and this can be changed if needed. See the Vivado® HLS user manuals for more information on how to do this.

²Similar capabilities are also available in C using the file `ap_cint.h`

To accomplish this, we must understand how the bitwidth increases as we perform arithmetic operations. Consider the operation $a = b + c$ where `ap_uint<10> b` and `ap_uint<10> c`. What is the data type for the variable `a`? We can perform a worst case analysis here, and assume both `a` and `b` are the largest possible value $2^{10} = 1024$. Adding them together results in $a = 2048$ which can be represented as an 11 bit unsigned number, i.e., `ap_uint<11>`. In general we will need one more bit than the largest bitwidth of the two number being added. That is, when `ap_uint<x> b` and `ap_uint<y> c`, the data type for `a` is `ap_uint<z>` where $z = \max(x, y) + 1$. This same is also true when adding signed integers.

That handles one part of the question for assigning the appropriate data type to `acc`, but we must also deal with the multiply operation. Using the same terminology, we wish to determine the value the bitwidth z given the bitwidths x and y (i.e., `ap_int<z> a`, `ap_int<x> b`, `ap_int<y> c`) for the operation $a = b * c$. While we will not go into the details, the formula is $z = x + y$.

Given these two formulas, determine the bitwidth of `acc` such that it is safe.

Ultimately we are storing `acc` into the variable `y` which is an output port of the function. Therefore, if the bitwidth of `acc` is larger than the bitwidth of `c`, the value in `acc` will be truncated to be stored into `y`. Thus, is it even important to insure that the bitwidth of `acc` is large enough so that it can handle the full precision of the multiply and accumulate operations?

The answer to the question lies in the tolerance of the application. In many digital signal processing applications, the data itself is noisy, meaning that the lower several bits may not have any significance. In addition, in signal processing applications, we often perform numerical approximations when processing data which can introduce additional error. Thus, it may not be important that we insure that the `acc` variable has enough precision to return a completely precise result. On the other hand, it may be desirable to keep more bits in the accumulation and then round the final answer again to reduce the overall rounding error in the computation. Other applications, such as scientific computing, more dynamic range is often required, which may lead to the use of floating point numbers instead of integer or fixed-point arithmetic. So what is the correct answer? Ultimately it is up to the tolerance of the application designer.

2.11 Complex FIR Filter

To this point, we have solely looked at filtering real numbers. Many digital wireless communication systems deal with complex numbers using in-phase (I) and quadrature (Q) components (see Chapter ?? for more details). Fortunately, it is possible to create a complex FIR filter using real FIR filter as we describe in the following.

To understand how to build a complex FIR filter from real FIR filters consider Equation 2.4. Assume that (I_{in}, Q_{in}) is one sample of the input data that we wish to filter. And one of the complex FIR filter coefficients is denoted as (I_{fir}, Q_{fir}) . There will be more than one input sample and complex coefficient, but let us not worry about that for now.

$$(I_{in} + jQ_{in})(I_{fir} + jQ_{fir}) = (I_{in}I_{fir} - Q_{in}Q_{fir}) + j(Q_{in}I_{fir} + I_{in}Q_{fir}) \quad (2.4)$$

Equation 2.4 shows the multiplication of the input complex number by one coefficient of the complex FIR filter. The right side of the equation shows that the real portion of the output of complex input filtered by a complex FIR filter is $I_{in}I_{fir} - Q_{in}Q_{fir}$ and the imaginary output is

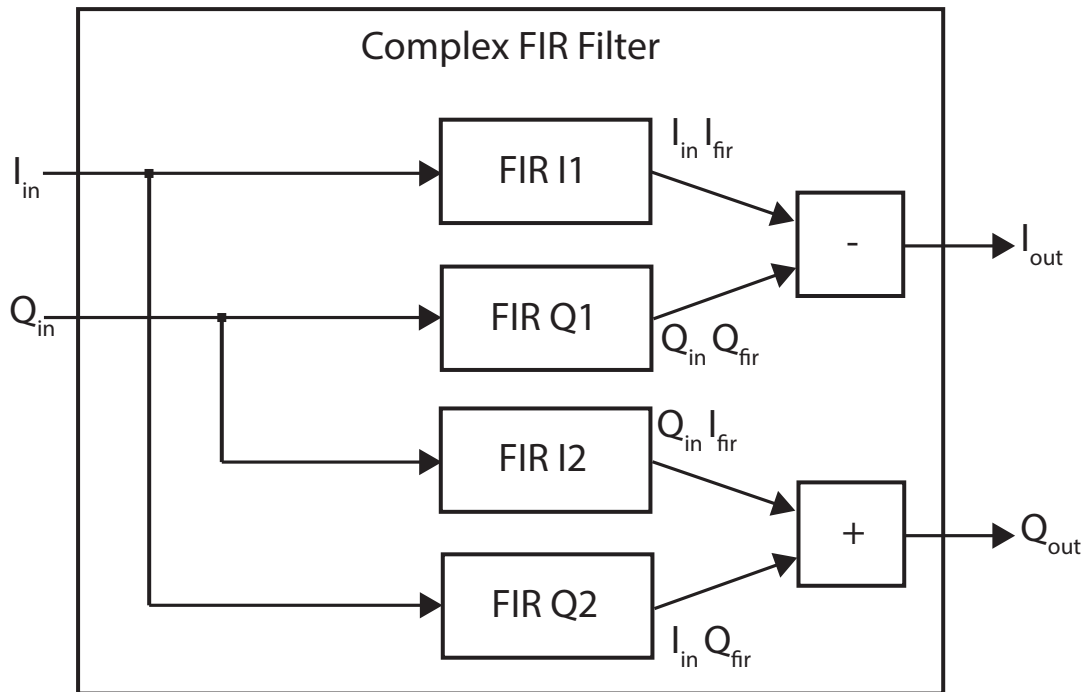


Figure 2.9: A complex FIR filter built from four real FIR filters. The input I and Q samples are feed into four different real FIR filters. The FIR filters hold the in-phase (FIR I) and quadrature (FIR Q) complex coefficients.

```
typedef int data_t;
void fir1(data_t *y, data_t x);
void firQ1(data_t *y, data_t x);
void firI2(data_t *y, data_t x);
void firQ2(data_t *y, data_t x);

void complexFIR(data_t lin, data_t Qin, data_t *lout, data_t *Qout) {

    data_t linIfir, QinQfir, QinIfir, linQfir;

    fir1(&linIfir, lin);
    firQ1(&QinQfir, Qin);
    firI2(&QinIfir, Qin);
    firQ2(&linQfir, lin);

    *lout = linIfir + QinQfir;
    *Qout = QinIfir - linQfir;
}
```

Figure 2.10: The Vivado[®] HLS code to hierarchically implement a complex FIR filter using four real FIR filters.

$Q_{in}I_{fir} + I_{in}Q_{fir}$. This implies that we can separate the complex FIR filter operation into four real filters as shown in Figure 2.9.

A complex FIR filter takes as input a complex number (I_{in}, Q_{in}) and outputs a complex filtered value (I_{out}, Q_{out}). Figure 2.9 provides a block diagram of this complex filter using four real FIR filters (FIR I1, FIR Q1, FIR I2, FIR Q2). The filters FIR I1 and FIR I2 are equivalent, i.e., they have the exact same coefficients. FIR Q1 and FIR Q2 are also equivalent. The output of each of these filters corresponds to a term from Equation 2.4. These output are then added or subtracted to provide the final filtered complex output (I_{out}, Q_{out}).

We used a hierarchical structure to define this complex FIR filter. Vivado[®] HLS implements hierarchy using functions. Taking the previous real FIR function **void fir** (**data_t *y, data_t x**) we can create another function that encapsulates four versions of this fir function to create the complex FIR filter. This code is shown in Figure 2.10.

The code defines four functions **fir1**, **firQ1**, **firI2**, and **firQ2**. Each of these functions has the exact same code, e.g., that of the **fir** function from Figure 2.1. Typically, we would not need to replicate the function; we could simply call the same function four times. However, this is not possible in this case due to the **static** keyword used within the **fir** function for the **shift_reg**.

The function calls act as interfaces. The Vivado[®] HLS tool does not optimize across function boundaries. That is, each fir function synthesized independently, and treated more or less as a black box in the **complexFIR** function. You can use the **inline** directive if you want the Vivado[®] HLS tool to co-optimize a particular function within its parent function. This will add the code from that function into the parent function and eliminate the hierarchical structure. While this can increase the potential for benefits in performance and area, it also creates a large amount of code that the tool must synthesize. That may take a long time, even fail to synthesize, or may result in a non-optimal design. Therefore, use the **inline** directive carefully. Also note that the Vivado[®]

```

float mul(int x, int y) {
    return x * y;
}

float top_function(float a, float b, float c, float d) {
    return mul(a, b) + mul(c, d) + mul(b, c) + mul(a, d);
}

float inlined_top_function(float a, float b, float c, float d) {
    return a * b + c * d + b * c + a * d;
}

```

Figure 2.11: A simple and trivial example to demonstrate the inline directive. The `top_function` has four function calls to the function `mul`. If we placed an inline directive on the `mul` function, the result is similar to what you see in the function `inlined_top_function`.

HLS tool may choose to inline functions on its own. These are typically functions with a small amount of code.

The inline directive removes function boundaries, which may enable additional opportunities for the Vivado[®] HLS tool at the cost of increasing the complexity of the synthesis problem, i.e., it will likely make the synthesis time longer. It also eliminates any overhead associated with performing the function call. It allows for different implementations while maintaining the structure of the code, and making it hierarchical and more readable.

The code in Figure 2.11 provides an example of how the inline directive works. The function `inlined_top_function` is the result of using the inline directive on the `mul` function.

The Vivado[®] HLS tool will sometimes choose to inline functions automatically. For example, it will very likely choose to inline the `mul` function from Figure 2.11 since it is small. You can force the tool to keep the function hierarchy by placing an inline directive in the function with the `off` argument.

The inline directive also has a `recursive` argument that inlines all functions called within the inlined function to also be inlined. That is, it will recursively add the code into the parent functions from every child function. This could create a substantial code base, so use this function carefully.

An inlined function will not have separate entries in the report since all of the logic will be associated with the parent function.

2.12 Conclusion

This chapter describes the specification and optimization of a FIR filter using the Vivado[®] HLS tool. The goal is to provide an overview of the HLS process. The first step in this process is understanding the basic concepts behind the computation of the FIR filter. This does not require a deep mathematical understanding, but certainly enough knowledge to write it in a manner that is synthesizable by the Vivado[®] HLS tool. This may require translating the code from a different language (e.g., MATLAB, Java, C++, Python, etc.). Many times it requires rewriting to use

simpler data structures, e.g., one that is explicitly implemented in an array. And it often involves removing system calls and other code not supported by the HLS tool.

Creating an optimum architecture requires a basic understanding about how the HLS tool performs its synthesis and optimization process to RTL code. It is certainly not unnecessary to understand the exact HLS algorithms for schedule, binding, resource allocation, etc. (and many times these are proprietary). But having a general idea of the process does aid the designer in writing code that maps well to hardware. Throughout the chapter, we talked about some of the key features of the HLS synthesis process that are necessary to understand when performing various optimizations. It is especially important to understand the way that the HLS tool reports performance, which we describe in Chapter [2.4](#).

Additionally, we presented some basic HLS optimizations (including loop and bitwidth optimizations). We highlighted their benefits and potential drawbacks using the FIR filter as an example. These are common optimizations that can be applied across a wide range of applications. We provide more details about these optimizations in subsequent chapters as we walk through the implementation of other more complex applications.

Finally, there is an entire project devoted to further optimizing the FIR filter. This is located in the Appendix in Chapter ??.

Chapter 4

Discrete Fourier Transform

The [Discrete Fourier Transform \(DFT\)](#) plays a fundamental role in digital signal processing systems. It is a method to change a discrete signal in the time domain to the same signal in the frequency domain. By describing the signal as the sum of sinusoids, we can more easily compute some functions on the signal, e.g., filtering and other linear time invariant functions. Therefore, it plays an important role in many wireless communications, image processing, and other digital signal processing applications.

This chapter provides an introduction to the [DFT](#) with a focus on its optimization for an FPGA implementation. At its core, the [DFT](#) performs a matrix-vector multiplication where the matrix is a fixed set of coefficients. The initial optimizations in Chapter [4.6](#) treat the [DFT](#) operation as a simplified matrix-vector multiplication. Then, Chapter [4.6](#) introduces a complete implementation of the [DFT](#) in Vivado[®] HLS code. Additionally, we describe how to best optimize the [DFT](#) computation to increase the throughput. We focus our optimization efforts on array partitioning optimizations in Chapter [4.5](#).

There is a lot of math in the first two sections of this chapter. This may seem superfluous, but is necessary to fully comprehend the code restructuring optimizations, particularly for understanding the computational symmetries that are utilized by the [Fast Fourier Transform \(FFT\)](#) in the next chapter. That being said, if you are more interested in the HLS optimizations, you can skip to Chapter [4.6](#).

4.1 Fourier Series

In order to explain the discrete Fourier transform, we must first understand the *Fourier series*. The Fourier series provides an alternative way to look at a real valued, continuous, periodic signal where the signal runs over one period from $-\pi$ to π . The seminal result from Jean Baptiste Joseph Fourier states that any continuous, periodic signal over a period of 2π can be represented by a sum of cosines and sines with a period of 2π . Formally, the Fourier Series is given as

$$\begin{aligned} f(t) &\sim \frac{a_0}{2} + a_1 \cos(t) + a_2 \cos(2t) + a_3 \cos(3t) + \dots \\ &\quad + b_1 \sin(t) + b_2 \sin(2t) + b_3 \sin(3t) + \dots \\ &\sim \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(nt) + b_n \sin(nt)) \end{aligned} \tag{4.1}$$

where the coefficients a_0, a_1, \dots and b_1, b_2, \dots are computed as

$$\begin{aligned} a_0 &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) dt \\ a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \cos(nt) dt \\ b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \sin(nt) dt \end{aligned} \quad (4.2)$$

There are several things to note. First, the coefficients $a_0, a_1, a_2, \dots, b_1, b_2, \dots$ in Equation 4.2 are called the Fourier coefficients. The coefficient a_0 is often called the *direct current (DC)* term (a reference to early electrical current analysis), the $n = 1$ frequency is called the fundamental, while the other frequencies ($n \geq 2$) are called higher harmonics. The notions of fundamental and harmonic frequencies originate from acoustics and music. Second, the function f , and the $\cos()$ and $\sin()$ functions all have a period of 2π ; changing this period to some other value is straightforward as we will show shortly. The DC value a_0 is equivalent to the coefficient of $\cos(0 \cdot t) = 1$, hence the use of symbol a . The b_0 value is not needed since $\sin(0 \cdot t) = 0$. Finally, the relation between the function f and its Fourier series is approximate in some cases when there are discontinuities in f (known as Gibbs phenomenon). This is a minor issue, and only relevant for the Fourier series, and not other Fourier Transforms. Therefore, going forward we will disregard this “approximation” (\sim) for “equality” ($=$).

Representing functions that are periodic on something other than π requires a simple change in variables. Assume a function is periodic on $[-L, L]$ rather than $[-\pi, \pi]$. Let

$$t \equiv \frac{\pi t'}{L} \quad (4.3)$$

and

$$dt = \frac{\pi dt'}{L} \quad (4.4)$$

which is a simple linear translation from the old $[-\pi, \pi]$ interval to the desired $[-L, L]$ interval. Solving for t' and substituting $t' = \frac{Lt}{\pi}$ into Equation 4.1 gives

$$f(t') = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos\left(\frac{n\pi t'}{L}\right) + b_n \sin\left(\frac{n\pi t'}{L}\right) \right) \quad (4.5)$$

Solving for the a and b coefficients is similar:

$$\begin{aligned} a_0 &= \frac{1}{L} \int_{-L}^L f(t') dt' \\ a_n &= \frac{1}{L} \int_{-L}^L f(t') \cos\left(\frac{n\pi t'}{L}\right) dt' \\ b_n &= \frac{1}{L} \int_{-L}^L f(t') \sin\left(\frac{n\pi t'}{L}\right) dt' \end{aligned} \quad (4.6)$$

We can use Euler’s formula $e^{jnt} = \cos(nt) + j \sin(nt)$ to give a more concise formulation

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{jnt}. \quad (4.7)$$

In this case, the Fourier coefficients c_n are a complex exponential given by

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) e^{-jnt} dt \quad (4.8)$$

which assumes that $f(t)$ is a periodic function with a period of 2π , i.e., this equation is equivalent to Equation 4.1

The Fourier coefficients a_n , b_n , and c_n are related as

$$\begin{aligned} a_n &= c_n + c_{-n} \text{ for } n = 0, 1, 2, \dots \\ b_n &= j(c_n - c_{-n}) \text{ for } n = 1, 2, \dots \\ c_n &= \begin{cases} \frac{1}{2}(a_n - jb_n) & n > 0 \\ \frac{1}{2}a_0 & n = 0 \\ \frac{1}{2}(a_{-n} + jb_{-n}) & n < 0 \end{cases} \end{aligned} \quad (4.9)$$

Note that the equations for deriving a_n , b_n , and c_n introduce the notion of a “negative” frequency. While this physically does not make much sense, mathematically we can think about as a “negative” rotation on the complex plane. A “positive” frequency indicates that the complex number rotates in a counterclockwise direction in the complex plane. A negative frequency simply means that we are rotating in the opposite (clockwise) direction on the complex plane.

This idea is further illustrated by the relationship of cosine, sine, and the complex exponential. Cosine can be viewed as the real part of the complex exponential and it can also be derived as the sum of two complex exponentials – one with a positive frequency and the other with a negative frequency as shown in Equation 4.10.

$$\cos(x) = \text{Re}\{e^{jx}\} = \frac{e^{jx} + e^{-jx}}{2} \quad (4.10)$$

The relationship between sine and the complex exponential is similar as shown in Equation 4.11. Here we subtract the negative frequency and divide by $2j$.

$$\sin(x) = \text{Im}\{e^{jx}\} = \frac{e^{jx} - e^{-jx}}{2j} \quad (4.11)$$

Both of these relationships can be visualized as vectors in the complex plane as shown in Figure 4.1. Part a) shows the cosine derivation. Here we add the two complex vectors e^{jx} and e^{-jx} . Note that the sum of these two vectors results in a vector on the real (in-phase or I) axis. The magnitude of that vector is $2\cos(x)$. Thus, by dividing the sum of these two complex exponentials by 2, we get the value $\cos(x)$ as shown in Equation 4.10. Figure 4.1 b) shows the similar derivation for sine. Here we are adding the complex vectors e^{jx} and $-e^{-jx}$. The result of this is a vector on the imaginary (quadrature or Q) axis with a magnitude of $2\sin(x)$. Therefore, we must divide by $2j$ in order to get $\sin(x)$. Therefore, this validates the relationship as described in Equation 4.11.

4.2 DFT Background

The previous section provided a mathematical foundation for the Fourier series, which works on signals that are continuous and periodic. The Discrete Fourier Transform requires *discrete* periodic signals. The DFT converts a finite number of equally spaced samples into a finite number of complex sinusoids. In other words, it converts a sampled function from one domain (most often the time domain) to the frequency domain. The frequencies of the complex sinusoids are integer multiples of the *fundamental frequency* which is defined as the frequency related to the sampling period of the input function. Perhaps the most important consequence of the discrete and periodic signal is that it can be represented by a finite set of numbers. Thus, a digital system can be used to implement the DFT.

The DFT works on input functions that uses both real and complex numbers. Intuitively, it is easier to first understand how the real DFT works, so we will ignore complex numbers for the time being and start with real signals in order to gain ease into the mathematics a bit.

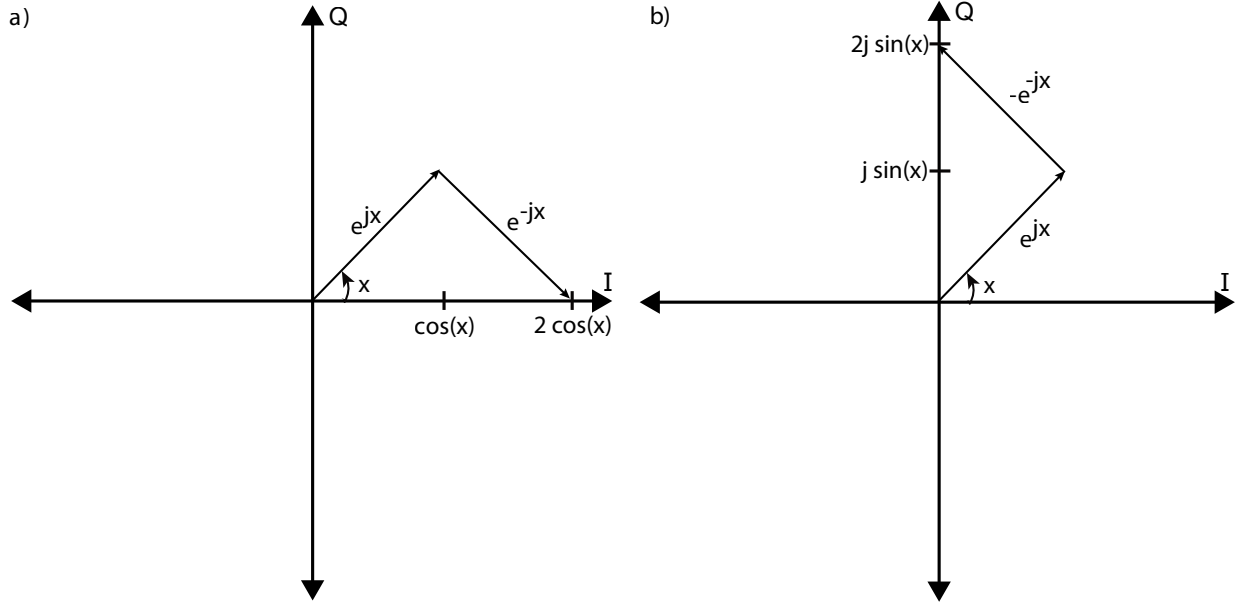


Figure 4.1: A visualization of the relationship between the cosine, sine, and the complex exponential. Part a) shows the sum of two complex vectors, e^{jx} and e^{-jx} . The result of this summation lands exactly on the real axis with the value $2 \cos(x)$. Part b) shows a similar summation except this time summing the vectors e^{jx} and $-e^{-jx}$. This summation lands on the imaginary axis with the value $2 \sin(x)$.

A quick note on terminology: We use lower case function variables to denote signals in the time domain. Upper case function variables are signals in the frequency domain. We use $()$ for continuous functions and $[]$ for discrete functions. For example, $f()$ is a continuous time domain function and $F()$ is its continuous frequency domain representation. Similarly $g[]$ is a discrete function in the time domain and $G[]$ is that function transformed into the frequency domain.

To start consider Figure 4.2. The figure shows on the left a real valued time domain signal $g[]$ with N samples or points running from 0 to $N-1$. The DFT is performed resulting in the frequency domain signals corresponding to the cosine and sine amplitudes for the various frequencies. These can be viewed as a complex number with the cosine amplitudes corresponding to the real value of the complex number and the sine amplitudes providing the imaginary portion of the complex number. There are $N/2 + 1$ cosine (real) and $N/2 + 1$ sine (imaginary) values. We will call this resulting complex valued frequency domain function $G[]$. Note that the number of samples in frequency domain ($N/2 + 1$) is due to the fact that we are considering a real valued time domain signal; a complex valued time domain signal results in a frequency domain signal with N samples.

An N point DFT can be determined through a $N \times N$ matrix multiplied by a vector of size N ,

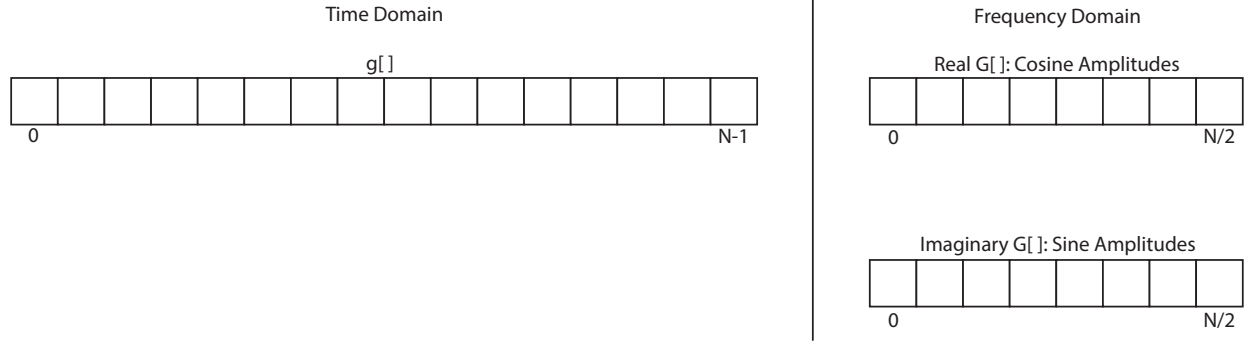


Figure 4.2: A real valued discrete function $g[]$ in the time domain with N points has a frequency domain representation with $N/2 + 1$ samples. Each of these frequency domain samples has one cosine and one sine amplitude value. Collectively these two amplitude values can be represented by a complex number with the cosine amplitude representing the real part and the sine amplitude the imaginary part.

$G = S \cdot g$ where

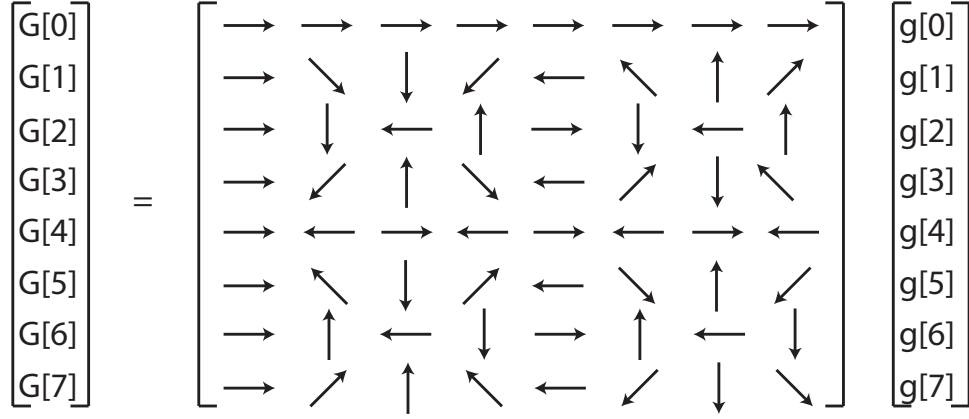
$$S = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & s & s^2 & \dots & s^{N-1} \\ 1 & s^2 & s^4 & \dots & s^{2(N-1)} \\ 1 & s^3 & s^6 & \dots & s^{3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & s^{N-1} & s^{2(N-1)} & \dots & s^{(N-1)(N-1)} \end{bmatrix} \quad (4.12)$$

and $s = e^{-j2\pi/N}$. Thus the samples in frequency domain are derived as

$$G[k] = \sum_{n=0}^{N-1} g[n]s^{kn} \text{ for } k = 0, \dots, N-1 \quad (4.13)$$

Figure 4.3 provides a visualization of the DFT coefficients for an 8 point DFT operation. The eight frequency domain samples are derived by multiplying the 8 time domain samples with the corresponding rows of the S matrix. Row 0 of the S matrix corresponds to the DC component which is proportional to the average of the time domain samples. Multiplying Row 1 of the S matrix with g provides the cosine and sine amplitudes values for when there is one rotation around the unit circle. Since this is an 8 point DFT, this means that each phasor is offset by 45° . Performing eight 45° rotations does one full rotation around the unit circle. Row 2 is similar except it performs two rotations around the unit circle, i.e., each rotation is 90° . This is a higher frequency. Row 3 does three rotations; Row 4 four rotations and so on. Each of these row times column multiplications gives the appropriate frequency domain sample.

Notice that the S matrix is diagonally symmetric, that is $S[i][j] = S[j][i]$. In addition, $S[i][j] = s^i * s^j = s^{(i+j)}$. There is also interesting symmetry around Row 4. The phasors in Rows 3 and 5 are complex conjugates of each other, i.e., $S[3][j] = S[5][j]^*$. Similarly, Rows 2 and 6 ($S[2][j] = S[6][j]^*$), and Rows 1 and 7 ($S[1][j] = S[7][j]^*$) are each related by the complex conjugate operation. It is for this reason that the DFT of a real valued input signal with N samples has only $N/2 + 1$ cosine and sine values in the frequency domain. The remaining $N/2$ frequency domain values provide redundant information so they are not needed. However, this is not true when the input signal is complex. In this case, the frequency domain will have $N + 1$ cosine and sine values.

Figure 4.3: The elements of the S shown as a complex vectors.

```

#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        dot_product_loop:
        for (j = 0; j < SIZE; j++) {
            sum += V_In[j] * M[i][j];
        }
        V_Out[i] = sum;
    }
}

```

Figure 4.4: Simple code implementing a matrix-vector multiplication.

4.3 Matrix-Vector Multiplication Optimizations

Matrix-vector multiplication is the core computation of a **DFT**. The input time domain vector is multiplied by a matrix with fixed special values. The result is a vector that corresponds to the frequency domain representation of the input time domain signal.

In this section, we look at the hardware implementation of matrix-vector multiplication. We break this operation down into its most basic form (see Figure 4.4). This allows us to better focus the discussion on the optimizations rather than deal with all the complexities of using functionally correct **DFT** code. We will build a **DFT** core in the next section.

The code in Figure 4.4 provides an initial starting point for synthesizing this operation into hardware. We use a custom data type called **BaseType** that is currently mapped as a **float**. This may seem superfluous at the time, but this will allow us in the future to easily experiment with different number representations for our variables (e.g., signed or unsigned fixed point with different precision). The **matrix_vector** function has three arguments. The first two arguments

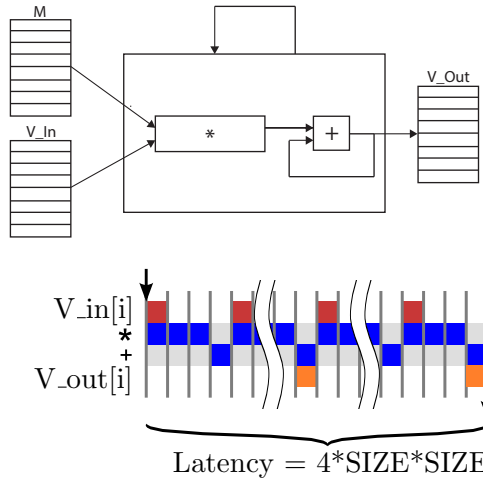


Figure 4.5: A possible implementation of matrix-vector multiplication from the code in Figure 4.4.

`BaseType M[SIZE][SIZE]` and `BaseType V_In[SIZE]` are the input matrix and vector to be multiplied. The third argument `BaseType V_Out[SIZE]` is the resultant vector. By setting $M = S$ and V_In to a sampled time domain signal, the V_Out will contain the `DFT`. $SIZE$ is a constant that determines the number of samples in the input signal and correspondingly the size of the `DFT`.

The algorithm itself is simply a nested **for** loop. The inner loop (`dot_product_loop`) computes the `DFT` coefficients starting from 0 and going to $SIZE - 1$. However, this relatively simple code has many design choices that can be performed when mapping to hardware.

Whenever you perform HLS, you should think about the architecture that you wish to synthesize. Memory organization is one of the more important decisions. The question boils down to *where do you store the data from your code?* There are a number of options when mapping variables to hardware. The variable could simply be a set of wires (if its value never needs saved across a cycle), a register, RAM or FIFO. All of these options provide tradeoffs between performance and area.

Another major factor is the amount of parallelism that is available within the code. Purely sequential code has few options for implementation. On the other hand, code with a significant amount of parallelism has implementation options that range from purely sequentially to fully parallel. These options obviously have different area and performance. We will look at how both memory configurations and parallelism effect the hardware implementation for the matrix-vector implementation of the `DFT`.

Figure 4.5 shows a sequential architecture for matrix-vector multiplication with one multiply and one addition operator. Logic is created to access the V_In and M arrays which are stored in BRAMs. Each element of V_Out is computed and stored into the BRAM. This architecture is essentially what will result from synthesizing the code from Figure 4.4 with no directives. It does not consume a lot of area, but the task latency and task interval are relatively large.

4.4 Pipelining and Parallelism

There is substantial opportunity to exploit parallelism in the matrix-multiplication example. We start by focusing on the inner loop. The expression `sum += V_In[j] * M[i][j];` is executed in each

```

#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        V_Out[i] = V_In[0] * M[i][0] + V_In[1] * M[i][1] + V_In[2] * M[i][2] +
        V_In[3] * M[i][3] + V_In[4] * M[i][4] + V_In[5] * M[i][5] +
        V_In[6] * M[i][6] + V_In[7] * M[i][7];
    }
}

```

Figure 4.6: The matrix-vector multiplication example with a manually unrolled inner loop.

iteration of the loop. The variable `sum`, which is keeping a running tally of the multiplications, is being reused in each iteration and takes on a new value. This inner loop can be rewritten as shown Figure 4.6. In this case, the `sum` variable has been completely eliminated and replaced with multiple intermediate values in the larger expression.

Loop unrolling is performed automatically by Vivado[®] HLS in a pipelined context. Loop unrolling can also be requested by using `#pragma HLS unroll` or the equivalent directive outside of a pipelined context.

It should be clear that the new expression replacing the inner loop has significant amount of parallelism. Each one of the multiplications can be performed simultaneously, and the summation can be performed using an adder tree. The data flow graph of this computation is shown in Figure 4.7.

If we wish to achieve the minimum task latency for the expression resulting from the unrolled inner loop, all eight of the multiplications should be executed in parallel. Assuming that the multiplication has a latency of 3 cycles and addition has a latency of 1 cycle, then all of the $V_In[j] * M[i][j]$ operations are completed by the third time step. The summation of these eight intermediate results using an adder tree takes $\log 8 = 3$ cycles. Hence, the body of `data_loop` now has a latency of 6 cycles for each iteration and requires 8 multipliers and 7 adders. This behavior is shown in the left side of Figure 4.8. Note that the adders could be reused across Cycle 4-6, which would reduce the number of adders to 4. However, adders are typically not shared when targeting FPGAs since an adder and a multiplexer require the same amount of FPGA resources (approximately 1 LUT per bit for a 2-input operator).

If we are not willing to use 8 multipliers, there is an opportunity to reduce resource usage in exchange for increasing the number of cycles to execute the function. For example, using 4 multipliers would result in a latency of 6 cycles for the multiplication of the eight $V_In[j] * M[i][j]$ operations, and an overall latency of 9 cycles to finish the body of `data_loop`. This behavior is shown in the right side of Figure 4.8. You could even use fewer multipliers at the cost of taking more cycles to complete the inner loop.

Looking at Figure 4.8, it is apparent that there are significant periods where the operators

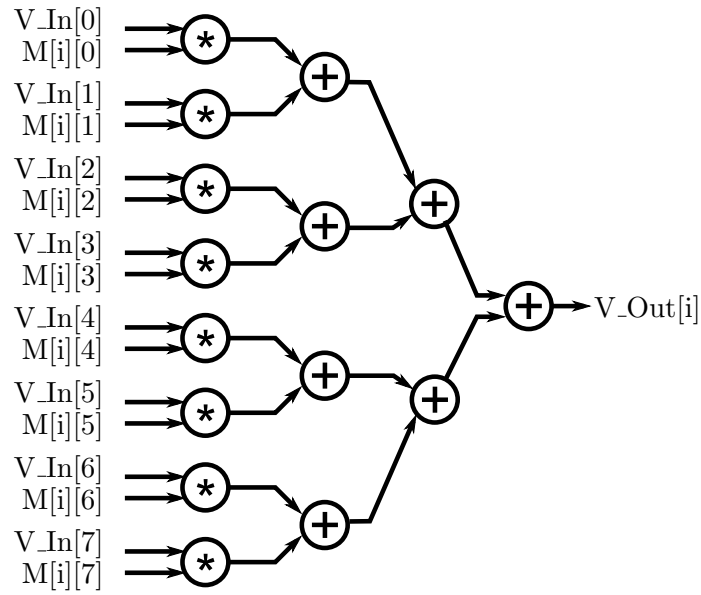


Figure 4.7: A data flow graph of the expression resulting from the unrolled inner loop from Figure 4.6.

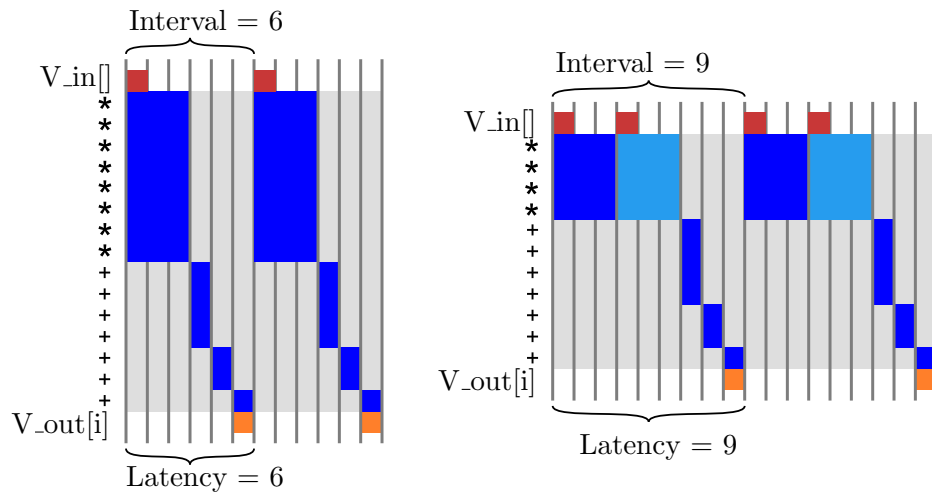


Figure 4.8: Possible sequential implementations resulting from the unrolled inner loop from Figure 4.6.

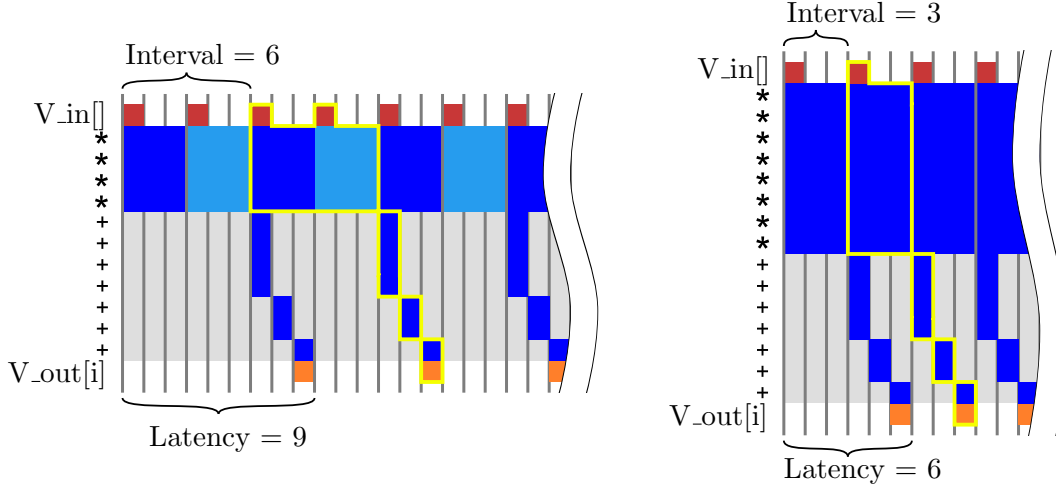


Figure 4.9: Possible pipelined implementations resulting from the unrolled inner loop from Figure 4.6.

are not performing useful work, reducing the overall efficiency of the design. It would be nice if we could reduce these periods. In this case we can observe that each iteration of `data_loop` is, in fact completely independent, which means that they can be executed concurrently. Just as we unrolled `dot_product_loop`, it's also possible to unroll `data_loop` and perform all of the multiplications concurrently. However, this would require a very large amount of FPGA resources. A better choice is to enable each iteration of the loop to start as soon as possible, while the previous execution of the loop is still executing. This process is called **loop pipelining** and is achieved in Vivado[®] HLS using `#pragma HLS pipeline`. In most cases, loop pipelining reduces the interval of a loop to be reduced, but does not affect the latency. Loop pipelined behavior of this design is shown in Figure 4.9.

Until now, we have only focused on operator latency. It is common for functional units to be also be pipelined and most functional units in Vivado[®] HLS are fully pipelined with an interval of one. Even though it might take 3 cycles for a single multiply operation to complete, a new multiply operation could start every clock cycle on a pipelined multiplier. In this way, a single functional unit may be able to simultaneously execute many multiply operations at the same time. For instance, a multiplier with a latency of 3 and an interval of 1 could be simultaneously executing three multiply operations.

By taking advantage of pipelined multipliers, we can reduce the latency of the unrolled inner loop without adding additional operators. One possible implementation using three pipelined multipliers is shown on the left in Figure 4.10. In this case, the multiplication operations can execute concurrently (because they have no data dependencies), while the addition operations cannot begin until the first multiplication has completed. In the figure on the right, a pipelined version of this design is shown, with an interval of 3, which is similar to the results of Vivado[®] HLS if `#pragma HLS pipeline II=3` is applied to the `data_loop`. In this case, not only are individual operations executing concurrently on the same operators, but those operations may come from different iterations of `data_loop`.

At this point you may have observed that pipelining is possible at different levels of hierarchy, including the operator level, loop level, and function level. Furthermore, pipelining at different levels are largely independent! We can use pipelined operators in a sequential loop, or we can use sequential operators to build a pipelined loop. It's also possible to build pipelined implementations

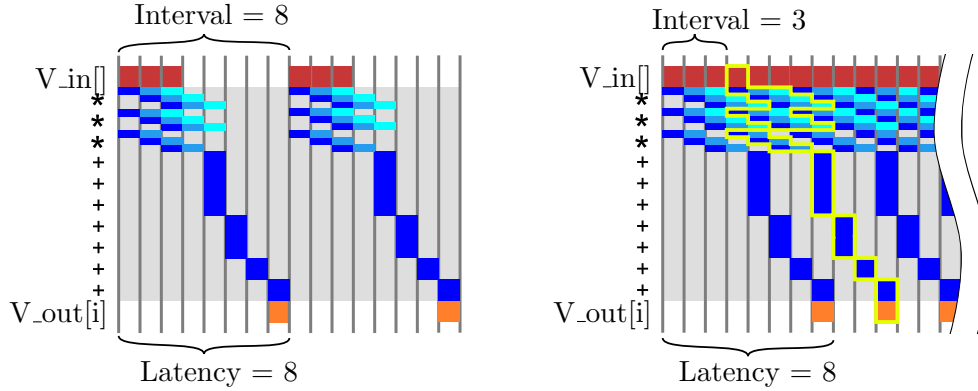


Figure 4.10: Possible implementations resulting from the unrolled inner loop from Figure 4.6 using pipelined multipliers.

of large functions which can be shared in Vivado[®] HLS just like primitive operators. In the end, what matters most is how many operators are being instantiated, their individual costs, and how often they are used.

4.5 Storage Tradeoffs and Array Partitioning

Up until this point, we have assumed that the data in arrays ($V_In[]$, $M[]$, and $V_Out[]$) are accessible at anytime. In practice, however, the placement of the data plays a crucial role in the performance and resource usage. In most processor systems, the memory architecture is fixed and we can only adapt the program to attempt to best make use of the available memory hierarchy, taking care to minimize register spills and cache misses, for instance. In HLS designs, we can also explore and leverage different memory structures and often try to find the memory structure that best matches a particular algorithm. Typically large amounts of data are stored in off-chip memory, such as DRAM, flash, or even network-attached storage. However, data access times are typically long, on the order of tens to hundreds (or more) of cycles. Off-chip storage also relatively large amounts of energy to access, because large amounts of current must flow through long wires. On-chip storage, in contrast can be accessed quickly and is much lower power. I contrast it is more limited in the amount of data that can be stored. A common pattern is to load data into on-chip memory in a block, where it can then be operated on repeatedly. This is similar to the effect of caches in the memory hierarchy of general purpose CPUs.

The primary choices for on-chip storage are in embedded memories (e.g., block RAMs) or in flip-flops (FFs). These two options have their own tradeoffs. Flip-flop based memories allow for multiple reads at different addresses in a single clock. It is also possible to read, modify, and write a Flip-flop based memory in a single clock cycle. However, the number of FFs is typically limited to around 100 Kbytes, even in the largest devices. In practice, most flip-flop based memories should be much smaller in order to make effective use of other FPGA resources. Block RAMs (BRAMs) offer higher capacity, on the order Mbytes of storage, at the cost of limited accessibility. For example, a single BRAM can store more than 1-4 Kbytes of data, but access to that data is limited to two different addresses each clock cycle. Furthermore, BRAMs are required to have a minimum amount of pipelining (i.e. the read operation must have a latency of at least one cycle). Therefore, the fundamental tradeoff boils down to the required bandwidth versus the capacity.

If throughput is the number one concern, all of the data would be stored in FFs. This would

allow any element to be accessed as many times as it is needed each clock cycle. However, as the size of arrays grows large, this is not feasible. In the case of matrix-vector multiplication, storing a 1024 by 1024 matrix of 32-bit integers would require about 4 MBytes of memory. Even using BRAM, this storage would require about 1024 BRAM blocks, since each BRAM stores around 4KBytes. On the other hand, using a single large BRAM-based memory means that we can only access two elements at a time. This obviously prevents higher performance implementations, such as in Figure 4.7, which require accessing multiple array elements each clock cycle (all eight elements of `V_in` along with 8 elements of `M`). In practice, most designs require larger arrays to be strategically divided into smaller BRAM memories, a process called **array partitioning**. Smaller arrays (often used for indexing into larger arrays) can be partitioned completely into individual scalar variables and mapped into FFs. Matching pipelining choices and array partitioning to maximize the efficiency of operator usage and memory usage is an important aspect of design space exploration in HLS.

Vivado® HLS will perform some array partitioning automatically, but as array partitioning tends to be rather design-specific it is often necessary to guide the tool for best results. Global configuration of array partitioning is available in the `config_array_partition` project option. Individual arrays can be explicitly partitioned using the **array_partition** directive. The directive **array_partition complete** will split each element of an array into its own register, resulting in a flip-flop based memory. As with many other directive-based optimizations, the same result can also be achieved by rewriting the code manually. In general, it is preferred to use the tool directives since it avoids introducing bugs and keeps the code easy to maintain.

Returning to the matrix-vector multiplication code in Figure 4.4, we can achieve a highly parallel implementation with the addition of only a few directives, as shown in Figure 4.11. The resulting architecture is shown in Figure 4.12. Notice that the inner `j` loop is automatically unrolled by Vivado® HLS and hence every use of `j` is replaced with constants in the implementation. This design demonstrates the most common use of array partitioning where the array dimensions that are partitioned (in this case, `V_in` and the second dimension of `M`) are indexed with the constants (in this case the loop index `j` of the unrolled loop). This enables an architecture where multiplexers are not required to access the partitioned arrays.

It's also possible to achieve other designs which use fewer multipliers and have lower performance. For instance, in Figure 4.10, these designs use only three multipliers, hence we need only need to read three elements of matrix `M` and vector `V_in` each clock cycle. Completely partitioning these arrays would result in extra multiplexing as shown in Figure 4.13. In actuality the arrays only need to be partitioned into three physical memories. Again, this partitioning could be implemented manually by rewriting code or in Vivado® HLS using the **array_partition** cyclic directive.

Beginning with an array `x` containing the values

$$[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9]$$

The directive **array_partition variable=x factor=2 cyclic** on the array would split it into two arrays which are

$$[1 \ 3 \ 5 \ 7 \ 9] \text{ and } [2 \ 4 \ 6 \ 8]$$

```

#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    #pragma HLS array_partition variable=M dim=2 complete
    #pragma HLS array_partition variable=V_In complete
    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        BaseType sum = 0;
        dot_product_loop:
        for (j = 0; j < SIZE; j++) {
            sum += V_In[j] * M[i][j];
        }
        V_Out[i] = sum;
    }
}

```

Figure 4.11: Matrix-vector multiplication with a particular choice of array partitioning and pipelining.

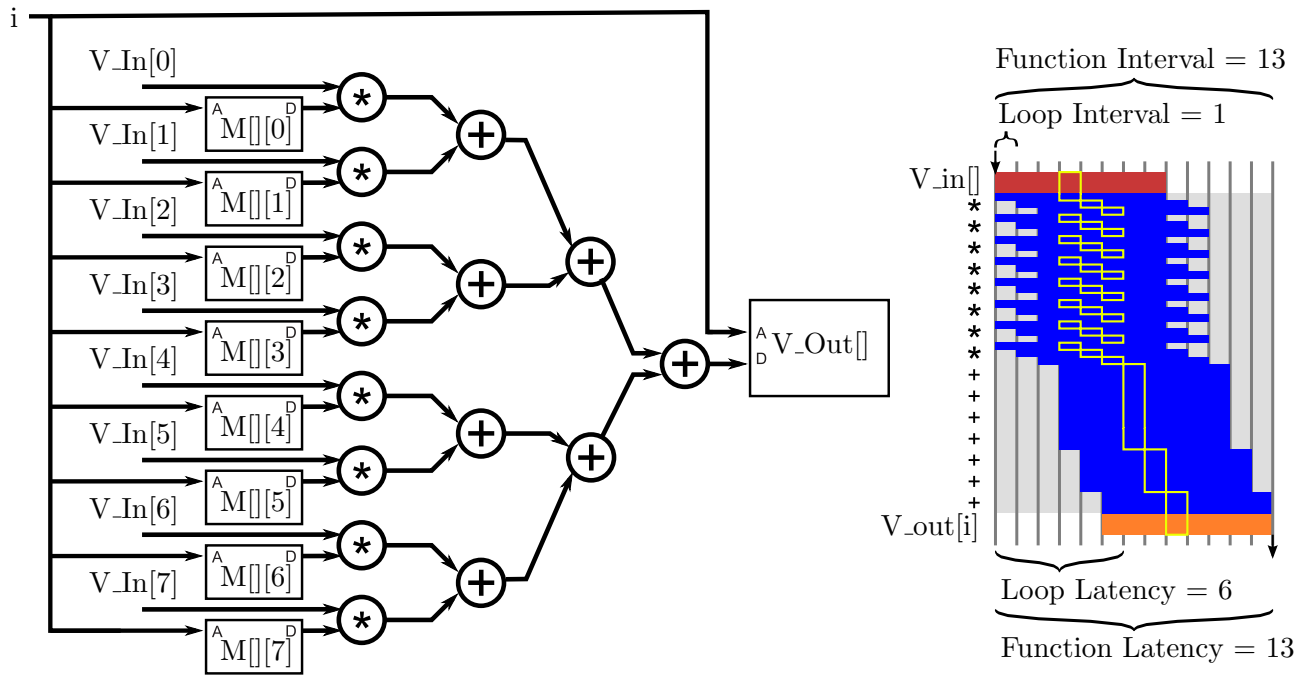


Figure 4.12: Matrix-vector multiplication architecture with a particular choice of array partitioning and pipelining. The pipelining registers have been elided and the behavior is shown at right.

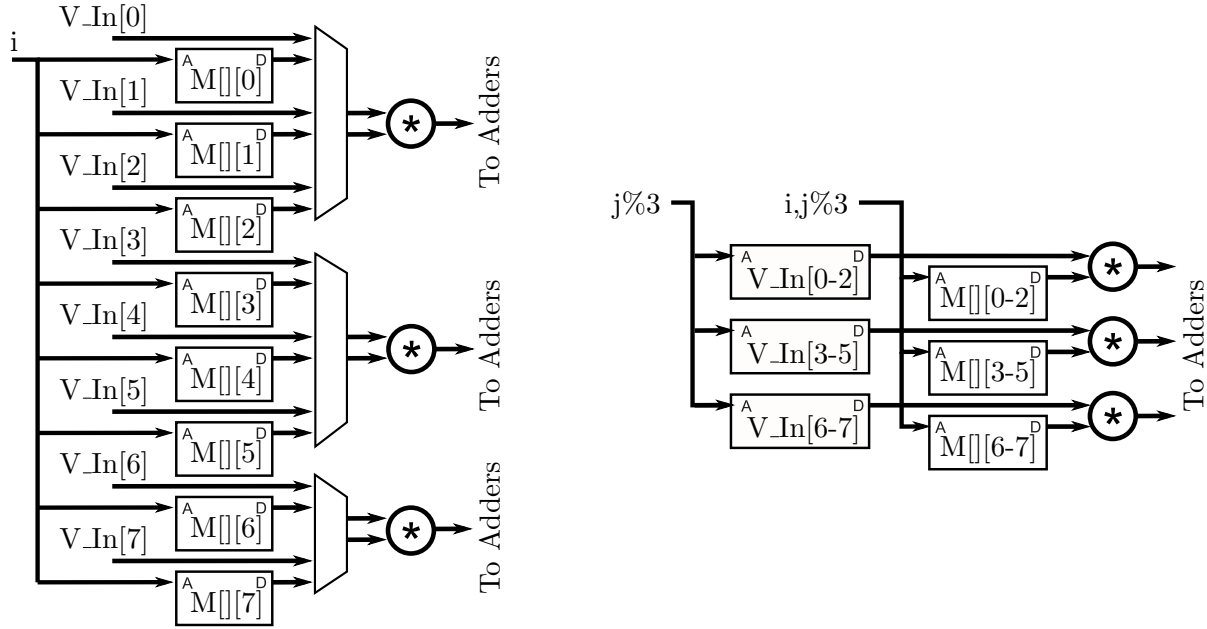


Figure 4.13: Matrix-vector multiplication architectures at $II=3$ with a particular choices of array partitioning. On the left, the arrays have been partitioned more than necessary, resulting in multiplexers. On the right, the arrays are partitioned with factor=3. In this case, multiplexing has been reduced, but the j loop index becomes a part of the address computations.

Similarly, the directive **array_partition** variable= x factor=2 block would split it into two arrays

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix} \text{ and } \begin{bmatrix} 6 & 7 & 8 & 9 \end{bmatrix}$$

Study the effects of varying pipeline II and array partitioning on the performance and area. Plot the performance in terms of number of matrix vector multiply operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for area (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?

Alternatively, similar results can be achieved by pipelining and applying **partial loop unrolling** to the inner **dot_product_loop**. Figure 4.14 shows the result of unrolling the inner loop of the matrix-vector multiplication code by a factor of 2. You can see that the loop bounds now increment by 2. Each loop iteration requires 2 elements of matrix $M[][]$ and vector $V_in[]$ each iteration and perform two multiplies instead of one. In this case after loop unrolling Vivado[®] HLS can implement the operations in both expressions in parallel, corresponding to two iterations of the original loop. Note that without appropriate array partitioning, unrolling the inner loop may offer no increase in performance, as the number of concurrent read operations is limited by the number of ports to the memory. In this case, we can store the data from the even columns in one BRAM and the data from the odd columns in the other. This is due to the fact that the unrolled loop is always performing one even iteration and one odd iteration.

```

#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    #pragma HLS array_partition variable=M dim=2 cyclic factor=2
    #pragma HLS array_partition variable=V_In cyclic factor=2
    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        dot_product_loop:
        for (j = 0; j < SIZE; j+=2) {
            #pragma HLS pipeline II=1
            sum += V_In[j] * M[i][j];
            sum += V_In[j+1] * M[i][j+1];
        }
        V_Out[i] = sum;
    }
}

```

Figure 4.14: The inner loop of matrix-vector multiply manually unrolled by a factor of two.

The HLS tool can automatically unroll loops using the `unroll` directive. The directive takes a `factor` argument which is a positive integer denoting the number of times that the loop body should be unrolled.

Manually divide `M[][]` and vector `V_in[]` into separate arrays in the same manner as the directive `array_partition cyclic factor=2`. How do you have to modify the code in order to change the access patterns? Now manually unroll the loop by a factor of two. How do the performance results vary between the original code (no array partitioning and no unrolling), only performing array partitioning, and performing array partitioning and loop unrolling? Finally, use the directives to perform array partitioning and loop unrolling. How do those results compare to your manual results?

In this code, we see that array partitioning often goes hand in hand with our choices of pipelining. Array partitioning by a factor of 2 enables an increase in performance by a factor of 2, which can be achieved either by partially unrolling the inner loop by a factor of 2 or by reducing the II of the outer loop by a factor of 2. Increasing performance requires a corresponding amount of array partitioning. In the case of matrix vector multiplication, this relationship is relatively straightforward since there is only one access to each variable in the inner loop. In other code, the relationship might be more complicated. Regardless, the goal of a designer is usually to ensure that the instantiated FPGA resources are used efficiently. Increasing performance by a factor of 2 should use approximately twice as many resources. Decreasing performance by a factor of 2 should use approximately half as many resources.

Study the effects of loop unrolling and array partitioning on the performance and area. Plot the performance in terms of number of matrix vector multiply operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for area (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?

4.6 Baseline Implementation

We just discussed some optimizations for matrix-vector multiplication. This is a core computation in performing a **DFT**. However, there are some additional intricacies that we must consider to move from the matrix-vector multiplication in the previous section to a functionally complete **DFT** hardware implementation. We move our focus to the **DFT** in this section, and describe how to optimize it to make it execute most efficiently.

One significant change that is required is that we must be able to handle complex numbers. As noted in Section 4.2 because the elements of the S matrix are complex numbers, the **DFT** of a real-valued signal is almost always a complex-valued signal. It is also common to perform the **DFT** of a complex-valued signal, to produce a complex-valued result. Additionally, we need to handle fractional or possibly floating point data, rather than integers. This can increase the cost of the implementation, particularly if floating point operations need to be performed. In addition, floating point operators, particularly addition, have much larger latency than integer addition. This can make it more difficult to achieve II=1 loops. A second change is that we'd like to be able to scale our design up to large input vector sizes, perhaps $N=1024$ input samples. Unfortunately, if we directly use matrix-vector multiplication, then we must store the entire S matrix. Since this matrix is the square of the input size, it becomes prohibitive to store for large input sizes. In the following sections, we'll discuss techniques to address both of these complexities.

As is typical when creating a hardware implementation using high level synthesis, we start with a straightforward or naive implementation. This provides us with a baseline code that we can insure has the correct functionality. Typically, this code runs in a very sequential manner; it is not highly optimized and therefore may not meet the desired performance metrics. However, it is a necessary step to insure that the designer understand the functionality of the algorithm, and it serves as starting point for future optimizations.

Figure 4.15 shows a baseline implementation of the **DFT**. This uses a doubly nested **for** loop. The inner loop multiplies one row of the S matrix with the input signal. Instead of reading the S matrix as an input, this code computes an element of S in each iteration of the inner loop, based on the current loop indices. This phasor is converted to Cartesian coordinates (a real part and an imaginary part) using the $\cos()$ and $\sin()$ functions. The code then performs a complex multiplication of the phasor with the appropriate sample of the input signal and accumulates the result. After N iterations of this inner loop, one for each column of S , one frequency domain sample is calculated. The outer loop also iterates N times, once for each row of S . As a result, the code computes an expression for w N times, but computes the $\cos()$ and $\sin()$ functions and a complex multiply-add N^2 times.

This code uses a function call to calculate $\cos()$ and $\sin()$ values. Vivado[®] HLS is capable of synthesizing these functions using its built-in math library. There are several possible algorithms [22] for implementing trigonometric functions including CORDIC, covered in Chapter 3. However, generating precise results for these functions can be expensive. There are several possibilities for

```
#include <math.h>    //Required for cos and sin functions
typedef double IN_TYPE; // Data type for the input signal
typedef double TEMP_TYPE; // Data type for the temporary variables
#define N 256        // DFT Size

void dft(IN_TYPE sample_real[N], IN_TYPE sample_imag[N]) {
    int i, j;
    TEMP_TYPE w;
    TEMP_TYPE c, s;

    // Temporary arrays to hold the intermediate frequency domain results
    TEMP_TYPE temp_real[N];
    TEMP_TYPE temp_imag[N];

    // Calculate each frequency domain sample iteratively
    for (i = 0; i < N; i += 1) {
        temp_real[i] = 0;
        temp_imag[i] = 0;

        //  $(2 * \pi * i) / N$ 
        w = (2.0 * 3.141592653589 / N) * (TEMP_TYPE)i;

        // Calculate the jth frequency sample sequentially
        for (j = 0; j < N; j += 1) {
            // Utilize HLS tool to calculate sine and cosine values
            c = cos(j * w);
            s = sin(j * w);

            // Multiply the current phasor with the appropriate input sample and keep
            // running sum
            temp_real[i] += (sample_real[j] * c - sample_imag[j] * s);
            temp_imag[i] += (sample_real[j] * s + sample_imag[j] * c);
        }
    }

    // Perform an inplace DFT, i.e., copy result into the input arrays
    for (i = 0; i < N; i += 1) {
        sample_real[i] = temp_real[i];
        sample_imag[i] = temp_imag[i];
    }
}
```

Figure 4.15: Baseline code for the DFT.

eliminating these function calls, since the inputs aren't arbitrary. We will discuss these tradeoffs in more detail later. A sequential implementation of this code is shown in Figure 4.16.

What changes would this code require if you were to use a CORDIC that you designed, for example, from Chapter 3? Would changing the accuracy of the CORDIC core make the DFT hardware resource usage change? How would it effect the performance?

Implement the baseline code for the DFT using HLS. Looking at the reports, what is the relative cost of the implementation of the trigonometric functions, compared to multiplication and addition? Which operations does it make more sense to try to optimize? What performance can be achieved by pipelining the inner loop?

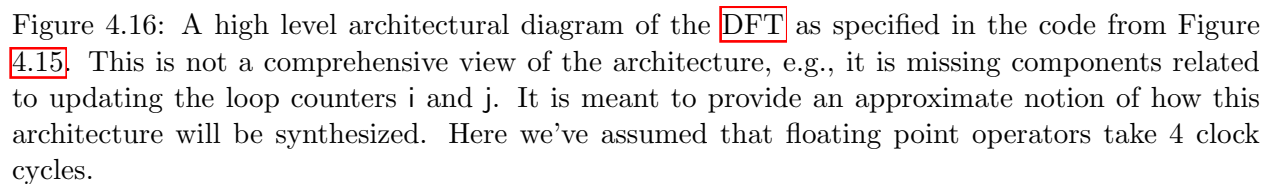
4.7 DFT optimization

The baseline DFT implementation of the previous section uses relatively high precision **double** datatypes. Implementing floating point operations is typically very expensive and requires many pipeline stages, particularly for double precision. We can see in Figure 4.16 that this significantly affects the performance of the loop. With pipelining, the affect of these high-latency operations is less critical, since multiple executions of the loop can execute concurrently. The exception in this code are the `temp_real[]` and `temp_imag[]` variables, which are used to accumulate the result. This accumulation is a **recurrence** and limits the achievable II in this design when pipelining the inner loop. This operator dependence is shown in Figure 4.17.

One possible solution is to reduce the precision of the computation. This is always a valuable technique when it can be applied, since it reduces the resources required for each operation, the memory required to store any values, and often reduces the latency of operations as well. For instance we could use the 32-bit **float** type or the 16-bit **half** types rather than double. Many signal processing systems avoid floating point data types entirely and use fixed point data types 3.5. For commonly used integer and fixed-point precisions, each addition can be completed in a single cycle, enabling the loop to be pipelined at II=1.

What happens to the synthesis result of the code in Figure 4.15 if you change all of the data types from *double* to *float*? Or from *double* to *half*? Or to a fixed point value? How does this change the performance (interval and latency) and the resource usage? Does it change the values of the output frequency domain samples?

A more general solution to achieve II=1 with floating point accumulations is to process the data in a different order. Looking at Figure 4.17 we see that the recurrence exists (represented by the arrow) because the *j* loop is the inner loop. If the inner loop were the *i* loop instead, then we wouldn't need the result of the accumulation before the next iteration starts. We can achieve this in the code by interchanging the order of the two loops. This optimization is often called **loop interchange** or pipeline-interleaved processing [40]. In this case, it may not be obvious that we can rearrange the loops because of the extra code inside the outer *i* loop. Fortunately, the *S* matrix is diagonally symmetric, and hence *i* and *j* can be exchanged in the computation of *w*. The result



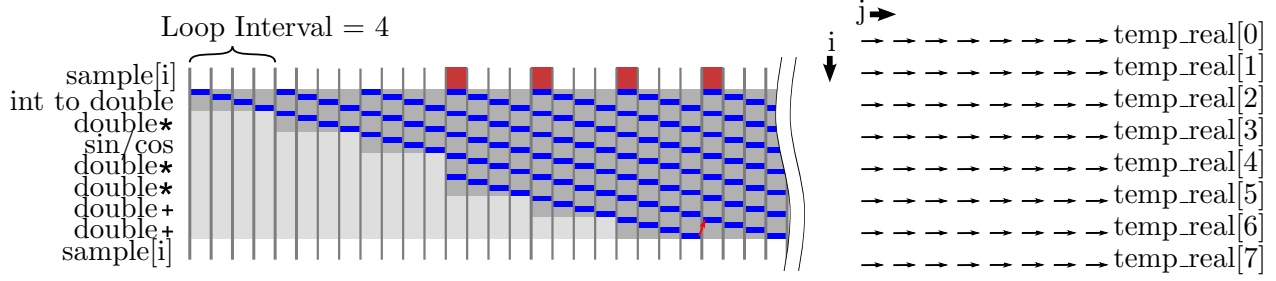


Figure 4.17: Pipelined version of the behavior in Figure 4.16. In this case, the initiation interval of the loop is limited to 4, since each floating point addition takes 4 clock cycles to complete and the result is required before the next loop iteration begins (the dependence shown in red). The dependencies for all iterations are summarized in the diagram on the right.

is that we can now achieve an II of 1 for the inner loop. The tradeoff is additional storage for the `temp_real` and `temp_imag` arrays to store the intermediate values until they are needed again.

Reorder the loops of the code in Figure 4.15 and show that you can pipeline the inner loop with an II of 1.

There are other optimizations that we can apply based on the structure of the S matrix in the **DFT** to eliminate the trigonometric operations entirely. Recall that the complex vectors for each element of the S matrix are calculated based upon a fixed integer rotation around the unit circle. Row $S[0][\cdot]$ of the S matrix corresponds to zero rotations around the unit circle, row $S[1][\cdot]$ corresponds to a single rotation, and the following rows correspond to more rotations around the unit circle. It turns out that the vectors corresponding to the second row $S[1][\cdot]$, which is one rotation around the unit circle (divided into $360/8 = 45^\circ$ individual rotations), cover all of the vectors from every other row. This can be visually confirmed by studying Figure 9.11. Thus it is possible to store only the sine and cosine values from this one rotation, and then index into this memory to calculate the requisite values for the corresponding rows. This requires only $2 \times N = \mathcal{O}(N)$ elements of storage. This results in a $\mathcal{O}(N)$ reduction in storage, which for the 1024 point **DFT** would reduce the memory storage requirements to 1024×2 entries. Assuming 32 bit fixed or floating point values, this would require only 8 KB of on-chip memory. Obviously, this is a significant reduction compared to storing the entire S matrix explicitly. We denote this one dimensional storage of the matrix S as S' where

$$S' = S[1][\cdot] = (1 \quad s \quad s^2 \quad \dots \quad s^{N-1}) \quad (4.14)$$

Derive a formula for the access pattern for the 1D array S' given as input the row number i and column element j corresponding to the array S . That is, how do we index into the 1D S' array to access element $S(i, j)$ from the 2D S array.

To increase performance further we can apply techniques that are very similar to the matrix-vector multiply. Previously, we observed that increasing performance of matrix-vector multiply required partitioning the $M[\cdot][\cdot]$ array. Unfortunately, representing the S matrix using the S' means that there is no longer an effective way to partition S' to increase the amount of data that we can

read on each clock. Every odd row and column of S includes every element of S' . As a result, there is no way to partition the values of S' like we were able to do with S . The only way to increase the number of read ports from the memory that stores S' is to replicate the storage. Fortunately, unlike with a memory that must be read and written, it is relatively easy to replicate the storage for an array that is only read. In fact, Vivado[®] HLS will perform this optimization automatically when instantiates a **Read-only Memory (ROM)** for an array which is initialized and then never modified. One advantage of this capability is that we can simply move the $\sin()$ and $\cos()$ calls into an array initialization. In most cases, if this code is at the beginning of a function and only initializes the array, then Vivado[®] HLS is able to optimize away the trigonometric computation entirely and compute the contents of the ROM automatically.

Devise an architecture that utilizes S' – the 1D version of the S matrix. How does this affect the required storage space? Does this change the logic utilization compared to an implementation using the 2D S matrix?

In order to effectively optimize the design, we must consider every part of the code. The performance can only as good as the “weakest link” meaning that if there is a bottleneck the performance will take a significant hit. The current version of the **DFT** function performs an in-place operation on the input and output data, i.e., it stores the results in the same array as the input data. The input array arguments `sample_real` and `sample_imag` effectively act as a memory port. That is, you can think of these arguments arrays as stored in the same memory location. Thus, we can only grab one piece of data from each of these arrays on any given cycle. This can create a bottleneck in terms of parallelizing the multiplication and summation operations within the function. This also explains the reason why we must store all of the output results in a temporary array, and then copy all of those results into the “sample” arrays at the end of the function. We would not have to do this if we did not perform an in-place operation.

Modify the **DFT** function interface so that the input and outputs are stored in separate arrays. How does this effect the optimizations that you can perform? How does it change the performance? What about the area results?

4.8 Conclusion

In this chapter, we looked at the hardware implementation and optimization of the Discrete Fourier Transform (**DFT**). The **DFT** is a fundamental operation in digital signal processing. It takes a signal sampled in the time domain and converts it into the frequency domain. At the beginning of this chapter, we describe the mathematical background for the **DFT**. This is important for understanding the optimizations done in the next chapter (**FFT**). The remainder of the chapter was focused on specifying and optimizing the **DFT** for an efficient implementation on an FPGA.

At its core, the **DFT** performs a matrix-vector multiplication. Thus, we spend some time initially to describe instruction level optimizations on a simplified code performing matrix-vector multiplication. These instruction level optimizations are done by the HLS tool. We use this as an opportunity to shed some light into the process that the HLS tool performs in the hopes that it will provide some better intuition about the results the tool outputs.

After that, we provide an functionally correct implementation for the **DFT**. We discuss a number of optimizations that can be done to improve the performance. In particular, we focus on the

problem of dividing the coefficient array into different memories in order to increase the throughput. Array partitioning optimization are often key to achieving the highest performing architectures.

Chapter 5

Fast Fourier Transform

Performing the **Discrete Fourier Transform (DFT)** directly using matrix-vector multiply requires $\mathcal{O}(n^2)$ multiply and add operations, for an input signal with n samples. It is possible to reduce the complexity by exploiting the structure of the constant coefficients in the matrix. This S matrix encodes the coefficients of the **DFT**; each row of this matrix corresponds to a fixed number of rotations around the complex unit circle (please refer to Chapter ?? for more detailed information). These values have a significant amount of redundancy, and that can be exploited to reduce the complexity of the algorithm.

The 'Big O' notation used here describes the general order of complexity of an algorithm based on the size of the input data. For a complete description of Big O notation and its use in analyzing algorithms, see [17].

The **Fast Fourier Transform (FFT)** uses a divide-and-conquer approach based on the symmetry of the S matrix. The **FFT** was made popular by the Cooley-Tukey algorithm [16], which requires $\mathcal{O}(n \log n)$ operations to compute the same function as the **DFT**. This can provide a substantial speedup, especially when performing the Fourier transform on large signals.

The divide-and-conquer approach to computing the **DFT** was initially developed by Karl Friedrich Gauss in the early 19th century. However, since Gauss' work on this was not published during his lifetime and only appeared in a collected works after his death, it was relegated to obscurity. Heideman et al. [32] provide a nice background on the history of the **FFT**.

The focus of this chapter is to provide the reader with a good understanding of the **FFT** algorithm since that is an important part of creating an optimized hardware design. Thus, we start by giving a mathematical treatment of the **FFT**. This discussion focuses on small **FFT** sizes to give some basic intuition on the core ideas. After that, we focus on different hardware implementation strategies.

5.1 Background

The **FFT** brings about a reduction in complexity by taking advantage of symmetries in the **DFT** calculation. To better understand how to do this, let us look at **DFT** with a small number of points,

starting with the 2 point **DFT**. Recall that the **DFT** performs a matrix vector multiplication, i.e., $G[] = S[][] \cdot g[]$, where $g[]$ is the input data, $G[]$ is the frequency domain output data, and $S[][]$ are the **DFT** coefficients. We follow the same notation for the coefficient matrix, and the input and output vectors as described in Chapter ??.

For a 2 point **DFT**, the values of S are:

$$S = \begin{bmatrix} W_2^{00} & W_2^{01} \\ W_2^{10} & W_2^{11} \end{bmatrix} \quad (5.1)$$

Here we use the notation $W = e^{-j2\pi}$. The superscript on W denotes values that are added to the numerator and the subscript on the W indicates those values added in the denominator of the complex exponential. For example, $W_4^{23} = e^{\frac{-j2\pi \cdot 2 \cdot 3}{4}}$. This is similar to the s value used in the **DFT** discussion (Chapter ??) where $s = e^{\frac{-j2\pi}{N}}$. The relationship between s and W is $s = W_N$.

The $e^{-j2\pi}$ or W terms are often called *twiddle factors*. This term has its origin in the 1966 paper by Gentleman and Sande [27].

$$\begin{bmatrix} G[0] \\ G[1] \end{bmatrix} = \begin{bmatrix} W_2^{00} & W_2^{01} \\ W_2^{10} & W_2^{11} \end{bmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \end{bmatrix} \quad (5.2)$$

Expanding the two equations for a 2 point **DFT** gives us:

$$\begin{aligned} G[0] &= g[0] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 0}{2}} + g[1] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 1}{2}} \\ &= g[0] + g[1] \end{aligned} \quad (5.3)$$

due to the fact that since $e^0 = 1$. The second frequency term

$$\begin{aligned} G[1] &= g[0] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 0}{2}} + g[1] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 1}{2}} \\ &= g[0] - g[1] \end{aligned} \quad (5.4)$$

since $e^{\frac{-j2\pi \cdot 1 \cdot 1}{2}} = e^{-j\pi} = -1$.

Figure 5.1 provides two different representations for this computation. Part a) is the data flow graph for the 2 point **DFT**. It is the familiar view that we have used to represent computation throughout this book. Part b) shows a butterfly structure for the same computation. This is a typically structure used in digital signal processing, in particular, to represent the computations in an **FFT**.

The butterfly structure is a more compact representation that is useful to represent large data flow graphs. When two lines come together this indicates an addition operation. Any label on the line itself indicates a multiplication of that label by the value on that line. There are two labels in this figure. The ‘-’ sign on the bottom horizontal line indicates that this value should be negated. This followed by the addition denoted by the two lines intersecting is the same as subtraction. The second label is W_2^0 . While this is a multiplication is unnecessary (since $W_2^0 = 1$ this means it is multiplying by the value ‘1’), we show it here since it is a common structure that appears in higher point **FFTs**.

Now let us consider a slightly larger **DFT** – a 4 point **DFT**, i.e., one that has 4 inputs, 4 outputs, and a 4×4 S matrix. The values of S for a 4 point **DFT** are:

$$S = \begin{bmatrix} W_4^{00} & W_4^{01} & W_4^{02} & W_4^{03} \\ W_4^{10} & W_4^{11} & W_4^{12} & W_4^{13} \\ W_4^{20} & W_4^{21} & W_4^{22} & W_4^{23} \\ W_4^{30} & W_4^{31} & W_4^{32} & W_4^{33} \end{bmatrix} \quad (5.5)$$

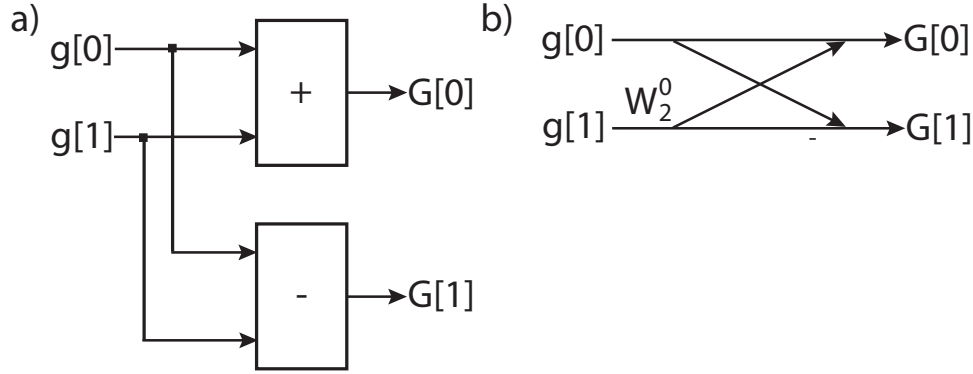


Figure 5.1: Part a) is a data flow graph for a 2 point **DFT/FFT**. Part b) shows the same computation, but viewed as a butterfly structure. This is a common representation for the computation of an **FFT** in the digital signal processing domain.

And the **DFT** equation to compute the frequency output terms are:

$$\begin{bmatrix} G[0] \\ G[1] \\ G[2] \\ G[3] \end{bmatrix} = \begin{bmatrix} W_4^{00} & W_4^{01} & W_4^{02} & W_4^{03} \\ W_4^{10} & W_4^{11} & W_4^{12} & W_4^{13} \\ W_4^{20} & W_4^{21} & W_4^{22} & W_4^{23} \\ W_4^{30} & W_4^{31} & W_4^{32} & W_4^{33} \end{bmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \\ g[2] \\ g[3] \end{bmatrix} \quad (5.6)$$

Now we write out the equations for each of the frequency domain values in $G[]$ one-by-one. The equation for $G[0]$ is:

$$\begin{aligned} G[0] &= g[0] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 3}{4}} \\ &= g[0] + g[1] + g[2] + g[3] \end{aligned} \quad (5.7)$$

since $e^0 = 1$.

The equation for $G[1]$ is:

$$\begin{aligned} G[1] &= g[0] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 3}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j2\pi}{4}} + g[2] \cdot e^{\frac{-j4\pi}{4}} + g[3] \cdot e^{\frac{-j6\pi}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j2\pi}{4}} + g[2] \cdot e^{-j\pi} + g[3] \cdot e^{\frac{-j2\pi}{4}} e^{-j\pi} \\ &= g[0] + g[1] \cdot e^{\frac{-j2\pi}{4}} - g[2] - g[3] \cdot e^{\frac{-j2\pi}{4}} \end{aligned} \quad (5.8)$$

The reductions were done based upon the fact that $e^{-j\pi} = -1$.

The equation for $G[2]$ is:

$$\begin{aligned} G[2] &= g[0] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 3}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j4\pi}{4}} + g[2] \cdot e^{\frac{-j8\pi}{4}} + g[3] \cdot e^{\frac{-j12\pi}{4}} \\ &= g[0] - g[1] + g[2] - g[3] \end{aligned} \quad (5.9)$$

The reductions were done by simplifications based upon rotations. E.g., $e^{\frac{-j8\pi}{4}} = 1$ and $e^{\frac{-j12\pi}{4}} = -1$ since in both cases use the fact that $e^{-j2\pi}$ is equal to 1. In other words, any complex exponential with a rotation by 2π is equal.

Finally, the equation for $G[3]$ is:

$$\begin{aligned}
G[3] &= g[0] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 3}{4}} \\
&= g[0] + g[1] \cdot e^{\frac{-j6\pi}{4}} + g[2] \cdot e^{\frac{-j12\pi}{4}} + g[3] \cdot e^{\frac{-j18\pi}{4}} \\
&= g[0] + g[1] \cdot e^{\frac{-j6\pi}{4}} - g[2] + g[3] \cdot e^{\frac{-j10\pi}{4}} \\
&= g[0] + g[1] \cdot e^{\frac{-j6\pi}{4}} - g[2] - g[3] \cdot e^{\frac{-j6\pi}{4}}
\end{aligned} \tag{5.10}$$

Most of the reductions that we have not seen yet deal with the last term. It starts out as $e^{\frac{-j18\pi}{4}}$. It is reduced to $e^{\frac{-j10\pi}{4}}$ since these are equivalent based upon a 2π rotation, or, equivalently, $e^{\frac{-j10\pi}{4}} \cdot e^{\frac{-j8\pi}{4}}$ and the second term $e^{\frac{-j8\pi}{4}} = 1$. Finally, a rotation of π , which is equal to -1 , brings it to $e^{\frac{-j6\pi}{4}}$. Another way of viewing this is $e^{\frac{-j6\pi}{4}} \cdot e^{\frac{-j4\pi}{4}}$ and $e^{\frac{-j4\pi}{4}} = -1$. We leave this term in this unreduced state in order to demonstrate symmetries in the following equations.

With a bit of reordering, we can view these four equations as:

$$\begin{aligned}
G[0] &= (g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}}(g[1] + g[3]) \\
G[1] &= (g[0] - g[2]) + e^{\frac{-j2\pi 1}{4}}(g[1] - g[3]) \\
G[2] &= (g[0] + g[2]) + e^{\frac{-j2\pi 2}{4}}(g[1] + g[3]) \\
G[3] &= (g[0] - g[2]) + e^{\frac{-j2\pi 3}{4}}(g[1] - g[3])
\end{aligned} \tag{5.11}$$

Several different symmetries are starting to emerge. First, the input data can be partitioned into even and odd elements, i.e., similar operations are done on the elements $g[0]$ and $g[2]$, and the same is true for the odd elements $g[1]$ and $g[3]$. Furthermore we can see that there are addition and subtraction symmetries on these even and odd elements. During the calculations of the output frequencies $G[0]$ and $G[2]$, the even and odd elements are summed together. The even and odd input elements are subtracted when calculating the frequencies $G[1]$ and $G[3]$. Finally, the odd elements in every frequency term are multiplied by a constant complex exponential W_4^i where i denotes the index for the frequency output, i.e., $G[i]$.

Looking at the terms in the parentheses, we see that they are 2 point **FFT**. For example, consider the terms corresponding to the even input values $g[0]$ and $g[2]$. If we perform a 2 point **FFT** on these even terms, the lower frequency (DC value) is $g[0] + g[2]$ (see Equation 5.3), and the higher frequency is calculated as $g[0] - g[2]$ (see Equation 5.4). The same is true for the odd input values $g[1]$ and $g[3]$.

We perform one more transformation on these equations.

$$\begin{aligned}
G[0] &= (g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}}(g[1] + g[3]) \\
G[1] &= (g[0] - g[2]) + e^{\frac{-j2\pi 1}{4}}(g[1] - g[3]) \\
G[2] &= (g[0] + g[2]) - e^{\frac{-j2\pi 0}{4}}(g[1] + g[3]) \\
G[3] &= (g[0] - g[2]) - e^{\frac{-j2\pi 1}{4}}(g[1] - g[3])
\end{aligned} \tag{5.12}$$

The twiddle factors in the last two equations are modified from $e^{\frac{-j2\pi 2}{4}} = -e^{\frac{-j2\pi 0}{4}}$ and $e^{\frac{-j2\pi 3}{4}} = -e^{\frac{-j2\pi 1}{4}}$. This allows for a reduction in the complexity of the multiplications since we can share multiplications across two terms.

Figure 5.2 shows the butterfly diagram for the four point **FFT**. We can see that the first stage is two 2 point **FFT** operations performed on the even (top butterfly) and odd (bottom butterfly) input values. The output of the odd 2 point **FFT**s are multiplied by the appropriate twiddle factor. We can use two twiddle factors for all four output terms by using the reduction shown in Equation 5.12.

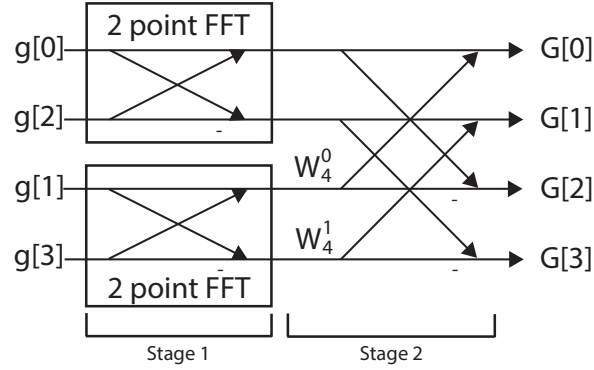


Figure 5.2: A four point **FFT** divided into two stages. Stage 1 has uses two 2 point **FFT**s – one 2 point **FFT** for the even input values and the other 2 point **FFT** for the odd input values. Stage 2 performs the remaining operations to complete the **FFT** computation as detailed in Equation 5.12.

We are seeing the beginning of trend that allows the a reduction in complexity from $\mathcal{O}(n^2)$ operations for the **DFT** to $\mathcal{O}(n \log n)$ operations for the **FFT**. The key idea is building the computation through recursion. The 4 point **FFT** uses two 2 point **FFT**s. This extends to larger **FFT** sizes. For example, an 8 point **FFT** uses two 4 point **FFT**s, which in turn each use two 2 point **FFT**s (for a total of four 2 point **FFT**s). An 16 point **FFT** uses two 8 point **FFT**s, and so on.

How many 2 point **FFT**s are used in a 32 point **FFT**? How many are there in a 64 point **FFT**? How many 4 point **FFT**s are required for a 64 point **FFT**? How about a 128 point **FFT**? What is the general formula for 2 point, 4 point, and 8 point **FFT**s in an N point **FFT** (where $N > 8$)?

Now let us formally derive the relationship, which provides a general way to describe the recursive structure of the **FFT**. Assume that we are calculating an N point **FFT**. The formula for calculating the frequency domain values $G[]$ given the input values $g[]$ is:

$$G[k] = \sum_{n=0}^{N-1} g[n] \cdot e^{-\frac{j2\pi kn}{N}} \text{ for } k = 0, \dots, N-1 \quad (5.13)$$

We can divide this equation into two parts, one that sums the even components and one that sums the odd components.

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{-\frac{j2\pi k(2n)}{N}} + \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{-\frac{j2\pi k(2n+1)}{N}} \quad (5.14)$$

The first part of this equation deals with the even inputs, hence the $2n$ terms in both $g[]$ and in the exponent of e . The second part corresponds to the odd inputs with $2n+1$ in both places. Also note that the sums now go to $N/2-1$ in both cases which should make sense since we have divided them into two halves.

We transform Equation 5.14 to the following:

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{-\frac{j2\pi kn}{N/2}} + \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{-\frac{j2\pi k(2n)}{N}} \cdot e^{-\frac{j2\pi k}{N}} \quad (5.15)$$

In the first summation (even inputs), we simply move the 2 into the denominator so that it is now $N/2$. The second summation (odd inputs) uses the power rule to separate the +1 leaving two complex exponentials. We can further modify this equation to

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-j2\pi kn}{N/2}} + e^{\frac{-j2\pi k}{N}} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-j2\pi kn}{N/2}} \quad (5.16)$$

Here we only modify the second summation. First we pull one of the complex exponentials outside of the summation since it does not depend upon n . And we also move the 2 into the denominator as we did before in the first summation. Note that both summations now have the same complex exponential $e^{\frac{-j2\pi kn}{N/2}}$. Finally, we simplify this to

$$G[k] = A_k + W_N^k B_k \quad (5.17)$$

where A_k and B_k are the first and second summations, respectively. And recall that $W = e^{-j2\pi}$. This completely describes an N point **FFT** by separating even and odd terms into two summations.

For reasons that will become clear soon, let us assume that we only want to use Equation 5.17 to calculate the first $N/2$ terms, i.e., $G[0]$ through $G[N/2 - 1]$. And we will derive the remaining $N/2$ terms, i.e., those from $G[N/2]$ to $G[N - 1]$ using a different equation. While this may seem counterintuitive or even foolish (why do more math than necessary?), you will see that this will allow us to take advantage of even more symmetry, and derive a pattern as we have seen in the 4 point **FFT**.

In order to calculate the higher frequencies $G[N/2]$ to $G[N - 1]$, let us derive the same equations but this time using $k = N/2, N/2 + 1, \dots, N/2 - 1$. Thus, we wish to calculate

$$G[k + N/2] = \sum_{n=0}^{N-1} g[n] \cdot e^{\frac{-j2\pi(k+N/2)n}{N}} \text{ for } k = 0, \dots, N/2 - 1 \quad (5.18)$$

This is similar to Equation 5.13 with different indices, i.e., we replace k from Equation 5.13 with $k + N/2$. Using the same set of transformations that we did previously, we can move directly to the equivalent to Equation 5.16, but replacing all instances of k with $k + N/2$ which yields

$$G[k + N/2] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-j2\pi(k+N/2)n}{N/2}} + e^{\frac{-j2\pi(k+N/2)}{N}} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-j2\pi(k+N/2)n}{N/2}} \quad (5.19)$$

We can reduce the complex exponential in the summations as follows:

$$e^{\frac{-j2\pi(k+N/2)n}{N/2}} = e^{\frac{-j2\pi kn}{N/2}} \cdot e^{\frac{-j2\pi(N/2)n}{N/2}} = e^{\frac{-j2\pi kn}{N/2}} \cdot e^{-j2\pi n} = e^{\frac{-j2\pi kn}{N/2}} \cdot 1 \quad (5.20)$$

The first reduction uses the power rule to split the exponential. The second reduction cancels the term $N/2$ in the second exponential. The final reduction uses that fact that n is a non-negative integer, and thus $e^{-j2\pi n}$ will always be a rotation of multiple of 2π . This means that this term is always equal to 1.

Now let us tackle the second complex exponential

$$e^{\frac{-j2\pi(k+N/2)}{N}} = e^{\frac{-j2\pi k}{N}} \cdot e^{\frac{-j2\pi N/2}{N}} = e^{\frac{-j2\pi k}{N}} \cdot e^{-j\pi} = -e^{\frac{-j2\pi k}{N}} \quad (5.21)$$

The first reduction splits the exponential using the power rule. The second reduction does some simplifications on the second exponential. We get the final term by realizing that $e^{-j\pi} = -1$.

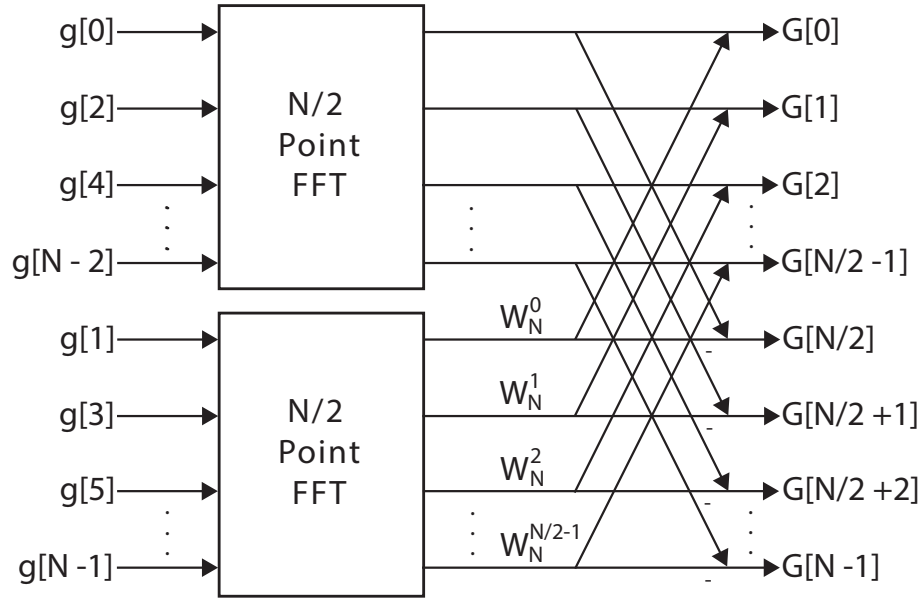


Figure 5.3: Building an N point FFT from two $N/2$ point FFTs. The upper $N/2$ point FFT is performed on the even inputs; the lower $N/2$ FFT uses the odd inputs.

By substituting Equations 5.20 and 5.21 into Equation 5.19, we get

$$G[k + N/2] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{-\frac{j2\pi kn}{N/2}} - e^{-\frac{j2\pi k}{N}} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{-\frac{j2\pi kn}{N/2}} \quad (5.22)$$

Note the similarity to Equation 5.16. We can put it in terms of Equation 5.17 as

$$G[k + N/2] = A_k - W_N^k B_k \quad (5.23)$$

We can use Equations 5.17 and 5.23 to create an N point FFT from two $N/2$ point FFTs. Remember that A_k corresponds to the even input values, and B_k is a function of the odd input values. Equation 5.17 covers the first $N/2$ terms, and Equation 5.23 corresponds to the higher $N/2$ frequencies.

Figure 5.3 shows an N point FFT derived from two $N/2$ point FFTs. A_k corresponds to the top $N/2$ FFT, and B_k is the bottom $N/2$ FFT. The output terms $G[0]$ through $G[N/2 - 1]$ are multiplied by W_N^0 while the output terms $G[N/2]$ through $G[N - 1]$ are multiplied by $-W_N^0$. Note that the inputs $g[]$ are divided into even and odd elements feeding into the top and bottom $n/2$ point FFTs, respectively.

We can use the general formula for creating the FFT that was just derived to recursively create the $N/2$ point FFT. That is, each of the $N/2$ point FFTs can be implemented using two $N/4$ point FFTs. And each $N/4$ point FFT uses two $N/8$ point FFTs, and so on until we reach the base case, a 2 point FFT.

Figure 5.4 shows an 8 point FFT and highlights this recursive structure. The boxes with the dotted lines indicate different sizes of FFT. The outermost box indicates an 8 point FFT. This is composed by two 4 point FFTs. Each of these 4 point FFTs have two 2 point FFTs for a total of four 2 point FFTs.

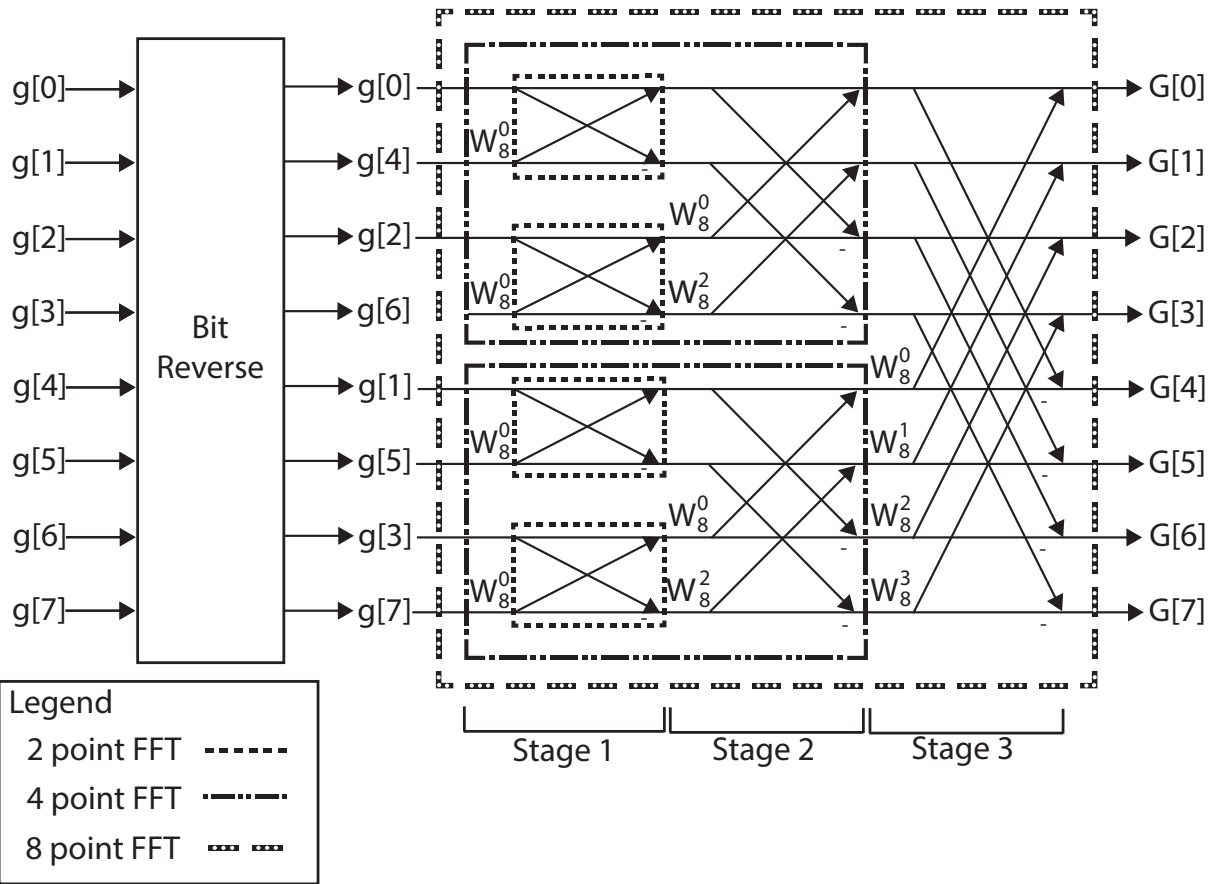


Figure 5.4: An 8 point **FFT** built recursively. There are two 4 point **FFTs**, which each use two 2 point **FFTs**. The inputs must be reordered to even and odd elements twice. This results in reordering based upon the bit reversal of the indices.

Also note that the inputs must be reordered before they are feed into the 8 point FFT. This is due to the fact that the different $N/2$ point FFTs take even and odd inputs. The upper four inputs correspond to even inputs and the lower four inputs have odd indices. However, they are reordered twice. If we separate the even and odd inputs once we have the even set $\{g[0], g[2], g[4], g[6]\}$ and the odd set $\{g[1], g[3], g[5], g[7]\}$. Now let us reorder the even set once again. In the even set $g[0]$ and $g[4]$ are the even elements, and $g[2]$ and $g[6]$ are the odd elements. Thus reordering it results in the set $\{g[0], g[4], g[2], g[6]\}$. The same can be done for the initial odd set yielding the reordered set $\{g[1], g[5], g[3], g[7]\}$.

The final reordering is done by swapping values whose indices are in bit reversed order. Table 5.1 shows the indices and their three bit binary values. The table shows the eight indices for the 8 point FFT, and the corresponding binary value for each of those indices in the second column. The third column is the bit reversed binary value of the second column. And the last column is the decimal number corresponding the reversed binary number.

Table 5.1: The index, three bit binary value for that index, bit reversed binary value, and the resulting bit reversed index.

Index	Binary	Reversed Binary	Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Looking at the first row, the initial index 0, has a binary value of 000, which when reversed remains 000. Thus this index does not need to be swapped. Looking at Figure 5.4 we see that this is true. $g[0]$ remains in the same location. In the second row, the index 1 has a binary value 001. When reversed this is 100 or 4. Thus, the data that initially started at index 1, i.e., $g[1]$ should end up in the fourth location. And looking at index 4, we see the bit reversed value is 1. Thus $g[1]$ and $g[4]$ are swapped.

This bit reversal process works regardless of the input size of the FFT, assuming that the FFT is a power of two. FFTs are commonly a power of two since this allows them to be recursively implemented.

In an 32 point FFT, index 1 is swapped with which index? Which index is index 2 is swapped with?

This completes our mathematical treatment of the FFT. There are plenty of more details about the FFT and how to optimize it. You may think that we spent too much time already discussing the finer details of the FFT; this is a book on parallel programming for FPGAs and not on digital signal processing. This highlights an important part of creating an optimum hardware implementation – the designer must have a good understanding of the algorithm under development. Without that,

it is difficult to create a good implementation. The next section deals with how to create a good `FFT` implementation.

5.2 Baseline Implementation

In the remainder of this chapter, we discuss different methods to implement the Cooley-Tukey `FFT` [16] algorithm using the Vivado[®] HLS tool. This is the same algorithm that we described in the previous section. We start with a common version of the code, and then describe how to restructure it to achieve a better hardware design.

When performed sequentially, the $\mathcal{O}(n \log n)$ operations in the `FFT` require $\mathcal{O}(n \log n)$ time steps. Typically, a parallel implementation will perform some portion of the `FFT` in parallel. One common way of parallelizing the `FFT` is to organize the computation into $\log n$ stages, as shown in Figure 5.8. The operations in each stage are dependent on the operations of the previous stage, naturally leading to a pipelining across the tasks. Such an architecture allows $\log n$ `FFTs` to be computed simultaneously with a task interval determined by the architecture of each stage. We discuss task pipelining using the **dataflow** directive in Section 5.4.

Each stage in the `FFT` also contains significant parallelism, since each butterfly computation is independent of other butterfly computations in the same stage. In the limit, performing $n/2$ butterfly computations every clock cycle with a Task Interval of 1 can allow the entire stage to be computed with a Task Interval of 1. When combined with a dataflow architecture, all of the parallelism in the `FFT` algorithm can be exploited. Note, however that although such an architecture can be constructed, it is almost never used except for very small signals, since an entire new block of `SIZE` samples must be provided every clock cycle to keep the pipeline fully utilized. For instance, a 1024-point `FFT` of complex 32-bit floating point values, running at 250 MHz would require $1024 \text{ points} \times (8 \text{ bytes/point}) \times 250 \times 10^9 \text{ Hz} = 1 \text{ Terabyte/second}$ of data into the FPGA. In practice, a designer must match the computation architecture to the data rate required in a system.

Assuming a clock rate of 250 MHz and one sample received every clock cycle, approximately how many butterfly computations must be implemented to process every sample with a 1024-point `FFT`? What about for a 16384-point `FFT`?

In the remainder of this section, we describe the optimization of an `FFT` with the function prototype `void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE])` where `DTYPE` is a user customizable data type for the representation of the input data. This may be `int`, `float`, or a fixed point type. For example, `#define DTYPE int` defines `DTYPE` as an `int`. Note that we choose to implement the real and imaginary parts of the complex numbers in two separate arrays. The `X_R` array holds the real input values, and the `X_I` array holds the imaginary values. `X_R[i]` and `X_I[i]` hold the i th complex number in separate real and imaginary parts.

There is one change in the `FFT` implementation that we describe in this section. Here we perform an `FFT` on complex numbers. The previous section uses only real numbers. While this may seem like a major change, the core ideas stay the same. The only differences are that the data has two values (corresponding to the real and imaginary part of the complex number), and the operations (add, multiply, etc.) are complex operations.

This function prototype forces an in-place implementation. That is, the output data is stored in the same array as the input data. This eliminates the need for additional arrays for the output data, which reduces the amount of memory that is required for the implementation. However, this may limit the performance due to the fact that we must read the input data and write the output data to the same arrays. Using separate arrays for the output data is reasonable if it can increase the performance. There is always a tradeoff between resource usage and performance; the same is true here. The best implementation depends upon the application requirements (e.g., high throughput, low power, size of FPGA, size of the `FFT`, etc.).

We start with code for an `FFT` that would be typical for a software implementation. Figure 5.5 shows a nested three `for` loop structure. The outer `for` loop, labeled `stage_loop` implements one stage of the `FFT` during each iteration. There are $\log_2(N)$ stages where N is the number of input samples. The stages are clearly labeled in Figure 5.4; this 8 point `FFT` has $\log_2(8) = 3$ stages. You can see that each stage performs the same amount of computation, or the same number of butterfly operations. In the 8 point `FFT`, each stage has four butterfly operations.

For an N point `FFT`, how many butterfly operations are there in each stage? How many total butterfly operations are there for the entire `FFT`?

The second `for` loop, labeled `butterfly_loop`, performs all of the butterfly operations for the current stage. `butterfly_loop` has another nested `for` loop, labeled `dft_loop`. Each iteration of `dft_loop` performs one butterfly operation. Remember that we are dealing with complex numbers and must perform complex additions and multiplications.

The first line in `dft_loop` determines the offset of the butterfly. Note that the “width” of the butterfly operations changes depending upon the stage. Looking at Figure 5.4, Stage 1 performs butterfly operations on adjacent elements, Stage 2 performs butterfly operations on elements with index differing by two, and Stage 3 performs butterfly operations on elements with index differing by four. This difference is computed and stored in the `i_lower` variable. Notice that this offset, stored in the variable `numBF`, is different in every stage.

The remaining operations in `dft_loop` perform multiplication by the twiddle factor and an addition or subtraction operation. The variables `temp.R` and `temp.I` hold the real and imaginary portions of the data after multiplication by the twiddle factor W . The variables `c` and `s` are the real and imaginary parts of W , which is calculated using the `sin()` and `cos()` builtin functions. We could also use the CORDIC, such as the one developed in Chapter 3, to have more control over the implementation. Lastly, elements of the `X.R[]` and `X.I[]` arrays are updated with the result of the butterfly computation.

`dft_loop` and `butterfly_loop` each execute a different number of times depending upon the stage. However the total number of times that the body of `dft_loop` is executed in one stage is constant. The number of iterations for the `butterfly for` loop depends upon the number of unique W twiddle factors in that stage. Referring again to Figure 5.4, we can see that Stage 1 uses only one twiddle factor, in this case W_8^0 . Stage 2 uses two unique twiddle factors and Stage 3 uses four different W values. Thus, `butterfly_loop` has only one iteration in Stage 1, 2 iterations in stage 2, and four iterations in stage 3. Similarly, the number of iterations of `dft_loop` changes. It iterates four times for an 8 point `FFT` in Stage 1, two times in Stage 2, and only one time in stage 3. However in every stage, the body of `dft_loop` is executed the same number of times in total, executing a total of four butterfly operations for each stage an 8 point `FFT`.


```

void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE]) {
    DTYPE temp_R; // temporary storage complex variable
    DTYPE temp_I; // temporary storage complex variable
    int i, j, k; // loop indexes
    int i_lower; // Index of lower point in butterfly
    int step, stage, DFTpts;
    int numBF; // Butterfly Width
    int N2 = SIZE2; // N2=N>>1

    bit_reverse(X_R, X_I);

    step = N2;
    DTYPE a, e, c, s;

    stage_loop:
    for (stage = 1; stage <= M; stage++) {
        // Do M stages of butterflies
        DFTpts = 1 << stage; // DFT = 2^stage = points in sub DFT
        numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT
        k = 0;
        e = -6.283185307178 / DFTpts;
        a = 0.0;
        // Perform butterflies for j-th stage
        butterfly_loop:
        for (j = 0; j < numBF; j++) {
            c = cos(a);
            s = sin(a);
            a = a + e;
            // Compute butterflies that use same W**k
            dft_loop:
            for (i = j; i < SIZE; i += DFTpts) {
                i_lower = i + numBF; // index of lower point in butterfly
                temp_R = X_R[i_lower] * c - X_I[i_lower] * s;
                temp_I = X_I[i_lower] * c + X_R[i_lower] * s;
                X_R[i_lower] = X_R[i] - temp_R;
                X_I[i_lower] = X_I[i] - temp_I;
                X_R[i] = X_R[i] + temp_R;
                X_I[i] = X_I[i] + temp_I;
            }
            k += step;
        }
        step = step / 2;
    }
}

```

Figure 5.5: A common implementation for the **FFT** using three nested **for** loops. While this may work well running as software on a processor, it is far from optimal for a hardware implementation.

Vivado[®] HLS performs significant static analysis on each synthesized function, including computing bounds on the number of times each loop can execute. This information comes from many sources, including variable bitwidths, ranges, and `assert()` functions in the code. When combined with the loop II, Vivado[®] HLS can compute bounds on the latency or interval of the `FFT` function. In some cases (usually when loop bounds are variable or contain conditional constructs), the tool is unable to compute the latency or interval of the code and returns “?”. When synthesizing the code in Figure 5.5, Vivado[®] HLS may not be able to determine the number of times that `butterfly_loop` and `dft_loop` iterate because these loops have variable bounds.

The `tripcount` directive enables the user to specify to the Vivado[®] HLS tool more information about the number of times a loop is executed which can be used by the tool to analyze the performance of the design. It takes three optional arguments `min`, `max`, and `average`. In this code, we could add a directive to `dft_loop`. By applying this directive, the Vivado[®] HLS tool can calculate bounds on the latency and interval value for the loop and the overall design. Note that since the Vivado[®] HLS tool uses the numbers that you provide, if you give the tool an incorrect tripcount then the reported task latency and task interval will be incorrect – garbage in, garbage out.

What is the appropriate way to use the `trip count` directive for the `FFT` in Figure 5.5? Should you set the `max`, `min`, and/or `average` arguments? Would you need to modify the tripcount arguments if the size of the `FFT` changes?

5.3 Bit Reversal

We have not talked about the bit reverse function, which swaps the input data values so that we can perform an in-place `FFT`. This means that the inputs values are mixed, and the result is that the output data is in the correct order. We discuss that function in some detail now.

Figure 5.6 shows one possible implementation of the bit reverse function. It divides the code into two functions. The first is the bit reversal function (`bit_reverse`), which reorders data in the given arrays so that each data is located at a different index in the array. This function calls another function, `reverse_bits`, which takes an input integer and returns the bit reversed value of that input.

Let us start with a brief overview of the `reverse_bits` function. The function goes bit by bit through the `input` variable and shifts it into the `rev` variable. The `for` loop body consists of a few bitwise operations that reorder the bits of the input. Although these operations are individually not terribly complex, the intention of this code is that the `for` loop is completely unrolled and Vivado[®] HLS can identify that the bits of the input can simply be wired to the output. As a result, the implementation of the `reverse_bits` function should require no logic resources at all, but only wires. This is a case where unrolling loops greatly simplifies the operations that must be performed. Without unrolling the loop, the individual ‘or’ operations must be performed sequentially. Although this loop can be pipelined, the ‘or’ operation would still be implemented in logic resources in the FPGA and executing the the loop would have a latency determined by the number of bits being reversed (`\glsl{fft}_BITS` in this case).

```

#define FFT_BITS 10 // Number of bits of FFT, i.e., log(1024)
#define SIZE 1024 // SIZE OF FFT
#define SIZE2 SIZE >> 1 // SIZE/2
#define DTYPE int

unsigned int reverse_bits(unsigned int input) {
    int i, rev = 0;
    for (i = 0; i < FFT_BITS; i++) {
        rev = (rev << 1) | (input & 1);
        input = input >> 1;
    }
    return rev;
}

void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE]) {
    unsigned int reversed;
    unsigned int i;
    DTYPE temp;

    for (i = 0; i < SIZE; i++) {
        reversed = reverse_bits(i); // Find the bit reversed index
        if (i < reversed) {
            // Swap the real values
            temp = X_R[i];
            X_R[i] = X_R[reversed];
            X_R[reversed] = temp;

            // Swap the imaginary values
            temp = X_I[i];
            X_I[i] = X_I[reversed];
            X_I[reversed] = temp;
        }
    }
}

```

Figure 5.6: The first stage in our **FFT** implementation reorders the input data. This is done by swapping the value at index i in the input array with the value at the bit reversed index corresponding to i . The function `reverse_bits` gives the bit reversed value corresponding to the input argument. And the function `bit_reverse` swaps the values in the input array.

```

void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
    #pragma HLS dataflow
    DTYPE Stage1_R[SIZE], Stage1_I[SIZE];
    DTYPE Stage2_R[SIZE], Stage2_I[SIZE];
    DTYPE Stage3_R[SIZE], Stage3_I[SIZE];

    bit_reverse(X_R, X_I, Stage1_R, Stage1_I);
    fft_stage_one(Stage1_R, Stage1_I, Stage2_R, Stage2_I);
    fft_stages_two(Stage2_R, Stage2_I, Stage3_R, Stage3_I);
    fft_stage_three(Stage3_R, Stage3_I, OUT_R, OUT_I);
}

```

Figure 5.7: The code divides an 8 point **FFT** into four stages, each of which is a separate function. The `bit_reverse` function is the first stages. And there are three stages for the 8 point **FFT**.

What is the latency of the `reverse_bits` function when no directives are applied? What is the latency when the loop is pipelined? What is the latency when the whole function is pipelined?

It is tempting to “blindly” apply directives in order to achieve a better design. However, this can be counterproductive. The best designer has a deep understanding of both the application and the available optimizations and carefully considers these together to achieve the best results.

Now let us optimize the parent `bit_reverse` function. This function has a single **for** loop that iterates through each index of the input arrays. Note that there are two input arrays `X_R[]` and `X_I[]`. Since we are dealing with complex numbers, we must store both the real portion (in the array `X_R[]`), and the imaginary portion (in the array `X_I[]`). `X_R[i]` and `X_I[i]` holds the real and imaginary values of the *i*-th input. In each iteration of the **for** loop, we find the index reversed value by calling the `reverse_bits` function. Then we swap both the real and imaginary values stored in the index *i* and the index returned by the function `reverse_bits`. Note that as we go through all `SIZE` indices, we will eventually hit the reversed index for every value. Thus, the code only swaps values the first time based on the condition **if**(*i* < `reversed`).

5.4 Task Pipelining

Dividing the **FFT** algorithm into stages enables Vivado[®] HLS to generate an implementation where different stages of the algorithm are operating on different data sets. This optimization, called **task pipelining** is enabled using the **dataflow** directive. This is a common hardware optimization, and thus is relevant across a range of applications.

We can naturally divide the **FFT** algorithm into $\log_2(N + 1)$ stages where *N* is the number of points of the **FFT**. The first stage swaps each element in the input array with the element located at the bit reversed address in the array. After this bit reverse stage, we perform $\log_2(N)$ stages of butterfly operations. Each of these butterfly stages has the same computational complexity. Figure

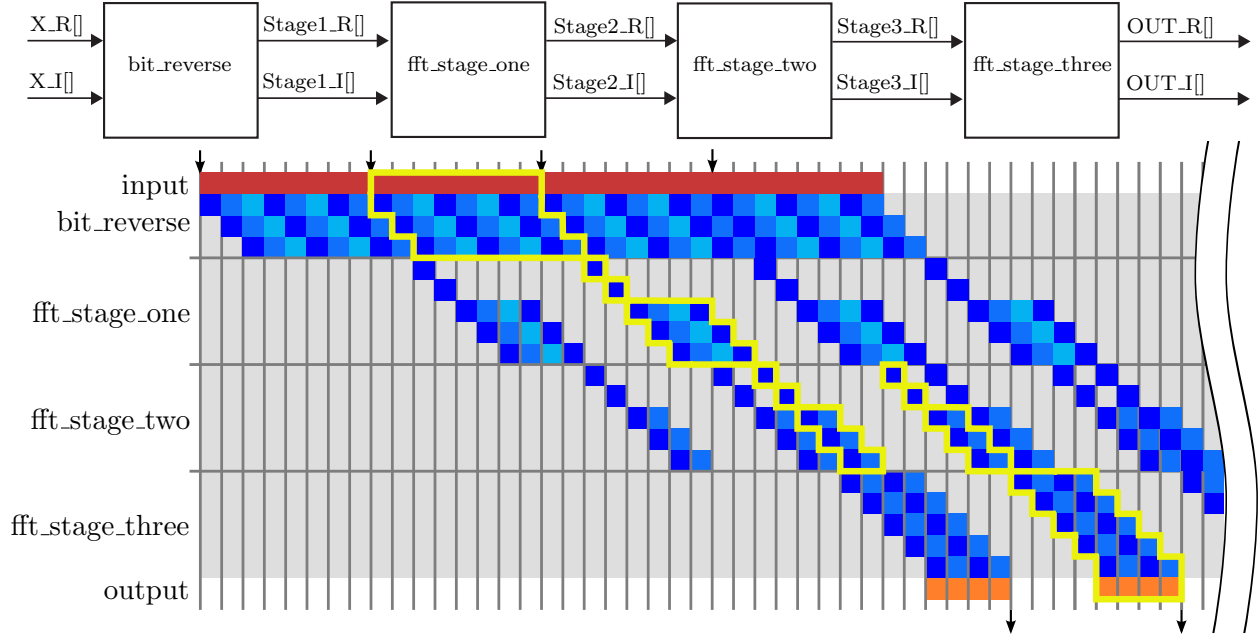


Figure 5.8: Dividing the `FFT` into different stages allows for task pipelining across each of these stages. The figure shows an example with three `FFT` stages (i.e., an 8 point `FFT`). The figure shows four 8 point `FFT` executing at the same time.

`5.7` describes how to devide an 8 point `FFT` into four separate tasks. The code has separate functions for each of the tasks: `bit_reverse`, `fft_stage_one`, `fft_stage_two`, and `fft_stage_three`. Each stage has two input arrays and two output arrays: one for the real portion and one for the imaginary portion of the complex numbers. Assume that the `DTYPE` is defined elsewhere, e.g., as an `int`, `float` or a fixed point data type.

Refactoring the `FFT` code allows us to perform `task pipelining`. Figure `5.8` gives an example of this. In this execution, rather than wait for the first task to complete all four four functions in the code before the second task can begin, we allow the second task to start after the first task has only completed the first function `bit_reverse`. The first task continues to execute each stage in the pipeline in order, followed by the remaining tasks in order. Once the pipeline is full, all four subfunctions are executing concurrently, but each one is operating on different input data. Similarly, there are four 8 point `FFT`s being computed simultaneously, each one executing on a different component of the hardware. This shown in the middle portion of Figure `5.8`. Each of the vertical four stages represents one 8 point `FFT`. And the horizontal denotes increasing time. Thus, once we start the fourth 8 point `FFT`, we have four `FFT`s running simultaneously.

The **dataflow** directive can construct separate pipeline stages (often called `processes`) from both functions and loops. The code in Figure `5.7` uses functions only, but we could achieve a similar result with four loops instead of four functions. In fact, this result could be achieved by unrolling the outer `stage_loop` in the original code either explicitly or using `#pragma HLS unroll`.

The **dataflow** directive and the **pipeline** directive both generate circuits capable of pipelined execution. The key difference is in the granularity of the pipeline. The **pipeline** directive constructs an architecture that is efficiently pipelined at the cycle level and is characterized by the II of the pipeline. Operators are statically scheduled and if the II is greater than one, then operations can be shared on the same operator. The **dataflow** directive constructs an architecture that is efficiently pipelined for operations that take a (possibly unknown) number of clock cycles, such as the the

behavior of a loop operating on a block of data. These coarse-grained operations are not statically scheduled and the behavior is controlled dynamically by the handshake of data through the pipeline. In the case of the `FFT`, each stage is an operation on a block of data (the whole array) which takes a large number of cycles. Within each stage, loops execute individual operations on the data in a block. Hence, this is a case where it often makes sense to use the **dataflow** directive at the toplevel to form a coarse-grained pipeline, combined with the **pipeline** directive within each loop to form fine-grained pipelines of the operations on each individual data element.

The **dataflow** directive must implement memories to pass data between different processes. In the case when Vivado[®] HLS can determine that processes access data in sequential order, it implements the memory using a FIFO. This requires that data is written into an array in the same order that it is read from the array. If this is not the case, or if Vivado[®] HLS can not determine if this streaming condition is met, then the memory can be implemented using a ping-pong buffer instead. The ping-pong buffer consists of two (or more) conceptual blocks of data, each the size of the original array. One of the blocks can be written by the source process while another block is read by the destination process. The term “ping-pong” comes from the fact that the reading and writing to each block of data alternates in every execution of the task. That is, the source process will write to one block and then switch to the other block before beginning the next task. The destination process reads from the block that the producer is not writing to. As a result, the source and destination processes can never writing and reading from the same block at the same time.

A ping-pong buffer requires enough memory to store each communication array at least twice. FIFOs can often be significantly smaller, although determining a minimal size for each fifo is often a difficult design problem. Unlike a FIFO, however, the data in a ping-pong buffer can be written to and read from in any order. Thus, FIFOs are generally the best choice when the data is produced and consumed in sequential order and ping-pong buffers are a better choice when there is not such regular data access patterns.

Using the **dataflow** directive effectively still requires the behavior of each individual process to be optimized. Each individual process in the pipeline can still be optimized using techniques we have seen previously such as code restructuring, pipelining, and unrolling. For example, we have already discussed some optimizations for the `bit_reverse` function in Section 5.3. In general, it is important to optimize the individual tasks while considering overall toplevel performance goals. Many times it is best to start with small functions and understand how to optimize them in isolation. As a designer, it is often easier to comprehend what is going on in a small piece of code and hopefully determine the best optimizations quickly. After optimizing each individual function, then you can move up the hierarchy considering larger functions given particular implementations of low level functions, eventually reaching the toplevel function.

However, the local optimizations must be considered in the overall scope of the goals. In particular for dataflow designs the achieved interval for the overall pipeline can never be smaller than the interval of each individual process. Looking again at Figure 5.7, assume that `bit_reverse` has an interval of 8 cycles, `fft_stage_one` takes 12 cycles, `fft_stage_two` requires 12 cycles, and `fft_stage_three` takes 14 cycles. When using **dataflow**, the overall task interval is 14, determined by the maximum of all of the tasks/functions. This means that you should be careful in balancing optimizations across different processes with the goal of creating a balanced pipeline where the interval of each process is approximately the same. In this example, improving the interval of the `bit_reverse` function cannot improve the overall interval of the `fft` function. In fact, it might be beneficial to increase the latency of the `bit_reverse` function, if it can be achieved with significantly fewer resources.

5.5 Conclusion

The overall goal is to create the most optimal design, which is a function of your application needs. This may be to create the smallest implementation. Or the goal could be creating something that can perform the highest throughput implementation regardless of the size of the FPGA or the power/energy constraints. Or the latency of delivering the results may matter if the application has real-time constraints. All of the optimizations change these factors in different ways.

In general, there is no one algorithm on how to optimize your design. It is a complex function of the application, design constraints, and the inherent abilities of the designer himself. Yet, it is important that the designer have a deep understanding of the application itself, the design constraints, and the abilities of the synthesis tool.

We attempted to illustrate these bits of wisdom in this chapter. While the `FFT` is a well studied algorithm, with a large number of known hardware implementation tricks, it still serves as a good exemplar for high-level synthesis. We certainly did not give all of the tricks for optimization; we leave that as an exercise in Chapter ?? where we task the designer to create a simple orthogonal frequency-division multiplexing receiver. The core of this is an `FFT`. Regardless, we attempted to provide some insight into the key optimizations here, which we hope serve as a guide to how to optimize the `FFT` using the Vivado[®] HLS tool.

First and foremost, understand the algorithm. We spent a lot of time explaining the basics of the `FFT`, and how it relates to the `DFT`. We hope that the reader understands that this is the most important part of building optimal hardware. Certainly, the designer could translate C/MATLAB/Java/Python code into Vivado[®] HLS and get a working implementation. And that same designer could somewhat blindly apply directives to achieve better results. But that designer is not going to get anywhere close to optimal results without a deep understanding of the algorithm itself.

Second, we provide an introduction to task level pipelining using the `dataflow` directive. This is a powerful optimization that is not possible through code restructuring. I.e., the designer must use this optimization to get such a design. Thus, it is important that the designer understand its power, drawbacks, and usage.

Additionally, we build upon some of the optimizations from previous chapters, e.g., loop unrolling and pipelining. All of these are important to get an optimized `FFT` hardware design. While we did not spend too much time on these optimizations, they are extremely important.

Finally, we tried to impress on the reader that these optimizations cannot be done in isolation. Sometimes the optimizations are independent, and they can be done in isolation. For example, we can focus on one of the tasks (e.g., in the `bit_reverse` function as we did in Section 5.3). But many times different optimizations will effect another. For example, the `inline` directive will effect the way the pipelining of a function. And in particular, the way that we optimize tasks/functions can propagate itself up through the hierarchy of functions. The takeaway is that it is extremely important that the designer understand the effects of the optimizations on the algorithm, both locally and globally.

Chapter 6

Sparse Matrix Vector Multiplication

Sparse matrix vector multiplication (SpMV) takes a sparse matrix, i.e., one in which most of its elements are zero, and multiplies it by a vector. The vector itself may be sparse as well, but often it is dense. This is a common operation in scientific applications, economic modeling, data mining, and information retrieval. For example, it is used as an iterative method for solving sparse linear systems and eigenvalue problems. It is an operation in PageRank and it is also used in computer vision, e.g., image reconstruction.

This chapter introduces several new HLS concepts, and reinforces some previously discussed optimization. One goal of the chapter is to introduce a more complex data structure. We use a **compressed row storage (CRS)** representation to hold the sparse matrix. Another goal is to show how to perform testing. We build a simple structure for a testbench that can be used to help determine if the code is functionally correct. This is an important aspect of hardware design, and Vivado® HLS makes it easy to test many aspects of the generated RTL with the same high-level C testbench. This is one of the big advantages of HLS over RTL design. We also show how you can perform C/RTL cosimulation using the testbench and Vivado® HLS tool. This is necessary to derive the performance characteristics for the different SpMV designs. Since the execution time depends upon the number of entries in the sparse matrix, we must use input data in order to determine the clock cycles for the task interval and task latency.

6.1 Background

Figure 6.1 shows an example of a 4×4 matrix **M** represented in two different ways. Figure 6.1 a) shows the normal representation of the matrix as a two-dimensional array of 16 elements. Each element is stored in its own location in the array. Figure 6.1 b) shows the same matrix represented in **CRS** format. The **CRS** representation is a data structure consisting of three arrays. The **values** array holds the value of each non-zero element in the matrix. The **columnIndex** and **rowPtr** arrays encode information about the location of these non-zero elements in the matrix. **columnIndex** stores the column of each element, while **rowPtr** contains the index in **values** of the first element in each row. The **CRS** format avoids storing values in the matrix that are zero, although there is nothing to prevent a zero from being explicitly represented in the **values** array. In the example, however, we see that the **values** array does not, in fact, contain any zero elements. The tradeoff is that some additional book-keeping information (the **columnIndex** and **rowPtr** arrays) is needed in order to properly interpret and manipulate the matrix. The **CRS** form is commonly used when large matrices contain only a small number of non-zero elements (typically 10 percent or less), enabling these matrices to be stored with less memory and manipulated with fewer operations. However, the

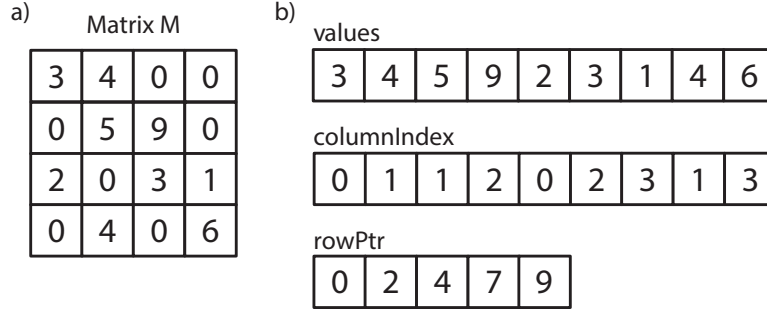


Figure 6.1: A 4×4 matrix \mathbf{M} represented in two different ways: as a ‘dense’ matrix stored in a two-dimensional array, and as a sparse matrix stored in the compressed row storage (CRS) form, a data structure consisting of three arrays.

CRS form has no requirements about the sparsity of the matrix and can be used for any matrix. This makes it a general approach that can be used for any matrix, but not necessarily the most efficient. The CRS form is also not the only efficient representation of sparse matrices. Depending on the characteristics of the matrix and the types of operations to be performed, other sparse representations can also be used.

More precisely, the CRS format uses a data structure consisting of three arrays: `values`, `colIndex`, and `rowPtr`. The `values` array and the `columnIndex` has an entry for each of the non-zero elements in the sparse matrix \mathbf{M} . These arrays represent the matrix \mathbf{M} stored in a row-wise fashion, i.e., left to right, and top to bottom. The data in the matrix is stored in the `values` array, while the `columnIndex` array contains the horizontal location of the data in the matrix. If `values[k]` represents M_{ij} then `colIndex[k] = j`. The array `rowPtr` has size $n + 1$ for an n -row matrix. `rowPtr[k]` contains the total number of elements in all the rows in the matrix prior to row k , with the first element `rowPtr[0] = 0` and the last element `rowPtr[n]` always giving the total number of non-zero elements the matrix. As a result, if `values[k]` represents M_{ij} , then `rowPtr[i] ≤ k < rowPtr[i+1]`. If row k contains any non-zero elements, then `rowPtr[k]` will contain the index of the first element in the row. Note that if there are rows in the matrix without a non-zero element, then values in the `rowPtr` array will repeat.

Looking at Figure 6.1 a), we can scan the matrix in row-major order to determine the `values` array in CRS form. Whenever we find a non-zero element, its value is stored at the next available index i in the `values` array and its column is stored at `columnIndex[i]`. In addition, whenever we start scanning a new row, we store the next available index i in the `rowPtr` array. As a result, the first element in the `rowPtr` array is always zero. Looking at Figure 6.1 b), we can also convert the matrix back to a two-dimensional array representation. The first step is to determine the number of elements in each row of the matrix from the `rowPtr` array. The number of elements in row i is the difference `rowPtr[i+1] - rowPtr[i]`. Then the row can be reconstructed by iterating through the `values` array starting at `values[rowPtr[i]]`. In our example matrix, the because the first two elements of the `rowPtr` array are 0 and 2, then we know that there are 2 elements in the first row, i.e., `values[0]` and `values[1]`. The first non-zero element in the `values` data structure, `values[0]`, is 3. This value is in column 0, since `columnIndex[0] = 0`. Similarly, the second non-zero value is the value 4 in column 1. The second row of the matrix has elements with $k \in [2, 4)$, the third row has elements with $k \in [4, 7)$, and so on. In this case, there are 9 non-zero entries, thus that last entry in the `rowPtr` data structure is 9.

```

#include "spmv.h"

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
    L1: for (int i = 0; i < NUM_ROWS; i++) {
        DTYPE y0 = 0;
        L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
            #pragma HLS unroll factor=8
            #pragma HLS pipeline
            y0 += values[k] * x[columnIndex[k]];
        }
        y[i] = y0;
    }
}

```

Figure 6.2: The baseline code for sparse matrix vector (SpMV) multiplication, which performs the operation $y = \mathbf{M} \cdot x$. The variables `rowPtr`, `columnIndex`, and `values` hold \mathbf{M} in `CRS` format. The first **for** loop iterates across the rows while the second nested **for** loop iterates across the columns of \mathbf{M} by multiplying each non-zero element by the corresponding element in the vector x which results in one element in the resulting vector y .

Given a 2-dimensional array representing a matrix, write the C code to convert the matrix to `CRS` form. Write the corresponding C code to convert the matrix in `CRS` form back to a 2-dimensional array.

It turns out that using the `CRS` form, we can multiply a sparse matrix with a vector relatively efficiently without explicitly converting the matrix back to a 2-dimensional array. In fact, for large matrices with a small number of non-zero elements, sparse matrix-vector multiply is much more efficient than the dense matrix-vector multiply we discussed in chapter 4. This is because we can compute the non-zero elements of the result by only looking at the non-zero elements of the operands.

6.2 Baseline Implementation

Figure 6.2 provides a baseline code for sparse matrix vector multiplication. The `spmv` function has five arguments. The arguments `rowPtr`, `columnIndex`, and `values` correspond to the input matrix \mathbf{M} in `CRS` format. These are equivalent to the data structures shown in Figure 6.1. The argument `y` holds the output result y and the argument `x` holds the input vector x to be multiplied by the matrix. The variable `NUM_ROWS` indicates the number of rows in the matrix \mathbf{M} . The variable `NNZ` is the number of non-zero elements in the matrix \mathbf{M} . Finally, the variable `SIZE` is the number of elements in the arrays `x` and `y`.

The outer **for** loop, labeled L1, iterates across each row of the matrix. Multiplying this row of the matrix with the vector x will produce one element of y . The inner loop labeled L2 loop across the elements in the columns of the matrix \mathbf{M} . The L2 loop iterates `rowPtr[i+1] - rowPtr[i]` times,

```

#ifndef __SPMV_H__
#define __SPMV_H__

const static int SIZE = 4; // SIZE of square matrix
const static int NNZ = 9; //Number of non-zero elements
const static int NUM_ROWS = 4; // SIZE;
typedef float DTYPE;
void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE]);

#endif // __MATRIXMUL_H__ not defined

```

Figure 6.3: The header file for `spmv` function and testbench.

corresponding to the number of non-zero entries in that row. For each entry, we read the value of the non-zero element of the \mathbf{M} matrix from the `values` array and multiply it by the corresponding value of the vector x read from the `x` array. That value is located at `columnIndex[k]` since the data structure `columnIndex` holds the column for the value k .

6.3 Testbench

Figure 6.4 shows a simple testbench for the `spmv` function. The testbench starts by defining the `matrixvector` function. This is a straightforward implementation of matrix vector multiplication. This does not assume a sparse matrix and does not use the `CRS` format. We will compare the output results from this function with the results from our `spmv` function.

A common testbench will implement a “golden” reference implementation of the function that the designer wishes to synthesis. The testbench will then compare the results of the golden reference with those generated from the code that is synthesized by the Vivado[®] HLS code. A best practice for the testbench is to use alternative implementations for the golden reference and the synthesizable code. This provides more assurance that both implementations are correct.

The testbench continues in the `main` function. Here we set the `fail` variable equal to 0 (later code sets this to 1 if the output data from `spmv` does not match that from the function `matrixvector`). Then we define a set of variables that correspond to the matrix \mathbf{M} , the input vector x and the output vector y . In case of \mathbf{M} , we have both the “normal” form and the CSR form (stored in the variables `values`, `columnIndex`, and `rowPtr`). The values of the \mathbf{M} matrix are the same as shown in Figure 6.1. We have two versions of the output vector y . The `y_sw` array stores the output from the function `matrixvector` and the `y` array has the output from the function `spmv`.

After defining all of the input and output variables, we call the `spmv` and `matrixvector` functions using the appropriate data. The following `for` loop compares the output results from both of the functions by comparing the elements from `y_sw` with those in `y`. If any of them are different, we set the `fail` flag equal to 1. Lastly, we print out the results of the test and then return the `fail` variable.

This testbench is relatively simple and probably insufficient to ensure that the implementation is correct. Primarily, it only tests one example matrix, whereas a better testbench would test

```
#include "spmv.h"
#include <stdio.h>

void matrixvector(DTYPE A[SIZE][SIZE], DTYPE *y, DTYPE *x)
{
    for (int i = 0; i < SIZE; i++) {
        DTYPE y0 = 0;
        for (int j = 0; j < SIZE; j++)
            y0 += A[i][j] * x[j];
        y[i] = y0;
    }
}

int main(){
    int fail = 0;
    DTYPE M[SIZE][SIZE] = {{3,4,0,0},{0,5,9,0},{2,0,3,1},{0,4,0,6}};
    DTYPE x[SIZE] = {1,2,3,4};
    DTYPE y_sw[SIZE];
    DTYPE values[] = {3,4,5,9,2,3,1,4,6};
    int columnIndex[] = {0,1,1,2,0,2,3,1,3};
    int rowPtr[] = {0,2,4,7,9};
    DTYPE y[SIZE];

    spmv(rowPtr, columnIndex, values, y, x);
    matrixvector(M, y_sw, x);

    for(int i = 0; i < SIZE; i++)
        if(y_sw[i] != y[i])
            fail = 1;

    if(fail == 1)
        printf("FAILED\n");
    else
        printf("PASS\n");

    return fail;
}
```

Figure 6.4: A simple testbench for our `spmv` function. The testbench generates one example and computes the matrix vector multiplication using a sparse (`spmv`) and non-sparse function (`matrixvector`).

multiple matrices. It is common, for instance, to randomly generate inputs for testing, in addition to explicitly verifying important corner-cases. In this case, we need to make sure to vary not only the values being operated on, which will be passed to our accelerator when it is executing, but also to vary the compile-time parameters which might be used to create different accelerators with different tradeoffs. The key difference is that we can randomly generate multiple data values to operate on and test them all in the same execution of the program, using multiple function calls. Compile-time parameters, on the other hand, require the code to be recompiled every time parameters change.

Create a more sophisticated testbench which generates multiple sets of test data using a random number generator. The compile-time parameters of the sparse matrix should be modifiable (e.g., `SIZE`, `NNZ`, etc.). Create an HLS synthesis script which executes the same code multiple times for different reasonable compile-time parameters.

6.4 Specifying Loop Properties

If you directly synthesize this code, you will get results for the clock period and utilization. However, you will not get the number of clock cycles either in terms of task latency or initiation interval. This is because this depends upon the input data, which is external to the `spmv` function itself. Primarily, the performance depends on the number of times the body of the inner loop is executed, which is equal to the number of non-zero elements in \mathbf{M} . We know that the number of non-zero elements is limited by the constant `NNZ` in the code, but it is possible to call the code with matrices of different sizes, so the actual number of iterations is data-dependent. In addition, the performance may vary depending on the location of the non-zero elements and the optimization directives utilized during synthesis. To make matters worse, the number of iterations depends on the input in a complex way and many potential inputs don't actually represent valid matrices. Thus, it is very difficult for a tool to determine the total number of clock cycles for the `spmv` function without complex analysis and additional information. Vivado[®] HLS is unable to perform this analysis.

What are the preconditions for the `spmv` function to work correctly? Prove that given these preconditions, the body of the inner loop does, in fact, execute exactly once for each non-zero element in the matrix.

There are several ways to leverage the tool to derive some performance estimates, however. One method is to provide the Vivado[®] HLS tool additional information about the loop bounds. This can be done using the `loop_tripcount` directive, which enables the designer to specify a minimum, maximum, and/or average number of iterations for each particular loop. By providing these values, the Vivado[®] HLS tool is capable of providing an estimate on the number of clock cycles.

Use the `loop_tripcount` directive to specify minimum, maximum, and/or average number of iterations for a loop with a variable bound. This enables the Vivado[®] HLS tool to provide an estimate on the number of clock cycles for the design. This does not impact the results of the synthesis; it only effects the synthesis report.

Add a **loop_tripcount** directive to the `spmv` function. The syntax for the pragma form of the directive is `#pragma HLS loop_tripcount min=X, max=Y, avg=Z` where X, Y, and Z are constant positive integers. Which loops require this directive? What happens to the synthesis report when you change the different parameters (min, max, and avg)? How does this effect the clock period? How does it change the utilization results?

The **loop_tripcount** directive enables the designer to get a rough idea about the performance of a function. This can enable comparison between different implementations of the same function either by applying different optimization directives or by restructuring the code itself. However, it may be difficult or impossible to determine the min, max, and avg parameters. It can also be difficult to provide tight bounds on the min and max parameters. If there is a testbench, there is another more accurate method to calculate the total number of clock cycles required for the `spmv` function. This is done by performing C/RTL cosimulation.

6.5 C/RTL Cosimulation

C/RTL cosimulation performs automatic testing of the register-transfer level (RTL) designs that are generated by the Vivado[®] HLS tool. It does this by executing the synthesized code together with the provided testbench. The execution is instrumented to record the input and output values for each execution of the synthesized code. The input values are converted to cycle-by-cycle *input vectors*. The input vectors are used in an RTL-level simulation of the generated RTL design and the resulting *output vectors* are captured. The testbench code can then be executed again replacing the synthesized code with the captured input and output values. The testbench code can then return a zero value (indicating success) or a non-zero value (indicating failure).

The C/RTL cosimulation flow combines the cycle-accurate RTL design generated from the Vivado[®] HLS tool with input values provided from the C testbench. As a result, it can generate accurate estimates of the performance of the generated RTL design which reflect any HLS optimizations, even in the presence of data-dependent behavior. The minimum, maximum, and average latency and interval of the synthesized function are automatically extracted after simulation completes.

Note that these numbers only correspond to the clock cycles derived from the input data used by the testbench. Thus, they are only as good as the testbench itself. To put it another way, if the testbench does not exercise the function in a manner that is consistent with how it will be used upon deployment, the results will not be accurate. In addition, the input testvectors are generated with idealized timing that does not accurately model the behavior of external interfaces. The actual performance may be lower if execution stalls waiting for input data, or if there is contention waiting for external memory access to complete. Nevertheless, it provides a convenient method for determining clock cycles that does not require the designer to estimate the loop bounds for a variable loop.

C/RTL cosimulation provides the latency for functions with variable loop bounds. It reports the minimum, maximum, and average clock cycles for function latency and function interval. These latency values are directly dependent upon the input data from the C testbench.

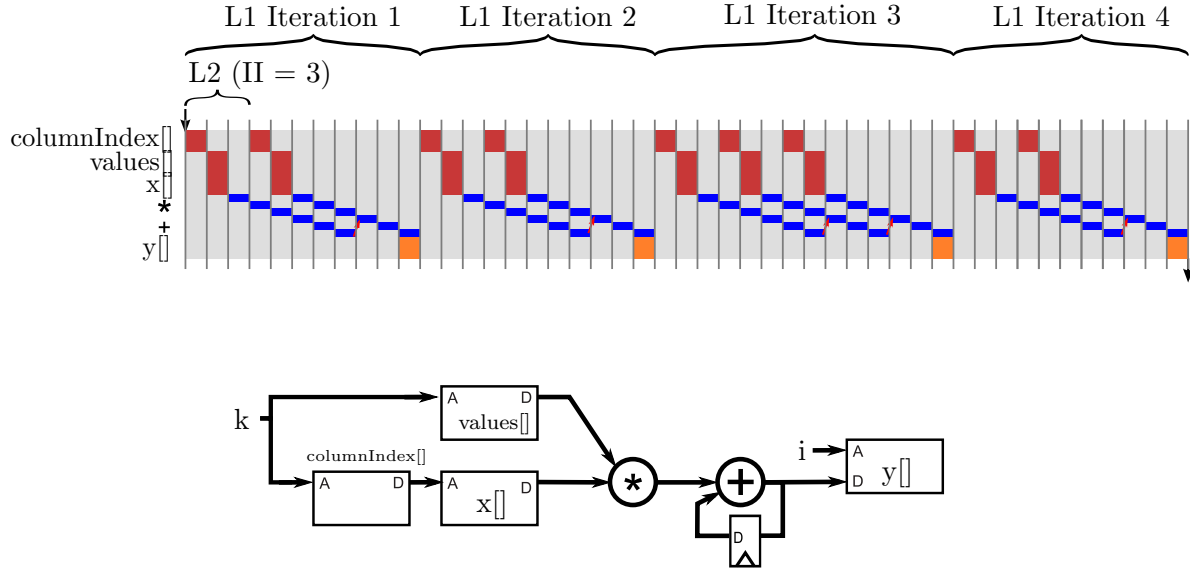


Figure 6.5: Architecture and behavior of the `spmv` code with a pipelined inner loop.

What are the minimum, maximum, and average clock cycles for the `spmv` function latency and function interval when using the testbench provided in Figure 6.4?

6.6 Loop Optimizations and Array Partitioning

Now that we have a method to gather all of the performance and utilization estimates from the Vivado® HLS tool, let us consider how to best optimize the function. Pipelining, loop unrolling, and data partitioning are the most common first approaches in optimizing a design. And the typical approach is to start with the innermost loop, and then move outwards as necessary.

In this example, pipelining the inner L2 loop is perhaps the first and easiest optimization to consider. This overlaps the execution of the consecutive iterations of this loop, which can result in a faster overall implementation. Without pipelining, each iteration of the L2 loop occurs sequentially. Note that the iterations of the L1 loop are still done sequentially.

Figure 6.5 illustrates the approximate manner in which the `spmv` function executes when pipelining the L2 **for** loop. Each iteration of the inner L2 loop is pipelined with $II=3$. Pipelining allows multiple iterations of the inner loop from the same iteration of the outer loop to execute concurrently. In this case, the II of the inner loop is limited by a recurrence through the accumulation. $II=3$ is achieved because we've assumed that the adder has a latency of 3 clock cycles. Iterations of the outer loop are not pipelined, so the inner loop must completely finish and flush the pipeline before the next iteration of the outer L2 loop begins.

Pipeline the innermost L2 **for** loop. This can be done by adding a pipeline directive to the `spmv` code from Figure 6.2. What is the achieved initiation interval (II)? What happens to the results as you specify an II argument, and increase or decrease the target II ?

Looking at this behavior, we see that there are several factors limiting the performance of the

loop. One factor is the recurrence through the adder that limits the achieved loop II. A second factor is that iterations of the outer loop are not pipelined. An efficient solution for sparse matrix-vector multiply would likely come close to using each multiplier and adder every clock cycle. This design is far from that.

In Section 4.3 we explored several design optimization techniques, including pipelining different loops, loop unrolling, and array partitioning. Understanding the tradeoffs between these techniques can be somewhat challenging, since they are often dependent on what another. We must often apply these techniques together with a carefully chosen goal in mind in order to get a benefit and applying one technique without applying another technique can actually make matters worse. For instance, when performing loop unrolling, the designer must be careful to understand the effects that this has upon memory accesses. Increasing the number of operations that can execute concurrently doesn't help if performance is limited by available memory ports. Similarly, providing more memory ports if there are insufficient operations to utilize those memory ports (or if the addresses of each memory operation can't be easily partitioned) can also incur a resource cost without increasing performance.

To see some of the complexity in applying these combinations of transforms, we encourage you to perform the following exercise:

Synthesize the `spmv` design using the directives specified in each of the ten cases from Table 6.1. Each case has different pipeline, unroll, and partitioning directives for the different loops and arrays. These partitionings should be done across the three arrays (`values`, `columnIndex`, and `x`). What sort of trends do you see? Does increasing the unroll and partitioning factors help or hurt when it comes to utilization? How about performance? Why?

Table 6.1: Potential optimizations for sparse matrix-vector multiplication.

	L1	L2
Case 1	-	-
Case 2	-	pipeline
Case 3	pipeline	-
Case 4	unroll=2	-
Case 5	-	pipeline, unroll=2
Case 6	-	pipeline, unroll=2, cyclic=2
Case 7	-	pipeline, unroll=4
Case 8	-	pipeline, unroll=4, cyclic=4
Case 9	-	pipeline, unroll=8
Case 10	-	pipeline, unroll=8, cyclic=8
Case 11	-	pipeline, unroll=8, block=8

If you performed the previous exercise, you should have seen that blindly applying optimization directives may not always provide you with the expected results. It is usually more effective to consider the properties of an application under design, and to select optimizations with a particular design goal in mind. Of course, this requires some intuition behind the capabilities and limitations of a particular tool being used. While it is certainly difficult (perhaps impossible?) to understand every detail of a complex tool like Vivado[®] HLS, we can build a mental model of the most critical aspects.

One of the options we considered in cases 3 and 4 above was to increase pipeline outer loops, such as the L1 loop in this code, rather than inner loops. This transformation has the effect of increasing the potential parallelism within one task. In order to perform this optimization, the Vivado[®] HLS tool must fully unroll inner loops, like the L2 loop in this code. If full unrolling is possible, this can reduce the cost of calculating the loop bounds and can also eliminate recurrences in the code. However, in this code, the inner loop cannot be unrolled by Vivado[®] HLS because the loop bound is not constant.

Add a directive to pipeline the outermost L1 loop, i.e., implement case 3 above. What is the initiation interval (II) when you do not set a target II? What happens to the utilization? How does explicitly increasing the II change the utilization results? How does this compare to pipelining the L2 loop? How does this compare to the baseline design (no directives)? What is happening when you attempt to pipeline this outer loop? (hint: check the synthesis log)

Another option to increase parallelism is partial loop unrolling of the inner loop, as in cases 5 through 10. This transformation exposes more parallelism by allowing more operations from the same loop iteration to be executed concurrently. In some cases, more operations can increase performance by enabling Vivado[®] HLS to instantiate more operators when pipelining the inner loop. However, in this case it is still difficult to improve the II of the inner loop because of the recurrence through the inner loop. However, in this case, because we have an II greater than 1, many of those operations can be shared on the same operators.

An partially unrolled version of the code is shown in Figure 6.6. In this code, the L2 loop has been split into two loops labeled L2_1 and L2_2. The innermost L2_2 executes a parameterized number of times, given by the compile-time parameter S . The body of the inner loop contains the body of the original L2 loop, along with a condition that arises from the loop bound of the original L2 loop. In this code, we now have an arbitrary number of multiply and add operations to execute in the body of the L2_1 loop, given by the parameter S , and a single recurrence through the accumulation $y0 += yt$.

Note that the code in Figure 6.6 is slightly different from the code that is generated from automatic loop unrolling. Automatic loop unrolling duplicates operations, but must also preserve the order of each operation (additions in this case). This results in a long chain of operation dependencies in the inner loop shown on the left side of Figure 6.7. Reordering the operations results in operation dependencies show on the right side of the figure. In this case, only the final accumulation results in a recurrence. When using floating-point data types, this reordering of operations can slightly change the behavior of the program, so Vivado[®] HLS does not apply this kind of operation reordering automatically.

A possible implementation of this design is shown in Figure 6.8. In this case, $S = 3$ to match the best achievable II where there is a latency of 3 through the adder. In this case, we see that all the operations have been successfully shared on a single multiplier and adder. Comparing this behavior to the behavior in Figure 6.5, we see that there are some disadvantages. In particular, the depth of the pipeline of the inner loop is much longer, which implies that the number of cycles to flush the pipeline to start a new iterations of the outer L1 loop is much larger. Processing of the non-zero elements in a row also occurs in blocks of size S . A row with 3 elements takes exactly the same time to compute as a row with one element. The remaining operations which are still scheduled in the loop pipeline must still ‘execute’ even though their results are discarded. In order to rigorously compare the characteristics of the two designs, we need to understand the expected number of non-zero elements in each row of the matrix. Fewer non-zero elements in each line would

```

#include "spmv.h"

const static int S = 7;

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
    L1: for (int i = 0; i < NUM_ROWS; i++) {
        DTYPE y0 = 0;
        L2.1: for (int k = rowPtr[i]; k < rowPtr[i+1]; k += S) {
            #pragma HLS pipeline II=S
            DTYPE yt = values[k] * x[columnIndex[k]];
            L2.2: for(int j = 1; j < S; j++) {
                if(k+j < rowPtr[i+1]) {
                    yt += values[k+j] * x[columnIndex[k+j]];
                }
            }
            y0 += yt;
        }
        y[i] = y0;
    }
}

```

Figure 6.6: A partially unrolled version of the `spmv` code from Figure 6.2

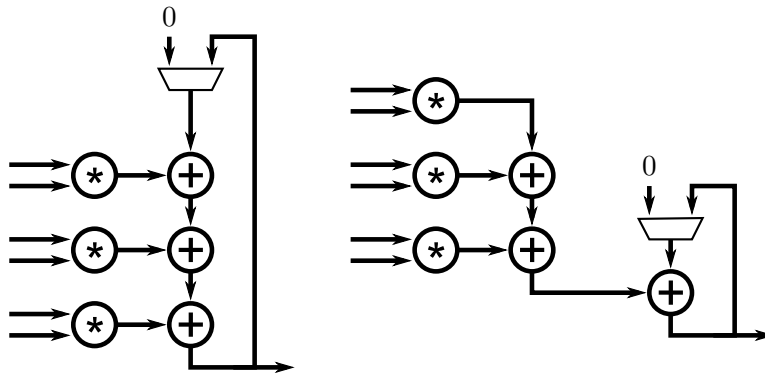


Figure 6.7: Two different partially unrolled versions of an accumulation. The version on the left has a recurrence with three additions, whereas the version on right only has one addition in the recurrence.

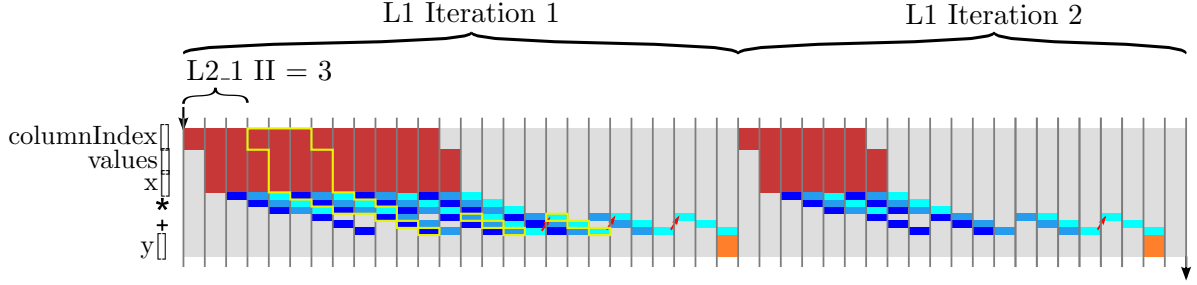


Figure 6.8: Architecture and behavior of the `spmv` code based on the partially unrolled and pipelined inner loop shown in Figure 6.6.

favor the first implementation, while more non-zero elements in each line would favor the second implementation.

Notice that there is, to some extent a chicken-and-egg problem here. We need to know the target device and clock period to determine the number of pipeline stages required for the adder to meet timing. Only after we know the number of pipeline stages (perhaps by running with $S=1$ and investigating the Vivado[®] HLS logs to identify the adder recurrence) can we select an appropriate version of the parameter S that achieves $II=1$. Once we've determined S , we can run `C/RTL cosimulation` to determine the achieved performance on a set of benchmark test data. Because of the variable loop bounds, the achieved performance is data-dependent so we might have to explore different values of S to determine the value that maximizes performance. Changing the target device or clock period might affect all of these decisions! Although it may seem like high-level synthesis provides little assistance in solving this problem, it's still much faster (and possible to easily script) compared to evaluating each new version with a new `RTL` design that must be verified!

The behavior in Figure 6.8 is achieved when S is the same as the number of pipeline stages for the adder. What happens to the behavior when S is set larger? What happens to the behavior when is it set smaller? What happens when the target II is smaller than S ? What happens when the target II is larger?

6.7 Conclusion

In this chapter, we looked at sparse matrix-vector multiplication (SpMV). This continues our study of matrix operations. This operation is particularly interesting because it uses a unique data structure. In order to reduce the amount of storage, the matrix is stored in a compressed row storage format. This requires a design that uses some indirect references to find the appropriate entry in the matrix.

This chapter is the first to discuss at length the testing and simulation abilities of the Vivado[®] HLS tool. We provide a simple testbench for SpMV and describe how it can be integrated into the HLS work-flow. Additionally, we describe the C/RTL cosimulation features of the Vivado[®] HLS tool. This is particularly important for us in order to get precise performance results. The task interval and task latency depends upon the input data. The less sparse the matrix, the more computation that must be performed. The cosimulation provides a precise trace of execution using the given testbench. This allows the tool to compute the clock cycles to include in the performance results. Finally, we discuss optimizing the code using loop optimizations and array partitioning.

Chapter 7

Matrix Multiplication

This chapter looks at a bit more complex design – matrix multiplication. We consider two different versions. We start with a “straightforward” implementation, i.e., one that takes two matrices as inputs and outputs the result of their multiplication. We call this complete matrix multiplication. Then, we look at a block matrix multiplication. Here the input matrices are feed into the function in portions, and the function computes partial results.

7.1 Background

Matrix multiplication is a binary operation that combines two matrices into a third. The operation itself can be described as a linear operation on the vectors that compose the two matrices. The most common form of matrix multiplication is call the *matrix product*. The matrix product \mathbf{AB} creates an $n \times p$ matrix when matrix \mathbf{A} has dimensions $n \times m$ and matrix \mathbf{B} has dimensions $m \times p$.

More precisely, we define the following:

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{bmatrix} \quad (7.1)$$

$$\mathbf{AB} = \begin{bmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{bmatrix} \quad (7.2)$$

where the operation $(\mathbf{AB})_{ij}$ is defined as $(\mathbf{AB})_{ij} = \sum_{k=1}^m A_{ik}B_{kj}$.

Now we provide a simple example. Let

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} \quad (7.3)$$

The result of the matrix product is

$$\mathbf{AB} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} & A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} \\ A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31} & A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} \end{bmatrix} \quad (7.4)$$

```

void matrixmul(int A[N][M], int B[M][P], int AB[N][P]) {
    #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
    #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
    /* for each row and column of AB */
    row: for(int i = 0; i < N; ++i) {
        col: for(int j = 0; j < P; ++j) {
            #pragma HLS PIPELINE II=1
            /* compute (AB)i,j */
            int ABij = 0;
            product: for(int k = 0; k < M; ++k) {
                ABij += A[i][k] * B[k][j];
            }
            AB[i][j] = ABij;
        }
    }
}

```

Figure 7.1: A common three **for** loop structure for matrix multiplication. The outer **for** loops, labeled **rows** and **cols**, iterate across the rows and columns of the output matrix **AB**. The innermost loop, labeled **product** multiplies the appropriate elements of one row of **A** and one column of **B** and accumulates them until it has the result for the element in **AB**.

Matrix multiplication is a fundamental operation in numerical algorithms. Computing the product between large matrices can take a significant amount of time. Therefore, it is critically important part of many of problems in numerical computing. Fundamentally, matrices represent linear transforms between vector spaces; matrix multiplication provides way to compose the linear transforms. Applications include linearly changing coordinates (e.g., translation, rotation in graphics), high dimensional problems in statistical physics (e.g., transfer-matrix method), and graph operations (e.g., determining if a path exists from one vertex to another). Thus it is a well studied problem, and there are many algorithms that aim to increase its performance, and reduce the memory usage.

7.2 Complete Matrix Multiplication

We start our optimization process with perhaps the most common method to compute a matrix multiplication – using three nested **for** loops. Figure 7.1 provides the code for such an implementation. The outer **for** loops, labeled **rows** and **cols**, iterate across the rows and columns of the output matrix **AB**. The innermost **for** loop computes a dot product of one row of **A** and one column of **B**. Each dot product is a completely independent set of computations that results in one element of **AB**. Conceptually, we are performing P matrix-vector multiplications, one for each column of **B**.

In this case, we’ve applied a **pipeline** directive to the **col** loop with a target initiation interval of 1. The result is that the innermost **for** loop is fully unrolled, and we expect the resulting circuit include roughly M multiply-add operators and to have an interval of roughly $N * P$ cycles. As discussed in Chapter 4, this is only one reasonable choice. We could choose to place the **pipeline** directive in

different locations in the function with the goal of achieving different resource-throughput tradeoffs. For instance, placing the same directive at the top of the function (outside all of the **for** loops) will result in all of the loops being completely unrolled, which would take roughly $N * M * P$ multiply-add operators and would have an interval of 1 cycle. Placing it inside the **row** loop would result in roughly $M * P$ multiply-add operators and an interval of roughly N cycles. These design points are relatively easy to achieve, given the corresponding array partitioning. It's also possible to pipeline the innermost loop with the goal of achieving a design with only one multiply-add operator, although achieving an $\Pi=1$ implementation at high clock frequencies can be difficult because of the recurrence involved in the accumulation of variable **ABij**. We can also partially unroll different loops to achieve yet more design points. The fundamental tradeoff here is between the complexity of the resulting architecture, i.e., the number of multiply-add operators, and performance, i.e., the number of cycles that the hardware is busy. In an ideal world, each doubling of the resource usage should result in exactly half the number of clock cycles being required, although in practice such 'perfect scaling' is difficult to achieve.

Change the location of the **pipeline** directive. How does the location effect the resource usage? How does it change the performance? Which alternative provides the best performance in terms of function interval? Which provides the smallest resource usage? Where do you think is the best place for the directive? Would increasing the size of the matrices change your decision?

Executing large numbers of operations every cycle requires being able to supply all of the required operands and to store the results of each operation. Previously we have used the **array_partition** directive to increase the number of accesses that can be performed on each memory. As long as the partition of the array that each memory access can be determined at compile time, then array partitioning is a simple and efficient way to increase the number of memory accesses that can be performed each clock cycle. In this case, we use the slightly different **array_reshape** directive to perform array partitioning. This directive not only partitions the address space of the memory into separate memory blocks, but then recombines the memory blocks into a single memory. This transformation increases the data width of the memory used to store the array, but doesn't change the overall number of bits being stored. The difference is shown in Figure 7.2.

Both **array_reshape** and **array_partition** increase the number of array elements that can be read each clock cycle. They also support the same options, enabling cyclic and block partitions or partitioning along different dimensions of a multi-dimensional array. In the case of **array_reshape**, the elements must each have the same address in the transformed array, whereas with **array_partition**, the addresses in the transformed array can be unrelated. Although it may seem like one would always want to use **array_partition** because it is more flexible, it makes each individual memory smaller, which can sometimes result in inefficient memory usage. The **array_reshape** directive results in larger memory blocks, which can sometimes be mapped more efficiently into primitive FPGA resources. In particular, the smallest granularity of **block RAM (BRAM)** blocks in Xilinx Virtex Ultrascale+ devices is 18 Kbits with several different supported combination of depths and widths. When the partitions of an array become smaller than around 18Kbits, then BRAMs are no longer used efficiently. If we start with an original array which is a 4-bit array with dimensions [1024][4], this array can fit in a single **BRAM** resource configured as a 4Kbit x 4 memory. Partitioning this array completely in the second dimension would result in 4 1Kbit x 4 memories, each of which are much smaller than one **BRAM** resource. Reshaping the array instead using the **array_reshape** directive results in a memory which is 1Kbit x 16, a supported **BRAM** configuration.

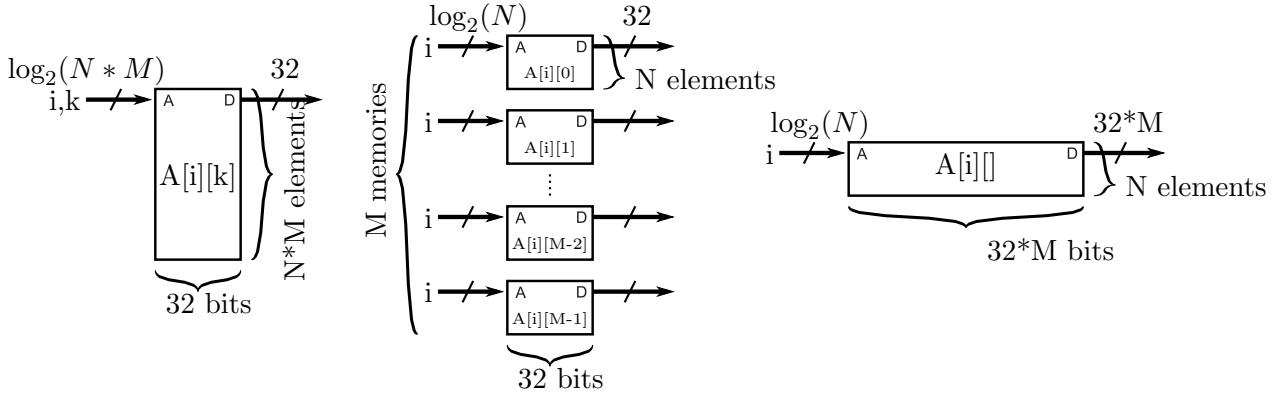


Figure 7.2: Three different implementations of a two-dimensional array. On the left is the original array consisting of $N \times M$ elements. In the middle, the array has been transformed using the **array_partition** directive, resulting in M memories, each with N elements. On the right, the array has been transformed using the **array_reshape** directive, resulting in one memory with N locations and each location contains M elements of the original array.

Note that this partitioning is effective because the partitioned dimension (dimension 2 of **A** or dimension 1 of **B**) is indexed by a constant. In addition, the non-partitioned dimension is indexed by the same (variable) value. When partitioning multi-dimensional arrays, this is a good rule of thumb to identify which dimension should be partitioned.

Remove the **array_reshape** directives. How does this effect the performance? How does it change the resource usage? Does it make sense to use any other **array_reshape** directives (with different arguments) on these arrays? In this case, how does the result differ if you use the **array_reshape** directive instead?

The size of the arrays can have a substantial effect on the optimizations that you wish to perform. Some applications might use very small matrices, say 2×2 or 4×4 . In this cases, it may be desirable to implement a design with the absolute highest performance, which is generally achieved by applying the **pipeline** directive on the entire function. As the size of the arrays increase, in the range of 32×32 , this approach quickly become infeasible because of the resource limits available in a single device. There will simply not be enough DSP resources to implement that many multiplications every clock cycle or enough external bandwidth to get data on and off the chip. Many FPGA designs are often coupled to the data rates of other components in a system, such as an Analog to Digital (A/D) converter, or the symbol rate in a communication system. In these designs it is common to instead apply the **pipeline** directive on inner loops with the goal of matching the interval of the computation with the data rate in a system. In such cases, we often need to explore different resource-throughput tradeoffs by moving the **pipeline** directive into an inner loop or partially unrolling loops. When dealing with very large matrices containing thousands or millions of elements, we often need to take into account more complex architectural considerations. The next section discusses a common approach to scaling matrix multiply to larger designs, called *blocking* or *tiling*.

Optimize your design for the 128×128 matrix multiplication. Then start increasing the size of the matrices by a factor of two (to 512×512 , 1024×1024 , 2048×2048 , etc. How does this effect the resource usage and performance? How about the runtime of the tool? What is the best way to optimize for large large matrix sizes?

7.3 Block Matrix Multiplication

A *block matrix* is interpreted as being partitioned into different submatrices. This can be visualized by drawing different horizontal and vertical lines across the elements of the matrix. The resulting “blocks” can be viewed as submatrices of the original matrix. Alternatively, we can view the original matrix as a matrix of blocks. This naturally leads to many hierarchical algorithms in Linear Algebra where we compute matrix operations, such as matrix multiply, on large block matrices by decomposing them into smaller matrix operations on the blocks themselves.

For instance, when we talk about the matrix multiplication operation between matrix \mathbf{A} and \mathbf{B} in Equations 7.3 and 7.4, we might normally think of each element of the matrices \mathbf{A}_{11} or \mathbf{B}_{23} as a single number or perhaps a complex number. Alternatively, we can consider each element in these matrix operations as a block of the original matrix. In this case, as long as the sizes of the individual blocks are compatible, we simply have to perform the correct matrix operations instead of the original scalar operations. For instance, to compute \mathbf{AB}_{11} , we would need to compute two matrix products and two matrix sums to compute $\mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} + \mathbf{A}_{13}\mathbf{B}_{31}$.

Matrix blocking turns out to be a very useful technique for a number of reasons. One reason is that blocking is an easy way to find more structure in the algorithm that we can explore. In fact, some of the optimizations that we have already seen as loop transformations, such as loop unrolling, can be viewed as specific simple forms of blocking. Another reason is that we can choose to block a matrix according to the natural structure of the matrix. If a matrix has a large block of zeros, then many individual products may be zero. If we want to skip these individual products then this can be difficult in a statically schedule pipeline, whereas it may be easier to skip a large block of zeros. Many matrices are *block-diagonal* where the blocks on the diagonal are non-zero and blocks off the diagonal are zero. Yet another reason is that the blocked decomposition results in lots of smaller problems operating on smaller sets of data. This increases the data locality of a computation. In processor systems, it is common to choose block sizes that conveniently match the memory hierarchy of a processor or the natural size of the vector data types supported by the processor. Similarly, in FPGAs we can choose the blocking sizes to match the available on-chip memory size or to the number of multiply-add operators that we can budget to support.

Until now, we have assumed that accelerators always have all of their data available before the start of a task. However in designs dealing with large datasets, such as large matrices, this can sometimes be an unneeded constraint. Since it is unlikely that our accelerator will be able to process all input data immediately, we can build an accelerator that receives input data only right before it is needed. This allows an accelerator to more efficiently use the available on-chip memory. We call this a *streaming architecture* since we transfer the input data (and potentially the output data) one portion at a time rather than all at once.

Streaming architectures are common in many applications. In some cases this is because of a conscious design choice that we make to decompose a large computation into multiple smaller computations. For instance, we may design a matrix multiplication system that reads and processes one block of data at a time from external memory. In other cases, we might process a stream of data because the data is being sampled in real time from the physical world, for instance, from an

A/D converter. In other cases, the data we are processing may simply be created in sequence from a previous computation or accelerator. In fact, we've already seen an instance of this in Section 5.4.

One potential advantage of streaming is a reduction in the memory that we can use to store the input and output data. The assumption here is that we can operate on the data in portions, create partial results, and then we are done with that data, thus we do not need to store it. When the next data arrives, we can overwrite the old data resulting in smaller memories.

In the following, we develop a streaming architecture for matrix multiplication. We divide the input arrays **A** and **B** into blocks, which are a contiguous set of rows and columns, respectively. Using these blocks, we compute a portion of the product **AB**. Then we stream the next set of blocks, compute another portion of **AB** until the entire matrix multiplication is complete.

Figure 7.3 provides the a description of the streaming architecture that we create. Our architecture has a variable `BLOCK_SIZE` that indicates the number of rows that we take from the **A** matrix on each execution, the number of columns taken from the **B** matrix, and the `BLOCK_SIZE` \times `BLOCK_SIZE` result matrix corresponding to the data that we compute each time for the **AB** matrix.

The example in Figure 7.3 uses a `BLOCK_SIZE` = 2. Thus we take two rows from **A**, and two columns from **B** on each execution of the streaming architecture that we define. The result of each call to the `blockmatmul` function is a 2×2 matrix for the **AB** architecture.

Since we are dealing with 4×4 matrices in the example, we need to do this process four times. Each time we get a 2×2 set of results for the **AB** matrix. The figure shows a progression of the rows and columns that we send. In Figure 7.3 a) we send the first two rows of **A** and the first two columns of **B**. The function will compute a 2×2 matrix corresponding to the first two elements in the rows and columns of the resulting matrix **AB**.

In Figure 7.3 b), we use again the first two rows of **A**, but this time we send the last two columns of **B**. We do not need to resend the data from the rows of **A** since they are the same as the previous data from the previous execution. And we get the results for the 2×2 matrix corresponding to the data in “upper left” corner of **AB**.

Figure 7.3 c) sends different data for both the **A** and **B** matrices. This time we send the last two rows of **A** and the first two columns of **B**. The results from this computation provide the “lower left” corner of the **AB** matrix.

The final execution of the streaming block matrix multiply, shown in Figure 7.3 d), uses the same last two rows of the **A** matrix from the previous iteration. And it sends the last two columns of the **B** matrix. The result provides the elements in the “lower right” corner of the **AB** matrix.

Before we show the code for the block matrix multiplication, we define some data types that we will use. Figure 7.4 shows the header file for the project. We create a custom data type `DTYPE` that specifies the type of data that we will multiply in the **A** and **B** matrices, and the corresponding **AB** matrix. This is currently set to an `int` data type.

It is good coding practice to use a custom data type in your designs. This allows you to easily change the data type, and to have one source of information so that you do not have errors when changing the data type in the future design iterations. And it is quite common to change the data type over the course of the design. For example, first you may start out with a **float** or **double** type while you get a functionally correct design. This also provides a baseline for error since later you will likely change your design to use fixed point data type. Fixed point data can reduce the number of resources, and increase the performance potentially at the cost of a reduction in the precision of the resulting data. You will likely try many different fixed

a)

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

 \times

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{31}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

 $=$

AB_{11}	AB_{12}	AB_{13}	AB_{14}
AB_{21}	AB_{22}	AB_{23}	AB_{24}
AB_{31}	AB_{32}	AB_{33}	AB_{34}
AB_{41}	AB_{42}	AB_{43}	AB_{44}

b)

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

 \times

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{31}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

 $=$

AB_{11}	AB_{12}	AB_{13}	AB_{14}
AB_{21}	AB_{22}	AB_{23}	AB_{24}
AB_{31}	AB_{32}	AB_{33}	AB_{34}
AB_{41}	AB_{42}	AB_{43}	AB_{44}

c)

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

 \times

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{31}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

 $=$

AB_{11}	AB_{12}	AB_{13}	AB_{14}
AB_{21}	AB_{22}	AB_{23}	AB_{24}
AB_{31}	AB_{32}	AB_{33}	AB_{34}
AB_{41}	AB_{42}	AB_{43}	AB_{44}

d)

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

 \times

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{31}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

 $=$

AB_{11}	AB_{12}	AB_{13}	AB_{14}
AB_{21}	AB_{22}	AB_{23}	AB_{24}
AB_{31}	AB_{32}	AB_{33}	AB_{34}
AB_{41}	AB_{42}	AB_{43}	AB_{44}

Figure 7.3: One possible blocked decomposition of the matrix multiplication of two 4×4 matrices. The entire \mathbf{AB} product is decomposed into four matrix multiply operations operating on a 2×4 block of \mathbf{A} and a 4×2 block of \mathbf{B} .

```
#ifndef _BLOCK_MM_H_
#define _BLOCK_MM_H_
#include "hls_stream.h"
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

typedef int DTYPE;
const int SIZE = 8;
const int BLOCK_SIZE = 4;

typedef struct {
    DTYPE a[BLOCK_SIZE]; } blockvec;

typedef struct {
    DTYPE out[BLOCK_SIZE][BLOCK_SIZE]; } blockmat;

void blockmatmul(hls::stream<blockvec> &Arows, hls::stream<blockvec> &Bcols,
blockmat & ABpartial, DTYPE iteration);
#endif
```

Figure 7.4: The header file for the block matrix multiplication architecture. The file defines the data types used within the function, the key constants, and the `blockmatmul` function interface.

point types until you find the right tradeoff between accuracy/error, performance, and resource usage.

SIZE determines the number of rows and columns in the matrices to be multiplied. We limit this to square matrices although handling arbitrary matrix sizes could be done by changing the code in a number of different places. We leave that as an exercise for the reader.

Change the code to allow it to handle matrices of arbitrary size.

The BLOCK_SIZE variable defines the number of rows from **A** and the number of columns from **B** that we operate upon in each execution. This also defines how much data that we stream at one time into the function. The output data that we receive from the function at each execution is an $\text{BLOCK_SIZE} \times \text{BLOCK_SIZE}$ portion of the **AB** matrix.

The `blockvec` data type is used to transfer the BLOCK_SIZE rows of **A** and columns of **B** to the function on each execution. The `blockmat` data type is where we store the partial results for the **AB** matrix.

Finally, the `blockmat` data type is a structure consisting of an $\text{BLOCK_SIZE} \times \text{BLOCK_SIZE}$ array. This holds the resulting values from one execution of the `matrmatmul` function.

The function prototype itself takes the two inputs which are both of the type `hls::stream<blockvec> &`. These are a sequence of `blockvec` data. Remember that a `blockvec` is a data type that consists of an array with BLOCK_SIZE elements.

The `hls::stream<>` template class is one way in Vivado[®] HLS of creating a FIFO data structure that works well in simulation and synthesis. The samples are sent in sequential order using the `write()` function, and retrieved using the `read()` function. This library was developed since streaming is a common methodology for passing data in hardware design, yet this same operation can be modeled in many different ways using the C programming language, for instance, by using arrays. In particular, it can be difficult for the Vivado[®] HLS tool to infer streaming behaviors when dealing complex access patterns or multi-dimensional arrays. The built-in stream library enables the programmer to explicitly specify the order of stream accesses, avoiding any limitations of this inference.

The `hls::stream` class must always be passed by reference between functions, e.g., as we have done in the `blockmatmul` function in Figure 7.5.

The code for executing one part of the streaming block matrix multiplication is shown in Figure 7.5. The code has three portions denoted by the labels `loadA`, `partialsum`, and `writeoutput`.

The first part of the code, denoted by the `loadA` label, is only executed on certain conditions, more precisely when `it % (SIZE/BLOCK_SIZE) == 0`. This is done to save some time in the cases when we can reuse the data from the **A** matrix from the previous execution of the function.

Remember that in each execution of this `blockmatmul` function we send BLOCK_SIZE rows from the **A** matrix and BLOCK_SIZE columns from the **B** matrix. We send multiple BLOCK_SIZE of columns for each BLOCK_SIZE of rows from **A**. The variable it keeps track of the number of times that we have called the `blockmatmul` function. Thus, we do a check on each execution of the function to determine if we need to load the rows from **A**. When we do not, this saves us some

```

#include "block_mm.h"
void blockmatmul(hls::stream<blockvec> &Arows, hls::stream<blockvec> &Bcols,
                blockmat &ABpartial, int it) {
    #pragma HLS DATAFLOW
    int counter = it % (SIZE/BLOCK_SIZE);
    static DTYPE A[BLOCK_SIZE][SIZE];
    if(counter == 0){ //only load the A rows when necessary
        loadA: for(int i = 0; i < SIZE; i++) {
            blockvec tempA = Arows.read();
            for(int j = 0; j < BLOCK_SIZE; j++) {
                #pragma HLS PIPELINE II=1
                A[j][i] = tempA.a[j];
            }
        }
    }
    DTYPE AB[BLOCK_SIZE][BLOCK_SIZE] = { 0 };
    partialsum: for(int k=0; k < SIZE; k++) {
        blockvec tempB = Bcols.read();
        for(int i = 0; i < BLOCK_SIZE; i++) {
            for(int j = 0; j < BLOCK_SIZE; j++) {
                AB[i][j] = AB[i][j] + A[i][k] * tempB.a[j];
            }
        }
    }
    writeoutput: for(int i = 0; i < BLOCK_SIZE; i++) {
        for(int j = 0; j < BLOCK_SIZE; j++) {
            ABpartial.out[i][j] = AB[i][j];
        }
    }
}

```

Figure 7.5: The `blockmatmul` function takes a `BLOCK_SIZE` set of rows from **A** matrix, a `BLOCK_SIZE` set of columns from the **B** matrix, and creates a `BLOCK_SIZE` \times `BLOCK_SIZE` partial result for the **AB** matrix. The first part of the code (denoted by the label `loadA`) stores the rows from **A** into a local memory, the second part in the nested `partialsum` **for** performs the computation for the partial results, and the final part (with the `writeoutput` label) takes these results and puts them the proper form to return from the function.

time. When it is executed, it simply pulls data from the `Arows` stream and puts it into a static local two-dimensional matrix `A[BLOCK_SIZE][SIZE]`.

Fully understanding this code requires some explanation about the `stream` class, and how we are using it. The `stream` variable `Arows` has elements of the type `blockvec`. A `blockvec` is a matrix of size `BLOCK_SIZE`. We use this in the following manner; each element in the `Arows` stream has an array that holds one element from each of the `BLOCK_SIZE` rows of the `A` matrix. Thus, in each call the `blockmatmul` function, the `Arows` stream will have `SIZE` elements in it, each of those holding one of each of the `BLOCK_SIZE` rows. The statement `tempA = Arows.read()` takes one element from the `Arows` stream. Then we load each of these elements into the appropriate index in the local `A` matrix.

The `stream` class overloads the `<<` operator to be equivalent to the `read()` function. Thus, the statements `tempA = Arows.read()` and `tempA << Arows` perform the same operation.

The next part of the computation calculates the partial sums. This is the bulk of the computation in the `blockmatmul` function.

The `Bcols` stream variable is utilized in a very similar manner to the `Arows` variable. However, instead of storing rows of `A`, it stores the data corresponding the columns of `B` that the current execution of the function is computing upon. Every call of the `blockmatmul` function will provide new data for the columns of the `B` matrix. Thus, we do not need to conditionally load this data as we do with the `A` matrix. The function itself works in a very similar manner to that from the `matmul` in Figure 7.1 except that we are only calculating $\text{BLOCK_SIZE} \times \text{BLOCK_SIZE}$ results from the `AB` matrix. Thus we only have to iterate across `BLOCK_SIZE` rows of `A` and `BLOCK_SIZE` columns of `B`. But each row and column has `SIZE` elements, hence the bounds on the outer `for` loop.

The final portion of the function moves the data from the local `AB` array, which has dimensions of $\text{BLOCK_SIZE} \times \text{BLOCK_SIZE}$; this holds the partial results of the `AB` output matrix.

Of the three parts of the function, the middle part, which calculates the partial sum, requires the most computation. By inspecting the code, we can see that this part has three nested `for` loops with a total of $\text{SIZE} \times \text{BLOCK_SIZE} \times \text{BLOCK_SIZE}$ iterations. The first part has $\text{SIZE} \times \text{BLOCK_SIZE}$ iterations; and the last part has $\text{BLOCK_SIZE} \times \text{BLOCK_SIZE}$ iterations. Thus, we should focus our optimizations on the middle part, i.e., the `partialsum` nested `for` loops.

The common starting point for optimizations of nested `for` loops is to pipeline the innermost `for` loop. Then, if that does not require too many resources, the designer can move the `pipeline` directive into higher level `for` loops. Whether the resulting design consume too many resource depends upon the specified `BLOCK_SIZE`; if this is small, then it is likely worth moving the `pipeline` directive. It may even be worthwhile to move it inside the outermost `for` loop. This will unroll the two inner `for` loops and thus very likely increase the resource usage by a substantial amount. However, it will increase the performance.

How does changing the `BLOCK_SIZE` effect the performance and resource usage? How about changing the `SIZE` constant? How does moving the `pipeline` directive across the three different nested `for` loops in the `partialsum` portion of the function change the performance and resource usage?

The `dataflow` directive at the start of the function creates a pipeline across the portions of the function, i.e., the `loadA` `for` loop, the `partialsum` nested `for` loop, and the `writeoutput` `for` loop.

Using this directive will decrease the interval of the `blockmatmul` function. However, this is limited by the largest interval of all three of the portions of the code. That is, the maximum interval for the `blockmatmul` function – let us call it `interval(blockmatmul)` – is greater than or equal to the the interval of the three parts which are defined as `interval(loadA)`, `interval(partialsum)`, and `interval(writeoutput)`. More formally,

$$\begin{aligned} \text{interval}(\text{blockmatmul}) \geq \max(\text{interval}(\text{loadA}), \text{interval}(\text{partialsum}), \\ \text{interval}(\text{writeoutput})) \end{aligned} \quad (7.5)$$

We need to keep Equation 7.5 in mind as we optimize the `blockmatmul` function. For example, assume that `interval(partialsum)` is much larger than the other two portions of the function. Any performance optimizations that minimize `interval(loadA)` and `interval(writeoutput)` are useless since the function interval, i.e., `interval(blockmatmul)` would not decrease. Thus, the designer should focus any performance optimization effort to decrease `interval(partialsum)`, i.e., target performance optimizations on those three nested **for** loops.

It is important to note that this only applies to performance optimizations. The designer can (and should) optimize the resource usage of these other two parts. In fact, they are ripe for such optimizations since reducing the resource usage often increases the interval and/or latency. In this case, it is ok to increase the interval as it will not effect the overall performance of the `blockmatmul` function. In fact, the ideal case is to optimize all three parts of the function such that they all have the same interval, assuming that we can easily tradeoff between the interval and resource usage (which is not always the case).

The testbench for the `blockmatmul` function is shown in Figures 7.6 and 7.7. We split it across two figures to make it more readable since it is a longer piece of code. Up until this point, we have not shown the testbenches. We show this testbench for several reasons. First, it provides insight into how the `blockmatmul` function works. In particular, it partitions the input matrices into blocks and feeds them into the `blockmatmul` function in a block by block manner. Second, it gives a complex usage scenario for using `stream` template for simulation. Finally, it gives the reader an idea about how to properly design testbenches.

The `matmatmul_sw` function is a simple three **for** loop implementation of matrix multiplication. It takes two two-dimensional matrices as inputs, and outputs a single two-dimensional matrix. It is very similar to what we have seen in the `matrixmul` function in Figure 7.1. We use this to compare our results from the blocked matrix multiplication hardware version.

Let us focus on the first half of the testbench shown in Figure 7.6. The beginning block of code initializes variables of the rest of the function. The variable `fail` keeps track of whether the matrix multiplication was done correctly. We will check this later in the function. The variables `strm_matrix1` and `strm_matrix2` are `hls::stream<>` variables that hold the rows and columns of the **A** and **B** matrices, respectively. Each element of these `stream` variables is a `<blockvec>`. Referring back at the `block_mm.h` file in Figure 7.4, we recall that a `blockvec` is defined as an array of data; we will use each `blockvec` to store one row or column of data.

The `stream` variable resides in the `hls` namespace. Thus, we can use that namespace and forgo the `hls::stream` and instead simply use `stream`. However, the preferred usage is to keep the `hls::` in front of the `stream` to insure code readers that the `stream` is relevant to Vivado[®] HLS and not C construct from another library. Also, it avoids having to deal with any potential conflicts that may occur by introducing a new namespace.

The next definitions in this beginning block of code are the variables `strm_matrix1_element` and `strm_matrix2_element`. These two variables are used as placeholders to populate each `blockvec` variable that we write into the `strm_matrix1` and `strm_matrix2` stream variables. The `block_out` variable is used to store the output results from the `blockmatmul` function. Note that this variable uses the data type `blockmat` which is a two-dimensional array of `BLOCK_SIZE × BLOCK_SIZE` as defined in the `block_mm.h` header file (see Figure 7.4). The final definitions are `A`, `B`, `matrix_swout`, and `matrix_hwout`. These are all `SIZE × SIZE` two-dimensional arrays with the `DTYPE` data type.

You can name the streams using an initializer. This is good practice as it gives better error messages. Without the name, the error message provides a generic reference to the stream with the data type. If you have multiple stream declarations with the same data type, then you will have to figure out which stream the error is referring to. Naming the stream variable is done by giving the variable an argument which is the name, e.g., `hls::stream<blockvec> strm_matrix1("strm_matrix1");`.

The next set of nested `initmatrices` **for** loops sets the values of the four two-dimensional arrays `A`, `B`, `matrix_swout`, and `matrix_hwout`. The variables `A` and `B` are input matrices. These are initialized to a random value between `[0, 512)`. We picked the number 512 for no particular reason other than it can fit any 9 bit value. Keep in mind that while the `DTYPE` is set as an **int**, and thus has significantly more range than `[0,512)`, we often move to fixed point values with much smaller ranges later in the design optimization process. The `matrix_swout` and `matrix_hwout` are both initialized to 0. These are filled in later by calls to the functions `matmatmul_sw` and `blockmatmul`.

The second part of the testbench is continued in Figure 7.7. This has the last portion of the code from the `main` function.

The first part of this figure has a complex set of nested **for** loops. The overall goal of the computation in these **for** loops is to set up the data from the input matrices **A** and **B** so that it can be streamed to the `blockmatmul` function. Then the results of the `blockmatmul` function are stored in the `matrix_hwout` array.

The outer two **for** loops are used to step across the input arrays in a blocked manner. You can see that these both iterate by a step of `BLOCK_SIZE`. The next two **for** loops write rows from **A** into `strm_matrix1_element` and the columns from **B** into `strm_matrix2_element`. It does this in an element by element fashion by using the variable `k` to access the individual values from the rows (columns) and write these into the one dimensional array for each of these “elements”. Remember that both `strm_matrix1_element` and `strm_matrix2_element` have the datatype `blockvec`, which is a one dimensional array of size `BLOCK_SIZE`. It is meant to hold `BLOCK_SIZE` elements from each row or column. The inner **for** loop iterates `BLOCK_SIZE` times. The `strm_matrix1` and `strm_matrix2` stream variables are written to `SIZE` times. That means that has a buffer of the entire row (or column) and each element in the buffer holds `BLOCK_SIZE` values.

The `stream` class overloads the `>>` operator to be equivalent to the `write(data)` function. This is similar to overloading the `read()` function to the `<<` operator. Thus, the statements `strm_matrix1.write(strm_matrix1_element)` and `strm_matrix1_element >> strm_matrix1` perform the same operation.

The final part of this portion of the code to highlight is the **if** statements. These are correspond to the values **A** matrix. Essentially, these are there so that we do not constantly write the same


```

#include "block_mm.h"
#include <stdlib.h>
using namespace std;

void matmatmul_sw(DTYPE A[SIZE][SIZE], DTYPE B[SIZE][SIZE],
DTYPE out[SIZE][SIZE]){
    DTYPE sum = 0;
    for(int i = 0; i < SIZE; i++){
        for(int j = 0; j<SIZE; j++){
            sum = 0;
            for(int k = 0; k < SIZE; k++){
                sum = sum + A[i][k] * B[k][j];
            }
            out[i][j] = sum;
        }
    }
}

int main() {
    int fail = 0;
    hls::stream<blockvec> strm_matrix1("strm_matrix1");
    hls::stream<blockvec> strm_matrix2("strm_matrix2");
    blockvec strm_matrix1_element, strm_matrix2_element;
    blockmat block_out;
    DTYPE A[SIZE][SIZE], B[SIZE][SIZE];
    DTYPE matrix_swout[SIZE][SIZE], matrix_hwout[SIZE][SIZE];

    initmatrices: for(int i = 0; i < SIZE; i++){
        for(int j = 0; j < SIZE; j++){
            A[i][j] = rand() % 512;
            B[i][j] = rand() % 512;
            matrix_swout[i][j] = 0;
            matrix_hwout[i][j] = 0;
        }
    }

    //the remainder of this testbench is displayed in the next figure

```

Figure 7.6: The first part of the testbench for block matrix multiplication. The function is split across two figures since it is too long to display on one page. The rest of the testbench is in Figure 7.7. This has a “software” version of matrix multiplication, and variable declarations and initializations.

```
// The beginning of the testbench is shown in the previous figure
int main() {
    int row, col, it = 0;
    for(int it1 = 0; it1 < SIZE; it1 = it1 + BLOCK_SIZE) {
        for(int it2 = 0; it2 < SIZE; it2 = it2 + BLOCK_SIZE) {
            row = it1; //row + BLOCK_SIZE * factor_row;
            col = it2; //col + BLOCK_SIZE * factor_col;

            for(int k = 0; k < SIZE; k++) {
                for(int i = 0; i < BLOCK_SIZE; i++) {
                    if(it % (SIZE/BLOCK_SIZE) == 0) strm_matrix1_element.a[i] = A[row+i][k];
                    strm_matrix2_element.a[i] = B[k][col+i];
                }
                if(it % (SIZE/BLOCK_SIZE) == 0) strm_matrix1.write(strm_matrix1_element);
                strm_matrix2.write(strm_matrix2_element);
            }
            blockmatmul(strm_matrix1, strm_matrix2, block_out, it);

            for(int i = 0; i < BLOCK_SIZE; i++)
                for(int j = 0; j < BLOCK_SIZE; j++)
                    matrix_hwout[row+i][col+j] = block_out.out[i][j];
            it = it + 1;
        }
    }

    matmatmul_sw(A, B, matrix_swout);

    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
            if(matrix_swout[i][j] != matrix_hwout[i][j]) { fail=1; }

    if(fail==1) cout << "failed" << endl;
    else cout << "passed" << endl;

    return 0;
}
```

Figure 7.7: The second portion of the block matrix multiply testbench. The first part is shown in Figure [7.6](#). This shows the computation required to stream the data to the `blockmatmul` function, and the code that tests that this function matches a simpler three **for** loop implementation.

values to `strm_matrix1`. Recall that the values from the **A** matrix are used across several calls to the `blockmatmul` function. See Figure 7.3 for a discussion on this. These **if** statements are placed there to highlight the fact that you should not continually write the same data over and over. This is important because the internal code of the `blockmatmul` only does a read of this data when it is necessary. So if we continued to write this consistently, the code would not function correctly do to the fact that this stream is written to more than it is read from.

Now that the input data, the testbench calls the `blockmatmul` function. After the function call, it receives the partial computed results in the `block_out` variable. The next two **for** loops but these results into the appropriate locations in the `matrix_hwout` array.

After this complex set of **for** loops, the block matrix multiplication is complete. And the testbench continues to insure that the code is written correctly. It does this by comparing the results from the multiple calls to the `blockmatmul` function to results that were computed in the `matmatmul_sw`, which is a much simpler version of matrix matrix multiplication. After this function call, the testbench iterates through both two-dimensional matrices `matrix_hwout` and `matrix_swout` and makes sure that all of the elements are equivalent. If there is one or more element that is not equal, it sets the fail flag equal to 1. The testbench completes by printing out failed or passed.

It is important that note that you cannot directly compare the performance of the function `blockmatmul` with that of code for matrix multiplication, such as the code in Figure 7.1. This is because it takes multiple calls to the `blockmatmul` function in order to perform the entire matrix multiplication. It is important to always compare apples to apples.

Derive a function to determine the number of times that `blockmatmul` must be called in order to complete the entire matrix multiplication. This function should be generic, e.g., it should not be assume a specific value of `BLOCK_SIZE` or size of the matrix (i.e., `SIZE`).

Compare the resource usage of block matrix multiplication versus matrix multiplication. How do the resources change as the size of the matrices increases? Does the block size play a role in the resource usage? What are the general trends, if any?

Compare the performance of block matrix multiplication versus matrix multiplication. How does the performance change as the size of the matrices increases? Does the block size play a role in the performance? Pick two architectures with similar resource usage. How does the performance for those architectures compare?

7.4 Conclusion

Block matrix multiplication provides a different way to compute matrix multiplication. It computes partial results of the result matrix by streaming a subset of the input matrices to a function. This function is then computed multiple times in order to complete the entire matrix multiplication computation.

Chapter 10

Sorting Algorithms

10.1 Introduction

Sorting is a common algorithm in many systems. It is a core algorithm for many data structures because sorted data can be efficiently searched in $\mathcal{O}(\log n)$ time using binary search. For example, given an sequence of elements:

$$A = \{1, 4, 17, 23, 31, 45, 74, 76\} \quad (10.1)$$

We can find whether a number exists in this set of data without comparing it against all 8 elements. Because the data is sorted we can start by picking an element in the middle of the array and checking to see whether our number is greater than or less than the element. For example, if we were looking to find the element 45, we could start by comparing with $A(4) = 31$. Since $45 > 31$ we can eliminate $A(0..4)$ from further comparison and only consider $A(5..7)$.

$$A = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare, \blacksquare, 45, 74, 76\} \quad (10.2)$$

Next if we compare with $A(6) = 74$, we see that $45 < 74$, so we can eliminate all but $A(5)$ from consideration.

$$A = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare, \blacksquare, 45, \blacksquare, \blacksquare\} \quad (10.3)$$

One additional comparison with $A(5)$ allows us to determine that 45 is indeed contained in the sequence.

In our example, the sequence A could represent a variety of different concepts. A could represent a mathematical set and searching for an element in the sequence could tell us whether a value exists in the set. A could also represent only a portion of the data, often called a *key*, which is useful for indexing the rest of the information. For instance, the key could be a person's name. After searching based on the key then we know the position where the rest of the data about the person, such as their birthdate, is stored. In yet other cases, the key could be something more abstract, such as a cryptographic hash of some data or a key. In this case, the order that the data is stored is likely to be randomized, but we can still find it if we know the right cryptographic hash to search for. In each case, the fundamental operations of required for sorting and searching are pretty much the same, primarily we need to have the ability to compare two different values. For the remainder of this chapter we will mostly ignore these differences.

There are a wide variety of different sorting techniques that have been studied in processor systems[36]. These different algorithms vary in terms of their fundamental $\mathcal{O}()$ complexity, but many require $\mathcal{O}(n \log n)$ comparisons to sort n elements. Conceptually, this is intuitive since we

could search for the right location to insert a new value in a sorted data set with $\mathcal{O}(\log n)$ comparisons using binary search. To insert n elements this process would need to be repeated n times, once for each element.

In practice, the cost of inserting an element can be significant, depending on the data structure being sorted. In processor systems a variety of factors influence overall performance, such as memory locality when processing large data sets or their ability to easily parallelize across multiple cores. In HLS, we have similar considerations where it is common to trade off increased resource usage for reduced processing time. In many cases, this might require a variety of algorithms and implementation techniques in order to obtain the best design. The best techniques for making these tradeoffs are an area of active research [45, 54, 49].

Characteristics other than performance also affect the choice of sorting algorithms. For instance, we might consider:

- **Stability:** A sort is a stable if when two items in the input data have the same key, then they will appear in the same order on the output. For example, we might sort a set of records containing people's names and ages using the ages as the sort key. In the input data, John appears before Jane, and both are 25 years old. A stable sort will ensure that John and Jane remain in the same order after sorting.
- **Online:** The algorithm allows for data to be sorted as it is received. This can be particularly valuable when data is not accessible when the sort starts or must be read in sequence from external storage.
- **In-place:** A list with n elements can be sorted using n memory elements. Some algorithms require additional storage during the sorting process.
- **Adaptive:** It is efficient for data that is already relatively sorted. For example, if data is already sorted, then some algorithms might run faster, e.g. in linear $\mathcal{O}(n)$ time.

10.2 Insertion Sort

Insertion sort is one of the basic sorting algorithms. It works by iteratively placing the items of an array into sorted order and builds the sorted array one element at a time. Each iteration selects an unsorted element and places it in the appropriate order within the previously sorted elements. It then moves onto the next element. This continues until all of the elements are considered, and the entire array is sorted.

To make this more formal, assume that we are given an input array A that should be put into sorted order. The base case is the first element of that array $A[0]$, which by default is a sorted subarray (since it is only one element). The next step is to consider element $A[1]$, and place it into the sorted subarray such that the new subarray (with two elements) is also sorted. We continue this process for each element $A[i]$ until we have iterated across all of the elements of A . At each step, we take the new element $A[i]$ and insert it into the proper location such that the subarray $A[0..i-1]$ remains sorted. Figure 10.1 gives a step by step view of insertion sort operating on an array. The first line is trivial. We consider only the first value 3 which makes a subarray with one element. Any subarray with one element is in sorted order. The second line places the second value 2 into the sorted subarray. The end result is that the value 2 is placed into the first element of the sorted subarray, which shifts the previous sorted element 3 to the right. The third line moves the third entry of the initial array into its appropriate place in the sorted subarray. In this case, since $A[2] = 5$, it is already in its correct location. Thus, nothing needs to happen. The fourth line

{3, 2, 5, 4, 1}
 {3, 2, 5, 4, 1}
 {2, 3, 5, 4, 1}
 {2, 3, 5, 4, 1}
 {2, 3, 4, 5, 1}
 {1, 2, 3, 4, 5}

Figure 10.1: The Insertion Sort algorithm operating on an array. The initial array is shown at the top. In each step of the algorithm, the underlined algorithm is considered and placed into sorted order of the elements to it's left. At each stage, the shaded elements are in sorted order.

```

#include "insertion_sort.h"
void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item && j > 0) {
            #pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
}
    
```

Figure 10.2: The complete code for insertion sort. The outer **for** loop iterates across the elements one at a time. The inner **while** loop moves the current element into sorted place.

considers the value 4. This is moved into its appropriate place, shifting 5 to the right. Finally, the fifth line considers the placement of the value 1. This is placed into the first location of the array, and all of the previous sorted values are shifted by one location.

Insertion sort is a stable, online, in-place, adaptive sorting algorithm. Because of these properties, insertion sort is often preferred when sorting small arrays or as a base case in a recursive sorting algorithm. For example, more complex algorithms might decompose a large data set into a number of smaller arrays and then these small arrays will be sorted using insertion sort. The result is that formed by combining the sorted arrays.

10.2.1 Basic Insertion Sort Implementation

Figure [10.2](#) shows basic C code for insertion sort. The outer loop, labeled L1, iterates from elements A[1] to A[SIZE - 1] where SIZE denotes the number of elements in the array A. We do not need to start at element A[0] since any one element is already in sorted order. Each iteration of the L1 loop starts by copying the current element that we wish to insert into the sorted subarray (i.e., A[i]) into the item variable and then executes the inner L2 loop. The inner loop walks down the

sorted portion of $A[]$ looking for the appropriate location to place the value `index`. The inner loop executes as long as it has not arrived at the end of the array (the condition $j > 0$) and the array elements are greater than the item being inserted (the condition $A[j-1] > \text{index}$). As long as the loop condition is satisfied, elements of the sorted subarray are shifted by one element (the statement $A[j] = A[j-1]$). This will make room for the insertion of `index` when we eventually find its correct location. When the loop exits, we have found the correct location for `index` and store it there. After the completion of iteration i , the elements from $A[0]$ to $A[i]$ are in sorted order.

The code in Figure 10.2 is a straightforward implementation without any optimizations. We can optimize it using different Vivado[®] HLS directives, such as **pipeline**, **unroll**, and **array_partition**. The simplest optimization would be to pipeline the inner loop, by applying the **pipeline** directive to the body of the inner loop. In this case, even though the inner loop is accessing different elements of $A[]$ there are no data dependencies in these array accesses, so we could expect to achieve a loop II of 1. The resulting accelerator would perform roughly $N^2/4$ data comparisons[58] on average and have a latency of roughly $N^2/4$ cycles, since it performs one comparison per clock cycle. In actuality, an accelerator generated from Vivado[®] HLS will have slightly higher latency to account for the sequential execution of the outer loop. In order to achieve higher performance, we could also attempt to move the **pipeline** directive to the outer L1 loop or to the function body itself. We could also combine these options with partial loop unrolling. Some of these options are shown in Table ??.

Table 10.1: Possible options to optimize the basic `insertion_sort` function in Figure 10.2 through directives.

	Directives	II	Period	Slices
1	L2: pipeline II=1	?	?	?
2	L2: pipeline II=1 L2: unroll factor=2 array_partition variable=A cyclic factor=2	?	?	?
3	L1: pipeline II=1	?	?	?
4	L1: pipeline II=1 L1: unroll factor=2 array_partition variable=A complete	?	?	?
5	function pipeline II=1 array_partition variable=A complete	?	?	?

Explore the options in Table 10.1. Synthesize each of these designs and determine the initiation interval (II), clock period, and required number of slices. Which options are successful in improving latency and/or throughput? What would happen if you combined the directives from multiple rows into one design?

Unfortunately, although this code seems similar to other nested loop programs we've looked at previously, it does have some aspects that can make it difficult to optimize. Even Option 1, which simply attempts to pipeline the inner loop, can fail to achieve II=1. Although there are no significant data recurrences, there is a recurrence in the control path that affects whether or not

```

#include "insertion_sort.h"
void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && t > item) {
            #pragma HLS pipeline II=1
            A[j] = t;
            t = A[j-2];
            j--;
        }
        A[j] = item;
    }
}

```

Figure 10.3: Refactored insertion sort for Option 1 in Table 10.1.

the pipeline can execute. In this case, the code must read $A[i-1]$ in order to determine whether the loop should actually execute. Vivado[®] HLS incorporates this read into the loop pipeline, but if the read of A is pipelined then the loop exit check cannot be made in the first stage of the pipeline. This is an example of a recurrence which includes the HLS-generated control logic for the loop. If such recurrences exist, then Vivado[®] HLS will issue a message indicating that the loop exit condition cannot be scheduled in the first II clock cycles. This situation can also occur a **break** or **continue** statement exists under a complex control condition. One solution is to explicitly speculate the read of $A[i-1]$ out of the loop. This enables the loop exit check to be scheduled with $II=1$ at the expense of an additional array access on the last iteration. This code is shown in Figure 10.3.

Option 2 in the Table unrolls the inner loop by a factor of 2, attempting to perform two shift operations every clock cycle. This could potentially reduce the latency to compute `insertion_sort` by a factor of two. Unfortunately, Vivado[®] HLS cannot achieve a loop II of 1 for this code using `array partitioning`, because each array access cannot be assigned to a different memory partition.

In Vivado[®] HLS, the `array_partition` directive results in implementing a number of completely separate memories. For instance, `array_partition variable=A cyclic factor=4` would result in generating four separate memories from array $A[]$, each of which contains a portion of the array contents. We can often think that this optimization provides four times the number of memory accesses each clock, but this is only the case if each memory access can be assigned to exactly one of the memory partitions. For example an array access $A[i]$ for unknown i could reference data stored in any partition, while an array access $A[4*i+2]$ would only access data in the third partition. More complex logic, often called *memory banking*, can resolve a number of independent accesses $A[i]$, $A[i+1]$, $A[i+6]$, $A[i+7]$ and perform these accesses in the same clock cycle. Memory banking requires additional crossbar logic to route these simultaneous accesses in a circuit, since i can be arbitrary. At compile time, we can guarantee that the constant offsets

of these accesses will hit in different banks, but the actual banks cannot be determined until i is known. Yet more complex logic could implement stalling logic, enabling a set of unrelated accesses to complete in a single clock cycle if they hit in different banks. If the accesses happen to all hit in the same bank, then the stalling logic can delay the progress of the circuit for a number of clocks until all of the accesses have completed. Lastly, multiport architectures have been designed that can allow a number of accesses guaranteed completion every clock cycle [62, 1, 37] by replicating data across normal memories with one or two physical ports.

Option 3 in the table also fails to achieve significant improvement. Since the inner L2 loop does not have a statically computable loop bound Vivado[®] HLS is unable to construct a pipeline from the body of the L1 loop. Unfortunately, this is a case where exploring the design space of interesting alternatives requires code restructuring in addition to the use of directives. Finding the best code restructuring requires not only understanding the algorithm, but also having a sense of the architecture that will be generated by the HLS process [28, 48]. For example, we discuss one such code restructuring for insertion sort in the following section. You will see that that code is significantly different from the code in Figure 10.2.

In the following, we attempt to demonstrate several concepts. First, writing efficient high-level synthesis code requires that the designer must understand hardware concepts like unrolling and partitioning. Second, the designer must be able to diagnose any throughput problems, which requires substantial knowledge about both the application and the hardware implementation of that design. Third, and most importantly, in order to achieve the best results, i.e. high performance and low-area, it is often required to rewrite the code in a manner that will create an efficient hardware architecture. This can be very different from code that results in the most efficient software.

10.2.2 Parallelising Insertion Sort

In order to significantly increase the performance of insertion sort, we'd like to get to the point where we can insert a new element every clock cycle. When inserting the last element into the sorted list, this might require shifting all of the elements in the array. For the code in Figure 10.2, this means that the inner while loop could actually execute over all of the elements in the array. Intuitively, we realize that inserting a new element into the sorted list every clock cycle requires enough hardware operators to perform a comparison on every element of the array in a single clock cycle. To enable pipelining of the outer loop, we can convert the inner L2 loop with variable loop bounds into a fixed-bound loop, enabling it to be unrolled and integrated into the L1 loop pipeline. Code that does this is shown in Figure 10.4.

This code contains the exit condition of the original loop (L2 in Figure 10.2) as an **if** condition in the body of the new L2 loop. The other branches of the conditional are added to handle the expanded iteration space of the loop, essentially performing no operations when the original loop would not be executing. In addition, the loop now contains the final assignment of `item` in the array, rather than performing the assignment outside of the loop. When the inner loop is unrolled, remember that `j` will become a constant in all of the unrolled instances of the loop. As a result, each read and write from `B[j]` will be performed at a constant index and comparisons between `j` and a constant will be completely optimized away. The `item` variable, on the other hand, is possibly assigned in every copy of the inner loop. During compilation, Vivado[®] HLS creates a separate copy of this variable for each separate possible assignment and each possible assignment results in a multiplexer in the implemented circuit.

```
#include "insertion_sort_parallel.h"
#include "assert.h"
void insertion_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
    #pragma HLS array_partition variable=B complete
    L1:
    for(int i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        DTYPE item = A[i];
        L2:
        for(int j = SIZE-1; j >= 0; j--) {
            DTYPE t;
            if(j > i) {
                t = B[j];
            } else if(j > 0 && B[j-1] > item) {
                t = B[j-1];
            } else {
                t = item;
                if (j > 0)
                    item = B[j-1];
            }
            B[j] = t;
        }
    }
}
```

Figure 10.4: Refactored insertion sort for Option 3 in Table 10.1.

The conversion of a single variable into multiple versions is a common internal transformation used by compilers. The resulting internal representation is called **static single assignment (SSA)**. To merge values coming from different points in the code, the **SSA** internal representation include artificial ‘phi-functions’ represented by the greek letter ϕ . These phi-functions often result in multiplexers in the circuit generated by Vivado[®] HLS and you’ll probably find the related resources in the tool reports if you look carefully.

The parallelized insertion sort in Figure 10.4 essentially results in a number of copies of the body of the inner loop. The contents of this inner loop consists of a few multiplexers, a comparator to determine the smallest of two elements, and a register to store an element of $B[]$. Incidentally, each stage might also include pipeline registers, if needed, to ensure that the resulting circuit runs at an effective clock frequency. We’ll call the contents of the inner loop a **sorting cell**. The whole insertion sort function consists of a one-dimensional array of sorting cells, combined with a small amount of additional logic to feed data in the input and capture the output at the right time, in this case after **SIZE** elements have been processed by the outer loop. This array of sorting cells has an interesting property where each sorting cell only communicates with it’s neighboring sorting cells, rather than with all cells. Designs like this are called **systolic arrays** and are a common technique for parallelizing algorithms. In many cases, including sorting, systolic array implementations can naturally arise when we unroll inner loops, as long as the communication between different loop iterations is limited. We call this type of design an implicit systolic array.

10.2.3 Explicit Systolic Array For Insertion Sort

Systolic arrays have been well researched and many parallel algorithms are published as systolic arrays. In particular, the idea of using a linear array of sorting cells to implement insertion sort is well understood [55, 9, 45, 5]. However, rather than being described as an unrolled loop, systolic arrays are often described as components communicating by streams of data. This section describes an alternative coding style based on explicit streams and the **dataflow** directive that can be a more intuitive way to describe a systolic array.

Figure 10.5 shows a **systolic array** implementing insertion sort. Each cell is identical and compares its input (**in**) with the value in current register **local**. The smaller value is passed to the output **out**, while the larger value is stored back in **local**. In other words, $out = \min(in, local)$. The output of cell i is passed as input to the next cell $i + 1$ in the linear array. As a new input arrives it will be compared against elements stored in the array until it finds it’s correct place. If a new input is larger than all of the values in the array, then the sorted values will shift one cell to the right. If a new input is smaller than all of the values in the array, then it will propagate through the array, and eventually become stored in the furthest right cell. After all the data has moved through the array, the smallest data element will be sorted in cell $N - 1$ and can be read from the output.

The code for one insertion cell is shown in Figure 10.6. The code uses a streaming interface by declaring the input and output variables as an `hls::stream` type. `DTYPE` is a type parameter enabling different types to be operated on. The **local** variable stores one element of the array being sorted. It is **static** because we want to preserve its value across multiple function calls. This does create a problem since we must replicate this cell function **N** times. Using the same function (e.g., calling the same cell function **N** times) would cause a problem since each cell must have a separate static variable. One static variable cannot be shared across **N** functions.

Linking the insertion cells together is straightforward. The function `insertion_cell_sort` in Figure 10.7 shows the code for sorting eight elements. Expanding this to a larger number of elements

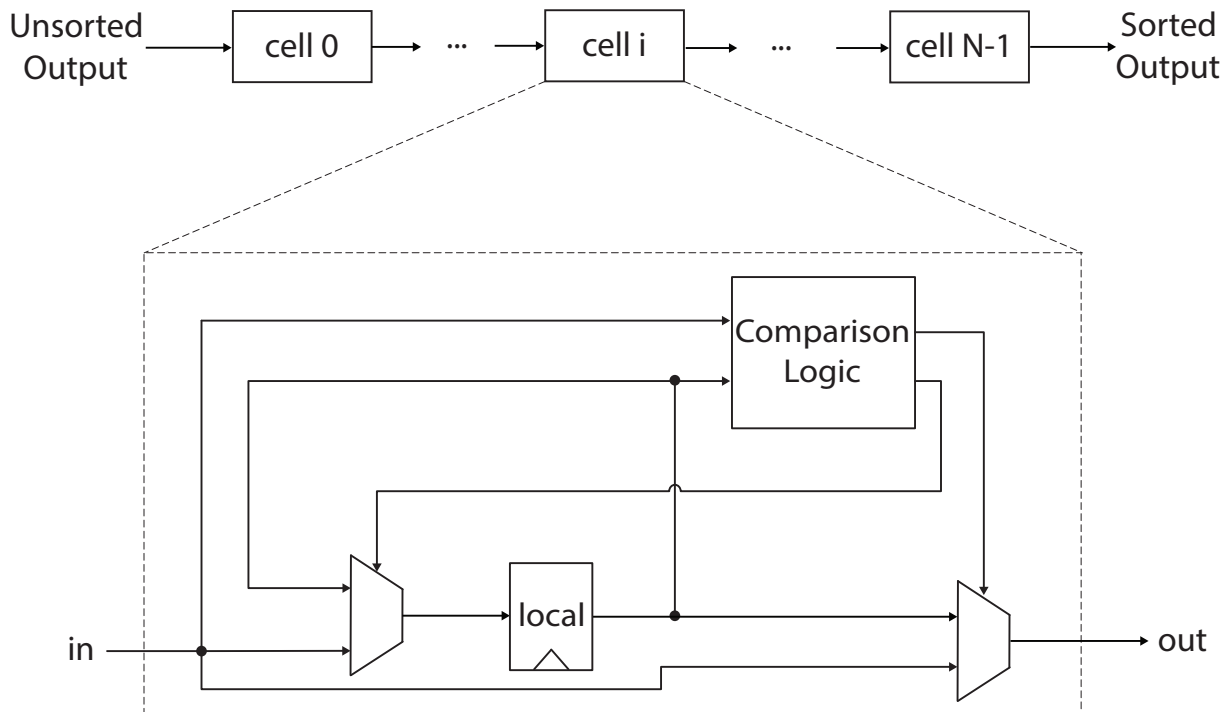


Figure 10.5: The architecture of one insertion cell. Each cell holds exactly one value in the register `local`. On each execution it receives an input value `in`, compares that to its `local` value, and writes the smaller of the two to `output`. Sorting N values requires N cells.

```
void cell0(hls::stream<DTYPE> & in, hls::stream<DTYPE> & out)
{
    static DTYPE local = 0;
    DTYPE in_copy = in.read();
    if(in_copy > local) {
        out.write(local);
        local = in_copy;
    }
    else
    {
        out.write(in_copy);
    }
}
```

Figure 10.6: The Vivado[®] HLS C code corresponding to one insertion cell `cell0`. The other cells have the exact same code except with a different function name (`cell1`, `cell2`, etc.). The code performs the same functionality as shown in the architectural diagram in Figure 10.5. It uses the `hls::stream` interface for the input and output variables. `hls::stream` provides a convenient method to create FIFOs that work both for synthesis and simulation.

```

void insertion_cell_sort(hls::stream<DTYPE> & in, hls::stream<DTYPE> & out)
{
    #pragma HLS DATAFLOW
    hls::stream<DTYPE> out0("out0_stream");
    hls::stream<DTYPE> out1("out1_stream");
    hls::stream<DTYPE> out2("out2_stream");
    hls::stream<DTYPE> out3("out3_stream");
    hls::stream<DTYPE> out4("out4_stream");
    hls::stream<DTYPE> out5("out5_stream");
    hls::stream<DTYPE> out6("out6_stream");

    // Function calls;
    cell0(in, out0);
    cell1(out0, out1);
    cell2(out1, out2);
    cell3(out2, out3);
    cell4(out3, out4);
    cell5(out4, out5);
    cell6(out5, out6);
    cell7(out6, out);
}

```

Figure 10.7: Insertion cell sorting for eight elements. The function takes an input `hls::stream` and outputs the elements in sorted order one at a time through the variable `out`. The order starts with the smallest element first, and then continues on to increasingly larger elements.

simply requires replicating more `cell` functions, additional `hls::stream` variables, instantiating these functions, and connecting their input and output function arguments in the appropriate manner.

The existing implementation of `insertion_cell_sort` outputs the data starting with the smallest element and continues outputting increasingly larger elements. What changes are required in order to reverse this order, i.e., to output the largest element first, and then the decreasingly smaller outputs?

The `insertion_cell_sort` function must be called multiple times in order to sort the entire data. Each call to `insertion_cell_sort` provides one data element to the array for sorting. The first call places the input data in its appropriate place. Where would this data be placed? To answer that question, we should point out that we make an assumption on the input data since we are initializing the local variable to 0. This is done in all of the cell functions.

Initializing the **static** variable `local` to 0 makes an assumption on the type of data that will be provided for sorting. What is this assumption? In other words, what is the range of values for the input data that will be properly handled? What would happen if input data from outside of this range was given to the `insertion_cell_sort` function? Is there a better way to initialize the local variable?

After making eight calls to the `insertion_cell_sort` function, all of the data will be put into the eight `local` variables in each of the `cell` functions.

How many times must we call the `insertion_cell_sort` function in order to get the first sorted element? How many calls are necessary in order to all of the sorted data? What if we increased the array to `N` cells? Can you generalize these two functions (number of times to call `insertion_cell_sort` to load `N` elements, and the number of calls required to output all `N` elements)?

To achieve a task-pipelined architecture consisting of the eight `cell` functions, the code specifies the **`dataflow`** directive. Each execution of the toplevel function processes a new input sample and inserts it into the sorted sequence. The actual insertion is pipelined, with one pipeline stage for each call to the `cell` function. With eight calls to the `cell` function, we can only sort sequences with eight values, but this can be extended almost arbitrarily since the code contains no recurrences between the cells.

How many cycles are required to sort an entire array? We define the sorting to be completed when all of the sorted data is output from the array of `cell` function inside of the `insertion_cell_sort`, i.e., all eight elements have been output from argument `out` in the function `insertion_cell_sort`? How does the cycle count change if we remove the **`dataflow`** directive? How does change the resource utilization?

Figure 10.8 shows the code for the testbench. The testbench generates random data to be sorted in `input[]`. This array is sorted by calling the `insertion_cell_sort()` function multiple times, with the result appearing in `cell_output[]`. Next, the same data is sorted in place using the `insertion_sort` function from Figure 10.2. Finally, the testbench compares the results of these two sort implementations. The testbench passes if the sorted order from both implementations is the same.

In the testbench, the `SIZE` constant is set to the number of elements being sorted. This would be 8 in the running example throughout this chapter. The `DEBUG` constant is used to provided output detailing the execution of the testbench. This should be set to a non-zero value if you wish to see the debugging data, and 0 if you want the output to be quiet. The input data is randomly generated using the `rand()` function and stored in the `input[]` array. You can change the data by modifying the argument in the call to `srand()`. The argument sets the seed for the random number generator to a specific value to ensure the generation of the same sequence of random numbers on each execution. Changing this value to a different integer will result in a different, but predictable, random sequence.

Note again that the testbench calls the `insertion_cell_sort()` function a total of `SIZE*2` times. Each of the first `SIZE` function calls feeds a single input element into function, but produce no useful output. The next `SIZE` calls provide dummy input data and produce one sorted element each function call. The data is produced one at a time starting with the smallest element.

After executing a sort, the code in Figure 10.6 leaves the `local` variable in a different state than when it started. Unfortunately, this means that we can only sort one array! In most cases, we not only want to sort more than one array, but we'd like to process each array back

```

#include "insertion_cell_sort.h"
#include <iostream>
#include <stdlib.h>

const static int DEBUG=1;
const static int MAX_NUMBER=1000;
int main () {
    int fail = 0;
    DTYPE input[SIZE];
    DTYPE cell_output[SIZE] = {0};
    hls::stream<DTYPE> in, out;

    //generate random data to sort
    if(DEBUG) std::cout << "Random_Input_Data\n";
    srand(20); //change me if you want different numbers
    for(int i = 0; i < SIZE; i++) {
        input[i] = rand() % MAX_NUMBER + 1;
        if(DEBUG) std::cout << input[i] << "\t";
    }

    //process the data through the insertion_cell_sort function
    for(int i = 0; i < SIZE*2; i++) {
        if(i < SIZE) {
            //feed in the SIZE elements to be sorted
            in.write(input[i]);
            insertion_cell_sort(in, out);
            cell_output[i] = out.read();
        } else {
            //then send in dummy data to flush pipeline
            in.write(MAX_NUMBER);
            insertion_cell_sort(in, out);
            cell_output[i-SIZE] = out.read();
        }
    }

    //sort the data using the insertion_sort function
    insertion_sort(input);

    //compare the results of insertion_sort to insertion_cell_sort; fail if they differ
    if(DEBUG) std::cout << "\nSorted_Output\n";
    for(int i = 0; i < SIZE; i++) {
        if(DEBUG) std::cout << cell_output[i] << "\t";
    }
    for(int i = 0; i < SIZE; i++) {
        if(input[i] != cell_output[i]) {
            fail = 1;
            std::cout << "golden=" << input[i] << "hw=" << cell_output[i] << "\n";
        }
    }

    if(fail == 0) std::cout << "PASS\n";
    else std::cout << "FAIL\n";
    return fail;
}

```

Figure 10.8: The testbench for the insertion_cell_sort function.

width = 1, $A[] = \{3, 7, 6, 4, 5, 8, 2, 1\}$
 width = 2, $A[] = \{3, 7, 4, 6, 5, 8, 1, 2\}$
 width = 4, $A[] = \{3, 4, 7, 6, 1, 2, 5, 8\}$
 width = 8, $A[] = \{1, 2, 3, 4, 5, 6, 7, 8\}$

Figure 10.9: The Merge Sort algorithm operating on an array. The initial state where each element is considered to be a sorted subarray of length one is shown at the top. At each step of the algorithm, subarrays are merged placing the shaded elements are in sorted order.

$in1[] = \{3, 4, 6, 7\}$	$in2[] = \{1, 2, 5, 8\}$	$out[] = \{ \}$
$in1[] = \{3, 4, 6, 7\}$	$in2[] = \{\blacksquare, 2, 5, 8\}$	$out[] = \{1\}$
$in1[] = \{3, 4, 6, 7\}$	$in2[] = \{\blacksquare, \blacksquare, 5, 8\}$	$out[] = \{1, 2\}$
$in1[] = \{\blacksquare, 4, 6, 7\}$	$in2[] = \{\blacksquare, \blacksquare, 5, 8\}$	$out[] = \{1, 2, 3\}$
$in1[] = \{\blacksquare, \blacksquare, 6, 7\}$	$in2[] = \{\blacksquare, \blacksquare, 5, 8\}$	$out[] = \{1, 2, 3, 4\}$
$in1[] = \{\blacksquare, \blacksquare, 6, 7\}$	$in2[] = \{\blacksquare, \blacksquare, \blacksquare, 8\}$	$out[] = \{1, 2, 3, 4, 5\}$
$in1[] = \{\blacksquare, \blacksquare, \blacksquare, 7\}$	$in2[] = \{\blacksquare, \blacksquare, \blacksquare, 8\}$	$out[] = \{1, 2, 3, 4, 5, 6\}$
$in1[] = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare\}$	$in2[] = \{\blacksquare, \blacksquare, \blacksquare, 8\}$	$out[] = \{1, 2, 3, 4, 5, 6, 7\}$
$in1[] = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare\}$	$in2[] = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare\}$	$out[] = \{1, 2, 3, 4, 5, 6, 7, 8\}$

Figure 10.10: The process of merging two sorted arrays. The initial state is shown at the top. In each step of the algorithm, the underlined elements are considered and one is placed into sorted order in the output array.

to back with no bubbles in the pipeline. Modify the code and the testbench to enable this and demonstrate that your modified code can sort multiple arrays.

10.3 Merge Sort

Merge sort is a stable, divide and conquer algorithm invented by John von Neumann in 1945 [36]. The basic idea of the merge sort algorithm is that combining two sorted arrays into a larger sorted array is a relatively simple operation, which can be completed in $\mathcal{O}(n)$ time. Conceptually, we divide the array into two subarrays, sort each subarray, and then combine the sorted subarrays into the final result.

Since we focus on sorting arrays of data, it turns out that the ‘divide’ operation is actually trivial. It requires no arithmetic or data movement and we can simply consider each individual element of the input array as a trivially sorted subarray. All of the computation is involved with merging subarrays into larger sorted subarrays. With other data representations, such as a linked-list, dividing an input array into subarrays can require traversing the data structure. The merge sort process is shown in Figure 10.9.

The process of combining two sorted arrays into one larger sorted array is sometimes called the “two finger algorithm”. Figure 10.10 describes the process using two sorted input arrays, named $in1[]$ and $in2[]$. These are merged into a sorted output array, named $out[]$.

The process starts with a “finger” pointing to the first element of each array. This conceptual finger is just an index into the array storing the data. As the algorithm proceeds, the fingers will point to different elements of the arrays. We underline these elements in order to show where the

fingers are placed. To begin with, the fingers point to the first element of each array, elements 3 and 1 in arrays `in1[]` and `in2[]`, respectively.

The first line of Figure 10.10 shows the initial state. There are four elements in each of the two input arrays and zero elements in the output array. We compare the first two elements of the input arrays, and move the smaller of these two to the output array. In this case, we compare 3 and 1, and move 1 into `out[]`. This reduces the number of elements in `in2[]`, and our “finger” moves to the next element in `in2[]` which is the next smallest element since the array is sorted. Once again, we compare the two elements from each of the input arrays, and move the smaller of the two elements to `out[]`. In this case, we compare 3 and 2, and move the element from `in2[]` to `out[]`. This process continues until all of the elements in one of the arrays is empty. In that case, we copy the remaining elements from the non-empty array into the output array.

Although the merge sort algorithm is a common textbook example of recursive function calls, most high-level synthesis tools do not support recursion or only support it in a limited manner. Thus, we will focus on a non-recursive implementation of the algorithm. The code might look substantially different from what you are used to, but the core of the algorithm is exactly the same.

10.3.1 Basic Merge Sort

Figure 10.11 shows basic code implementing a non-recursive merge sort. This code sorts an array by performing roughly $N \log N$ comparisons, but requires a temporary array to store the partially sorted data. The code starts by considering each element of the array as a length one sorted subarray. Each iteration of the outer loop merges pairs of sorted subarrays into larger sorted subarrays. After the first iteration we have sorted subarrays with maximum size 2, after the second iteration the sorted subarrays have maximum size 4, then 8 and so on. Note that if the size of the input array is not a power of two, then it is possible that we might end up with some subarrays which are smaller than the maximum size.

The sorting process starts in the `merge_sort()` function. The function primarily operates on the input `A[]` array, but leverages internal storage in `temp[]`. The size of both of these arrays is determined by the `SIZE` parameter. The parameter `DTYPE` determines the type of data being sorted.

The computation of the function consists of two nested **for** loops. The outer **stage** loop keeps track of the number of elements in each sorted subarray in the `width` variable. The function starts by considering each element as a separate subarray; hence `width` is initialized as 1. Every iteration of the **stage** loop results in the generation of longer sorted subarrays. These new subarrays potentially have twice as many elements, which is why `width` doubles on each iteration of the **stage** loop. The **stage** loop terminates when `width` is greater than or equal to `SIZE`, indicating that all elements of `A[]` are in a single sorted subarray.

Each iteration of the inner **for** loop, labeled `merge_arrays` merges two consecutive subarrays. These subarrays each consist of up to `width` elements, starting at index `i1` and `i2`. These two subarrays are merged and copied into a single subarray stored in `temp[]` using the `merge()` function. The main complexity here is dealing with the boundary condition at the end of the loop if `SIZE` is not exactly a power of two. In this case the subarrays might contain less than `width` elements. After merging subarrays into `temp[]`, the final loop copies the data back into `A[]` for the next iteration of the loop.

The code in Figure attempts to handle a wide variety of values for the parameter `SIZE`. What values are allowed? When the `merge()` function is called, where are the possible relationships

```
#include "merge_sort.h"
#include "assert.h"

// subarray1 is in[i1..i2-1], subarray2 is in[i2..i3-1], result is in out[i1..i3-1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
        // Select the smallest available element.
        if((f1 < i2 && in[f1] <= in[f2]) || f2 == i3) {
            out[index] = in[f1];
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = in[f2];
            f2++;
        }
    }
}

void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    // Each time through the loop, we try to merge sorted subarrays of width elements
    // into a sorted subarray of 2*width elements.
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        merge_arrays:
        for (int i1 = 0; i1 < SIZE; i1 = i1 + 2 * width) {
            // Try to merge two sorted subarrays:
            // A[i1..i1+width-1] and A[i1+width..i1+2*width-1] to temp[i1..2*width-1]
            int i2 = i1 + width;
            int i3 = i1 + 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge(A, i1, i2, i3, temp);
        }

        // Copy temp[] back to A[] for next iteration
        copy:
        for(int i = 0; i < SIZE; i++) {
            A[i] = temp[i];
        }
    }
}
```

Figure 10.11: A non-recursive implementation of merge sort. The `merge_sort()` function iteratively merges subarrays until the entire array has been sorted.

between `i1`, `i2`, and `i3`? If we restrict the allowed values of the parameter `SIZE` can the code be simplified? What is the affect on the resulting HLS generated circuit?

The `merge()` function performs the “two finger” algorithm on two subarrays within the `in[]` array. The function takes input in the `in[]` array and produces output in the `out[]` array. The function also takes as input variables `i1`, `i2`, and `i3` which describe the extent of the two subarrays to be merged. One subarray starts at index `i1` and includes all the elements before `i2` and the second subarray starts at index `i2` and includes the elements up to `i3`. The merged output subarray will be stored from index `i1` up to `i3` in `out[]`.

The `merge()` function consists of a single loop which iterates over the elements being stored into `out[]`. Each iteration places an element into its correctly sorted location in `out[]`. The variables `f1` and `f2` within the function correspond to the position of the fingers for each subarray. The `if` condition selects the smaller of `in[f1]` or `in[f2]` to copy to the next sorted position in `out[]`. However, the `if` condition is more complex since it has to deal with several special cases. One case is where `f1 == i2` and we have run out of elements to consider for `in[f1]`, in which case we must select `in[f2]` as the smallest element. Alternatively, if `f2 == i3`, then we have run out of elements to consider for `in[f2]`, in which case we must select `in[f1]` as the smallest element.

What happens to the `in[]` array over the course of the computation? Describe the state of the elements in the `in[]` array after each iteration of the outer `for` loop. What is the final order of the elements in the `in[]` array when `merge_sort()` returns?

The performance report after synthesis may not be able to determine the number of cycles for the latency and interval. Why is that the case? What are appropriate `min`, `max`, and `avg` values to provide in a `loop_tripcount` directive(s)?

The code is not particularly optimized for any particular high-level synthesis (HLS) implementation. The best starting point for optimization is by adding directives. Given that we have several nested `for` loops, we generally look first at optimizing the inner loops. Optimizations of inner loops are usually much more significant than optimizations of outer loops, which are executed relatively rarely. By this point, you should be familiar with the common `pipeline` and `unroll` directives for loop optimization.

Perform different optimizations using the `pipeline` and `unroll` directives on the `for` loops. What provides the best performance? Which gives the best tradeoff between resource utilization and performance? What aspects of the code prevent higher performance? Are these aspects fundamental to the algorithm, or only because of the way the algorithm is captured in the code?

Pipelining and unrolling can be hindered by resource constraints; in particular, we must carefully consider the number of memory ports for the arrays. The arrays in this code seem relatively straightforward as they are both one-dimensional. Yet, the designer must carefully consider the access patterns to insure that performance optimizations match with the resource constraints.

Optimize the code using loop optimizations and array partitioning, i.e., create a set of designs using the **array_partition**, **pipeline**, and **unroll** directives. Were you able to achieve better results than by using the **pipeline** and **unroll** directives alone? What was the best strategy for performing design space exploration using these directives? What was your best design in terms of performance? What was the best design that provides a good tradeoff between resource utilization and performance?

Many times the best designs are only possible by performing code restructuring. Although Vivado[®] HLS provides many directives to enable common code optimizations, it is impractical to provide directives for every optimization. Sometimes we must resort to rewriting the code in addition to providing directives in order to achieve a design that meets our requirements. In the next section we describe one way to significantly restructure the merge sort code in order to increase throughput.

10.3.2 Restructured Merge Sort

Looking first at the inner loop of the `merge()` function, you might have found that it was difficult to achieve a loop II of 1. One challenge is that there are actually four reads of `in[]`, but only at two different addresses. The HLS tool must recognize that these reads are redundant, since **block RAM (BRAM)** memories can only support two accesses each clock. However because these reads are in different basic blocks, it is more difficult for a compiler to eliminate the redundant loads. By eliminating the redundant reads, the compiler needs to do less optimization to achieve a loop II of 1. The restructured code is shown in Figure 10.12. In addition, there is a recurrence through the `f1` and `f2` variables. These variables are incremented in one of the branches of the **if** conditional, which must be used in the next iteration to determine which locations in `in[]` to compare and subsequently which branch of the conditional to take. Because the floating point comparison is relatively complex, this recurrence can also limit the achievable II and clock period.

The behavior of this code is shown in Figure 10.13. Although the inner loop achieves a loop II of 1, this inner loop often has a very small number of loop iterations. When the inner loop finishes, the pipeline must empty before code executing after the pipeline can execute. Although the loop pipeline is relatively short in this case, the bubble caused by the loop completing is significant, since the number of iterations is also small. Unfortunately, because of the limits of static loop analysis, the performance of this particular code is somewhat hard to visualize. In this case, the number of iterations of the inner loop is data dependent.

A common approach is to flatten loop nests like these into a single loop, reducing the number of times that the pipeline must flush when exiting a loop. In fact, Vivado[®] HLS will automatically flatten perfect loop nests. In this case, however, since the code does not contain a perfect loop nest, we can resort to flattening the loops manually. Code resulting from manually flattening the `merge_arrays` loop with the loop inside the `merge()` function is shown in Figure 10.14. Note that one advantage of this code is that the `merge_arrays` loop also has a constant number of loop iterations, making understanding performance much easier.

Estimate the performance of the code in Figure 10.14. Even though the inner loops have achieved a loop II of 1, is the design using hardware efficiently? Is there a way to further reduce the latency of the `merge_sort()` function to the point where it is using approximately $N \log N$ clock cycles?

```

#include "merge_sort.h"
#include "assert.h"

// subarray1 is in[i1..i2-1]; subarray2 is in[i2..i3-1]
// sorted merge is stored in out[i1..i3-1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // For each element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
#pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        // Select the smallest available element.
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[index] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = t2;
            f2++;
        }
    }
}
    
```

Figure 10.12: Restructured code for the `merge()` function, which can achieve a loop II of 1 in Vivado[®] HLS.

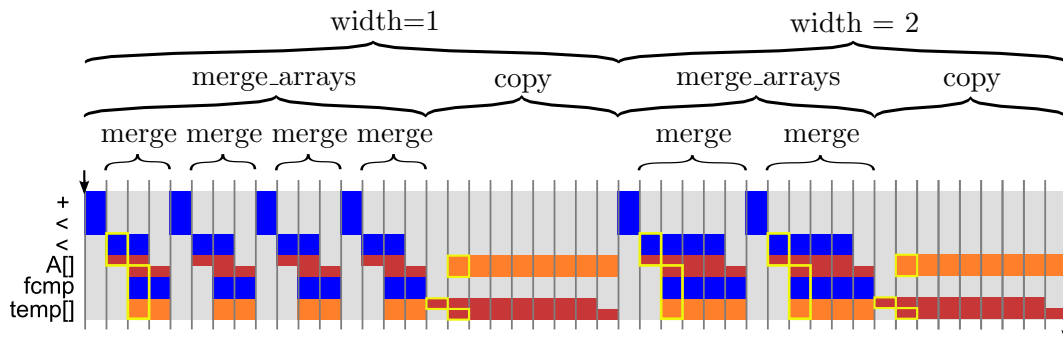


Figure 10.13: Behavior of the restructured code in Figure 10.12.

```
#include "merge_sort.h"
#include "assert.h"

void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        int f1 = 0;
        int f2 = width;
        int i2 = width;
        int i3 = 2*width;
        if(i2 >= SIZE) i2 = SIZE;
        if(i3 >= SIZE) i3 = SIZE;
    merge_arrays:
        for (int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
            DTYPE t1 = A[f1];
            DTYPE t2 = A[f2];
            if((f1 < i2 && t1 <= t2) || f2 == i3) {
                temp[i] = t1;
                f1++;
            } else {
                assert(f2 < i3);
                temp[i] = t2;
                f2++;
            }
            if(f1 == i2 && f2 == i3) {
                f1 = i3;
                i2 += 2*width;
                i3 += 2*width;
                if(i2 >= SIZE) i2 = SIZE;
                if(i3 >= SIZE) i3 = SIZE;
                f2 = i2;
            }
        }

    copy:
        for(int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
            A[i] = temp[i];
        }
    }
}
```

Figure 10.14: Restructured code for Merge Sort which can achieve a loop II of 1 with fewer pipeline bubbles in Vivado® HLS.

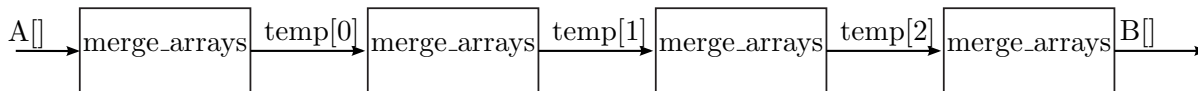


Figure 10.15: Dataflow pipelined architecture for implementing 4 stages of Merge Sort. This architecture can sort up to 16 elements.

So far we have focused on optimizing the `merge_sort()` function to reduce the latency of the computation without significantly increasing the number of resources. As a result, the accelerator is becoming more efficient. However, after achieving a reasonably efficient design, the only way to further reduce latency and/or increase throughput is increase parallelism. Previously we have seen ways of unrolling inner loops and partitioning arrays as a way to perform more work each clock cycle. An alternative way to increase parallelism is leverage pipelining. In addition to operator-level pipelining we can also look for coarser-granularity task-level pipelining.

With Merge Sort it turns out that we can make a dataflow pipeline out of each iteration of the `stage` loop, assuming that we have a fixed size array to sort. In Vivado[®] HLS this implementation can be conceptually achieved by unrolling the `stage` loop and using the **dataflow** directive. Each instance of the `merge_arrays` loop then becomes an independent process which can operate on a different set of data. The resulting architecture is shown in Figure 10.15.

The code to implement this is shown in Figure 10.16. This code is similar in many ways to the original code, but has several important differences. One key difference is that the `merge_arrays` loop has been extracted into a function, making it easier simpler to rewrite the toplevel code. A second key difference is that the output of `merge_sort_parallel()` is produced in a separate array than the input, enabling Vivado[®] HLS to build an architecture that pipelined architecture processing using the **dataflow** directive. Additionally, the `temp[][]` array is now used to model the dataflow ping-pong channels between the processes implemented by the `merge_arrays()` function, making extra array copies unnecessary. This two-dimensional array is partitioned in the first dimension, making it easy to represent a parameterized number of channels.

The `merge_sort_parallel()` function consists of `STAGES` calls to the `merge_arrays()` function. Each The first call reads from the input and writes to `temp[0]`. The loop performs the intermediate stages and writes to the remaining partitions of `temp[]`. The final call writes to the output array `B[]`. The code is parameterized in terms of `SIZE` and `STAGES` and supports array lengths 4 or greater.

Estimate the performance of the code in Figure 10.16. What is the interval and latency of the implementation? How much memory is required to support that processing rate? Is all of the memory necessary?

10.4 Conclusion

This chapter introduced a number of basic sorting algorithms with fundamentally different algorithmic tradeoffs. Insertion sort operating on an array performs on average $N^2/4$ comparisons, but requires much fewer (approximately N comparisons) when operating on sorted data. We showed several ways of parallelizing Insertion sort which can increase performance at the expense of using more hardware resources. Because of bubbles in the statically scheduled pipeline, we end up needing roughly N comparators with an interval of N cycles to perform insertion sort. Merge sort operating on an array performs generally fewer comparisons, roughly $N \log N$, but requires

```
#include "merge_sort_parallel.h"
#include "assert.h"

void merge_arrays(DTYPE in[SIZE], int width, DTYPE out[SIZE]) {
    int f1 = 0;
    int f2 = width;
    int i2 = width;
    int i3 = 2*width;
    if(i2 >= SIZE) i2 = SIZE;
    if(i3 >= SIZE) i3 = SIZE;
    merge_arrays:
    for (int i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[i] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[i] = t2;
            f2++;
        }
        if(f1 == i2 && f2 == i3) {
            f1 = i3;
            i2 += 2*width;
            i3 += 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            f2 = i2;
        }
    }
}

void merge_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
    #pragma HLS dataflow

    DTYPE temp[STAGES-1][SIZE];
    #pragma HLS array_partition variable=temp complete dim=1
    int width = 1;

    merge_arrays(A, width, temp[0]);
    width *= 2;

    for (int stage = 1; stage < STAGES-1; stage++) {
        #pragma HLS unroll
        merge_arrays(temp[stage-1], width, temp[stage]);
        width *= 2;
    }

    merge_arrays(temp[STAGES-2], width, B);
}
```

Figure 10.16: Restructured code for Merge Sort which can be implemented as a dataflow pipeline by Vivado[®] HLS.

additional memory to store partially sorted results. The more complex loop structure of Merge Sort means that we required additional refactoring to reach an efficient solution that took roughly $N \log N$ cycles with one comparator. We also showed a task-pipelined implementation which performs $\log N$ comparisons every clock cycle with an interval of N cycles to perform merge sort. In contrast with insertion sort, this requires much fewer comparisons to achieve the same interval, but requires more memory (in the form of dataflow channels) and has much higher latency.

In practice, many FPGA-based implementations of sorting will have to address these fundamental tradeoffs in order to achieve the best integration with other aspects of a system. Another sort with a fundamentally different tradeoff is the Radix Sort, which focuses on data that exists within a bounded range, unlike the more general sorting algorithms in this chapter which only require comparison between items. Radix sort will be implemented as part of the Huffman Coding in Chapter [11](#).

Parallel implementations of sorting algorithms are often described as sorting networks. Sorting networks are sometimes described as [systolic arrays](#) and other times as pipelines. We can often gain inspiration for HLS designs by looking investigating these existing parallel implementations and then capturing them as C code. In Vivado[®] HLS, these networks can be described using either loop pipelines, dataflow pipelines, or a combination of both.

Glossary

array partitioning Dividing a single logical array into multiple physical memories.. [87](#), [152](#), [177](#)

bitstream The configuration data used to program the functionality of an FPGA. [13](#)

BRAM A block RAM is a configurable random access memory that is embedded throughout an FPGA for data storage and communication.. [17](#), [131](#), [150](#), [164](#), [189](#), [232](#)

C/RTL cosimulation The process of verifying an [RTL](#) design generated by HLS using testvectors captured from the C testbench.. [128](#)

Compressed Row Storage Compressed Row Storage is a technique for representing a sparse matrix. It allows large matrices with a small number of elements to be stored and operated on efficiently.. [117](#)

data rate The frequency at which a task can process the input data. This is often expressed in bits/second and thus also depends on the size of the input data. [20](#)

Discrete Fourier Transform An transformation that takes a discrete signal and converts it to a frequency-domain representation.. [4](#), [77](#), [99](#), [231](#)

EDA Electronic design automation (EDA) are a set of software tools used to aid the hardware design process. . [11](#)

Fast Fourier Transform An optimized version of the [Discrete Fourier Transform \(DFT\)](#) which requires fewer operations.. [77](#), [99](#)

FF A flip-flop (FF) is a circuit that can store information. We typically think of it as storing one bit of data and are a fundamental building block for creating memories in digital circuits.. [14](#), [150](#), [164](#), [232](#)

finite impulse response A common digital signal processing task that performs a convolution on the input signal with a fixed signal that is defined by its coefficients. The FIR is often performed in hardware and can be efficiently implemented. [22](#)

FPGA A field-programmable gate array (FPGA) is an integrated circuit that can be customized or programmed after it is manufactured (“in the field”). . [11](#), [232](#)

HLS High-level synthesis is a hardware design process that translates an algorithmic description (which is decoupled from the cycle to cycle behavior) into a register transfer level (RTL) hardware description language which specifies the exact behavior of the circuit on a cycle-by-cycle basis. [11](#), [188](#)

- I/O block** An I/O block provides the interface between the FPGA fabric and the remainder of the system. I/O blocks can talk to memories (e.g., on-chip caches and off-chip DRAM, microprocessors (using AXI or other protocols), sensors, actuators, etc.. . [16](#), [233](#)
- IP core** An RTL-level component with well-defined interfaces enabling it to be incorporated into a larger design. Often used as a way of hiding the ‘intellectual property’ from another company, hence the name.. [18](#)
- logic synthesis** The process of converting an glsrtl design into a netlist of device-level primitives.. [13](#)
- loop interchange** A code transformation that changes the order of loop operations. This transformation is often a useful approach to addressing recurrences in code.. [94](#)
- loop pipelining** Enabling multiple iterations of a loop to run concurrently sharing the same functional units.. [84](#)
- LUT** A lookup table (LUT) is a memory where the address signal are the inputs and the corresponding outputs are contained in the memory entries. It is a key computational component of modern [field-programmable gate array \(FPGA\)](#)s.. [13](#), [232](#)
- netlist** An intermediate design artifact consisting of device-level primitive elements and the connections between them. In FPGA designs, the primitive elements include [lookup table \(LUT\)s](#), [flip-flop \(FF\)s](#), and [block RAM \(BRAM\)s](#).. [13](#)
- partial loop unrolling** A transformation where the body of a loop is replicated multiple times. This is often used in processor systems to reduce loop condition overhead or to provide opportunities for vectorization. In HLS, it can have a similar effect, enabling more operations from the same loop nest to be considered in scheduling. This can improve the performance of a design.. [90](#), [126](#)
- place and route** The process of converting a netlist of device-level primitives into the configuration of a particular device.. [13](#)
- process** An individual component in a dataflow architecture. [114](#)
- processing element** A coarse-grained concurrently executing component in a design. In HLS, this is often used in the context of a dataflow design.. [153](#)
- recurrence** A code structure that results in a feedback loop when implemented in a circuit. Recurrences limit the throughput of the circuit.. [26](#), [94](#)
- ROM** A Read-only Memory is a memory which is initialized to a particular value and then read but never written. In many cases the storage for ROMs can be highly optimized because their value never changes.. [96](#)
- routing channel** A routing channel provides a flexible set of connections between the FPGA programmable logic elements. . [15](#), [16](#), [15](#), [16](#), [233](#)
- RTL** Register-transfer level (RTL) is a hardware design abstraction which models a synchronous digital circuit using logical operations that occur between between hardware registers. It is common design entry for modern digital design. [11](#), [123](#), [202](#)

- slice** A (typically small) set of LUTs, FFs and multiplexors. These are often reported in FPGA resource utilization reports. . 14, 15, 14, 15, 16
- sorting cell** An simple stateful component that forms part of a larger sorting network or algorithm. Commonly a cell performs a compare-and-swap operation between two elements.. 180
- stable sort** A sorting algorithm that keeps different elements with the same sorting key in their original sequence after sorting.. 174, 205
- Static Single Assignment** Static Single Assignment is an intermediate representation in compilers where each variable is assigned only once. This form makes many common optimizations simpler to write.. 180
- switchbox** A switchbox connects routing channels to provide a flesible routing structure for data routed between the programmable logic and I/O block. . 15, 16, 15, 16
- systolic array** An array of processing elements that coordinate to perform a more complex algorithm. Systolic arrays are usually designed so that each processing element encapsulates some local information and only communicates with its local neighbors. This often enables systolic arrays to easily scale to large problem sizes by increasing the size of the array.. 180, 194
- task** A fundamental atomic unit of behavior or high-level synthesis computation; this corresponds to a function invocation in high-level synthesis. 20
- task pipelining** Being able to execute more than one task concurrently on the same accelerator in a pipelined fashion.. 113, 114, 154
- task interval** The time between when one task starts and the next starts or the difference between the start times of two consecutive tasks. 20
- task latency** The time between when a task starts and when it finishes. 20