Parallel Programming for FPGAs

Ryan Kastner and Stephen Neuendorffer

2017-10-10

"When someone says, 'I want a programming language in which I need only say what I wish done', give him a lollipop." -Alan Perlis

This book focuses on the use of algorithmic high-level synthesis (HLS) to build application-specific FPGA systems. Our goal is to give the reader an appreciation of the process of creating an optimized hardware design using HLS. Although the details are, of necessity, different from parallel programming for multicore processors or GPUs, many of the fundamental concepts are similar. For example, designers must understand memory hierarchy and bandwidth, spatial and temporal locality of reference, parallelism, and tradeoffs between computation and storage.

This book is a practical guide for anyone interested in building FPGA systems. In a university environment, it is appropriate for advanced undergraduate and graduate courses. At the same time, it is also useful for practicing system designers and embedded programmers. The book assumes the reader has a working knowledge of C/C++ and includes a significant amount of sample code. In addition, we assume familiarity with basic computer architecture concepts (pipelining, speedup, Amdahl's Law, etc.). A knowledge of the RTL-based FPGA design flow is helpful, although not required.

The book includes several features that make it particularly valuable in a classroom environment. It includes questions within each chapter that will challenge the reader to solidify their understanding of the material. It provides specific projects in the Appendix. These were developed and used in the HLS class taught at UCSD (CSE 237C). We will make the files for these projects available to instructors upon request. These projects teach concepts in HLS using examples in the domain of digital signal processing with a focus on developing wireless communication systems. Each project is more or less associated with one chapter in the book. The projects have reference designs targeting FPGA boards distributed through the Xilinx University Program (`http://www.xilinx.com/support/university.html`). The FPGA boards are available for commercial purchase. Any reader of the book is encouraged to request an evaluation license of Vivado® HLS at `http://www.xilinx.com`.

This book is *not* primarily about HLS algorithms. There are many excellent resources that provide details about the HLS process including algorithms for scheduling, resource allocation, and binding [36, 15, 20]. This book is valuable in a course that focuses on these concepts as supplementary material, giving students an idea of how the algorithms fit together in a coherent form, and providing concrete use cases of applications developed in a HLS language. This book is also *not* primarily about the intricacies of FPGA architectures or RTL design techniques. However, again it may be valuable as supplementary material for those looking to understand more about the system-level context.

This book focuses on using Xilinx tools for implementing designs, in particular Vivado® HLS to perform the translation from C-like code to RTL. C programming examples are given that are specific to the syntax used in Vivado® HLS. In general, the book explains not only Vivado® HLS specifics, but also the underlying generic HLS concepts that are often found in other tools. We encourage readers with access to other tools to understand how these concepts are interpreted in any HLS tool they may be using.

Good luck and happy programming!

Contents

Add section describing formatting used to denote matrices, vector, math, etc.?

Chapter 1

Introduction

1.1 High-level Synthesis (HLS)

The goal of high-level synthesis (HLS) is to raise the level of abstraction in hardware design. Over the years, hardware designers have moved from specifying individual transistors and wires, to combinational and sequential Boolean models of computation using logic gates and flip-flops, to register-transfer level (RTL) operations with arithmetic and simple control flow operators. HLS raised the abstraction beyond RTL design. The goal is to free the designer from making every decision on a cycle-by-cycle basis and allow them to focus on a more functional specifications.

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 [10], 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
- 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, the 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. And the ability to perform recursion is often limited. But 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, a tool may output testbenches to aid in the verification process. Additionally, 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.

1.2 Hardware Design Process

Move this after FPGA architecture section?

The goal of the field-programmable gate array (FPGA) design process produces a bitstream – the configuration data used to program the FPGA. The bitstream contains the low level details about how to program the FPGA logic elements, route individual wires, and configure on-chip memories and all the other FPGA resources.

The hardware design process has evolved 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 of transistors. The Mead and Conway approach [35] 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. RTL was one step in abstraction; here the designer can simply specify the registers and the operations performed on those registers. EDA tools can translate RTL specifications into logic circuits and then subsequently into a bitstream. HLS is yet another step in abstraction. This allows a hardware designers to specify more functional behaviors. The goal of the HLS tools is to compile a RTL specification which is then synthesized into a bitstream using the FPGA EDA tools.

add figure that provides an overview of the hardware design process?

1.3 FPGA Architecture

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 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. This is a flexible and fairly efficient method for encoding smaller Boolean logic functions.

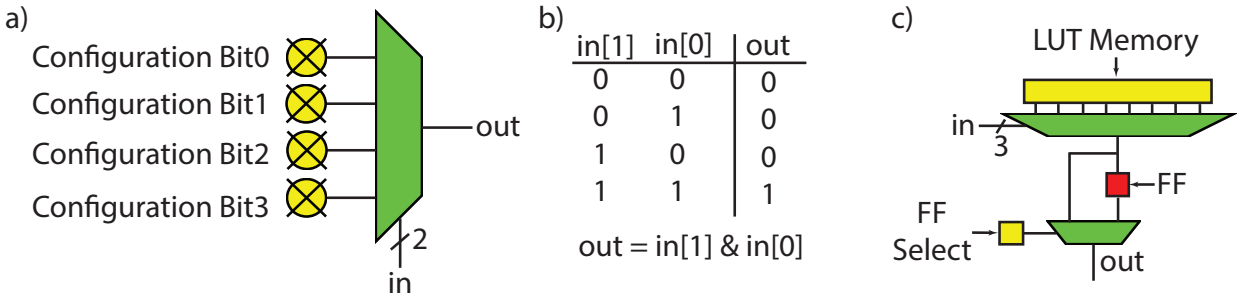


Figure 1.1: Part a) shows a 2 input LUT – or 2-LUT. Each of the four configuration bits can be programmed to change the function of the 2-LUT making it a fully programmable 2 input logic gate. Part b) provides a sample programming to implement an AND gate. The values in the "out" column from top to bottom correspond directly to configuration bits 0 through 3. Part c) shows a slightly more complex 3-LUT with the possibility of storing the output into a flip-flop (FF). Note that there are nine configuration bits: 8 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. This can be viewed a simple slice.

Most FPGAs use a LUTs with 4-6 input bits as their base element for computation. Larger FPGAs can have millions of LUTs.

The FF is the basic memory element for the FPGA. They are typically colocated 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 arrays block (LAB) or slice depending on the vendor or design tool. We use the term slicet since it is the resource reported by the Vivado® HLS tool. A slice is a small number of LUTs, FFs and multiplexors combined to make a more powerful programmable logic element. The exact number and combination of LUTs, FFs and multiplexors 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 one FF.

Programmable interconnect is the other key element of an FPGA. It provides a flexible network of wires to create connections between the programmable logic elements. The inputs and outputs of the programmable logic 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. Switchboxes provide a programable switch that routes data between different routing channels. Figure 1.2 shows how a slice, routing channel, and switchbox are generally 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 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. The slices represent "logic islands" that

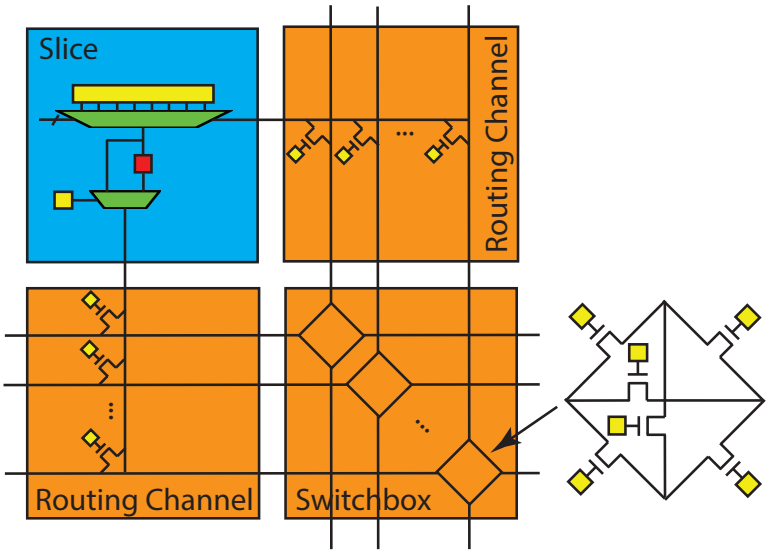


Figure 1.2: A slice contains a small numbrer of LUTs and FF. These are connected to one another using a routing channel and switchbox. These two provide a programmable interconnect that provide the data movement between the programmable logic elements (slices).

are connected using the routing channels and switchboxes. A switchboxes connects to four routing channels to the north, east, south, and west directions. The exact programming of the pass transistors 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 [9, 7, 23]. Generally, it is not necessary to understand all of the nitty-gritty details of the FPGA architecture, 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. The figure shows the two dimensional layout of the programmable logic (e.g., slices), routing channels, and switchboxes. The FPGA uses an 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 a actuator (e.g., a motor through an 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 the fabric in order to increase performance.

something about BRAMs, DSP48s and the generally more hetereogenous modern FPGAs. Talk about Figure 1.4.

The I/O interface is very important, and we will discuss that in more detail throughout the book, but for the time being we can assume that our HLS block is put onto the FPGA fabric and the rest of the system will dictate the interface through the I/O blocks. At

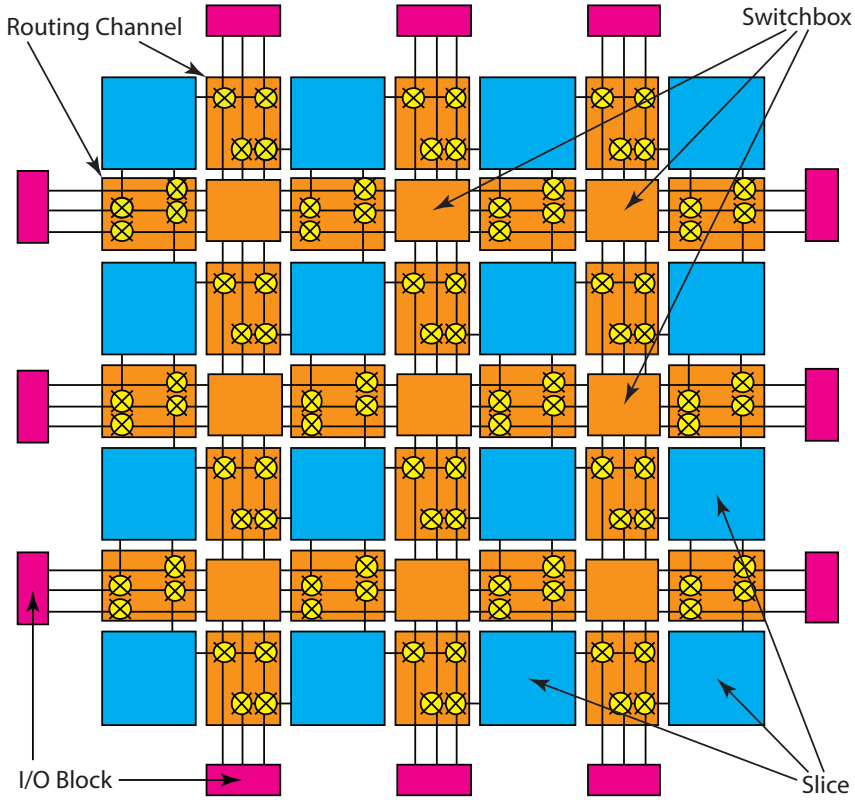


Figure 1.3: The two-dimensional structure of an FPGA. The slices are interconnected using routing channels and switchboxes. The I/O blocks provide an external interface, e.g., to a memory, microprocessor, sensor, and/or actuator. On some FPGAs, the IØdirectly connects to the chip pins. Other FPGAs use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

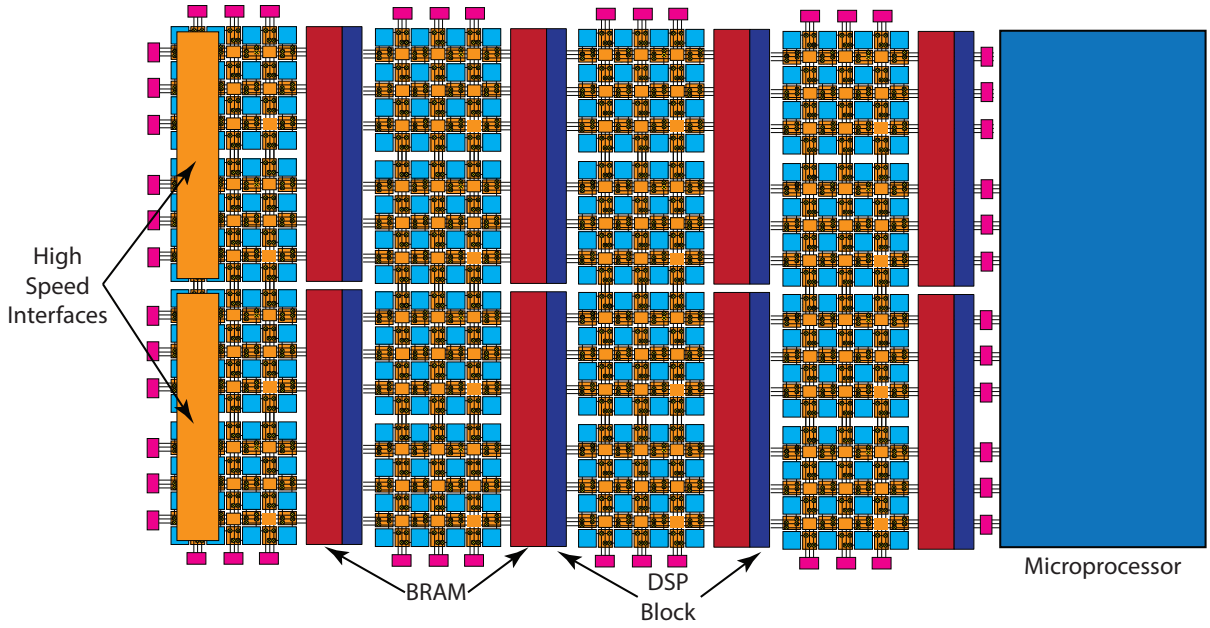


Figure 1.4: Modern FPGAs are becoming more heterogenous with a mix of programmable logic elements and architectural elements like register files, custom datapaths, and high speed interconnect (e.g., AXI and PCI-e). The FPGA is often paired with one or more microprocessors, e.g., ARM or x86 cores, that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

the periphery of the design, close to the I/O pins, is a some 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 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 repeatably capture data in a register.

Away from I/O pins, FPGA designs often contain soft processors, 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 *custom cores*.

	External Memory	Block Memory	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: Types of memories in FPGA-based systems.

As with standard cores, custom 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. This book will focus on the development of custom cores from HLS.

One of the challenges of FPGA design is that a circuit must be implemented using the fixed set of primitives available in the FPGA. Typically, given an RTL description in a Hardware Description Language (HDL), vendor-specific tools will transform the circuit into the FPGA primitives. In this process, combinational logic elements are typically mapped onto LUTs, whereas sequential logic elements may be mapped onto a variety of storage elements including FFs or embedded memories.

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 the 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.

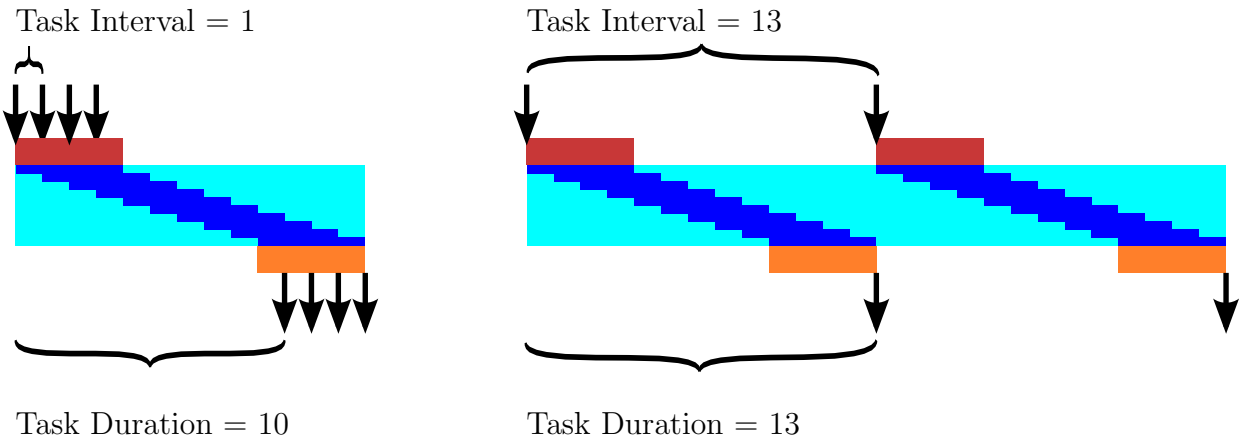


Figure 1.6: The task interval and task latency for two different designs. The left design is pipelined while the right one uses a more sequential implementation.

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 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 1.6 shows two different designs of some hypothetical application. On the left, there is an architecture that starts a new task every cycle. This corresponds to a 'fully-pipelined' implementation. On the right, there is task with the same latency and interval (13 cycles). This means that the architecture works sequentially on the tasks – one after the other. In other words, there is only one task being executed at any given time. This is in contrast to the pipelined design which has multiple tasks operating at any given instance in time. Perhaps the most common understanding of "pipelining" is the 5-stage microprocessor pipeline. One can think of HLS pipelining in a similar way except that HLS performs the pipelining in a custom manner. Thus the Vivado® HLS tool decides on the number of pipeline stages, the initiation interval (the time between successive data provided to the pipeline – similar to the task interval), and other constraints.

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 1.7 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.

Figure 1.8 shows one potential design – a "one tap per clock" architecture, consisting of a single multiplier and single adder. 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 it will output the result of that filtered sample after 4 cycles. The implementation in Figure 1.9 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. That is, it can start a new task, which in this case means accept a new value to filter, every cycle. Other implementations are also possible, such as architectures with "two taps per clock" or "two

```
#define NUM_TAPS 4

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

  int i;
  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 = result;
}
```

Figure 1.7: Code for a four tap FIR filter.

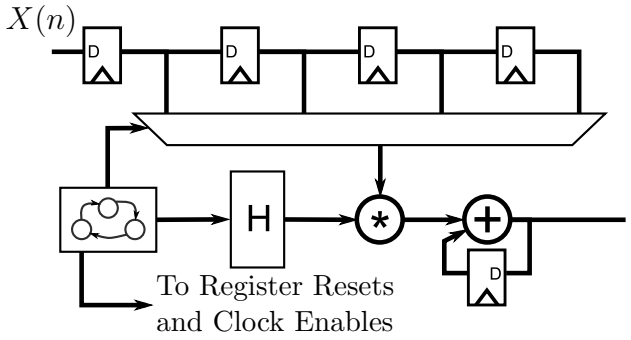


Figure 1.8: A "one tap per clock" architecture for an FIR filter. This architecture can be implemented from sequential implementation of the code in Figure 1.7.

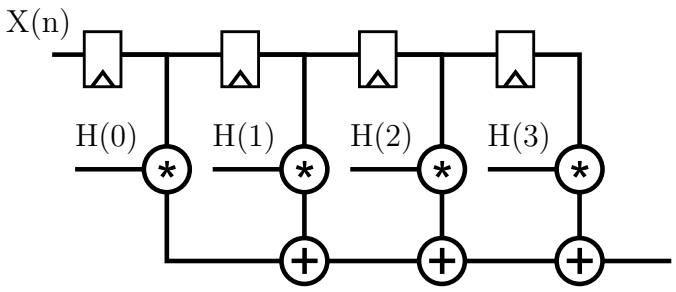


Figure 1.9: A "one sample per clock" architecture for an FIR filter. This architecture can be implemented from the code in Figure 1.7 using function pipeline.

samples per clock", which might be more appropriate for a particular application.

The "one tap per clock" architecture in Figure 1.8 is often called a *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. A sequential architecture is often appropriate for code that executes relatively rarely, where resource sharing needs to be maximized in order to make a design that is as "small" or uses few FPGA resources. One disadvantage to a sequential architecture is that analyzing and understanding data rates is often much harder. The control logic dictates the number of cycles for the task interval and task latencies. The control can be quite complex making it difficult to analyze. Or it may depend on dynamic information. For example, whether the circuit in Figure 1.8 actually consumes one sample per clock cycle or not depends on the structure of the generated control logic.

The "one sample per clock" architecture in Figure 1.9 is often called a *function pipeline*. A function pipeline 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, function pipelines are relatively simple to characterize, analyze, and understand. Function pipelines are often used for simple, high data rate designs where data is processed continuously. Function pipelines 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 pipeline architecture is that not all code can be effectively pipelined. We discuss these restrictions throughout the remainder of this book, e.g., in Chapter 2.

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?

By default, Vivado® HLS will generate a sequential architecture from C code. Since a sequential implementation can be generated from most programs, this makes it easy for users to get started with HLS. In order to generate a function pipelined architecture, Vivado® HLS requires a pipeline directive to be applied to the function. We discuss

multiple ways to optimize the FIR function in Chapter 2.

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 [38, 30]. 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 1.7), sequential loops (Figure 1.8). Recurrences can also appear in a sequential architecture and disappear when pipelining is applied (as in Figures 1.8 and 1.9). 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.

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 1.9 or 1.8 from the code in Figure 1.7, an alternative coding style such as that in Figure 1.10 would map to a much more 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.

```
#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 1.10: Alternative code implementing an FIR filter.

This would provide coax the HLS tool to produce an architecture that looks more like the pipelined one in Figure 1.9.

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 [33, 32, 12, 11, 29]. 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

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

	Chapter									
	2	3	4	5	6	7	8	9	10	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		

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. 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 partitioning. 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 a 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 steaming 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 (see Chapter 12). 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.

The FIR deals with a sampled 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 would sample the voltage across an antenna at a regular interval with an analog-to-digital converter. Or we would sample the voltage on a image sensor to get the light intensity for a pixel.

The format of the data in a sample changes depending upon the application. Digital communications often uses in-phase and quadrature (I/Q) values to represent a sample.Later in this chapter we will describe how to design a complex FIR filter to handle this complex numbers. 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.

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 described in more depth in subsequent chapters.

2.2 Background

A finite impulse response (FIR) filter derives its name from that fact that the system is fully determined by providing an input in the form of a impulse. The output of the filter is its *impulse response*; it contains the complete information about the filter. As the name implies, the output signal from the impulse response is finite in nature.

A FIR filter performs a convolution on the input signal with a fixed signal that is defined by its coefficients. There are two primary different types of coefficients – one for operating in the time domain and the other for the frequency domain. We will provide our explanation using the time domain as it is more intuitive and easier to understand.

Moving average filters work in the time domain; they attempt to smooth out the signal, e.g., to remove white noise. A moving average filter looks at an adjacent subset of the samples on the input signal and averages them together. That is, the filter sums all of the samples and then divides the total by the number of samples. This is mathematically represented as:

$$y[i] = \frac{1}{N+1} \sum_{j=0}^{N} x[i-j] \tag{2.1}$$

where x is the input signal, y is the output signal, and N is the number of points that we wish to average across.

Each sample in the output signal requires N additions for its computation. For example, when $N = 2$, each output sample $y[i]$ requires two additions in order to average the $N+1 = 3$ points around an output sample $y[i]$. To calculate $y[12]$, we perform the operation

$$y[12] = \frac{1}{3} \cdot (x[12] + x[11] + x[10]) \tag{2.2}$$

This is a causal system, 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 largely unimportant for the hardware implementation as it only requires some buffering and/or reindexing of the data.

The filter operation is a discrete convolution. The convolution kernel that we use in this example is $\cdots, 0, 0, 0, 0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0, 0, 0, 0, \cdots$. In general, a moving average filter can be described as the convolution of a rectangular pulse with an area of 1. It is also called a *box car filter* due to the shape of its convolution kernel.

As N gets larger, we average over a larger number of samples, and we correspondingly must perform more computations. N is the *order* of the filter. In general, a larger value of

N provides better characteristics of the filter. For a moving average filter, larger values of N corresponds to reducing the bandwidth of the output signal. In essence, it is acting like a low pass filter (though not a very optimum 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.

To make our discussion on FIR more general, we focus on the notion that the FIR filter implements a discrete convolution operation where the values of the convolution kernel define the type of filter. An FIR filter is weighted sum as described by the difference equation

$$y[i] = \sum_{j=0}^{N} h_j \cdot x[i-j] \tag{2.3}$$

In the case of the three point moving filter, the kernel $h = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. These kernel values are called the *coefficients* of the filter. These are also called the *taps* of the filter. The taps indicate the number of multiply accumulate operations. The number of taps is equal to one plus the order of the filter, i.e., $N + 1$.

The taps can be crafted so that they create all kinds of different filters: low pass, high pass, band pass, etc.. There is substantial amount of literature devoted to generating these filter coefficients in order to create a filter with the best response. The actual values of these coefficients are largely irrelevant, though many filter designs do transform the constant multiplications into shift and add operations. In that case, the values of the coefficients can drastically change the performance and area. 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`. It is an example of a streaming architecture since for each time that we execute the function, we provide one input sample and receive one output sample.

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.

```
#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.

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 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 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.

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
```

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 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 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 300 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 the important to

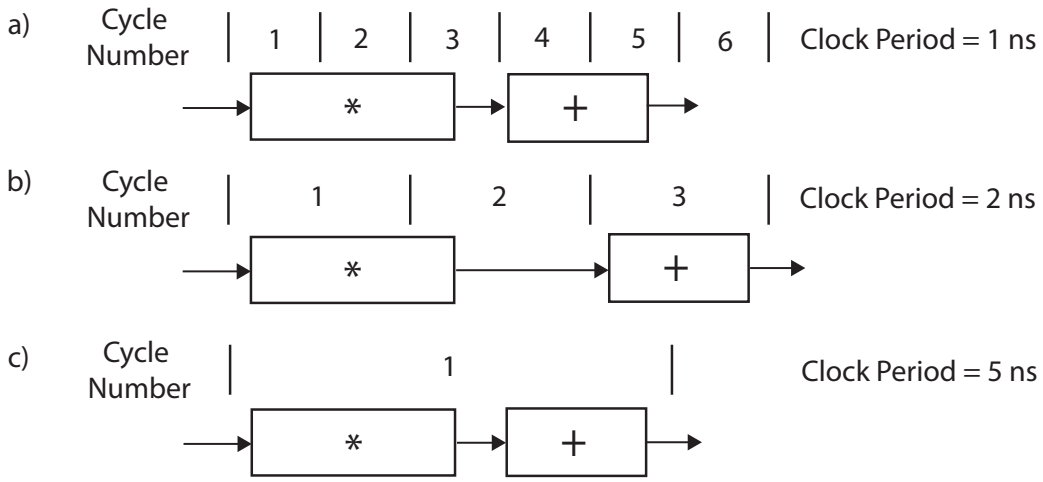


Figure 2.2: The performance of multiply accumulate operation changes depending upon the target clock period. Assume the `multiply` operation takes 3 ns and `add` operation takes 2 ns. Part a) has a clock period of 1 ns, and one MAC operation takes 5 cycles. Thus the performance is 200 million MACs/sec. 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.

fully understand the entire process of the Vivado® HLS tool. This is especially true since the tool is constantly undergoing optimizations 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.

However, to this point, clock period optimization is a bit of a black art. 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. This 100 MHz clock frequency is relatively easy to achieve, yet it is provides a good first order result. It is possible to create designs that run on faster clock rate; 200 MHz or even higher are possible. Achieving such high frequencies require very careful optimization. You can change this 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?

```
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.

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 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 loop partitioning. 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:
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.

`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 partitioning. What is the difference in performance? How do the number of resources change?

Loop partitioning 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...

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.

```
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.

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 pragma: `#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` pragma. To automatically perform the unrolling done manually in Figure 2.5, we should put the pragma: `#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` pragma. 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` pragma in conjunction with the `unroll` pragma? What happens if you do not use the `unroll` pragma?

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

```
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.

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 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 pragma, 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 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 you 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 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 is 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

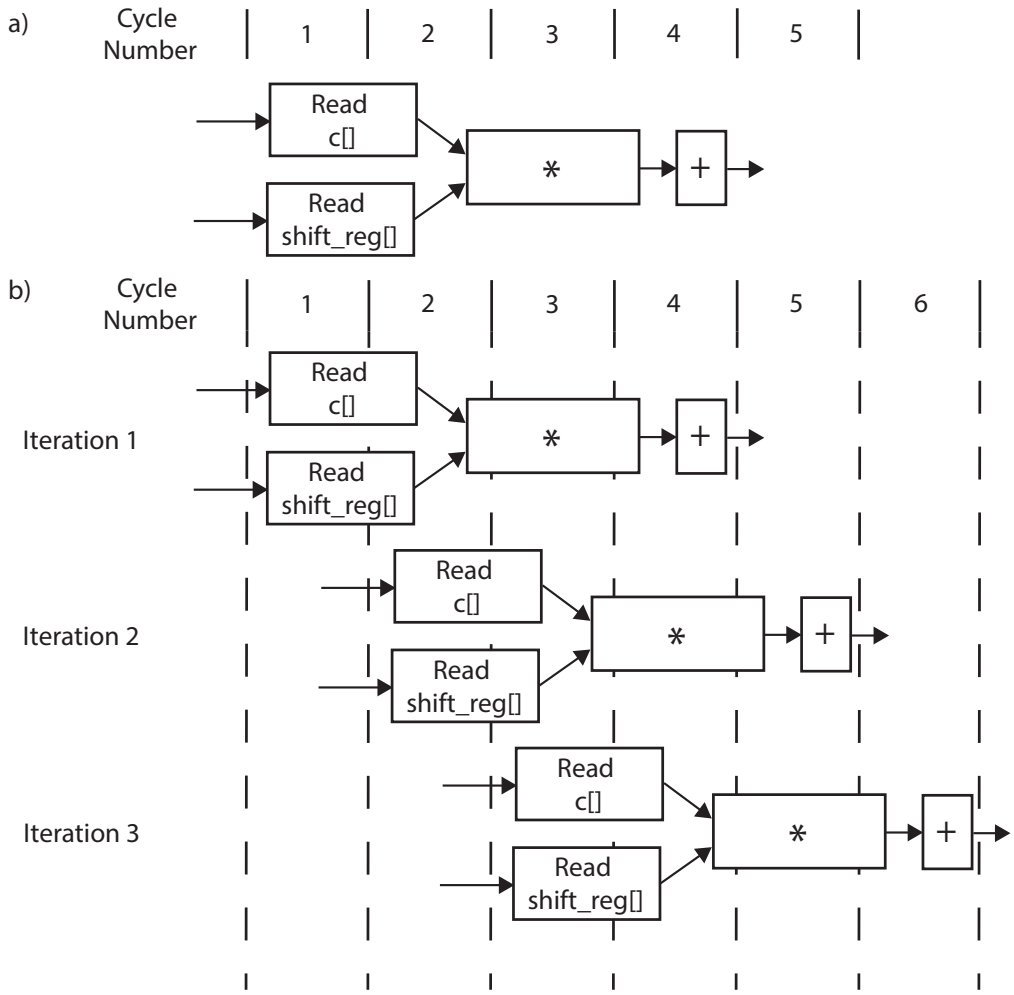


Figure 2.7: Part a) shows a schedule for the body of the `MAC for` loop. Part b) shows the schedule for three iterations of a pipelined version of the `MAC for` loop.

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 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 the 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 pragma `#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 as 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 not `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 pragma `#pragma HLS PIPELINE` after the loop header. The result of the synthesis is a loop initiation interval equal to 1 cycle. This means

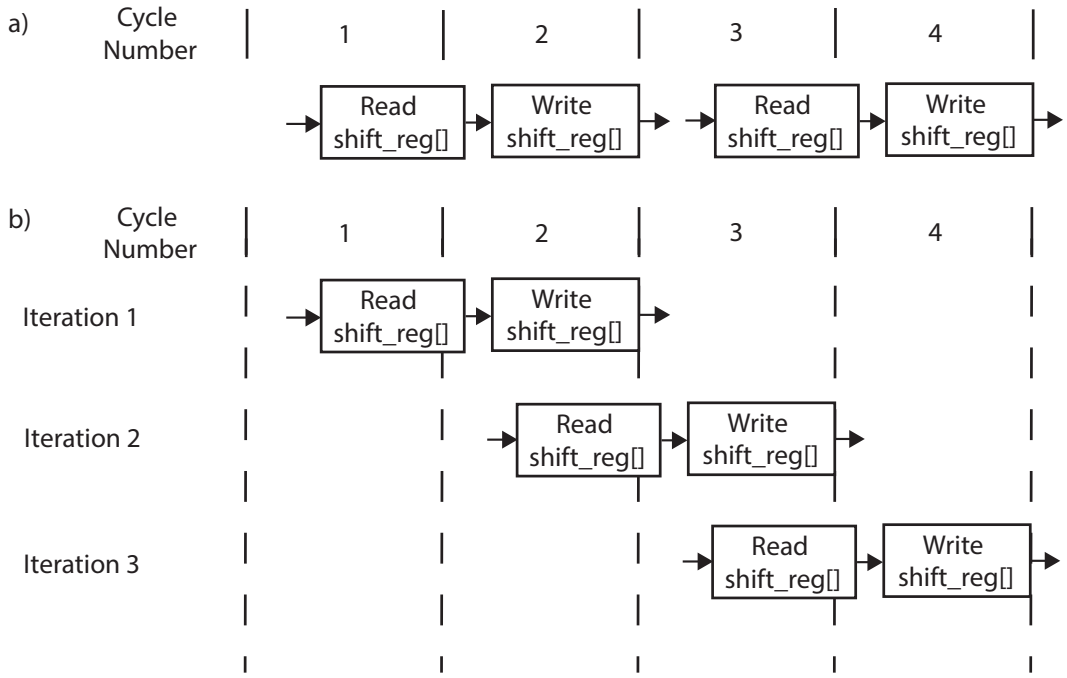


Figure 2.8: Part a) shows a schedule for two iterations of the body of the `TDL for` loop. Part b) shows the schedule for three iterations of a pipelined version of the `TDL for` loop with II=1.

that we can start a loop iteration every cycle.

By modifying the example slightly, we can demonstrate a scenario where the resource constraints to 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 pragma `#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. This is not possible with a single port RAM. Thus, we must have an II greater than 1, specifically II=2.

> 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 countless other options for specifying different resources that can be found in the Vivado® HLS documentation. 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

Bitwidth optimization is the process of assigning each of the variables to a data type. Up until this point, we have used `int` as the data type for all of the variables in our `fir` example. Vivado® HLS treats this as a 32 bit signed integer.

The C language allows us some flexibility in assigning data types. There are floating point data types such as `float` and `double`. And the C language has other integer data types, e.g., `char`, `short`, `long`, `long long`, etc. Not to mention `unsigned` versions of many of these data types. All of these data types are aligned on 8-bit boundaries.

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, 8 bit values versus a 16 bit values can make a substantial difference in memory usage (2× savings). The drawback is the range of values that you can represent. 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].

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 resources that fit the specified data types. For example, the statement `a = b * c` will change both in terms of the latency and the area depending upon the data type. If all of the variables are 32 bit, then the operation requires 5 cycles. When they are 8 bits, the operation only requires 1 cycle. Additionally, the multiply operation will be mapped to the exact number of bits in each case. Thus, the 8 bit implementation will be mapped to resources that multiply two 8 bit numbers, while the 32 bit implementation will require a larger multiplier.

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?

In many cases, we will need a data type that is not aligned on an 8-bit boundary. 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 we would potentially lose out on some performance and savings in resources. The Vivado® HLS tool has an *arbitrary precision data type* which allows for a signed or unsigned data type for 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 1024[1]. 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 include the file `ap_int.h`, i.e., add the code `#include "ap_int.h"` in your project when using C++. If you are using the C function, you should include the file `ap_cint.c`.

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 set last `ap_int<10>`.

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

[1] 1024 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.

We can also change 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 the least 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 least 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.

Before we do that, 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 $a = 2024$ 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 is also true for the signed case.

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 to 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. And we perform a number of estimation functions in order to decode the data, which functionally are not always 100% accurate. Thus, it may not be important that we insure that the `acc` variable has enough precision to avoid loss of data. However, in other applications (e.g., scientific computing) precision is important. And even if we are truncating the `acc` variable into a smaller bitwidth to store the value into `y`, not using a full precision for `acc` can lead to rounding errors in `y`. So what is the correct answer? Ultimately it is up to the tolerance of the application designer.

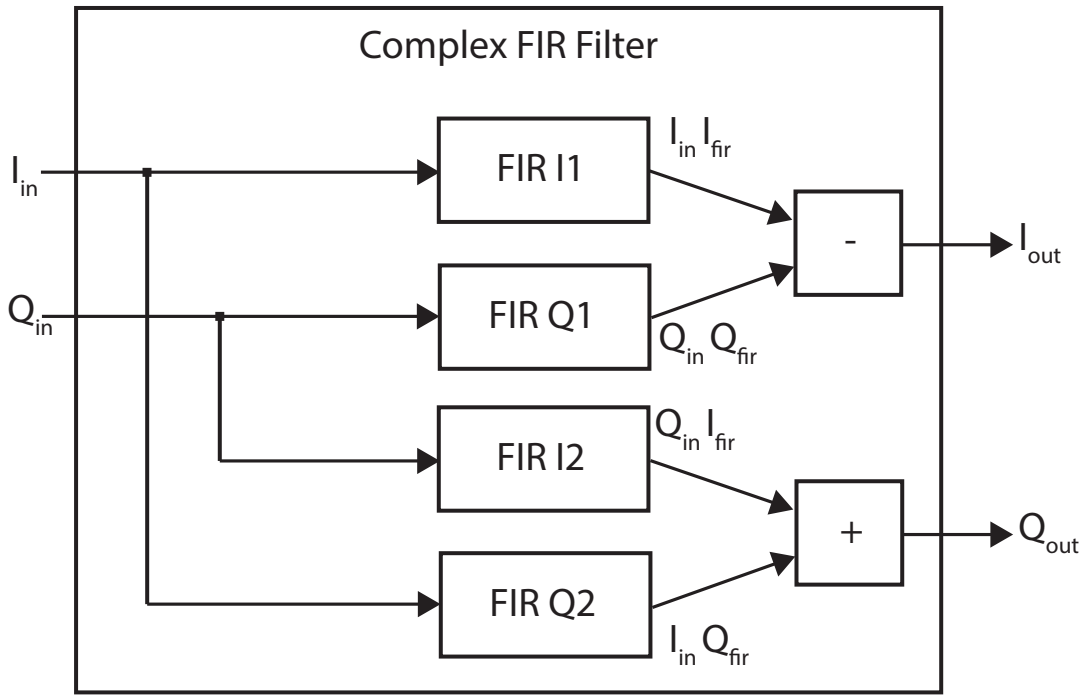


Figure 2.9: A complex FIR filter built from four real FIR filters. The input I and Q samples are feed into four different real FIR filters. The FIR filters hold the in-phase (FIR I) and quadrature (FIR Q) complex coefficients.

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 12 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.

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

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 $Q_{in}I_{fir} + I_{in}Q_{fir}$. 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

```
typedef int data_t;
void firI1(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 Iin, data_t Qin, data_t *Iout, data_t *Qout) {

    data_t IinIfir, QinQfir, QinIfir, IinQfir;

    firI1(&IinIfir, Iin);
    firQ1(&QinQfir, Qin);
    firI2(&QinIfir, Qin);
    firQ2(&IinQfir, Iin);

    *Iout = IinIfir + QinQfir;
    *Qout = QinIfir - IinQfir;
}
```

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

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 `firI1`, `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® HLS tool may choose to

```
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`.

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 13.

Chapter 3

CORDIC

3.1 Overview

The Coordinate Rotation DIgital Computer (CORDIC) method calculates trigonometric and hyberbolic functions. It employs a digit-by-digit algorithm that typically produces one digit of the function per iteration. CORDIC works by efficiently performing simple arithmetic using addition, subtraction, bit shifting and a table lookup, which are efficient to implement in FPGAs and more generally in hardware.

> The CORDIC method was developed by Jack Volder in the 1950's as a digital solution to replace an analog resolver for real-time navigation on a B-58 bomber. A resolver measures degrees of rotation. At that time hardware implementations of multiply operations were prohibitively expense and CPUs had very limited amount of state. Thus the algorithm needed to have low complexity and use simple operations. Over the years, it has been used in math co-processors [18], linear systems [1], radar signal processing [2], Fourier transforms [16], and many other digital signal processing algorithms. It is now commonly used in FPGA designs. Vivado® HLS uses a CORDIC core for calculating trigonometric functions, and it is a common element of modern FPGA IP core libraries.

In the remainder of this chapter, we discuss the CORDIC method and how to create an optimized hardware implementation using high-level synthesis. Our goal is to continue to increase the complexity of the types of hardware cores that we are developing as we progress through the book. The CORDIC method is an iterative algorithm; thus most of the computation is performed within a single `for` loop. The code itself is not all that complex. However, understanding the code such that we can create an optimal hardware implementation requires deep insight. And a good HLS designer must always understand the computation if she wishes to create the optimal restructured code. Thus, we spend the next two subsections describing the computational background for the CORDIC method.

The major optimization that we wish to highlight in this chapter is choosing the correct number representation for the variables. As we discuss later in the chapter, the designer must carefully tradeoff between the accuracy of the results and the performance and resource utilization of the design. Number representation is one big factor in this tradeoff – "larger"

numbers (i.e., those with more bits) generally provide more precision at the cost of increased area and delay. We provide a background on number representation and arbitrary data types in Chapter 3.6.5.

3.2 Background

Does there be a bit more of an introduction of modulation? This motivates the need for phasors, and the use of sines and cosines in DSP, cartesian vs polar planes, etc.? Currently it fairly quickly goes into complex numbers without much motivation as to why.

Need to emphasize that I/Q is just another way of looking at a signal; it is a transformation of the data into a different domain and one that has some useful properties. In this domain the signal is described as a magnitude and phase. Often data is modulated using magnitude and phase (AM radio using amplitude modulation and phase is a key component in more advanced systems) so this allows us to easily modulate or demodulate this into a system. The explanation at whiteboard.ping.se/SDR/IQ is pretty good.

It might be useful to have an aside that shows the relationships between I, Q and phase and magnitude. Some formulas calculating with I/Q Signals translating between polar and rectangular form etc.

move some of the later discussion from Chapter 12 here, and remove chapter 12?

Key idea: We very often want to view a signal in terms of frequency components. A frequency has an amplitude and phase. We often modulate data onto a signal using three components: amplitude, frequency and phase. Can talk about these three and give examples?

How does the "complex" come into this whole thing? Well through a magical formula called Euler's equation that relates sine and cosines to a complex exponential.

it is possible to encode information on a sine and cosine separately because they are orthogonal signals.

Negative frequencies: a "crazy" consequence of math? But actually show how there is more information when you consider the complex plane. A cosine can have both a positive and negative frequency. But this cannot be determined until you look at the orthogonal sine value (i.e., the "imaginary" portion). The negative frequency is a rotation in the "opposite" direction.

Before we dive into the details, recall some basics of complex numbers. When multiplying two complex numbers, their phases add and their amplitudes (magnitudes) multiply. Consider two complex numbers

$$\begin{aligned} C_1 &= I_1 + j \cdot Q_1 = A_1(\cos\phi_1 + j \cdot \sin\phi_1) \\ C_2 &= I_2 + j \cdot Q_2 = A_2(\cos\phi_2 + j \cdot \sin\phi_2) \end{aligned} \tag{3.1}$$

The first part of both equations is the complex number represented in Cartesian coordinates, where the real and imaginary parts are denoted as in-phase (I) and quadrature (Q) values.

The second part is in polar form.

Multiplying these two complex number yields

$$C_1 \cdot C_2 = (I_1 I_2 - Q_1 Q_2) + j \cdot (I_1 Q_2 + I_2 Q_1) = A_1 A_2(\cos(\phi_1 + \phi_2) + j \cdot \sin(\phi_1 + \phi_2)) \quad (3.2)$$

You can see from the polar representation that the multiplication of these two complex numbers results in a summation of their phases (angles) and a multiplication of their amplitudes. Therefore, if we wish to rotate the complex number C_1 by some angle, we should multiply it by a second complex number C_2 that has a phase of that desired rotation angle. Going forward, we denote the rotating complex number as C_R. If we wish to subtract an angle, we can use the complex conjugate of C_R, i.e., $C_R^* = I_R - j \cdot Q_R$. Therefore to subtract the angle of complex number C_R from C_1 we perform the complex multiplication

$$C_1 \cdot C_R^* = (I_1 I_R + Q_1 Q_R) + j \cdot (I_R Q_1 - I_1 Q_R) = A_1 A_R(\cos(\phi_1 - \phi_R) + j \cdot \sin(\phi_1 - \phi_R)) \quad (3.3)$$

Note the similarities between Equations 3.2 and 3.3. The major difference is that the signs are flipped in real and imaginary summations of the Cartesian multiply. This shows that the polar multiply performs addition and subtraction of the angles, respectively, as expected. An optimized CORDIC hardware design can take advantage of these similarities.

Complex numbers are a useful representation for digital signal processing because they provide a simple way to model the phase and amplitude of a sinusoidal signal. For more information about this, please see Chapter 12.1. Most often in digital communications you will deal with complex numbers since it is easy to generate an analog to digital convertor that provides in-phase (real) and quadrature (complex) data.

An *imaginary number* was originally derived such that the roots of any polynomial equation, which are defined the solutions to any polynomial equation

$$x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n \quad (3.4)$$

such that the polynomial equation is equal to zero, i.e.,

$$x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n = 0. \quad (3.5)$$

Consider the polynomial $x^2 - 1$, which we can factor as

$$x^2 - 1 = (x - 1)(x + 1). \quad (3.6)$$

This means the roots are $+1$ and -1. In general, if we can factor the polynomial in Equation 3.4 as

$$x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n = (x - r_1)(x - r_2) \cdots (x - r_n), \quad (3.7)$$

any value such that $x = r_i$ for any $i \in \{1, \cdots, n\}$ satisfies Equation 3.5.

Now consider the polynomial $x^2 + 1$. This has no solution if we only consider real numbers. However, imaginary numbers provide a solution where the root is $\sqrt{-1} = j$. And with the introduction of imaginary numbers we can insure that every polynomial of degree n can be factored into n polynomials of degree one as demonstrated in Equation 3.7.

Once again, the CORDIC algorithm calculates various trigonometric functions through a set of rotations. First, consider rotating a complex number by 90°. To do this we multiply by $C_R = 0 + j$ since the phase of $C_R = 90°$. By substituting this into Equation 3.2 and doing some simplification we get

$$C_1 \cdot C_R = -Q_1 + j \cdot I_1 = A_1(\cos(\phi_1 + 90°) + j \cdot \sin(\phi_1 + 90°)) \tag{3.8}$$

which is equivalent to negating the quadrature (imaginary) value of C_1 and swapping the in-phase (real) and quadrature values. Note that $A_R = 1$ and thus the amplitude complex number does not change with the rotation. A similar effect occurs when we want to rotate C_1 by $-90°$. In this case, we multiply by the complex number $C_R = 0 - j$ which gives us

$$C_1 \cdot C_R = Q_1 - j \cdot I_1 = A_1(\cos(\phi_1 - 90°) + j \cdot \sin(\phi_1 - 90°)) \tag{3.9}$$

which is equivalent to negating the in-phase value of C_1 and then swapping the in-phase and quadrature values. Thus, the computation for the rotation by $\pm 90°$ is simply a negation and swapping of values which is simple and efficient in an FPGA.

Of course, we also need to efficiently rotate by values other than 90°. To do this, we perform rotations by multiplying by a complex number in the form of

$$C_R = 1 + j \cdot 2^{-L} \tag{3.10}$$

where $L = 0, 1, 2, 3, ...$ and thus the quadrature value $Q_R = 1, 0.5, 0.25, 0.125, ...$. This probably seems like an odd choice but it will allow us to efficiently perform the complex multiplication by changing the multiplication into shifts which are essentially free in an FPGA. By substituting Equation 3.10 into Equation 3.2 we get

$$C_1 \cdot C_R = (I_1 - Q_1 \cdot 2^{-L}) + j \cdot (I_1 \cdot 2^{-L} + Q_1) = (I_1 - Q_1 >> L) + j \cdot (I_1 >> L + Q_1) \tag{3.11}$$

Note that calculating both the in-phase and quadrature values of the is done by a shift and add operation. The rotation of the negative angle, which is done using the conjugate of C_R, provides a similarly efficient calculation with shifts and adds

$$C_1 \cdot C_R^* = (I_1 + Q_1 \cdot 2^{-L}) + j \cdot (Q_1 - I_1 \cdot 2^{-L}) = (I_1 + Q_1 >> L) + j \cdot (Q_1 - I_1 >> L) \tag{3.12}$$

The next obvious question is what angle are we rotating by? And what about the amplitude of the resulting complex number after the rotation is performed? The phase of a complex number is calculated as

$$\arctan(\frac{Q}{I}) \tag{3.13}$$

and the amplitude as

$$\sqrt{I^2 + Q^2} \tag{3.14}$$

Thus the phase and amplitude of C_R varies at shown in Table 3.1. The CORDIC gain is the running change in amplitude of the a complex number (C_1) who has gone through the sequential series of rotations of C_R from $L = 0$ to the current value. Thus, the CORDIC gain when $L = 0$ is simply the amplitude of C_R when $L = 0$, i.e., $|1 + j| = 1.41421$. The CORDIC gain when $L = 1$ assumes that we have rotated by both $1 + j$, i.e., C_R when $L = 0$ and $1 + j \cdot 0.5$, i.e., C_R when $L = 1$. Thus the CORDIC gain is the amplitudes of these two C_R numbers multiplied together, $1.41421 * 1.11803 = 1.58114$. You can see that as L increases, the quadrature portion of C_R starts to approach 0. And as the angle gets smaller, the amplitude approaches 1, and the CORDIC gain stabilizes to approximately 1.647.

Table 3.1: The phase, amplitude, and CORDIC gain of the rotating complex number C_2 as L varies.

L	2^{-L}	C_R	C_R Phase (°)	$\lvert C_R \rvert$	CORDIC Gain
0	1.0	$1 + j$	45.000	1.41421	1.41421
1	0.5	$1 + j \cdot 0.5$	26.565	1.11803	1.58114
2	0.25	$1 + j \cdot 0.25$	14.036	1.03078	1.62980
3	0.125	$1 + j \cdot 0.125$	7.125	1.00778	1.64248
4	0.0625	$1 + j \cdot 0.0625$	3.576	1.00195	1.64569
5	0.03125	$1 + j \cdot 0.03125$	1.790	1.00049	1.64649
6	0.015625	$1 + j \cdot 0.015625$	0.895	1.00012	1.64669

3.3 Calculating Sine and Cosine

Now consider the calculation of a sine or cosine value given some angle ϕ. In order to do this using CORDIC, we perform a series of rotations until we are approximately at given angle. Then we can simply read the in-phase and quadrature values to get the values $\cos(\phi)$ and $\sin(\phi)$, respectively. This assumes that the amplitude of the final complex number is equal to 1, which as you will see is not too difficult to achieve.

Let us illustrate this with an example: calculating $\cos(60°)$ and $\sin(60°)$, which is depicted graphically in Figure 3.1. Here we perform five rotations in order to give a final complex number with approximately a 60° angle. We start with the imaginary number with a 0° angle, i.e., one with a quadrature value of 0. Since we want to get to 60°, we rotate it by the imaginary number $C_R = 1 + j$ which corresponds to the C_R when $L = 0$ (see Table 3.1). The resulting vector has a 45° angle; also note that its amplitude is longer by approximately 1.414. As we wish to get to a 60° angle, we rotate again by C_R with $L = 1$. This multiplication results a complex number with an angle of $45° + 26.565° = 71.565°$ and an amplitude which is 1.118 times larger for a total amplitude $1.414 \times 1.118 = 1.581$ (the CORDIC gain) larger than the amplitude of the first complex number. Now the angle is larger than our 60° target, so we rotate by a negative angle using the complex conjugate of C_R with $L = 2$ resulting in a complex number with a 57.529° angle and amplitude 1.630 larger than the initial complex number. This process continues by rotating by C_R with incrementally larger L values, resulting in smaller and smaller rotations that will eventually approximate the desired angle. Also, note that the CORDIC gain begins to stabilize.

After we perform a sufficient number of rotations, which is a function of the desired accuracy, we get a complex number with a phase close to the desired input angle. The in-phase and quadrature values of that complex number correspond to approximately $A_R \cos(60°)$ and $A_R \sin(60°)$, which is exactly what we wish to calculate assuming $A_R = 1$. Since we typically know a priori the number of rotations that we will perform, we can insure that $A_R = 1$ by setting the amplitude of the initial complex number to the reciprocal of the CORDIC gain. In the case of our example, assuming that we perform five rotations as shown in Figure 3.1, this value is $1.64649^{-1} = 0.60735$ (the reciprocal of the CORDIC gain when $L = 5$, see Table

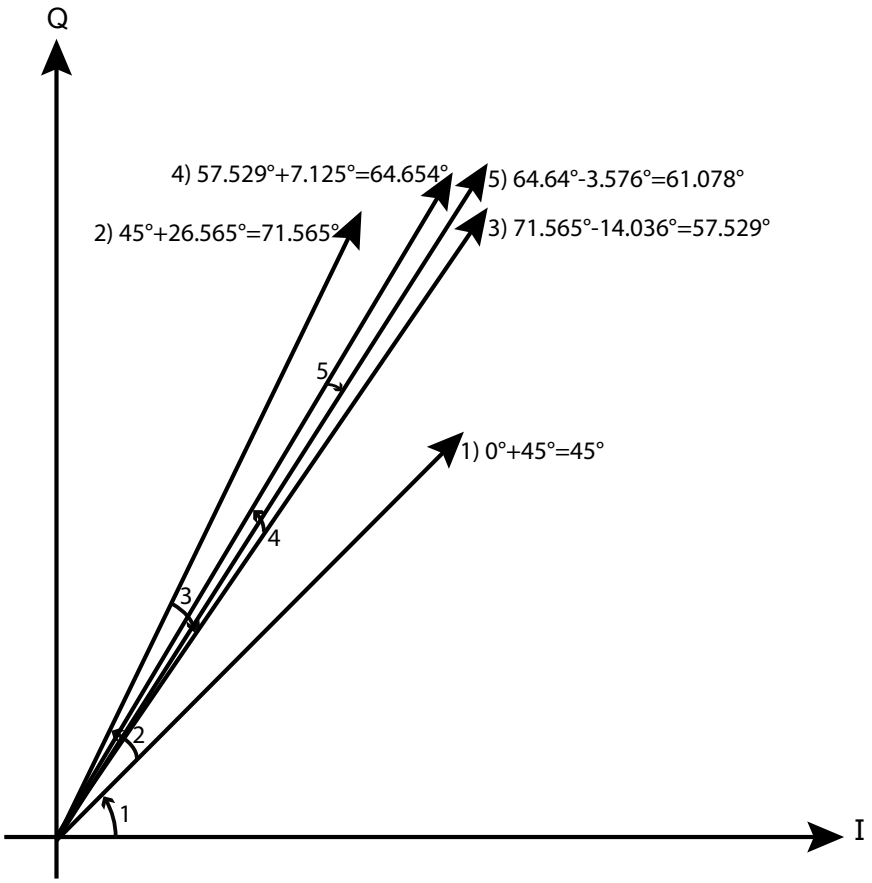


Figure 3.1: Calculating cos(60°) and sin(60°) using the CORDIC algorithm. Five rotations are performed using C_R with incrementally larger L values (0,1,2,3,4). The result is a complex number with an angle of 61.078°. The in-phase and quadrature values of that final complex number give the approximate desired cosine and sine values.

3.1). Setting the amplitude of the initial vector is simple since we start at a complex number with 0° angle, i.e., $C = 0.60735 + j \cdot 0$.

Figure 3.2 provides code that implements sine and cosine calculation using the CORDIC algorithm. It takes as input a target angle, and outputs the sine and cosine values corresponding to that angle. The code uses an array $cordic_phase$ to hold the angle for each rotation. This corresponds to the values in the column C_R Phase in Table 3.1. We assume that the "cordic.h" defines the different data types (i.e., `COS_SIN_TYPE` and `THETA_TYPE`) and sets `NUM_ITERATIONS` to some constant value. The data types can be changed to different fixed or floating point types, and `NUM_ITERATIONS` set depending on our desired accuracy, area, and throughput.

> Notice that the variable `sigma` is set as a two bit integer. Since we know that this will only take the value of ± 1 we can change its data type which will result in smaller area and better performance than if we were to use the current `int` data type. We discuss data types and how to specify them in Vivado® HLS shortly.

This code is close to a "software" version. It can be optimized in many ways to increase its performance and reduce its area. We will discuss how to optimize this code later in the chapter.

3.4 Calculating Amplitude and Phase

Calculating the amplitude and phase of a given complex number is another common CORDIC function. To do this, we rotate the given complex number to 0°. Once this rotation is complete, the magnitude is the in-phase component of the final rotated complex number. To determine the phase, we simply keep track of the cumulative angle of the rotations that we are performing. The angles of the rotating complex number (C_R for $L = 0, 1, 2, 3, \dots$) are known. Therefore, we simply need to perform the appropriate addition or subtraction of these phases depending on the direction of rotation.

The algorithm is similar to that of calculating the sine and cosine of a given angle. We perform a set of rotations by C_R with increasing values of L which results in a final complex number that resides on the positive in-phase axis (i.e., an angle of 0°). This can be done using positive or negative rotations by C_R which is predicated on the quadrature value of the complex number whose amplitude and phase we wish to determine.

The first step of the algorithm performs a rotation to get the initial complex number into either Quadrant I or IV. This rotates the number by $\pm 90°$ depending on the sign of the quadrature component of the initial complex number. If the quadrature value is positive, we know that we are in either Quadrant I or II. A rotation by $-90°$ will put us into Quadrant IV or I, respectively. Once we are in either of those quadrants, we can guarantee that we will be able to asymptotically approach the target 0° angle. If we are in Quadrant III or IV, the value of the quadrature component of the initial complex number will be negative. And a rotation by 90° will put us into Quadrant IV or I, respectively. Recall that a $\pm 90°$ rotation is done by negating either the in-phase or quadrature component of the complex

```
// The file cordic.h holds definitions for the data types and constant values
#include "cordic.h"

// The cordic_phase array holds the angle for the current rotation
THETA_TYPE cordic_phase[NUM_ITERATIONS] = {45, 26.565, 14.036, 7.125,
                                           3.576, 1.790, 0.895, ...};

void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c)
{
  // Set the initial vector that we will rotate
  // current_cos = I; current_sin = Q
  COS_SIN_TYPE current_cos = 0.60735;
  COS_SIN_TYPE current_sin = 0.0;

  // Only takes values -1/1
  int sigma = 0;

  // Factor is the 2^(-L) value
  COS_SIN_TYPE factor = 1.0;

  // This loop iteratively rotates the initial vector to find the
  // sine and cosine values corresponding to the input theta angle
  for (int j = 0; j < NUM_ITERATIONS; j++)
  {
    // Determine if we are rotating by a positive or negative angle
    if (theta < 0)
    {
      sigma = -1;
    }
    else
    {
      sigma = 1;
    }

    // Save the current_cos, so that it can be used in the sine calculation
    COS_SIN_TYPE temp_cos = current_cos;

    // Perform the rotation
    current_cos = current_cos - current_sin * sigma * factor;
    current_sin = temp_cos * sigma * factor + current_sin;
```

```
    // Determine the new theta
    theta = theta - sigma * cordic_phase[j];

    // Calculate next 2^(-L) value
    factor = factor >> 1;
  }

  // Set the final sine and cosine values
  s = current_sin;
  c = current_cos;
}
```

Figure 3.2: CORDIC code implementing the sine and cosine of a given angle.

number and then swapping those values (see Equations 3.8 and 3.9). The concept of these $\pm 90°$ is shown in Figure 3.3.

There is an issue with the final magnitude of the rotated number. Its magnitude is not the same as the initial magnitude before the rotations by C_R. It differs by a multiple of the CORDIC gain. Or course, one could calculate the precise value of the initial complex number's magnitude by multiplying by the reciprocal of the appropriate CORDIC gain (approximately $1/1.647 = 0.607$)[1]. However, this defeats the purpose of having a CORDIC, which eliminates the need for costly multipliers. And unfortunately this multiplication cannot be performed trivially using shifts and adds. Fortunately, this factor is often not important. e.g., in amplitude shift keying used in modulation in wireless communications, you only need to know a relative magnitude. Or in other times, this amplitude gain can be compensated by other parts of the system.

3.5 Removing if/else Conditional

One effect code restructuring technique is to make use of the `?` ternary operator. The `?` operator performs a selection between two input operations based upon an input condition. Vivado® HLS will synthesize the `?` operator as a multiplexor. Thus, when you perform this code restructuring, you are explicitly telling the Vivado® HLS tool to instantiate the code in a certain manner. In this case, a multiplexor implementation is typically more efficient than the equivalent implementation using an `if/else` clause. **In most cases, this optimization is dubious and should be done by the tool automatically. I think the more important concept is to talk about function pipelining?**

Looking again at the code in Figure 3.2, we can replace the lines corresponding to the if/else `sigma` clause (these lines are replicated again in Figure 3.4 for convenience) with an equivalent implementation using the `?` ternary conditional operator. This is shown in the bottom part of Figure 3.4.

[1]Recall that the CORDIC gain is a function of the number of rotations as show in Table 3.1.

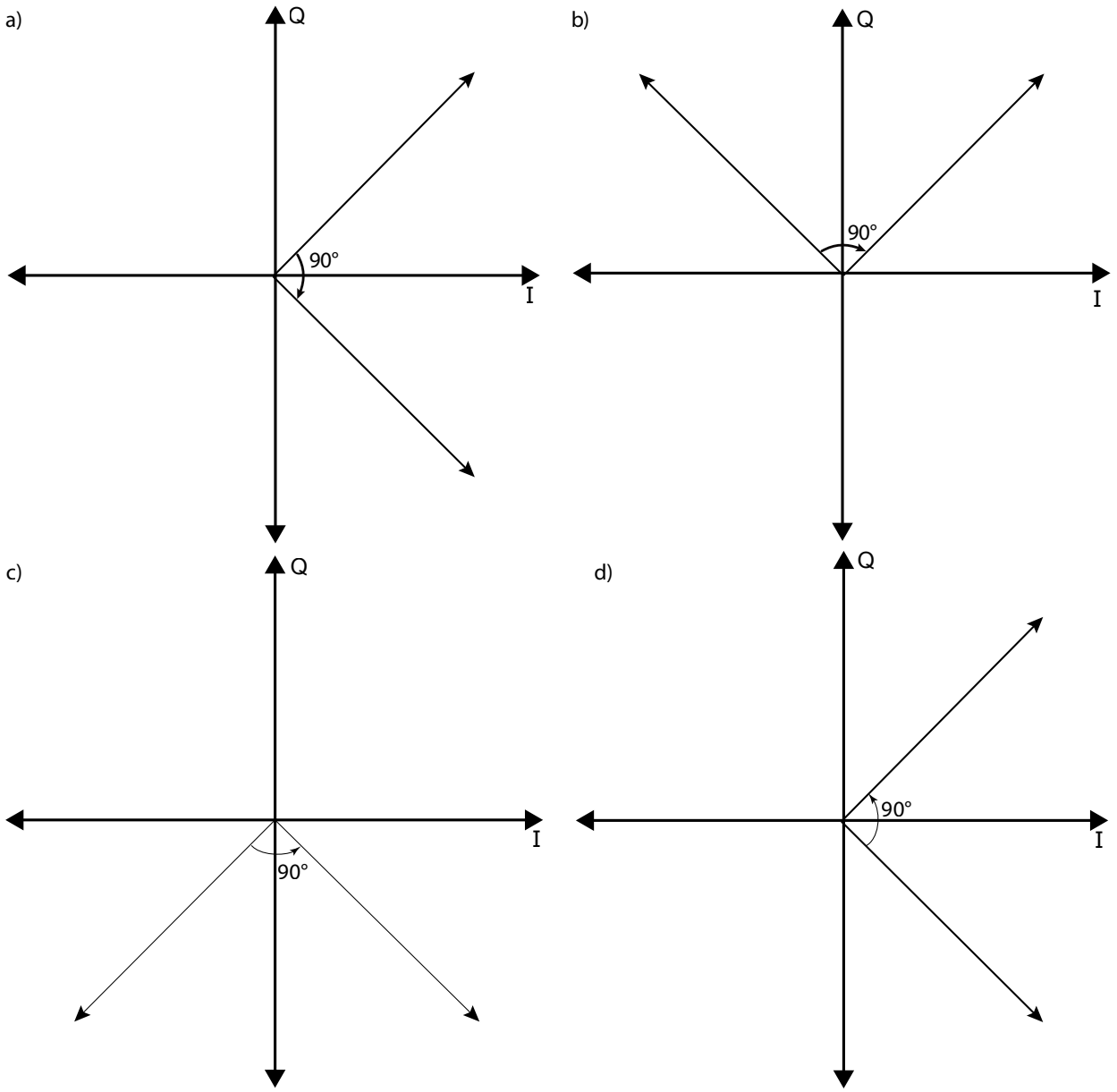


Figure 3.3: The first step in calculating the amplitude and phase of a given complex number is to perform a rotation by $\pm 90°$ in order to get the complex number into either Quadrant I or IV. Once it is in either of these two quadrants, subsequent rotations by C_R will allow the complex number to be rotated to a final angle of $0°$. At this point, the amplitude of the initial number is the in-phase value and the phase of the initial number is the summation of the angles of the rotated C_R numbers. Parts a) and b) show an example with the quadrature value is positive, which means that the complex number resides in either Quadrant I or II. Rotating by $-90°$ puts them into the appropriate quadrant. Parts c) and d) show a similar situation when the quadrature portion is negative. Here we wish to rotate by $90°$ to get the complex number into Quadrant I or IV.

```
// Determine if we are rotating by a positive or negative angle
if (theta < 0) {
    sigma = -1;
}
else {
    sigma = 1;
}

//A functionally equivalent version of the above using ? operator
sigma = theta < 0 ? -1 : 1;
```

Figure 3.4: Replacing the if/else conditional code from the CORDIC function with the functionally equivalent conditional ternary operator (`?`). Vivado® HLS synthesizes the ternary operation at a multiplexor.

How does code restructuring using the `?` ternary operator change the FPGA resource utilization? How does it affect the performance? Does it change the number of cycles? Will it every make the clock period different? Why?

3.6 Number Representation

The `cordic` function is currently using common types for the variables. For example, the `sigma` is defined as an `int` and other variables use custom data types (e.g., `THETA_TYPE` and `COS_SIN_TYPE`. In many cases, HLS tools are able to further optimize the representation of these values to simplify the generated hardware. For instance, in Figure 3.4, the variable `sigma` is restricted to be either `1` or `-1`. Even though the variable is declared as an `int` type of at least 32 bits, many fewer bits can be used to implement the variable without changing the behavior of the program. In other cases, particularly function inputs, memories, and variables that appear in recurrences, the representation cannot be automatically optimized. In these cases, modifying the code to use smaller datatypes, is a key optimization to avoid unnecessary resource usage.

Although reducing the size of variables is generally a good idea, this optimization can change the behavior of the program. A data type with fewer number of bits will not be able to express as much information as a data type with more bits and no finite binary representation can represent all real numbers with infinite accuracy. Fortunately, as designers we can pick numeric representations that are tuned to accuracy requirements of particular applications and tradeoff between acccuracy, resource usage, and performance.

Before discussing these number representation optimizations further using our `cordic` function, we first give a background on number representation. We provide the basics, as this is important in urderstand the data type specific representations provided by Vivado® HLS. The next section starts with a fundamental background on number representation, and

then proceeds to discuss the arbitrary precision variables available in Vivado® HLS.

3.6.1 Binary and Hexadecimal Numbers

Computers and FPGAs typically represent numbers using *binary representation*, which enables numbers to be efficiently represented using on-off signals called binary digits, or simply *bits*. Binary numbers work in most ways like normal decimal numbers, but can often be the cause of confusing errors if you are not familiar with how they work. This is particularly true in many embedded systems and FPGAs where minimizing the number of bits used to represent variables can greatly increase the overall performance or efficiency of a system. In this section, we will summarize binary arithmetic and the basic ways that computers represent numbers.

Many readers may already be familiar with these ideas. In that case, you may skim these sections or skip them entirely. We do suggest that look at Section 3.6.5 as this provides information specific to Vivado® HLS on how to declare arbitrary data types. This is a key idea for optimizing the number representation of the `cordic` function and any HLS code.

When we write a normal integer, such as 4062, what we really mean is implicitly $(4 * 1000) + (0 * 100) + (6 * 10) + (2 * 1) = 4062$, or written in columns:

10^3	10^2	10^1	10^0	unsigned
4	0	6	2	= 4062

A binary number is similar, except instead of using digits from zero to nine and powers of ten, we use numbers from zero to one and powers of 2:

2^3	2^2	2^1	2^0	unsigned
1	0	1	1	= 11

since $(1 * 8) + (0 * 4) + (1 * 2) + (1 * 1) = 11$. To avoid ambiguity, binary numbers are often prefixed with "0b". This makes it obvious that `0b1011` is the number decimal 11 and not the number 1011. The bit associated with the highest power of two is the *most significant bit*, and the bit associated with the lowest power of two is the *least significant bit*.

Hexadecimal numbers use the digits representing numbers from zero to 15 and powers of 16:

16^3	16^2	16^1	16^0	unsigned
8	0	3	15	= 32831

In order to avoid ambiguity, the digits from 10 to 15 are represented by the letters "A" through "F", and hexadecimal numbers are prefixed with "0x". So the number above would normally be written in C code as `0x803F`.

Note that binary representation can also represent fractional numbers, usually called *fixed-point* numbers, by simply extending the pattern to include negative exponents, so that "0b1011.01" is equivalent to:

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	unsigned
1	0	1	1	0	1	=11.25

since $8 + 2 + 1 + \frac{1}{4} = 11.25$. Unfortunately, the C standard doesn't provide a way of specifying constants in binary representation, although gcc and many other compilers allow integer constants (without a decimal point) to be specified with the "0b" prefix. The C99 standard does provide a way to describe floating-point constants with hexadecimal digits and a decimal exponent, however. Note that the decimal exponent is required, even if it is zero.

```
float p1 = 0xB.4p0; // Initialize p1 to "11.25"
float p2 = 0xB4p-4; // Initialize p2 to "11.25"
```

Notice that in general, it is only necessary to write non-zero digits and any digits not shown can be assumed to be zero without changing the represented value of an unsigned number. As a result, it is easy to represent the same value with more digits: simply add as many zero digits as necessary. This process is often called *zero-extension*. Note that each additional digit multiplies the amount of numbers that can be represented. Adding an additional bit to a binary number doubles the amount of numbers that can be represented, while an additional hexadecimal digit increases the amount of numbers by a factor of 16.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	unsigned
0	0	0	0	1	0	1	1	0	1	0	0	=11.25

Note that it is possible to have any number of bits in a binary number, not just 8, 16, or 32. SystemC [26], for instance, defines several template classes for handling arbitrary precision integers and fixed-point numbers (including sc_int<>, sc_uint<>, sc_bigint<>, sc_ubigint<>, sc_fixed<>, and sc_ufixed<>). These classes can be commonly used in HLS tools, although they were originally defined for system modeling and not necessarily synthesis. Vivado® HLS, for instance, includes similar template classes (ap_int<>, ap_uint<>, ap_fixed<>, and ap_ufixed<>) that typically work better than the SystemC template classes, both in simulation and synthesis.

Arbitrary precision numbers are even well defined (although not terribly useful) with zero digits. List all the numbers that are representable with zero digits.

3.6.2 Negative numbers

Negative numbers are slightly more complicated than positive numbers, partly because there are several common ways to do it. One simple way is represent negative numbers with a sign bit, often called *signed-magnitude* representation. This representation just includes an additional bit to the front of the number to indicate whether it is signed or not. One somewhat odd thing about signed-magnitude representation is that there is more than one way

to represent zero. This tends to make even apparently simple operations, like operator ==(), more complex to implement.

+/-	2^1	2^0	signed magnitude
0	1	1	$= 3$
0	1	0	$= 2$
0	0	1	$= 1$
0	0	0	$= 0$
1	0	0	$= -0$
1	0	1	$= -1$
1	1	0	$= -2$
1	1	1	$= -3$

Another way to represent negative numbers is with *biased* representation. This representation adds a constant offset (usually equal in magnitude to the value of the largest bit) to the value, which are otherwise treated as positive numbers:

2^2	2^1	2^0	biased
1	1	1	$= 3$
1	1	0	$= 2$
1	0	1	$= 1$
1	0	0	$= 0$
0	1	1	$= -1$
0	1	0	$= -2$
0	0	1	$= -3$
0	0	0	$= -4$

However by far the most common technique for implementing negative numbers is known as *two's complement*. In two's complement representation, the most significant bit represents the sign of the number (as in signed-magnitude representation), and *also* whether or not an offset is applied. One way of thinking about this situation is that the high order bit represents a negative contribution to the overall number.

-2^2	2^1	2^0	two's complement
0	1	1	$= 3$
0	1	0	$= 2$
0	0	1	$= 1$
0	0	0	$= 0$
1	1	1	$= -1$
1	1	0	$= -2$
1	0	1	$= -3$
1	0	0	$= -4$

-2^4	2^3	2^2	2^1	2^0	two's complement
0	0	0	1	1	$= 3$
0	0	0	1	0	$= 2$
0	0	0	0	1	$= 1$
0	0	0	0	0	$= 0$
1	1	1	1	1	$= -1$
1	1	1	1	0	$= -2$
1	1	1	0	1	$= -3$
1	1	1	0	0	$= -4$

One significant difference between unsigned numbers and two's complement numbers is that we need to know exactly how many bits are used to represent the number, since the most significant bit is treated differently than the remaining bits. Furthermore, when widening a signed two's complement number with more bits, the sign bit is replicated to all the new most significant bits. This process is normally called *sign-extension*. For the rest of the book, we will generally assume that all signed numbers are represented in two's complement unless otherwise mentioned.

What is the largest positive number representable with N bits in two's complement? What is the largest negative number?

Given a positive number x, how can you find the two's complement representation of $-x$? What is -0 in two's complement? if x is the largest negative number representable with N bits in two's complement, what is $-x$?

3.6.3 Overflow, Underflow, and Rounding

Overflow occurs when a number is larger than the largest number that can be represented in a given number of bits. Similarly, *underflow* occurs when a number is smaller than the smallest number that can be represented. One common way of handling overflow or underflow is to simply drop the most significant bits of the original number, often called *wrapping*.

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
0	0	1	0	1	1	0	1	0	0	$= 11.25$
	0	1	0	1	1	0	1	0	0	$= 11.25$
		1	0	1	1	0	1	0	0	$= 11.25$
			0	1	1	0	1	0	0	$= 3.25$

Handling overflow and underflow by wrapping two's complement numbers can even cause a positive number to become negative, or a negative number to become positive.

-2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	two's complement
1	0	1	1	0	1	0	0	$= -4.75$

	-2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	two's complement
	0	1	1	0	1	0	0	$= 3.25$

Similarly, when a number cannot be represented precisely in a given number of fractional bits, it is necessary apply *rounding*. Again, there are several common ways to round numbers. The simplest way is to just drop the extra fractional bits, which tends to result in numbers that are more negative. This method of rounding is often called *rounding down* or *rounding to negative infinity*, and corresponds to the floor() function.

0b0100.00	$= 4.0$			0b0100.0	$= 4.0$
0b0011.11	$= 3.75$			0b0011.1	$= 3.5$
0b0011.10	$= 3.5$			0b0011.1	$= 3.5$
0b0011.01	$= 3.25$			0b0011.0	$= 3.0$
0b0011.00	$= 3.0$	Round to		0b0011.0	$= 3.0$
0b1100.00	$= -4.0$	$\rightarrow$ Negative	$\rightarrow$	0b1100.0	$= -4.0$
0b1011.11	$= -4.25$	Infinity		0b1011.1	$= -4.5$
0b1011.10	$= -4.5$			0b1011.1	$= -4.5$
0b1011.01	$= -4.75$			0b1011.0	$= -5.0$
0b1011.00	$= -5.0$			0b1011.0	$= -5.0$

It is also possible to handle rounding in other similar ways which force rounding to a more positive numbers (called *rounding up* or *rounding to positive infinity* and corresponding to the ceil() function), to smaller absolute values (called *rounding to zero* and corresponding to the trunc() function), or to larger absolute values (called *rounding away from zero* or *rounding to infinity* and corresponding to the round() function)). None of these operations always minimizes the error caused by rounding, however. A better approach is called *rounding to nearest even*, *convergent rounding*, or *banker's rounding* and is implemented in the lrint() function. As you might expect, this approach to rounding always picks the nearest representable number. In addition, If there are two numbers equally distant, then the *even* one is always picked. This approach is the default handling of rounding with IEEE floating point, as it not only minimizes rounding errors but also ensures that the rounding error tends to cancel out when computing sums of random numbers.

0b0100.00	$= 4.0$			0b0100.0 $= 4.0$
0b0011.11	$= 3.75$			0b0100.0 $= 4.0$
0b0011.10	$= 3.5$			0b0011.1 $= 3.5$
0b0011.01	$= 3.25$			0b0011.0 $= 3.0$
0b0011.00	$= 3.0$	$\rightarrow$	Round to Nearest Even	$\rightarrow$ 0b0011.0 $= 3.0$
0b1100.00	$= -4.0$			0b1100.0 $= -4.0$
0b1011.11	$= -4.25$			0b1100.0 $= -4.0$
0b1011.10	$= -4.5$			0b1011.1 $= -4.5$
0b1011.01	$= -4.75$			0b1011.0 $= -5.0$
0b1011.00	$= -5.0$			0b1011.0 $= -5.0$

3.6.4 Binary arithmetic

Binary addition is very similar to decimal addition, simply align the binary points and add digits, taking care to correctly handle bits carried from one column to the next. Note that the result of adding or subtracting two N-bit numbers generally takes N+1 bits to represent correctly without overflow. The added bit is always an additional most significant bit for fractional numbers

	2^3	2^2	2^1	2^0	unsigned
		0	1	1	$= 3$
+		0	1	1	$= 3$
=	0	1	1	0	$= 6$

	2^3	2^2	2^1	2^0	2^{-1}	unsigned
		1	1	1	1	$= 7.5$
+		1	1	1	1	$= 7.5$
=	1	1	1	1	0	$= 15$

Note that since the result of subtraction can be negative, the 'extra bit' becomes the sign-bit of a two's complement number.

	2^3	2^2	2^1	2^0	unsigned
	0	0	1	1	$= 3$
-	0	0	1	1	$= 3$
=	0	0	0	0	$= 0$

	-2^4	2^3	2^2	2^1	2^0	unsigned
		0	0	1	1	$= 3$
-		1	1	1	1	$= 15$
=	1	0	1	0	0	$= -12$ (two's complement)

Multiplication for binary numbers also works similarly to familiar decimal multiplication. In general, multiplying 2 N-bit numbers results in a 2*N bit result.

	2^6	2^5	2^4	2^3	2^2	2^1	2^0	two's complement
				1	0	0	1	= 9
*				1	0	0	1	= 9
				1	0	0	1	= 9
			0	0	0	0		= 0
		0	0	0	0			= 0
+	1	0	0	1				= 72
	1	0	1	0	0	0	1	= 81

Operations on signed numbers are somewhat more complex because of the sign-bit handling and won't be covered in detail. However, the observations regarding the width of the result still applies: adding or subtracting two N-bit signed numbers results in an N+1-bit result, and Multiplying two N-bit signed numbers results in an 2*N-bit result.

What about division? Can the number of bits necessary to exactly represent the result a division operation of 2 N-bit numbers be computed?

3.6.5 Representing Arbitrary Precision Integers in C and C++

According to the C99 language standard, the precision of many standard types, such as **int** and **long** are implementation defined. Although many programs can be written with these types in a way that does not have implementation-defined behavior, many cannot. One small improvement is the `inttypes.h` header in C99, which defines the types int8_t,int16_t,int32_t, and int64_t representing signed numbers of a given width and the corresponding types uint8_t,uint16_t,uint32_t, and uint64_t representing unsigned numbers. Although these types are defined to have exactly the given bitwidths, they can still be somewhat awkward to use. For instance, even relatively simple programs like the code below can have unexpected behavior.

```
#include "inttypes.h"
uint16_t a =0x4000;
uint16_t b = 0x4000;
// Danger! p depends on sizeof(int)
uint32_t p = a*b;
```

Although the values of `a` and `b` can be represented in 16 bits and their product (`0x10000000`) can be represented exactly in 32 bits, the behavior of this code by the conversion rules in C99 is to first convert `a` and `b` to type **int**, compute an integer result, and then to extend the result to 32 bits. Although uncommon, it is correct for a C99 compiler to only have 16 bits of precision. Furthermore, the C99 standard only defines 4 bitwidths for integer numbers, while FPGA systems often use a wide variety of bitwidths for arithmetic. Also, printing these datatypes using `printf()` is awkward, requiring the use of additional macros to write portable code. The situation is even worse if we consider a fixed-point arithmetic example. In the code below, we consider `a` and `b` to be fixed point numbers, and perform normalization correctly to generate a result in the same format.

```
#include "inttypes.h"
// 4.0 represented with 12 fractional bits
uint16_t a =0x4000;
// 4.0 represented with 12 fractional bits.
uint16_t b = 0x4000;
// Danger! p depends on sizeof(int)
uint32_t p = (a*b) >> 12;
```

The correct code in both cases requires casting the input variables to the width of the result before multiplying.

```
#include "inttypes.h"
uint16_t a = 0x4000;
uint16_t b = 0x4000;
// p is assigned to 0x10000000
uint32_t p = (uint32_t) a*(uint32_t) b;
```

```
#include "inttypes.h"
// 4.0 represented with 12 fractional bits.
uint16_t a =0x4000;
// 4.0 represented with 12 fractional bits.
uint16_t b = 0x4000;
// p assigned to 16.0 represented with 12 fractional bits
uint32_t p = ( (uint32_t) a*(uint32_t) b ) >> 12;
```

When using integers to represent fixed-point numbers, it is very important to document the fixed point format used, so that normalization can be performed correctly after multiplication. Usually this is described using "Q" formats that give the number of fractional bits. For instance, "Q15" format uses 15 fractional bits and usually applies to 16 bit signed variables. Such a variable has values in the interval $[-1, 1)$. Similarly "Q31" format uses 31 fractional bits.

For these reasons, it's usually preferable to use C++ and the Vivado® HLS template classes ap_int<>, ap_uint<>, ap_fixed<>, and ap_ufixed<> to represent arbitrary precision numbers. The ap_int<> and ap_uint<> template classes require a single integer template parameter that defines their width. Arithmetic functions generally produce a result that is wide enough to contain a correct result, following the rules in section 3.6.4. Only if the result is assigned to a narrower bitwidth does overflow or underflow occur.

```
#include "ap_int.h"
ap_uint<15> a =0x4000;
ap_uint<15> b = 0x4000;
// p is assigned to 0x10000000.
ap_uint<30> p = a*b;
```

The ap_fixed<> and ap_ufixed<> template classes are similar, except that they require two integer template arguments that define the overall width (the total number of bits) and the number of integer bits.

```
#include "ap_fixed.h"
// 4.0 represented with 12 fractional bits.
ap_ufixed<15,12> a = 4.0;
// 4.0 represented with 12 fractional bits.
ap_ufixed<15,12> b = 4.0;
// p is assigned to 16.0 represented with 12 fractional bits
ap_ufixed<18,12> p = a*b;
```

Note that the ap_fixed<> and ap_ufixed<> template classes require the overall width of the number to be positive, but the number of integer bits can be arbitrary. In particular, the number of integer bits can be 0 (indicating a number that is purely fractional) or can be the same as the overall width (indicating a number that has no fractional part). However, the number of integer bits can also be negative or greater than the overall width! What do such formats describe? What are the largest and smallest numbers that can be represented by an ap_fixed<8,−3>? ap_fixed<8,12>?

floating point?

3.7 Further Optimizations

In this section, we provide some brief thoughts and suggestions on the best way to optimize the CORDIC function. We focus on how the different optimizations change the precision of the result while providing the ability tradeoff between throughput, precision, and area.

Ultimately, CORDIC produces an approximation. The error on that approximation generally decreases as the number of iterations increases. This corresponds to the number of times that we execute the `for` loop in the `cordic` function, which is set by `NUM_ITERATIONS`. Even if we perform a very large number of iterations, we may still have an approximation. One reason for this is that we may approach but never exactly match the desired target angle. We can, however, tune precision by choosing to perform greater or fewer iterations. All that needs to change in the algorithm is to modify the value of `NUM_ITERATIONS`. The choice of `NUM_ITERATIONS` depends on the number of digits of precision required by application using this CORDIC core.

How do the area, throughput, and precision of the sine and cosine results change as you vary the data type?

How does the constant `NUM_ITERATIONS` affect the area, throughput, and precision? How does this affect the initial values of `current_cos` and `current_sin`? Do you need to modify the array `cordic_phase`? Can you optimize the data types depending on the value of `NUM_ITERATIONS`?

The computations in the *for* loop occupy most of the overall time. How do you best perform code transforms and/or use pragmas to optimize it?

Setting the variable *sigma* can be efficient in hardware using a two input multiplexor. Can you transform the code so that the high level synthesis tool implements it in this manner?

The current code assumes that the given angle is between $\pm 90°$. Can you add code to allow it to handle any angle between $\pm 180°$?

3.8 Conclusion

In this chapter, we looked the Coordinate Rotation DIgital Computer (CORDIC) method for calculating trigonometric and hyberbolic functions based on vector rotations. We start with a background on the computation being performed by the CORDIC method. In particular, we focus on how to use the CORDIC method to calculate the sine and cosine values for a given angle. Additionally, we discuss how the same CORDIC method can be used to determine the amplitude and phase of a given complex number.

After this, we focus on the optimizations that can be done on the CORDIC method. Since it is an iterative method, there are fundamental tradeoffs between the number of iterations that are performed and the precision and accuracy of the resulting computation. We discuss how to reduce the precision/accuracy and get savings in FPGA resource usage and increases in performance.

We introduce the notion of using custom arbitrary data types for the variables in our `cordic` function. This provides another method to reduce the latency, increase the throughput, and minimize the area while changing the precision of the intermediate and final results. Vivado® HLS provides a method to specifically generate a large number of data types. We provide a background on number representation and introduce these custom data types.

In general, there is a complex relationship between precision, resource utilization, and performance. We touch on some of these tradeoffs, and provide some insights on how to best optimize the `cordic` function. We leave many of the optimization as well as the analysis of these tradeoffs, as an exercise to the reader. The CORDIC method is an integral part of the Phase Detector project described in Chapter 14 – a lab provided in the Appendix.

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 effectively 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 4.5 treat the DFT operation as a simplified matrix-vector multiplication. Then, Chapter 4.5 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 4.6.

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 FFT in the next chapter. That being said, if you are more interested in the HLS optimizations, you can skip to Chapter 4.5.

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

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

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

$$\begin{aligned} a_0 &= \tfrac{1}{\pi}\int_{-\pi}^{\pi} f(t)\,\mathrm{d}t \\ a_n &= \tfrac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(nt)\,\mathrm{d}t \\ b_n &= \tfrac{1}{\pi}\int_{-\pi}^{\pi} f(t)\sin(nt)\,\mathrm{d}t \end{aligned} \tag{4.2}$$

There are several things to note. First, the coefficients $a_0, a_1, a_2, \ldots, b_1, b_2, \ldots$ 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

$$t \equiv \frac{\pi t'}{L} \tag{4.3}$$

and

$$\mathrm{d}t = \frac{\pi \mathrm{d}t'}{L} \tag{4.4}$$

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

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

Solving for the a and b coefficients is similar:

$$\begin{aligned} a_0 &= \tfrac{1}{L}\int_{-L}^{L} f(t')\,\mathrm{d}t' \\ a_n &= \tfrac{1}{L}\int_{-L}^{L} f(t')\cos(\tfrac{n\pi t'}{L})\,\mathrm{d}t' \\ b_n &= \tfrac{1}{L}\int_{-L}^{L} f(t')\sin(\tfrac{n\pi t'}{L})\,\mathrm{d}t' \end{aligned} \tag{4.6}$$

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

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

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

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

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

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

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.

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

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 $2j$.

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

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*

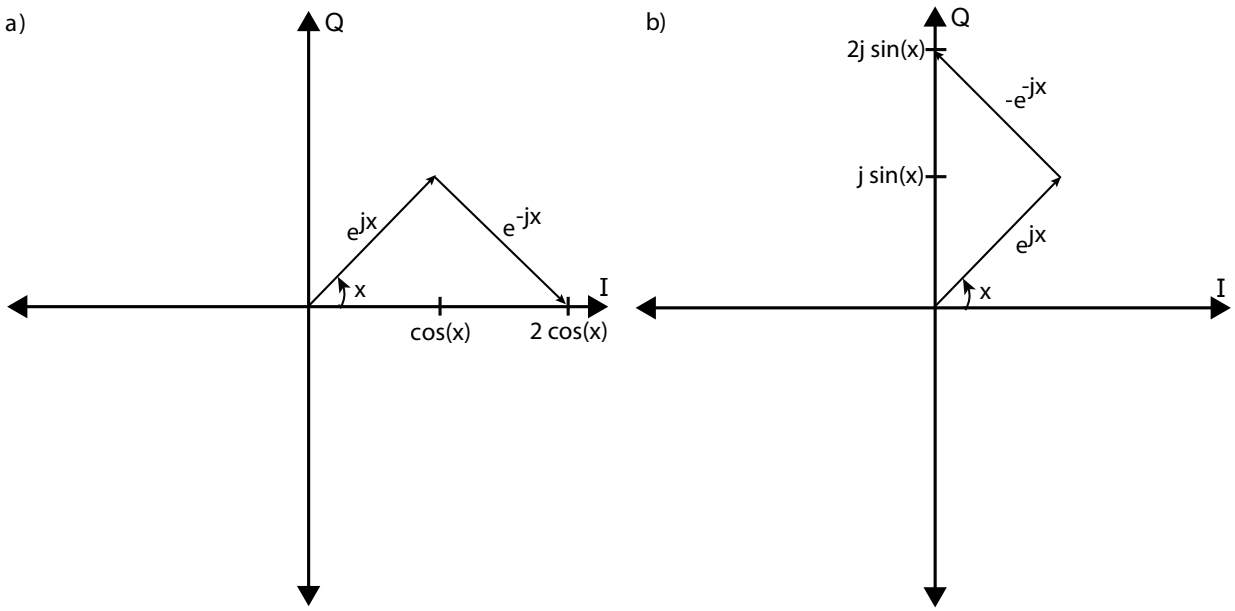


Figure 4.1: A visualization of the relationship between the cosine, sine, and the complex exponential. Part a) shows the sum of two complex vectors, e^{jx} and e^{-jx}. The result of this summation lands exactly on the real axis with the value $2\cos(x)$. Part b) shows a similar summation except this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

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.

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

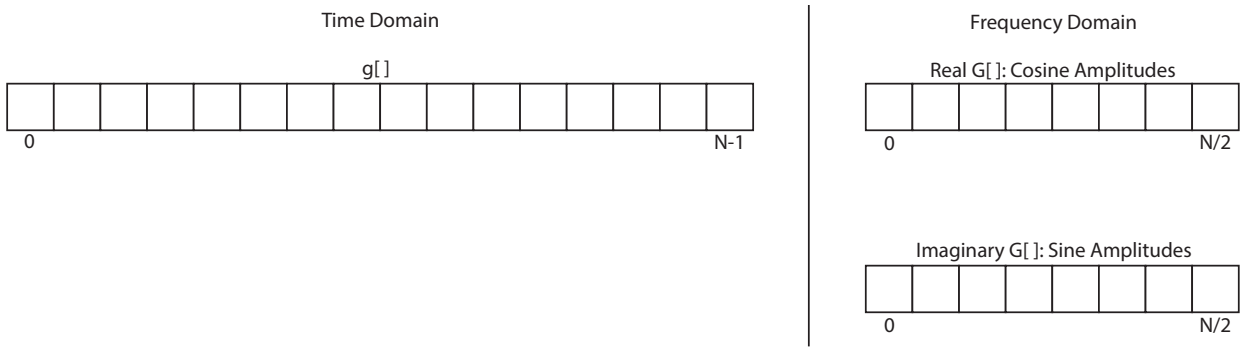


Figure 4.2: A real valued discrete function $g[]$ in the time domain with N points has a frequency domain representation with $N/2+1$ samples. Each of these frequency domain samples has one cosine and one sine amplitude value. Collectively these two amplitude values can be represented by a complex number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

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 discrete Fourier transform (DFT) can be determined through a $N \times N$ matrix multiplied by a vector of size N, $G = S \cdot g$ where

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

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

$$G[k] = \sum_{n=0}^{N-1} g[n] s^{kn} \text{ for } k = 0, \ldots, N-1 \tag{4.13}$$

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

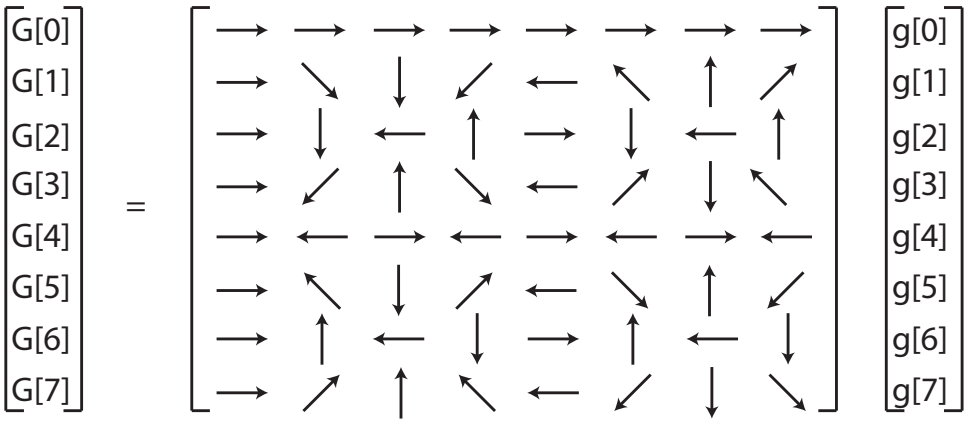


Figure 4.3: The elements of the S shown as a complex vectors.

phasor is offset by 45°. Performing eight 45° rotations does one full rotation around the unit circle. Row 2 is similar except is 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 there is symmetry around Row 4. The phasors in Rows 3 and 5 are complex conjugates of each other, i.e., $G[3] = G[5]^*$. Similarly, Rows 2 and 6 ($G[2] = G[6]^*$), and Rows 1 and 7 ($G[1] = G[7]^*$) 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.

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 `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

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE],
                   BaseType V_Out[SIZE]) {
  BaseType i, j;
  for (i = 0; i < SIZE; i++) {
    BaseType sum = 0;
    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.

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 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 largely sequential architecture for matrix-vector multiplication. Here the multiply and addition operations from the inner loop are instantiated as functional units. 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 happen if you synthesize the code from Figure 4.4. It does not consume a lot of area, but the task latency and task interval are relatively large.

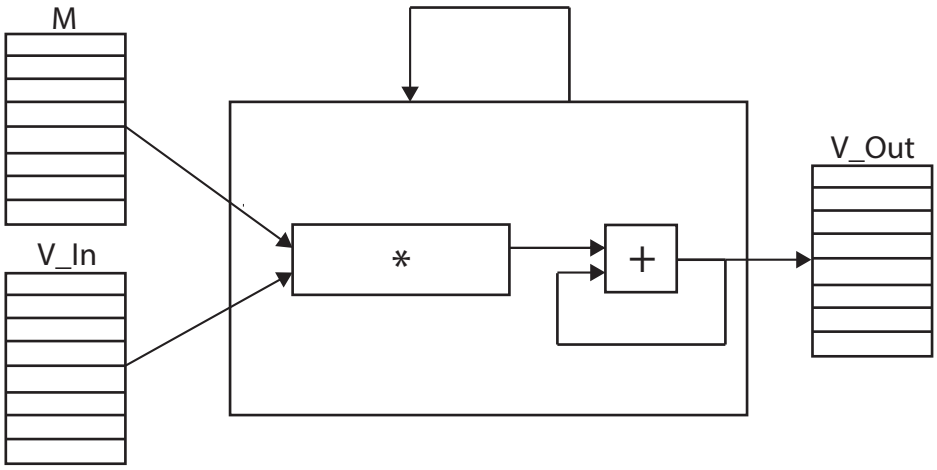


Figure 4.5: An architectural depiction of the sequential implementation of matrix-vector multiplication which corresponds synthesis from the code in Figure 4.4.

4.4 Loop Optimizations

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 largely independent across the iterations of the loop. The variable `sum`, which is keeping a running tally of the multiplications, is being reused in each iteration. This inner loop can be rewritten as shown Figure 4.6. As you see, the variable sum is unnecessary and completely eliminated.

> Loop unrolling can be done automatically by Vivado® HLS, e.g., by using `#pragma HLS unroll` or the equivalent directive.

It should be clear that this new expression which totally replaced 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 maximum task interval for the expression resulting from the unrolled inner loop, all eight of the multiplications should be executed in parallel. Assume that the multiplication takes 3 cycles and addition requires 1 cycle; therefore, 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, in total this unrolled inner loop requires 6 cycles. The resource usage for this architecture would be 8 multipliers and 7 adders. 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 multiplexor require the same amount of logic.

If we are not willing to use 8 multipliers, there is opportunity to make the design using fewer resources but with a larger number of cycles. For example, using 4 multipliers would

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE],
                   BaseType V_Out[SIZE]) {
  BaseType i, j;
  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.

require 6 cycles for the multiplication of the eight `V_In[j] * M[i][j]` operations, and an overall 9 cycles to finish the entire inner loop. You could even use fewer multipliers at the cost of taking more cycles to complete the inner loop.

Many functional units can be designed to handle data in a pipelined fashion. As an example consider a hardware instantiation of the multiplier. Assume as before that it takes 3 cycles to complete the operation. This means that after three cycles it can start another multiply operation. On the other hand, a pipelined multiplier can start executing another separate multiply operation even before it fully completes the previous multiply operation. In fact, it may be able to simultaneously be executing many multiply operations at the same time, e.g., the 3 cycle hardware multiplier could be simultaneously executing three multiply operations.

By taking advantage of this pipelined multiplier, we can reduce the latency of the unrolled inner loop without adding a substantial amount of additional hardware (multipliers). Figure 4.8 shows the unrolled inner loop using a three stage pipelined multiplier. Here the first three multiply operations (corresponding to the operations `V_In[0] * M[i][0]`, `V_In[1] * M[i][1]`, `V_In[2] * M[i][2]` , `V_In[3] * M[i][3]`) can use the same multiplier. Note that this multiplier can only start one multiply operation in any clock cycle, and the operations still requires 3 cycles to complete. Likewise the operations `V_In[4] * M[i][4]`, `V_In[5] * M[i][5]`, `V_In[6] * M[i][6]` , `V_In[7] * M[i][7]` can use the same multiplier. Thus only 2 multipliers are required.

Up until this point, we have assumed that the data in arrays (`V_In[]`, `M[][]` and `V_Out[]` are accessible at anytime. This may not be the case depending on the memory structure that is used to stored these arrays. For instance, using three single port BRAMs (one for each array) would restrict the access to one element per array per cycle. The "fully parallel" implementation in Figure 4.7 requires that we access all eight elements of `V_In[]` in the first cycle along with 8 elements of `M[][]`. However, the single port BRAM only allows one element to be accessed per cycle. We could increase the number of ports that we used for the arrays though this comes with an inherent cost. A port is essentially a multiplexor and

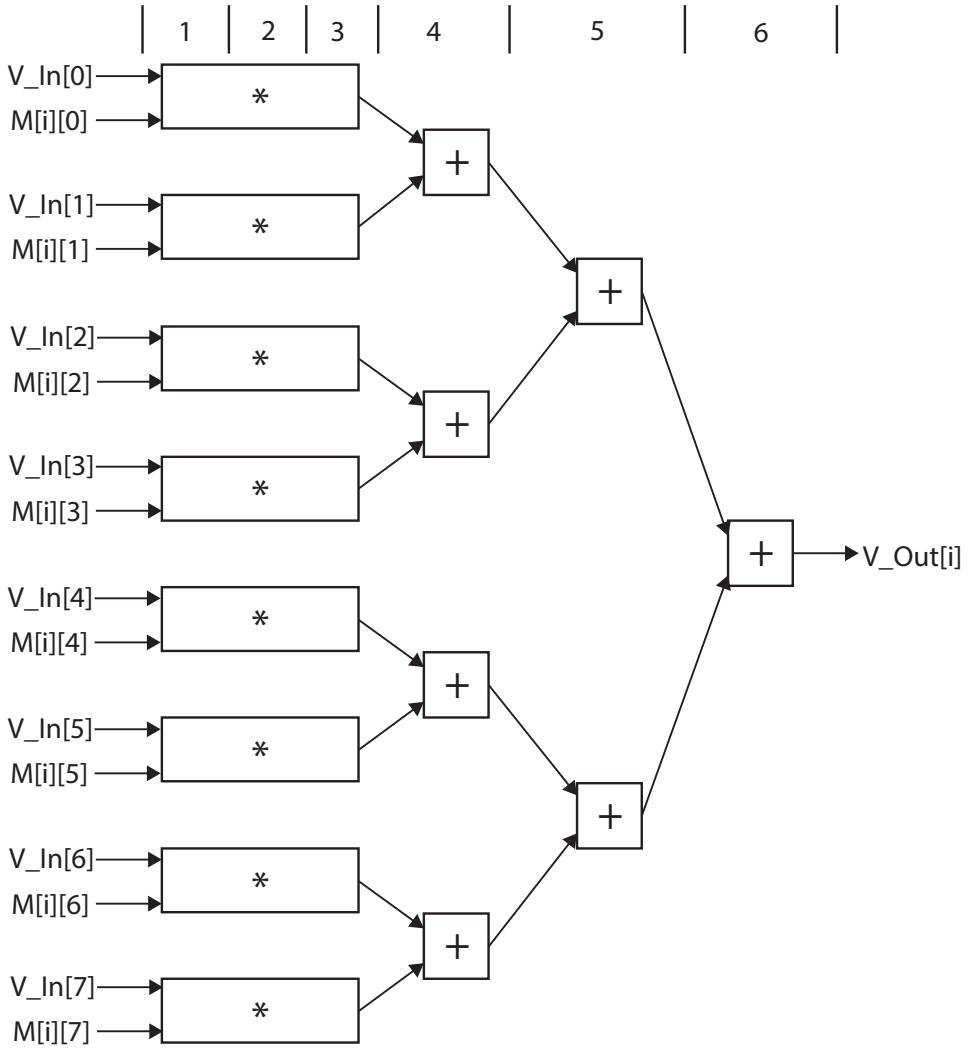


Figure 4.7: A data flow graph of the expression resulting from the unrolled inner loop from Figure 4.6.

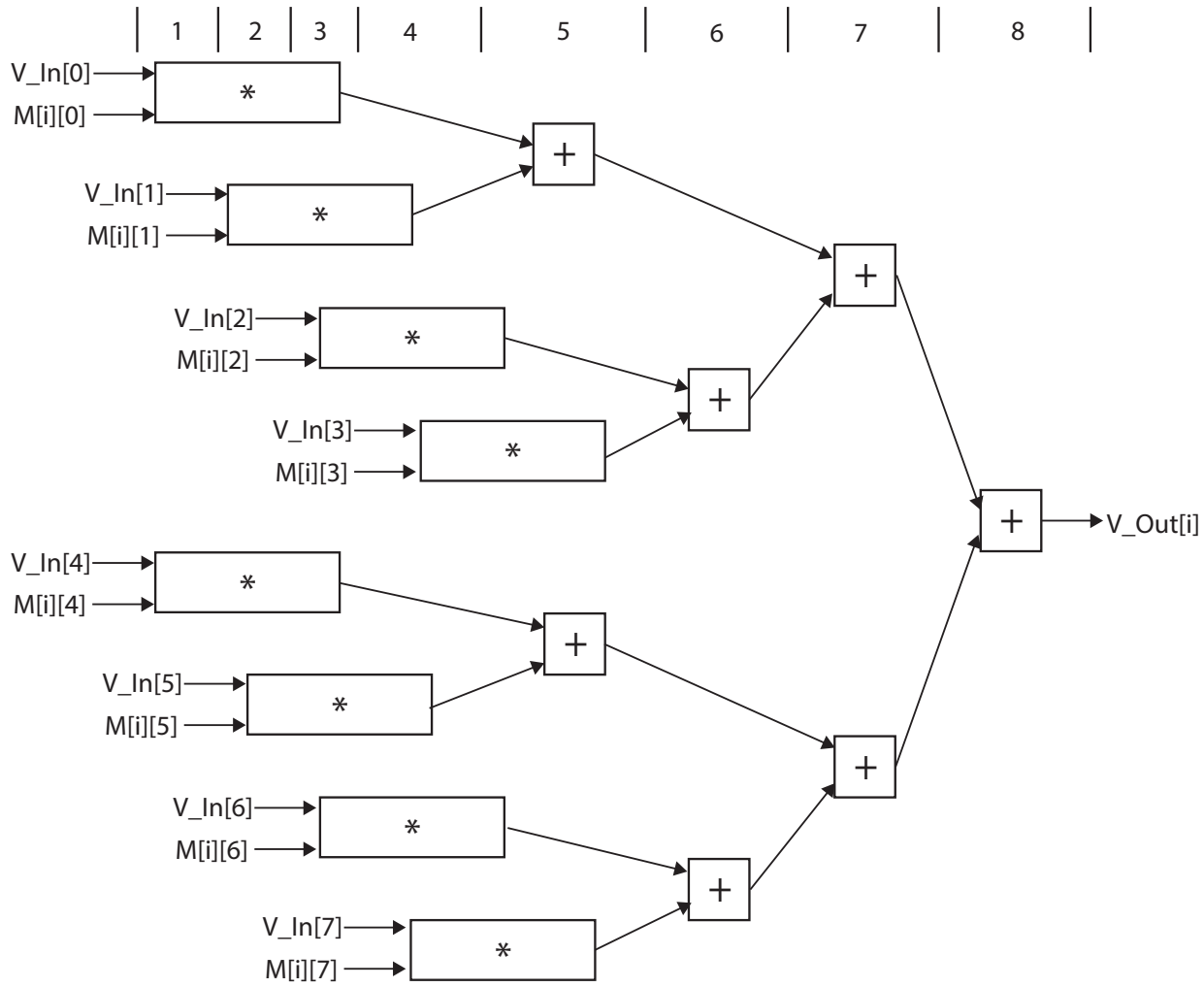


Figure 4.8: A data flow graph of the expression resulting from the unrolled inner loop from Figure 4.6 using pipelined multipliers.

adding more ports can increase the area depending on the size of the array. A single BRAM is limited to two ports. Note that you can partition the array across multiple BRAMs, and this is an important part of design space exploration. That essentially allows you to increase the number of ports while still utilizing BRAMs. The most flexible access scenario would be to put every element of the array into its own register. While this is attractive for increasing the performance, it often requires a significant amount of area. We discuss different way to partition the arrays in Chapter 4.6. But first, the next section moves past this simple and small matrix-vector multiplication example and provide a functionally complete HLS code for the DFT.

4.5 Baseline Implementation

We just discussed some optimizations for matrix-vector multiplication. This is a core computation in performing a DFT. However, there are some additionally 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.

There are several major changes to transform the matrix-vector multiplication into a DFT. First, we wish to perform a DFT on complex numbers. That changes the data types and the computations (e.g., multiplication turn into complex multiplication). Additionally, we compute the S matrix online rather than use precomputed values.

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 4.9 shows a baseline implementation of the DFT. This uses a doubly nested *for* loop. The inner *for* loop performs multiplies one row with the input signal. In each iteration, it calculates the sine and cosine values corresponding to the appropriate phasor. It then performs a complex multiplication of that phasor with the appropriate sample from the input signal. Finally, it adds the result of that complex multiplication to a running total. After N iterations of this inner *for* loop, one frequency domain sample is calculated. The outer *for* loop also iterates N times - one for each of the N frequency domain samples.

This code uses a function call to calculate $cos()$ and $sin()$ values. The high level synthesis tool is capable of handling these functions. In fact it uses a CORDIC implementation to calculate these values. The CORDIC core requires a number of cycles to calculate these values, which depends upon the desired precision. The results of these $sin()$ and $cos()$ function calls are known a priori assuming that N is constant and known at design time. This is often the case, e.g., you will know that you wish to do a 256 or 1024 point DFT. Therefore, it is possible to eliminate these $cos()$ and $sin()$ function calls by using a lookup table that holds the values of the S matrix. This replaces the costly CORDIC implementation of $cos()$ and $sin()$ with a memory access. The S matrix can be stored in BRAMs for smaller size DFT. However, larger DFTs can quickly exhaust the available BRAM space on-chip.

```
#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 * (TEMP_TYPE)i / (TEMP_TYPE)N;

    // 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 4.9: Baseline DFT code with primarily sequential execution. This follows a matrix-vector multiplication implementation.

This provides one possible design tradeoff, which utilizes more memory in order to perform computation faster. We discuss this tradeoff in more detail later.

Figure 4.10 provides an overview of the one potential DFT architecture. This hardware implementation is highly sequential. The data path corresponding to the inner loop is executed $N \times N$ times, which corresponds to N^2 uses of CORDIC and N^2 complex multiplications and additions. In each iteration of the outer loop, the variable w is computed. This is essentially determining the amount of rotation that each row of the S matrix requires. The exact phasor angle is then calculated in each iteration of the inner loop through a multiplication by j. This is fed into a CORDIC core to determine the sine and cosine values used in the complex multiplication and subsequent summation. The result of the inner loop is summed into the appropriate *temp*[] variable. Finally, after these two nested loops are finished, the values in the *temp*[] array are copied into the *sample*[] array. This is called an in-place DFT, i.e., the output frequency samples are stored in the same location as the input signal samples. This saves memory at the expense of N additional read and write operations.

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

4.6 Array Partitioning

One important design space tradeoff in generating a hardware DFT architecture focuses on the DFT coefficients. We could compute them on-demand, using a sine or cosine function call, which is typically implemented as a CORDIC core. This is what we did in the baseline implementation in the previous section. The other option is to precompute these coefficients and store them in a memory. The latter is much faster; it requires only a memory lookup which, if the coefficients are stored on-chip in BRAMs or LUTs, requires one cycle or less. However, this utilizes precious on-chip resources, and can quickly become infeasible as for larger DFT cores. For example, assume we are doing a 1024 point DFT. A naive implementation would store the corresponding sine and cosine values, which would require one for each element of the S matrix. A 1024 point DFT has an S matrix of size $1024 \times 1024 = 2^{10} \times 2^{10} = 2^{20} = 1$ Mega "entries". If each entry is a 32 bit fixed or floating point number, this results in 4 MBytes of memory for each sine and cosine table, requiring a total of 8 MBytes of on-chip storage. While this is within the realm of possibilities for modern FPGAs, it would not only require a high end (i.e., expensive) FPGA, but also utilize a large portion of the on-chip memory. Thus it would be beneficial to reduce the memory requirements if at all possible.

The code shown in Figure 4.9 performs a dynamic calculation of the phasor values (i.e., the *sin*() and *cos*() function calls). Rewrite the code to eliminate these function

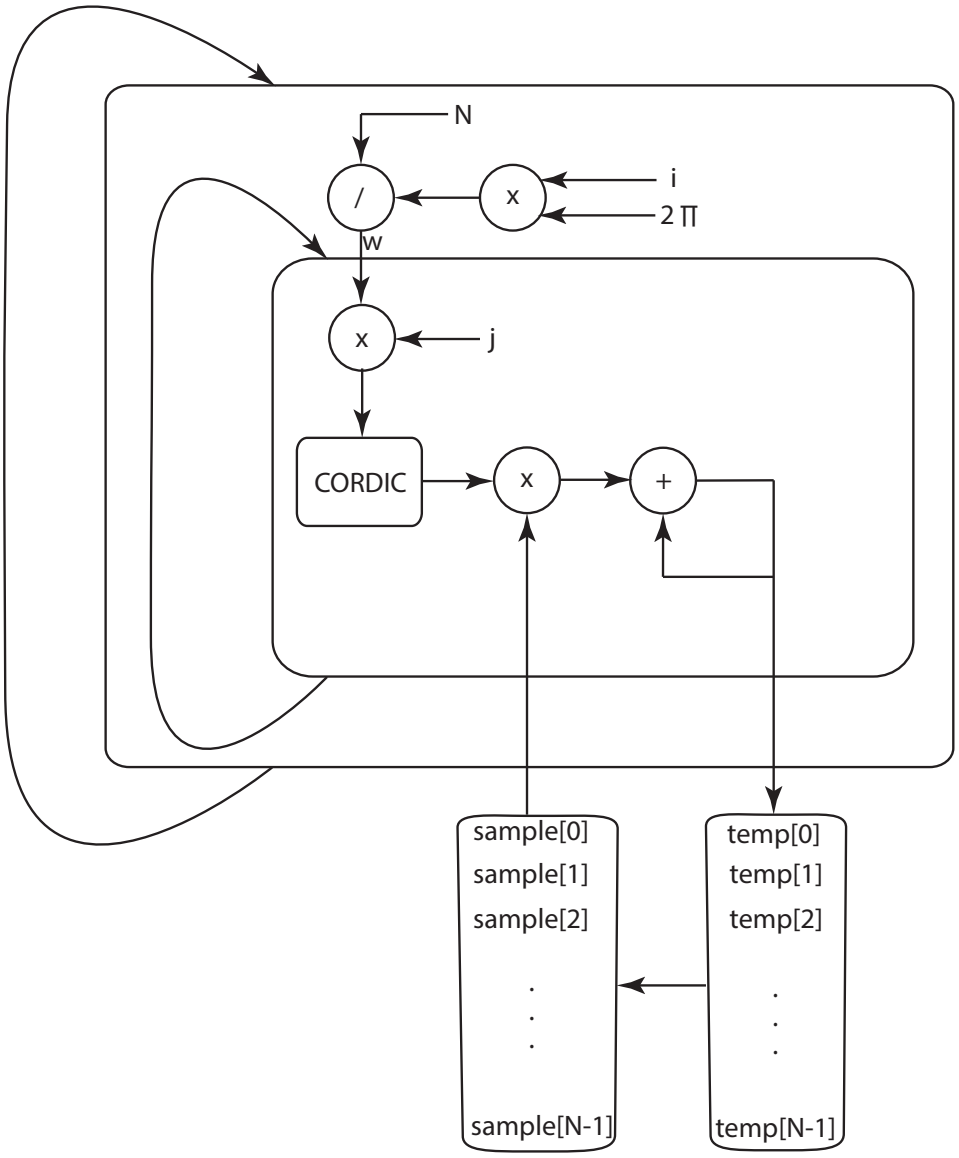


Figure 4.10: A high level architectural diagram of the DFT as specified in the code from Figure 4.9. This is not a comprehensive view of the architecture, e.g., it is missing components related to updating the loop counters i and j. It is meant to provide an approximate notion of how this architecture will be synthesized.

calls by utilizing a table lookup. How does this change the throughput and area? What happens to the table lookup when you change the value of the constant N?

The first obvious way of reducing the memory storage requirement would use smaller data types. Unfortunately this can only get us so far. For example, by restricting each element to 8 bits (1 byte), we reduce the memory footprint by a factor of four to 2 MBytes. And this comes at the potential costs of a loss in precision.

What would happen to the code in Figure 4.9 if you changed all of the data types from *double* to $float$? Or from *double* to *int*? Or some other fixed point value? How does this change the number of cycles? Would it affect the resource usage? Does it change the values of the output frequency domain samples?

The placement of the data plays a crucial role in the performance. There are different types of storage elements where the data can reside. Off-chip storage in DRAM, flash, or other storage devices holds large amounts of data. However, data access times are long, on the order of tens to hundreds (or more) of cycles. Therefore, when possible, we should store the data on-chip.

The primary choices for on-chip storage on in embedded memories (e.g., block RAMs) or in flip-flops (FFs). These two options have their own tradeoffs. FFs allow for sub-cycle access times, i.e., you can access, compute, and store data in one cycle. However, the number of FFs is limited to on the order of 100 Kbytes. Block RAMs (BRAMs) offer additional 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 by two read ports and one write port. Furthermore, the access time is around one cycle because you must present the address of the data that you wish to access on the clock cycle before you will receive the data. 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 allows the data to be accessed exactly when it is needed. However, as the size of the DFT grows large, this is not feasible since we have on the order of 100 Kbytes of FFs and a 1024 DFT requires Mbytes of storage. Therefore, we must use BRAMs for larger DFT sizes.

The `array_partition` directive partitions an array into smaller arrays. The directive `array_partition complete` will split each element of an array into its own register. This is equivalent to putting every bit of data into a FF.

When using BRAMs, the fundamental questions boils down to how many BRAMs should be used, and how should the data be distributed into these different BRAMs. Obviously, we will need at least as many BRAMs to hold the desired amount of data. Assume we have an FPGA architecture where each BRAM has 36 Kbits (= 4.5 Kbytes) of storage. Recall that

storing only the sine values using 8 bits of precisions requires 1 Mbyte of storage. Thus, we need at least (1000/4.5 = 222.2) 223 BRAMs to store this data alone.

The question then becomes how do we store the data across these different BRAMs. For example, we could store the sine values in the S matrix in row-major or column-major order. This may effect the performance depending upon the data access patterns. For example, if we could partition the data in such a way that none of the accesses at any time occur to the same BRAM, then this would allow for all of the memory access operations to be scheduled at the same cycle. However, if the data was partitioned such that the data accesses all went to the same BRAM at any given time, these access must be sequentialized since the BRAM only has a limited number of read ports. Therefore, it is important to carefully consider how the data access patterns required by the code when you are partitioning your data.

To make this more concrete, consider the storage of the $N \times N$ S matrix. Also, assume that the S matrix will fit into one BRAM, e.g., it is a 32 point DFT where each entry of the matrix is 1 byte; this case requires 1 Kbyte of storage. To further simplify the discussion, assume the BRAM has one read port and one write port. The question that we want to answer is how to best partition the S matrix to maximize the performance of the resulting architecture.

We augment the code from Figure 4.9 to that shown in Figure 4.11. We replaced the sine and cosine function calls with a lookup into two precomputed arrays `sin_table[]` and `cos_table[]`, respectively. We assume these $N \times N$ arrays hold the sine and cosine values for the complex vectors in the S matrix as denoted in Figure 9.6. We further assume that the values are stored in row-major order.

Row-major order stores the values of an array with the elements in a row in consecutive order. For example the matrix when stored in row-major order is

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

The same matrix in column-major order is

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

We can now start exploring methods to parallelize the DFT computation. Loop unrolling is an important optimization in this regard. This essentially replicates several instances of the loop body. This has the effect of enabling multiple iterations of the loop to execute concurrently, assuming that their are no loop carried dependencies that serialize the computations. The inner loop of the DFT is a prime example where the computation in the iterations of the loop can run in parallel.

Consider the case when the inner loop is unrolled by a factor of two; Figure 4.12 shows the result of this code. You can see that the loop bounds now increment by 2. We fetch two precomputed sine and cosine values instead of one. And we perform two complex multiplies instead of one. The benefit of this loop unrolling is that it gives the HLS tool the flexibility to

```
#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;

    // Calculate the jth frequency sample sequentially
    for (j = 0; j < N; j += 1) {
      // Fetch precomputed sine and cosine values
      c = cos_table[i * j];
      s = sin_table[i * j];

      // Multiply c and s 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 4.11: Baseline DFT code using precomputed sine and cosine values for the S matrix.

```
...

    // Calculate the jth frequency sample sequentially
    for (j = 0; j < N; j += 2) {
  // Fetch precomputed sine and cosine values
  c_0 = cos_table[i * j];
  s_0 = sin_table[i * j];
  c_1 = cos_table[i * (j + 1)];
  s_1 = sin_table[i * (j + 1)];

  // Multiply c and s with the appropriate input sample and keep running sum
  temp_real[i] += (sample_real[j] * c_0 - sample_imag[j] * s_0) +
                  (sample_real[j + 1] * c_1 - sample_imag[j + 1] * s_1);
  temp_imag[i] += (sample_real[j] * s_0 + sample_imag[j] * c_0) +
                  (sample_real[j + 1] * s_1 + sample_imag[j + 1] * c_1);
}
}

...
```

Figure 4.12: The inner loop of the DFT manually unrolled by a factor of two.

implement the operations in the two subexpressions, corresponding to two iterations of the loop, in parallel. While the HLS tool may or may not fully take advantage of this increased scheduling flexibility, it at least has the option. When the loop was not unrolled, the HLS tool has no choice but to schedule the operations in consecutive iterations sequentially.

> 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.

The unrolled code attempts to do two loads from each of the arrays `cos_table[]` and `sin_table[]`. If both of these arrays are stored in one BRAM that has only one read port, then these load operations must be scheduled sequentially. This limits the flexibility of the HLS tool, and can lead to suboptimal results. Ideally, the HLS tool has as many options to consider. By intelligently dividing the `cos_table[]` into two BRAMs, the two load operations can be done in parallel; one from each BRAM[1]. In order to do this, we should 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.

[1]We focus on the `cos_table[]` to simplify the discussion; the same optimizations should be done on the `sin_table[]`.

We could manually partition the data into the two loops or use the `array_partition` directive. In this case, we would want to use `cyclic` partitioning with a `factor` = 2. The `factor` argument states to divide the array into two arrays. A `cyclic` argument interleaves the elements from the original array into the two partitioned arrays. The combination of these two arguments divides the original array into two arrays; one has the even elements and the other has the odd elements.

Using the directive `array_partition -factor 2 -type cyclic` on the array

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

splits it into two arrays which are

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

What is the relationship between array partitioning and loop unrolling? Does it help to perform one without the other? What is the relationship between the array partitioning factor and the unroll factor?

Many (but not all) of the directives can be done manually. For example, a user could manually split the `sin_table[]` into two arrays which hold the even and odd elements. The user would not only have to split the matrices, but also change the code in order to divide the array access operations so that they access the appropriate array. This is not only a tedious and error prone process, but it also complicates the code making it difficult to read and modify.

Manually divide `sin_table[]` and `cos_table[]` into two arrays in the same manner as the directive `array_partition -factor 2 -type cyclic`. 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?

Loop unrolling and array partitioning often go hand in hand. We described how unrolling by a factor of two required that we partition the arrays by a factor of two. Unrolling by larger factors similarly requires the appropriate array partitioning. In the case of DFT (matrix vector multiplication), this relationship is straightforward – the loop unroll factor and partition factor are equivalent.

Loop unrolling and array partitioning have a cost. Array partitioning has a direct effect on the number of BRAMs. In general, loop unrolling results in a design that uses more

resources, e.g., LUTs, FFs, DSP blocks, etc. The exact amount of area (or resource usage) is not straightforward. But increased performance typically results in a larger design.

> 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?

4.7 Application Specific Code Restructuring

We can employ application specific tricks to reduce the amount of data that is required for the sine and cosine values corresponding to the S matrix. Recall that the complex vectors for each element of the S matrix are calculated based upon a fixed integer rotation around the unit circle. The "first" row of the S matrix corresponds to one rotation around the unit circle while the other rows correspond to integer rotations around the unit circle. Note that recording the sine and cosine values corresponding to the first row, i.e., one rotation around the unit circle, provides all of the requisite sine and cosine values for every other row. We denote the one dimensional storage of the matrix S as S' where

$$S' = S(1,:) = (1 \quad s \quad s^2 \quad \cdots \quad s^{N-1}) \tag{4.14}$$

We use MATLAB convention $S(1,;)$, meaning that S' corresponds to the first row of the S matrix. Please refer to Equation 4.12 for a mathematical definition of the S matrix. **MAKE SURE THAT I DID THE MATLAB NOTATION CORRECT.**

This is be visually confirmed by studying Figure 9.6. The vectors corresponding to the first row, which is one rotation around the unit circle (divided into $360/8 = 45°$ individual rotations), cover all of the vectors from every other row. Thus it is possible to store only the sine and cosine values from this one rotation, and then intelligently 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 8 KB of on-chip memory. Obviously, this is a significant (order of magnitude) reduction from using a similar 32 bit value for the elements.

> 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?

This seems like a great idea, and it certainly is if we are trying to create a design that uses the smallest amount of data storage resources. But it comes at a cost. Storing all of the data into a 1D array complicates the partitioning of that S' array in order to parallelize the computation.

Previously we argued that loop unrolling exposes parallelism that can increase the performance. This required that we divide the array S across multiple memories, e.g., by using the `array_partition` pragma, so that we did not have a memory access bottleneck. We have many different options on how to do this, and we argued that `cyclic` was the most appropriate for partitioning the 2D array S.

Unfortunately, compressing the S into the 1D S' array complicates the partitioning. The problem is due to the access pattern on S'. The accesses are dynamic depending on which output value that we are computing. For example, when we are computing the DC component, we require the data from the first row of S. This corresponds to continually accessing the first element of S', i.e., $S'[0] = 1$. Computing the data corresponding to the second frequency output requires that we access every element of S' in consecutive order. The computation of the third output value requires accesses every other element (and wraps around once we reach the end). Computing the third output accesses every third element of S'. And so on.

> 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.

This dynamic access pattern makes partitioning the S' array difficult. For example, consider the previous scenario where we unroll the inner loop of the DFT calculation by a factor of two, i.e., `unroll -factor 2`. These means that we are enabling two consecutive iterations to operate in parallel. How should we partition the S' array? We could partition it into two arrays where one array holds the even elements, and the next array holds the

odd elements, i.e., `array_partition -factor 2 -type cyclic` as we did before with the S array.

Consider how this effects the computation of the different elements. When we compute the first output value – the DC value – we access the first element of the S' on every iteration of the inner loop of the DFT. This may or may not be a problem depending on how you specify the accesses. If they written as array accesses with constant indices (i.e., $S'[0]$), the HLS tool should be intelligent enough to know that we are accessing the same value, and share that value across the two complex multiplies in the inner loop. Let assume that this is the case, thus everything works fine.

When we are computing the first harmonic, corresponding to using the values from the first row of S, we access the values of S' in sequential order as described before. In the unrolled inner loop, we will need an even element from S' and an odd element from S' on every iteration. Therefore, the cyclic partitioning works. So far so good.

The issue arises in computing the higher harmonic output values. When computing the second harmonic, which uses the constants from the second row of S, we are always accessing even elements of S'. The first complex multiply in the inner loop requires $S'[0]$; the second uses $S'[2]$; the third accesses $S'[4]$, and so on. In this case, the cyclic partitioning does not help us since would always access one memory – the memory holding the even values of S'. Computing the higher harmonic output frequency values runs into similar problems. Sometimes the partitioning works; other times it does not.

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 4.2 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 [14].

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 [13], 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. [24] 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 4.2.

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

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

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 4.2) 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 [21].

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

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

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

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

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

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

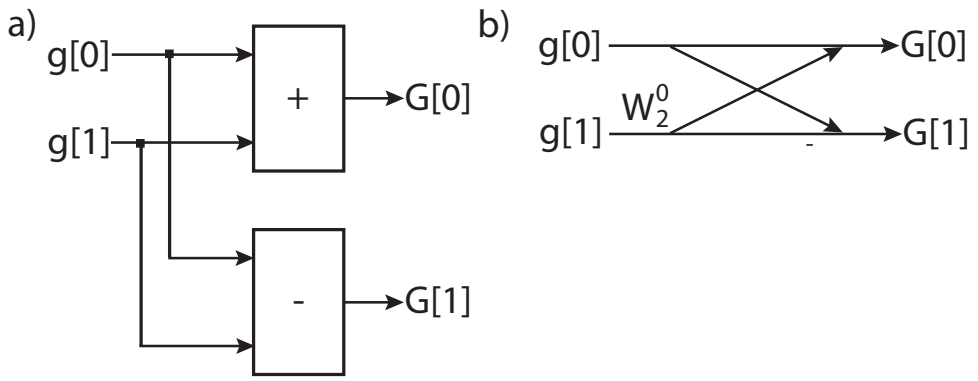


Figure 5.1: Part a) is a data flow graph for a 2 point DFT/FFT. Part b) shows the same computation, but viewed as a butterfly structure. This is a common representation for the computation of an FFT in the digital signal processing domain.

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:

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

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

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

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

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

since $e^0 = 1$.

The equation for $G[1]$ is:

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

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

The equation for $G[2]$ is:

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

The reductions were done by simplifications based upon rotations. E.g., $e^{\frac{-j8\pi}{4}} = 1$ and $e^{\frac{-12j\pi}{4}} = -1$ 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 $g[3]$.

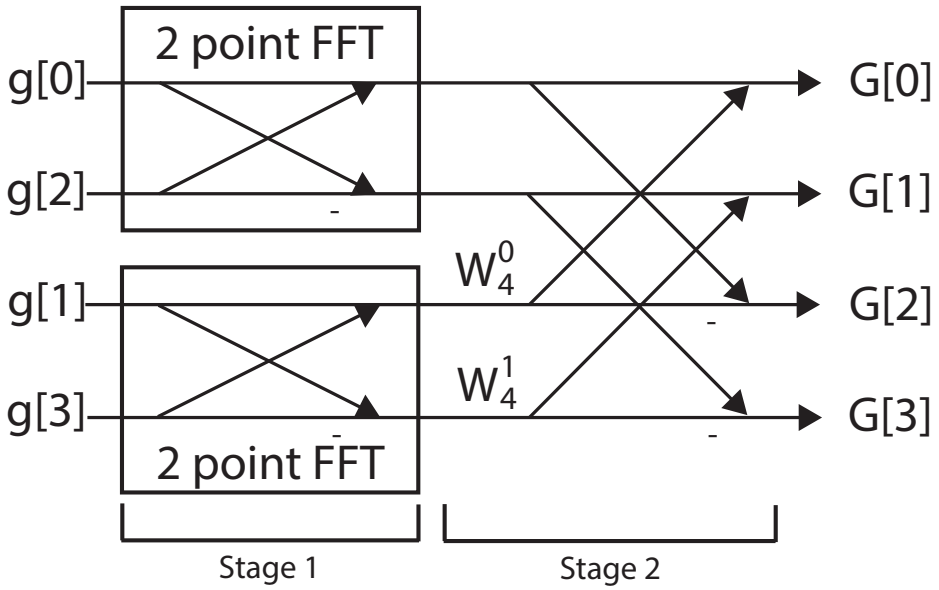


Figure 5.2: A four point FFT divided into two stages. Stage 1 has uses two 2 point FFTs – one 2 point FFT for the even input values and the other 2 point FFT for the odd input values. Stage 2 performs the remaining operations to complete the FFT computation as detailed in Equation 5.12.

We 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 FFTs 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.

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 FFTs. This extends to larger FFT sizes. For example, an 8 point FFT uses two 4 point FFTs, which in turn each use two 2 point FFTs (for a total of four 2 point FFTs). An 16 point FFT uses two 8 point FFTs, and so on.

How many 2 point FFTs are used in a 32 point FFT? How many are there in a 64 point FFT? How many 4 point FFTs 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 FFTs in an N point FFT (where $N > 8$)?

Now let us formal 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:

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

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

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

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:

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

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

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

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

$$G[k] = A_k + W_N^k B_k \tag{5.17}$$

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 $k = N/2, N/2+1, \ldots, N/2-1$. Thus, we wish to calculate

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

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

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

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

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

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

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

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 $e^{-j\pi} = -1$.

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

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

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

$$G[k+N/2] = A_k - W_N^k B_k \tag{5.23}$$

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

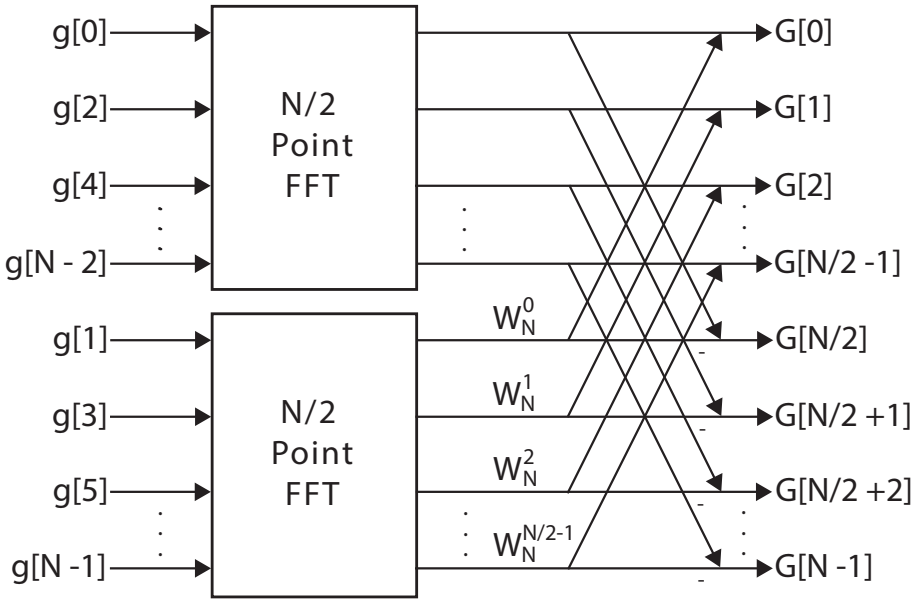


Figure 5.3: Building an N point FFT from two $N/2$ point FFTs. The upper $N/2$ point FFT is performed on the even inputs; the lower $N/2$ FFT uses the odd inputs.

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.

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 $\{g[1], g[5], g[3], g[7]\}$.

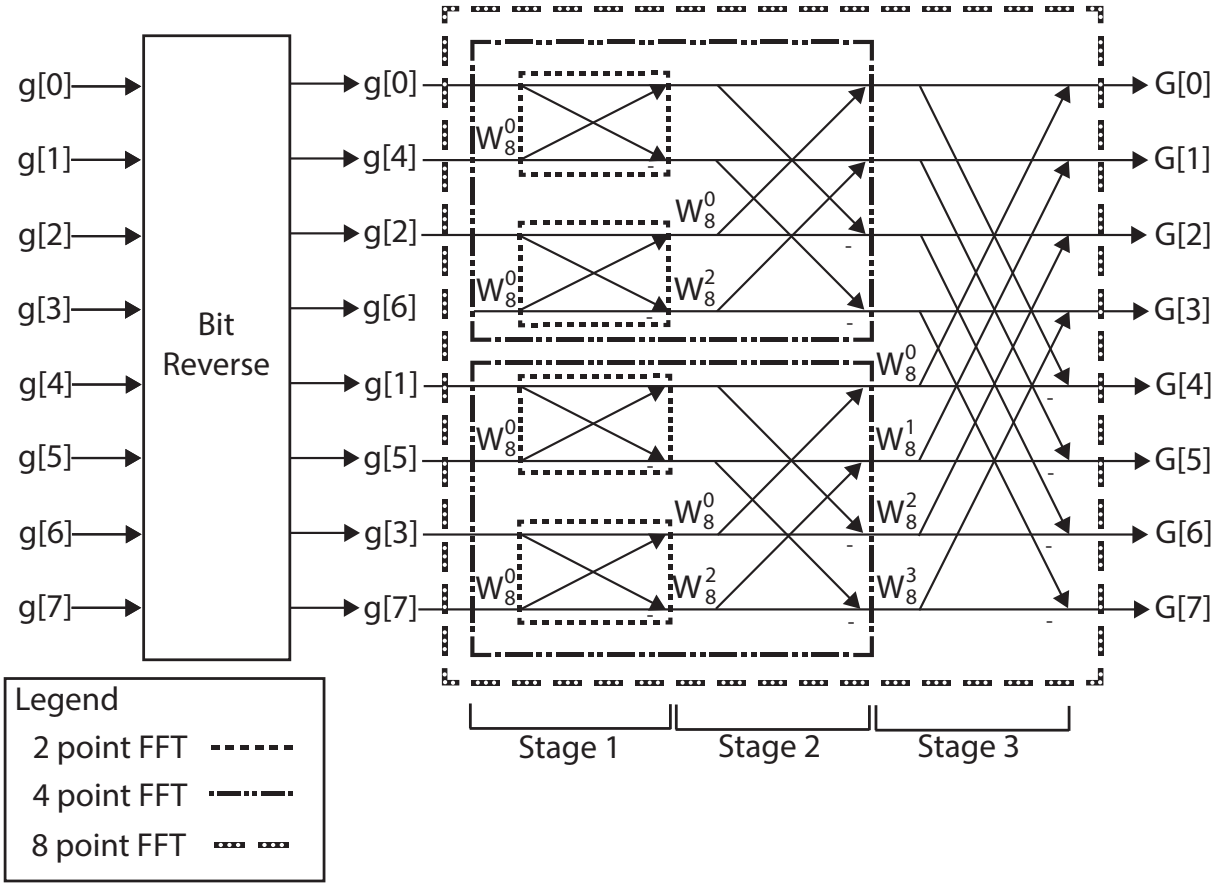


Figure 5.4: An 8 point FFT built recursively. There are two 4 point FFTs, which each use two 2 point FFTs. The inputs must be reordered to even and odd elements twice. This results in reordering based upon the bit reversal of the indices.

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. FFT 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 [13] 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. Note that this architecture can also allow some other optimizations e.g., each of the tasks can themselves be pipelined..

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 points*(8 bytes/point)*250*10^9 Hz = 1Terabyte/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 ith 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 with the label `stages` 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` performs all of the butterfly operations for the current stage. This `butterfly for` loop has another nested `for` loop with the label `DFTpts`; each iteration of this `for` loop performs one butterfly operation. Remember that we are dealing with complex numbers, thus we must perform complex additions and multiplications.

The first line in this `DFTpts for` 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 with adjacent elements; Stage 2 does the butterfly operations two elements away; and Stage 3 butterfly has a width of four elements. That is exactly what the `i_lower` variable is calculating. Notice that this offset – the variable `numBF` – is set differently in every stage.

The remaining operations in the `DFTpts for` loop perform the multiplication by the twiddle factor, and the 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 or W. The variables `c` and `s` are the real and imaginary parts of W, which is calculated using at sin() and cos() function. These are builtin operations. We could also use a CORDIC, e.g., one that we developed in Chapter **??**, if we wanted to have more control over the implementation. The next two lines, corresponding to setting the elements `X_R[i_lower]` and `X_I[i_lower]` perform the subtraction operation corresponding to the bottom part of the butterfly. The final two statements do the addition operation on the upper part of the butterfly.

```
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;

stages:
  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:
    for (j = 0; j < numBF; j++) {
      c = cos(a);
      s = sin(a);
      a = a + e;
    // Compute butterflies that use same W**k
    DFTpts:
      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.

The `DFTpts for` loop iterates a different number of times depending upon the stage. The total number of times that this `DFTpts for` loop is executed in one stage is constant; however, the number of times that the `butterfly for` loop iterates depends upon the stage. 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 all of the same W twiddle factors, in this case W_8^0. Stage 2 uses two unique twiddle factors; and Stage 3 has four different W values. Thus, in Stage 1, the `butterfly for` loop has only one iteration; in Stage 2 the `butterfly for` loop iterates twice; and it iterates four times in Stage 3. In each case, the number of times that the `DFTpts for` loop is called and iterates per call changes. In Stage 1, it is called once, but would iterate four times for an 8 point FFT. In Stage 2, the `DFTpts for` loop is called two times and each time it iterates two times. And in Stage 3, it is called four times, and each time it iterates on one time. But in every stage, the body of the `DFTpts` function is executed the same number of times – four iterations for an 8 point FFT.

When synthesizing the code in Figure 5.5, the Vivado® HLS tool may not provide a value for the latency or interval of the FFT function. When the tool reports a value of '?', this indicates that it cannot reason about how long some part of the code executes. In this case, the Vivado® HLS may not be able to determine the number of times that the `butterfly` and `DFTpts for` loops iterate. This due to the fact that these loops have variable bounds.

The `tripcount` directive enables the user to specify to the Vivado® HLS tool the number of times a loop is executed. It takes three optional arguments `min`, `max`, and `average`. In this case, we could add a directive to the `DFTpts` loop. By applying this directive, the Vivado® HLS tool can calculate 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 operation (`bit_reverse`). 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` value and shifts it into the `rev` variable. The `for` loop body consists of a few bitwise operations. These are very efficient when implemented on an FPGA. Thus the entire loop body can very likely be implemented within one cycle.

The `for` loop imposes a sequential flow upon the execution of this function. This can be parallelized by using `unroll` and `pipeline` directives.

Assume that the loop body can be implemented in one cycle. In this case, pipelining will not help since performing the loop in a pipelined manner does not save any cycles compared to implementing it without pipelining. That is, when the loop is performed in a pipelined manner, each iteration of the `for` loop is completed before the next iteration begins because an iteration only requires one cycle. This is the same thing that happens when we execute the `for` loop in the "sequential" (i.e., non-pipelined) manner. In either implementation, pipelined or non-pipelined, we are essentially taking one cycle for each bit operation. Thus, if we are dealing with `int` data, as we are here, the loop latency is around 32 cycles.

How many cycles does the `reverse_bits` function require without pipelining? How many cycles does the pipelined version take?

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 then she carefully considers these in tandem to achieve the best results.

We should be able to parallelize these rather simple operations more effectively. One manner of doing this would be to unroll the loop. Since the implementation of the loop body is relatively small in hardware, we can fully unroll the loop without a huge explosion in the number of resources. This provides a very efficient and fast implementation of this loop.

How many cycles does the `reverse_bits` function require when you unroll the `for` loop completely? How many resources does it use? How do these change when you change the unroll factor to 2, 4, 8, 16, ...?

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

```
#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 croresponding 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 bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE]
                 DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);
void fft_stage_one(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                 DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);
void fft_stages_two(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                 DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);
void fft_stage_three(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                 DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);

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_R);
   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.

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, we only swap values the first time, which is done by the condition `if(i < reversed)`.

We can optimize this function in a number of different ways. Since the `reverse_bits` function is small and efficient, it make sense to use the `inline` pragma. This essentially copies all of the code from within the `reverse_bits` function directly into the `bit_reverse` function. Thus there is no function call, and any overhead associated with it goes away. Furthermore, this allows the Vivado® HLS tool to optimize the code within the `reverse_bits` function along with the code in the `bit_reverse` function, which can lead to some additional efficiencies. That being said, it may be the case when the `inline` directive does not help.

5.4 Task Pipelining

The best way to implement the FFT in hardware is to divide the algorithm into stages. This allows us to perform task level pipelining 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

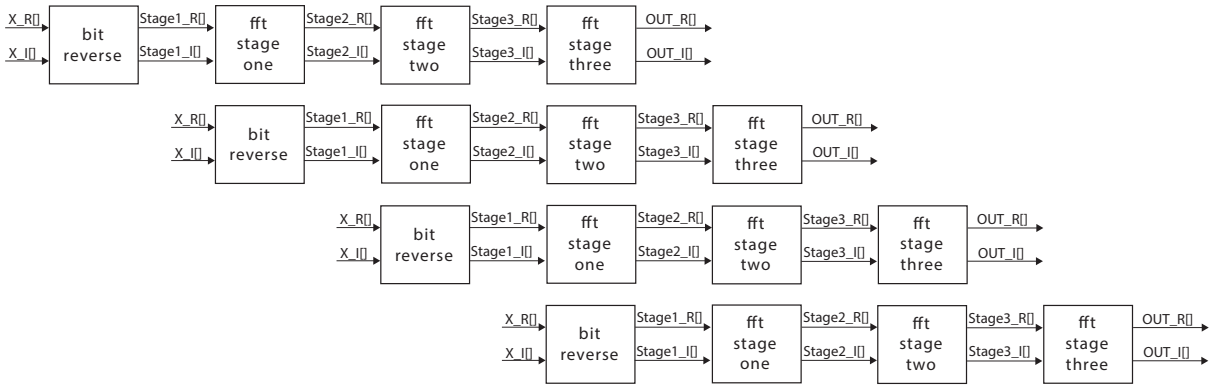


Figure 5.8: Dividing the FFT into different stages allows for task pipelining across each of these stages. The figure shows an example with three FFT stages (i.e., and 8 point FFT). The figure shows four 8 point FFT executing at the same time.

of points of the FFT. Bit reverse is the first stage. This stage swaps a value of the input data 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 5.7 describes how to devise 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 real and one for imaginary portions 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 level pipelining. Figure 5.8 gives an example of this. This is much like other types of pipelining, e.g., using the `pipeline` directive in a loop or a function, with one major difference – the level of granularity of the computation being pipelined. Here we are pipelining the individual functions. That is rather than wait for all four functions to complete, we start the second function, `fft_stage_one`, immediately after we complete the first function `bit_reverse`. And the third function `fft_stages_two` starts after the second function completes, and so on. Once the pipeline is full, all four functions are always executing, but each one has different input data. That is, there are four 8 point FFTs being computed simultaneously – each one on a different function. 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 FFTs running simultaneously.

The dataflow directive tells the Vivado® HLS tool to perform task level pipelining. The directive works on both functions and loops. Our code in Figure 5.7 works uses functions only, but it would perform in a similar manner if we had four loops instead of four functions.

The `dataflow` directive informs the Vivado® HLS tool to perform task level pipeline. This is similar to the `pipeline` pragma in that it performs several computations at the same time in a pipelined manner. It differs in the granularity of the computation. The

`pipeline` directive works on operations within a function or on the iterations of a loop; the `dataflow` is done across functions or loops. We will see an example of the `dataflow` directive as we optimize the FFT. It will be used to pipeline across FFT stages, which will be each implemented as a separate function.

The `dataflow` directive attempts to best implement the memory used to pass data between the two functions. In the case when the Vivado® HLS tool can explicitly determine that the tasks work in a streaming manner, it implements the data transfer using a FIFO. These requires that the data from the previous task is consumed by the next task in the same manner it which it is outputted.

If the is not the case (or if the Vivado® HLS tool can not determine if this streaming condition is met), then the Vivado® HLS tool will generate a memory transfer using a ping-pong buffer. This is two memories. One of the memories will be written to by the previous task. And the other memory will be read from by the next tasks. The term "ping-pong" comes from the fact that the reading and writing to the buffers alternates in every execution of the task. That is, the producer task will write to one memory, and than switch to the other memory in the next execution of that task. The consumer task will read from the memory that the producer is not writing to. And in the next execution it will read from the other memory. The producer and consumer tasks are never writing and reading from the same memory.

A ping-pong buffer requires twice the amount of memory as a FIFO. However, the data in a ping-pong buffer can be written to and read from in any order. Thus, FIFOs are the best choice when the data is produced and consumed in a streaming manner. And ping-pong buffers are necessary when there is not such a regular data access.

Using the `dataflow` directive does not preclude other optimizations on the code. For example, it is beneficial to look carefully at each of the tasks or stages. We have already discussed some optimizations for the `bit_reverse` function in Section 5.3. These include restructuring the code, pipelining, and unrolling. Furthermore, each of the FFT stages are open to similar optimizations.

In general, it is important to optimize the individual tasks while considering the overall goal – the top level function. Many times it is best to start with the tasks themselves since they are smaller chunks of code. As a designer, you can more easily comprehend what is going on, and hopefully determine the best optimizations quickly. After you optimize each of the task, then you can move up the hierarchy and perform task pipelining and possibly other optimizations (e.g., code restructuring).

However, the local optimizations must be considered in the overall scope of the goals. For example, the initiation interval for task pipelining is only as small as the largest latency for any of the tasks. Looking again at Figure 5.7, assume that `bit_reverse` takes 8 cycles to complete, `fft_stage_one` takes 12 cycles, `fft_stage_two` requires 12 cycles, and `fft_stage_three` takes 14 cycles. When using `dataflow`, the initiation interval is 14 – the maximum of all of the tasks/functions. This means that you should be careful in balancing the optimizations across the tasks. For example, in terms of throughput, it does not help to optimize the latency of `bit_reverse`. In fact, it would be beneficial to increase the latency if that leads to a reduction in the resource usage.

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 16 where we task the designer to create an simple orthogonal frequency-division multiplexing receiver. The core of this 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 an 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 give 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 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 translate your HLS testbench into a RTL 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 a) shows an example of a sparse matrix $\mathbf{M}$. For the sake of simplicity, we assume $\mathbf{M}$ has a size of 4×4. There are different ways to represent the sparse matrix. We use compressed row storage (CRS) matrix format. The CRS format compresses the sparse matrix to save memory. It makes no assumptions about the sparsity of the matrix, and does not store any unnecessary elements. This makes it a general approach that can be used for any matrix, but not necessarily the most efficient.

Figure 6.1 b) shows the corresponding compressed row storage format. The CRS format uses three data structures: `values`, `colIndex`, and `rowPtr`. These data structures corresponding to matrix $\mathbf{M}$ are shown in Figure 6.1 b). The data structure `values` has an entry for each of the non-zero elements in the sparse matrix $\mathbf{M}$. They are stored into the data structure by traversing in a row-wise fashion across $\mathbf{M}$ (i.e., left to right, and top to bottom).

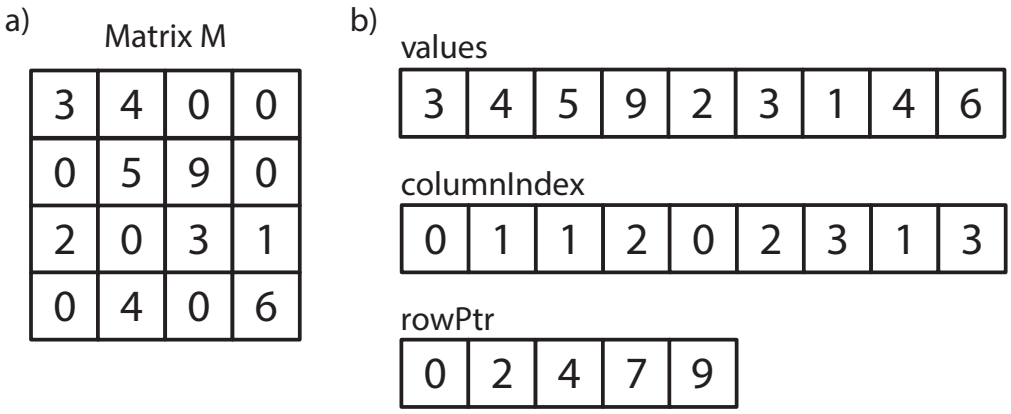


Figure 6.1: A compressed row storage (CRS) representation of the matrix **M**. CRS has three data structures. The first data structure `values` holds the value of each non-zero element in M. The `columnIndex` data structure indicates the column for each of these elements in `values` at the same index. The `rowPtr` vector stores the index of the first element in each row.

The `columnIndex` also has an entry for each non-zero element in **M**. This data structure indicates the column for each of the elements in `values`. That is, if $value[k] = M_{ij}$ then $colIndex[k] = j$. The data structure `rowPtr` is a list of the values of the starting index of each row. In other words, it stores the location in the `values` data structure that start a row. Since the first two elements are 0 and 2 respectively, the first row contains the elements $[0, 2)$, i.e., $[0, 1]$. The second row has elements $[2, 4)$; the third row has element $[4, 7)$, and so on. Another way to look at this is if $value[k] = M_{ij}$, then $rowPtr[i] \leq k < rowPtr[i + 1]$. We define $rowPtr[n + 1]$ as the number of non-zero of entries . In this case, there are 9 non-zero entries, thus that last entry in the `rowPtr` data structure is 9.

The first element in the `values` data structure is 3; it is in column 0 as indicated by indexing into the first element of the data structure `columnIndex`. The `rowPtr` data structure indicates that the first row contains the elements [0, 2). Thus, the first element is in the first row. Now consider $values[4] = 2$. We can find its column by $columnIndex[4] = 0$ thus it is in the zeroth column. Finding the row is a bit more difficult. For this, we search through the `rowPtr` data structure. We see that the third element is 4, thus we know that the fourth element from values is in the third row. In fact, this is the first non-zero element in the third row. You will see that this `rowPtr` data structure is efficient for our implementation of sparse matrix vector multiplication since you must iterate across the matrix rows when multiplying them by the vector to get the final result.

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 in CRS format. These are equivalent to the data structures shown in Figure

```
#include "spmv.h"

void spmv(DTYPE rowPtr[NUM_ROWS+1], DTYPE 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++) {
           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 compressed row storage (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`.

```
#ifndef __SPMV_H__
#define __SPMV_H__

const int SIZE = 4; // SIZE of square matrix
const int NNZ = 9; //Number of non-zero elements
const int NUM_ROWS = SIZE;
typedef int DTYPE;
void spmv(DTYPE rowPtr[NUM_ROWS+1], DTYPE 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.

6.1. The argument `y` holds the output result. And the argument `x` holds the input vector to be multiplied by the matrix. That is, we have $y = \mathbf{M} \cdot x$ where $\mathbf{M}$ is represented in CRS format by `rowPtr`, `columnIndex`, and `values`. 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 vector `x`, which is the same as the number of elements in the output vector `y`.

The first `for` loop iterates across each row. The second nested `L2 for` loop iterates across the elements in the columns of the matrix $\mathbf{M}$. We use the fact that the `rowPtr` data structure holds the first elements of each row. Thus we should iterate between the values stored in the i and $i+1$ indices of `rowPtr`. This `L2 for` loop will iterate `rowPtr[i+1] − rowPtr[i]` times which is the number of non-zero entries in that row. For each of those entries, we grab the value of that non-zero entry from the $\mathbf{M}$ matrix from `values` and multiply it by the corresponding value in the input vector `x`. 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 vector `y_sw` stores the output from the function `matrixvector` and the vector `y` 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. We print out the results of the test and then return the `fail` variable.

This testbench is very naive and does a poor job of exercising any corner cases. It only tests one small example. Certainly it should not be relied upon to insure that the

```
#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};
  DTYPE columnIndex[] = {0,1,1,2,0,2,3,1,3};
  DTYPE 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`).

implementation is correct. A good testbench would test a large number of examples. These would have a range of sizes in terms of rows, columns, and the number of non-zero elements.

Create a more sophisticated testbench. This should test different size matrices. The matrices could be autogenerated using random numbers. The parameters of the sparse matrix should be modifiable (e.g., `SIZE`, `NNZ`, etc.).

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 of initiation interval. This is because this depends upon the input data, which is external to the `spmv` function itself. In other words, the number of clock cycles depends $\mathbf{M}$. More specifically, the number of non-zero elements in $\mathbf{M}$, i.e., the constant `NNZ`. It may also vary based on the location of these non-zero elements depending upon the optimization directives utilized during synthesis. In summary, the `L2 for` loop has variable loop bounds which cannot be derived without knowing the input data. Thus, the synthesis tool cannot possibly determine the total number of clock cycles for the `spmv` function without additional information.

There are several ways that one can derive the clock cycle numbers. The first manner is to directly provide the Vivado® HLS tool 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 filling in all of 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 better idea about the performance. And it allows for a comparison between different implementations of the `spmv` function either by applying different optimizations or by restructuring the code itself.

However, it may be difficult to determine the `min`, `max`, and `avg` parameters. Or it could be difficult to provide tight bounds on `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 does more than just provide an accurate clock cycle count; it performs automatic testing of the register transfer level (RTL) designs that are generated by the Vivado® HLS tool. It does this by utilizing the provided testbench. It records the input and outputs generated during the testbench. It saves the inputs provided to the `spmv` function from the C testbench into an input vector. And it saves the results from executing the `spmv` on those inputs into an output vector. Then it invokes a simulation tool on the generated RTL design. It provides the input testbench to this RTL design and compares the output results from the RTL design to those generated in the C testbench. If this matches, then the cosimulation passes. Otherwise, it fails.

The C/RTL cosimulation executes upon the RTL design generated from the Vivado® HLS tool in order to perform the cosimulation. The input data is provided from the C testbench. Thus, it can also be used to get the minimum, maximum, and average clock cycles numbers for the latency and interval of the `spmv` function. The cosimulation report provides these numbers upon completion.

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. 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.

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

Figure 6.5 illustrates the approximate manner in which the `spmv` function executes when pipelining the `L2 for` loop. Part 6.5 a) shows one iteration of the outer `L1 for` loop. Since the `L2 for` loop is pipelined, the figure shows each of the `L2 for` loop iterations for one iteration of the `L1 for` loop. Notice that they are overlapping; the next iteration of the `L2 for` loops starts before the previous iteration is completed. This is the source of the performance advantage that pipelining provides. The amount of time between two consecutive `L2 for` loop iterations is the initiation interval (II); this is typically reported as the number of cycles. Figure 6.5 b) shows the consecutive executions of the `L1 for` loop. These are done in a sequential manner though in each iteration the `L2 for` loop execution is performed in a pipelined manner.

When the designer specifies the pipeline directive on a loop without an explicit target II (e.g., `#pragma HLS PIPELINE`), the Vivado® HLS tool attempts to minimize the II as much as possible, i.e., it attempts to create a design with $II = 1$. This is not always possible due to utilization constraints. It is possible to specify a target II using an argument to the directive (e.g., `#pragma HLS PIPELINE II=3`).

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 initiation interval (II)? What happens to the results as you specify an II argument, and increase or decrease the target II?

Now consider pipelining the outer `L1 for` loop. In order to perform this optimization, the Vivado® HLS tool will attempt to fully unroll the inner `L2 for` loop. If full unrolling is possible, this removes any control flow due to calculating the loop bounds. In general, pipelining any outer `for` loop results in fully unrolling any inner loop.

Add a directive to pipeline the outermost `L1 for` for loop. 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 for` loop? How does this compare to the baseline design (no directives)? What is happening when you pipeline this outer loop (hint: check the synthesis log)?

Loop unrolling is another optimization that typically has a positive effect on performance (at the cost of increased number of resources). Fully unrolling a loop completely removes the control flow. But the designer can also partially unroll a loop. This typically exposes more parallelism by enabling the Vivado® HLS tool to schedule operations from consecutive iteration to occur in parallel (when dependency constraints allow for this).

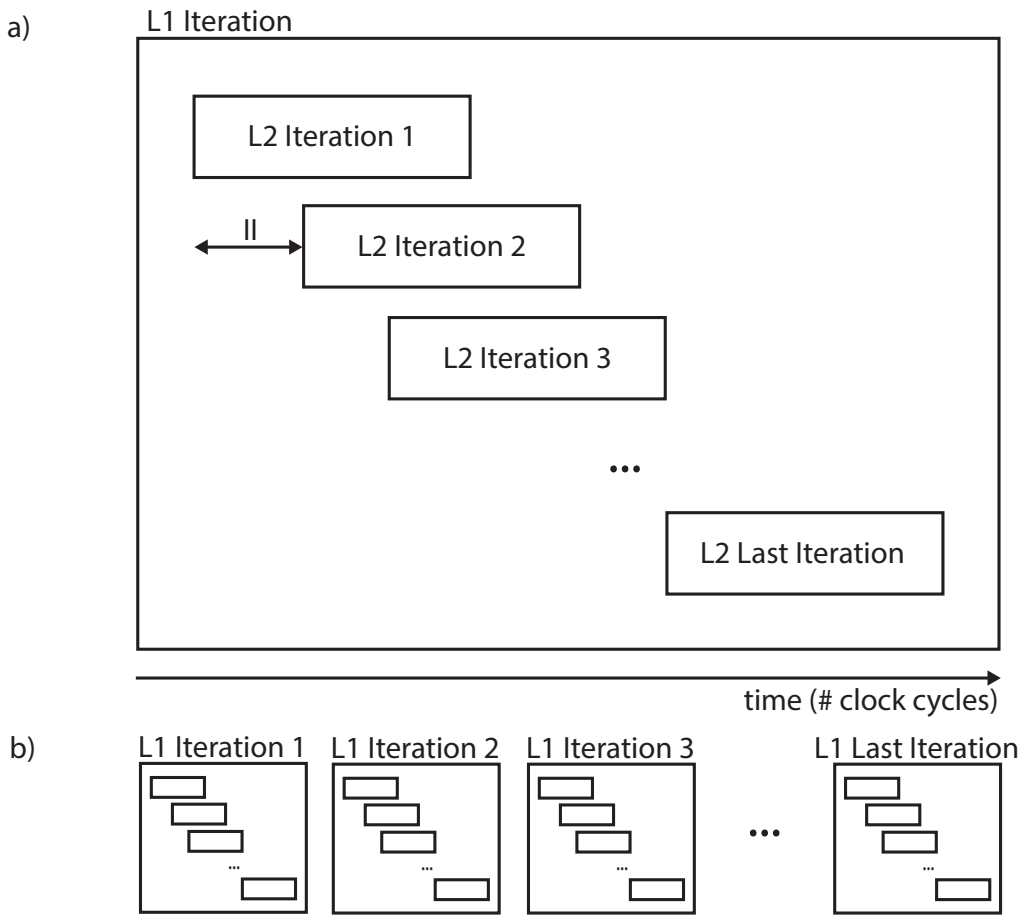


Figure 6.5: Part a) shows one iteration of the `L1 for` loop. Since the `L2 for` loop is pipelined, the execution of each of these iterations are overlapping. The `II` arrow denotes the amount of time between the start of consecutive iterations. Part b) shows the `L1 for` loop iterations. They are performed sequentially with each of the `L2 for` loop executions happening in a pipelined manner.

When performing loop unrolling, the designer must be careful to understand the additional constraints that this adds upon the Vivado® HLS tool. In particular, the designer should carefully consider the effects that this has upon memory accesses. In other words, does the amount of data that is read and written to any given array increase? In the `spmv` function, we read from three arrays on every iteration of the `L2 for` loop – `values`, `x`, and `columnIndex`. Thus, when we unroll this loop, we need to consider the additional read operations that occur upon these arrays.

Each iteration of the `L2 for` loop, accesses `values` and `columnIndex` using the variable `k`, which increases by 1 on every iteration. In fact, we know that we are accessing these arrays sequentially across the entire function, i.e., last iteration of `L2` and the first iteration of the following execution of `L2` (as we move to the next iteration of `L1`) are sequential. The access pattern of the `x` array is a bit more complicated and depends upon the locations of the non-zero elements in the input matrix $\mathbf{M}$. This makes it difficult to optimize these memory accesses. Therefore, we only consider the optimization of the arrays `values` and `columnIndex`.

Assume that all of the data in the array `values` is stored in one memory. Also assume that the entire array `columnIndex` is stored in memory, which is separate from the memory storing `values`. Finally, assume that we unroll the `L2 for` loop by a factor of four. In this case, each iteration of the unrolled loop requires four read operations from both memories. If the memories do not have four read ports, then the Vivado® HLS tool must sequentialize these read operations. This will reduce the performance. And we are not able to take advantage of all of the parallelism that is exposed by the unroll optimization.

We can eliminate the need for sequentialization by creating more read ports. The way to do this is through data partitioning, i.e., separate the `values` and `columnIndex` into multiple memories. This can be done manually by refactoring the code. Or it can be done automatically by the Vivado® HLS tool using the `ARRAY_PARTITION` directive. This directive splits the array into multiple smaller memories based upon the `factor` argument. For example, setting `factor = 2` splits the array into two memories, and `factor = 4` divides the array across four memories.

The next question is how exactly to divide the arrays. This can be done in a `block`, `cyclic`, or `complete` manner. This is specified using the `type` argument. A `block` partition takes consecutive elements of the array and puts them in the same memory. For example, the directive `#pragma HLS ARRAY_PARTITION variable=values block factor=2` will put the elements of the first half of the array `values` into one memory and the elements of the second half into another memory. A `cyclic` partition takes consecutive elements and puts them in different arrays. Thus, the directive `#pragma HLS ARRAY_PARTITION variable=values block factor=2` puts every even element of `values` into one memory, and every odd element into another separate memory.

Given that we are accessing the elements from the arrays `values` and `columnIndex` in a sequential manner, we want to make sure that each consecutive element is in a separate memory. This would be done using a `cyclic` partitioning. It should be clear that it is important to consider the data partitioning when we perform loop unrolling.

Loop unrolling can be used in combination with pipelining. Unrolling reduces the number of iterations of the loop while performing more work per iteration. The addition of pipelining enables the iterations to occur in an overlapping fashion.

It is also possible to perform pipelining, unrolling, and data partitioning in combination. However, the effects of each of these is not necessarily straightforward. To understand these effects, we encourage you to perform the following exercise:

> Synthesize the `spmv` design using the optimizations 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. The optimizations only consider `cyclic` partitioning for reasons outlined earlier. 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

If you performed the previous exercise, you should have seen that blindly applying optimization directives will not always provide you with the expected results. In fact, it is rarely effective. It is more important to understand the application under design, and to consider the effects of the optimization upon that design. Additionally, the designer needs to have an understanding of the synthesis tool. And while it is difficult (perhaps impossible?) to understand every detail of the Vivado® HLS tool, it is important to know the effects of the different directives.

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:

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

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

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

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

```
#define N 128
#define M 128
#define P 128

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 i of A */
    row: for(int i = 0; i < N; ++i) {
        /* for each column j of B */
        col: for(int j = 0; j < P; ++j) {
            #pragma HLS PIPELINE II=1
            /* compute (AB)i,j */
            int ABij = 0;// = C[i][j];
            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 outermost `for` loop iterates across the rows of $\mathbf{A}$; the `col for` loop iterates across the columns of $\mathbf{B}$; and the innermost `product for` loop multiplies the appropriate elements of $\mathbf{A}$ and $\mathbf{B}$ and accumulates them until it has the result for the element in $\mathbf{AB}$.

The result of the matrix product is

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

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 first `for` loop iterates over each row of the input matrix **A**. The second `for` loop iterates over each column of the input matrix **B**. The innermost `for` loop creates the product of the appropriate elements from the **A** and **B** matrices, and accumulates them into the final element $(\mathbf{AB})_{ij}$; one element of the final matrix **AB** is calculated after the completion of each execution of the innermost `for` loop.

We put a `pipeline` directive in the second `col for` loop with a target initiation interval $II = 1$. The result is that the innermost `for` loop is fully unrolled. It is certainly possible place the `pipeline` directive in different places throughout the function. Putting it outside all of the `for` loops will result in all of the `for` loops being completely unrolled. Placing it inside the first `row for` loop will unroll the `col` and `product for` loops. The tradeoff here is the complexity of the resulting architecture and its performance, i.e., this is a tradeoff between resource usage and performance.

Change the location of the `pipeline` directive. How does the location effect the resource usage? How does it change the performance? Which of the locations 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?

The other `array_reshape` directives modify the data layout for the input arrays **A** and **B**. You can think of this directive as performing two operations. The first partitions the array into separate memories. The second operation combines these memories back into a single memory. This is similar to performing the `array_partition` directive followed by the `array_map` directive. The end result is still mapped into one array, but there are a smaller number of elements that are caused by grouping some of the elements from the original array into one element in the new array.

There are multiple arguments to the `array_reshape` directive. The `variable` argument specifies the name of the array to perform the reshaping.The `dim` argument indicates the dimension of the array to perform reshaping upon. This is only appropriate for multidimensional arrays. Another argument is `type` which is one of `block`, `cyclic`, or `complete`. `Block` combines adjacent elements into one larger element; `cyclic` combines interleaved elements into larger elements; and `complete` makes all of the elements into one very large element. The final argument is `factor`. This is only appropriate when using `block` and `cyclic` reshaping. It determines the size of the new elements.

The first `array_reshape` directive works on the **A** array. It states that the second dimension should be completely partitioned. That is, all of the elements in the second dimension should be combined into a single element. This has the same effect as making **A** into a one dimensional array where each element consists of the elements corresponding to the entire row of **A**.

The second `array_reshape` directive performs a similar operation on the **B** array. But here, it does a complete partitioning on the first dimension. This has the effect of combining all of the elements corresponding to one column into a single element. Thus, it is a one dimensional array, and when we index into the array we get a full columns with of data with each index.

Performing these two `array_reshape` directives allows us to fetch an entire row of the **A** matrix and an entire column of the **B** matrix with only two memory accesses. One access on **A**; and one access on **B**.

> 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?

The size of the arrays can have a substantial effect on the optimizations that you wish to perform. The example uses relatively small arrays – square matrices of size 128×128. Some applications will use much smaller matrices. In this case, it may be possible to do some extreme operations, e.g., using the `pipeline` directive on the entire function, and not just one of the loops. As the size of the arrays increase, these operations quickly become infeasible; you simply do not have the resources for the same optimizations. Thus, you may need to move the `pipeline` directive into the inner `for` loops. And in this case you may wish to think about using the `unroll` directive with a small factor on the loops that are not pipelined.

> 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 a particular submatrix.

Many matrices can be naturally partitioned into blocks based upon the structure of the data. For example, a block diagonal matrix has is a square matrix whose blocks on the diagonal are also square matrices, and all of the remaining non-diagonal blocks are zero matrices.

We can use the notion of a block matrix to create a more efficient architecture. In particular, we stream the matrix blocks and perform partial product on these blocks. We call this a *streaming architecture* since we transfer the input data (and potentially the output data) one portion at a time rather than sending the entire input matrices at once, and the complete output matrix at the end of the computation. This is how we performed the matrix multiplication in the previous section.

Streaming architectures are common in many applications. Often times we will not get all of the data at once; for example, the **A** and **B** input matrices that we wish to multiply could come one row or one column at a time. This maybe be due to the way that the data is

being sampled – perhaps it is coming from a sensor or the previous computation generates the data in a specific manner.

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 7.2 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 7.2 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 7.2 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 7.2 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 7.2 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 7.2 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 7.3 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

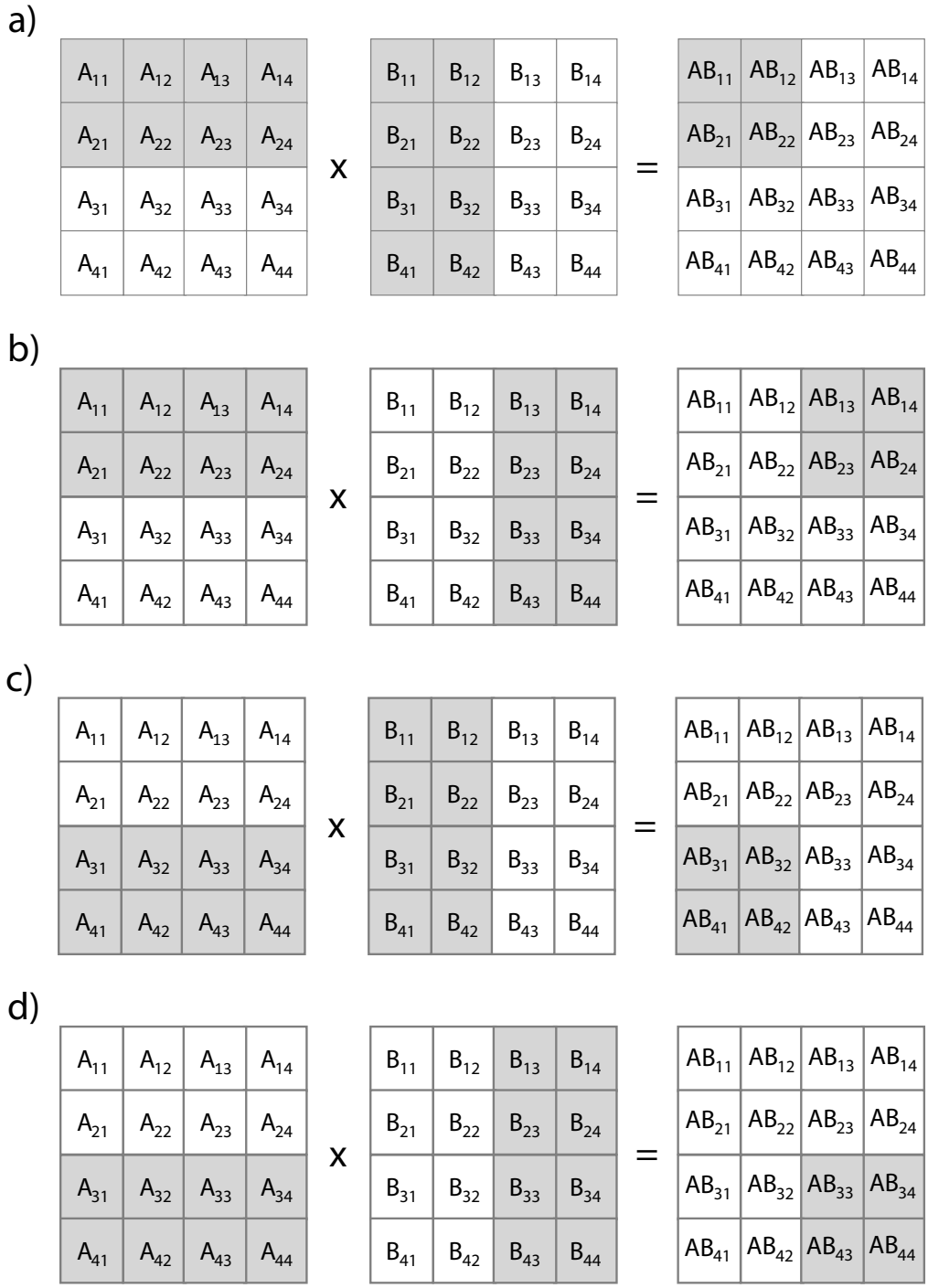


Figure 7.2: A streaming architecture to compute the matrix multiplication of two 4×4 matrices. The entire $\mathbf{AB}$ product requires four consecutive streams. Each time we take different set of two rows and two columns from the $\mathbf{A}$ and $\mathbf{B}$ matrices. The `BLOCK_SIZE` here is 2.

```
#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 7.3: 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.

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

The `SIZE` defines the number of rows and columns in the matrices to be multiplied. We limit this to square matrices though 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 `BLOCK_SIZE` $\times$ `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 `BLOCK_SIZE` $\times$ `BLOCK_SIZE` array. This holds the resulting values from one execution of the `matrmatmul` function.

The function prototype itself takes the two input 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. The *stream library* was developed since streaming is a common methodology for passing data in hardware design, yet this same operation can be developed in many different ways using the C programming language. In particular, it is difficult for the Vivado® HLS tool to infer streaming behaviors especially when the data types that the designer wishes to stream are complex. For example, coding up the seemingly simple `blockvec` data type in a streaming manner can result in a difficult to analyze code. The streaming behavior will not be obvious to the Vivado® HLS tool. This becomes even more difficult with more complex data types, and different code styles. Thus, the need for a builtin stream library.

The `hls::stream` class must always be passed by reference between functions, e.g., as we have done in the `blockmatmul` function in Figure 7.4.

```
#include "block_mm.h"

void blockmatmul(hls::stream<blockvec> & Arows, hls::stream<blockvec> & Bcols,
                 blockmat & ABpartial, DTYPE it) {
    #pragma HLS DATAFLOW
    blockvec m;
    blockmat out_temp;
    int counter = it % (SIZE/BLOCK_SIZE);
    blockvec tempA;
    blockvec tempB;

    static DTYPE A[BLOCK_SIZE][SIZE];
    if(counter == 0){ //only load the A rows when necessary
        loadA: for(int i = 0; i < SIZE; i++) {
            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++){
        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 7.4: The `blockmatmul` function takes a `BLOCK_SIZE` set of rows from $\mathbf{A}$ matrix, a `BLOCK_SIZE` set of columns from the $\mathbf{B}$ matrix, and creates a `BLOCK_SIZE` $\times$ `BLOCK_SIZE` partial result for the $\mathbf{AB}$ matrix. The first part of the code (denoted by the label `loadA`) stores the rows from $\mathbf{A}$ 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.

The code for executing one part of the streaming block matrix multiplication is shown in Figure 7.4. 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 `blockmatmat` function we send a `BLOCK_SIZE` of rows from the **A** and a `BLOCK_SIZE` of columns from the **B**. And 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 `blockmatmat` 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 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` class has an array that holds one element from each of the `BLOCK_SIZE` rows of the **A** matrix. Thus, in each call the 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 `blcokmatmul` function.

The `Bcols` stream variable is utilized in a very similar manner at 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 the function will have 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 `BLOCK_SIZE` $\times$ `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 `BLOCK_SIZE` $\times$ `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 inspectation of the code alone, we can see that this part has three nested `for` loops with a total of `SIZE` $\times$ `BLOCK_SIZE` $\times$ `BLOCK_SIZE` iterations. The

first part has SIZE × BLOCK_SIZE iterations; and the last part has BLOCK_SIZE × 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. In our case, moving the `pipeline` directive into the second `for` loop will certainly use more resources since this, by definition, will unroll the innermost loop. Whether this makes the 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,

$$interval(\texttt{blockmatmul}) \geq \max(interval(\texttt{loadA}), interval(\texttt{partialsum}), interval(\texttt{writeoutput})) \tag{7.5}$$

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 of *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 7.5 and 7.6. We split it across two figures to make it more readable since it is quite long function. 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 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 7.5. 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 7.3, 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 7.3). 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 7.6. 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 values to `strm_matrix1`. Recall that the values from the **A** matrix are used across several calls to the `blockmatmul` function. See Figure 7.2 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

```
#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 7.5: 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 7.6. This has a "software" version of matrix multiplication, and variable declarations and initializations.

```
int main() {
  //The beginning of the testbench is shown in the previous figure

  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 7.6: The second portion of the block matrix multiply testbench. The first part is shown in Figure 7.5. 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.

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, e.g., any of the code from Section **??**. 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?

Chapter 8

Prefix Sum and Histogram

Where does this best fit? As a separate combined chapter? Two separate chapters (probably not – it is too short). It could go in the Huffman Sorting chapter – in the radix sort section. But this make this already very long chapter longer, and I think might interrupt the flow of that chapter. Maybe this could be moved earlier in the book (perhaps after FIR or after CORDIC)? But then this chapter assumes the reader understands array partitioning which is best introduced in the DFT chapter. So in that sense, it is best after that. This could also be moved into a radix sort section in the sorting chapter. In this case, it needs to be better integrated.

8.1 Prefix Sum

Prefix sum is a common kernel used in many applications, e.g., recurrence relations, compaction problems, string comparison, polynomial evaluation, histogram, radix sort, and quick sort [8]. Prefix sum requires restructuring in order to create an efficient FPGA design.

Prefix sum is the cumulative sum of a sequence of numbers. Assume that the input sequence is stored in the array $in[]$, and the prefix sum is put into array $out[]$. In other words, `out[n]` is the summation of `in[0] + in[1] + in[2] + ··· + in[n-1] + in [n]`. The following shows the computation for the first four elements of the output array `out`.

$$
\begin{aligned}
out[0] &= in[0] \\
out[1] &= in[0] + in[1] \\
out[2] &= in[0] + in[1] + in[2] \\
out[3] &= in[0] + in[1] + in[2] + in[3] \\
&\cdots
\end{aligned}
$$

Figure 8.1 presents the baseline C code for prefix sum. The ideal optimization yields $II = 1$ for the loop in the code. Unfortunately, even for a relatively simple kernel like prefix sum, the designer must change the code in order to get an $II = 1$.

We start the optimization process using directives without changing the code. We start by performing loop unrolling. Ideally, each time we unroll the loop, the performance increases.

```
void prefixsum(int in[SIZE], int out[SIZE]) {
    out[0]=0;
    for(i=1;i<SIZE-1;i++){
        out[i]=out[i-1]+in[i]
    }
}
```

Figure 8.1: The initial code for implementing prefix sum. This requires restructuring and optimization in order to create an optimal hardware design.

```
void prefixsum(int in[SIZE], int out[SIZE]) {
    #pragma HLS ARRAY_PARTITION variable=out cyclic factor=4 dim=1
    #pragma HLS ARRAY_PARTITION variable=in cyclic factor=4 dim=1
    out[0]=0;
    for(int i=1; i<SIZE-1; i++) {
        #pragma HLS UNROLL factor=4
        #pragma HLS PIPELINE
        out[i]=out[i-1]+in[i]
    }
}
```

Figure 8.2: Optimizing the prefixsum code using `unroll`, `pipeline`, and `array_partition` directives.

If we unroll by a factor of two, then the performance doubles. A factor of four would increase the performance fourfold, i.e., the performance scales in a linear manner as it is unrolled. Unfortunately, this is not the case for the baseline C code for prefix sum. We must rewrite the code to allow unrolling to scale as expected.

> Unroll the `for` loop corresponding to the prefix sum code in Figure 8.1 by different factors, e.g., 2,3,4,6,8,16, etc.. How does the `prefixsum` function latency change? What are the trends in the resource usages? Why do you think you are seeing these trends?

Inevitably, the first optimizations that a designer will apply in this case will focus on the `for` loop. In particular, pipelining the `for` loop is often productive. And using the `unroll` directive on a `for` loop typically exposes parallelism that can be exploited by the Vivado® HLS tool. Finally, when working on arrays, we must be careful that the lack of memory ports do not limit the availability of the data, and cause serialization of the hardware design. Thus, it is important to properly partition the arrays `in` and `out` using the appropriate `factor` and ordering of the data. The pragmas in the code unroll the `for` loop by a factor of four, pipeline the loop, and partition the two arrays by a factor of four (in order to match the loop unrolling factor) and in a cyclic manner.

```
void prefixsum(int in[SIZE], int out[SIZE]) {
    #pragma HLS ARRAY_PARTITION variable=out cyclic factor=4 dim=1
    #pragma HLS ARRAY_PARTITION variable=in cyclic factor=4 dim=1
    A = in[0];
    for(i=0;i<SIZE;i++) {
        #pragma HLS UNROLL factor=4
        #pragma HLS PIPELINE
        A = A + in[i];
        out[i] = A;
    }
}
```

Figure 8.3: This code adds an intermediate variable `A` to hold the running sum from the previous `in[]` values. This simple restructuring breaks the dependency between `out[i]` and `out[i-1]` from the previous code, and enables the performance to scale with the unrolling factor.

One might expect that these optimizations would provide a 4× speedup over the original code in Figure 8.1. Yet, this is not the case. Even for this relatively "simple" code, the designer must restructure the code in order to meet theses expected performance improvements. In this case, the designer must break the data dependency between `out[i]` and `out[i-1]`.

Synthesize the code from Figure 8.2. How does the function latency of this optimized code compare to the code from Figure 8.1.

In order to obtain the desired ratio between unrolling and speed, the designer should eliminate the dependency on the `out[]` array. This can be done with a simple code restructuring. The code in Figure 8.3 slightly modifies the code to introduce an intermediate variable `A` that holds the running sum of the prefix sum through the previous iteration, i.e., it is the prefix sum value for the previous iteration (`out[i-1]`. Adding this variable eliminates the dependency, and allows the code to scale as expected as the loop is unrolled, i.e., more unrolling increases the performance.

This code seems superfluous when writing for a pure software implementation. Why introduce another variable? This would only be additional work, which would likely increase the runtime. But, as we describe in the following, it makes a better hardware architecture.

Figure 8.4 a) provides a graphical description of the hardware architecture from synthesizing the code in Figure 8.2. Figure 8.4 b) shows the same architecture, but when compiling the code from Figure 8.3. This `A` variable maps to a hardware register. A register is helpful because a it can be read to and written from on every cycle. Unlike a memory, which has limited number of read and write ports, the register value can be read from and subsequently sent to a large number of places that need that data on every cycle with very limited penalty. The only major issues is wire routing, which in the grand scheme of things is not much ad-

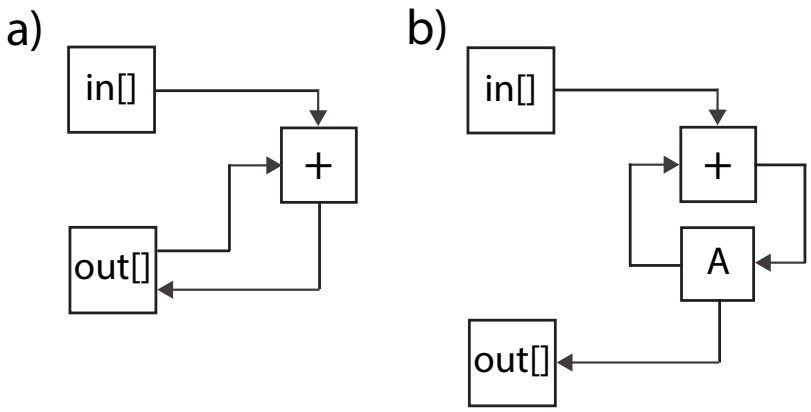


Figure 8.4: Part a) displays an architecture corresponding to the code in Figure 8.2. The dependence on the `out[]` array does not allow for full optimization. Adding a variable to keep the running sum eliminates this dependence (as shown in the code in Figure 8.3). This code provides a better base for optimization using unrolling, pipelining, and array partitioning directives.

ditional overhead in terms of resources. Nor does it typically incur a large penalty in terms of performance.

> Synthesize the code from Figure 8.3. How does the function latency of this code compare to the code change as you vary the unroll factor? Do you need to change the values in any of the other directives? Why?

Could add a whole part about doing reduction. Probably not necessary. Should have something here about what to do with floating point accumulation. This is fundamentally more problematic than what's above (which is relatively easily handled by improving store-¿load optimization.

The goal of this section is to show that even a small change in the code can have a significant effect on the hardware design. And that this change may not necessarily be intuitive from software design standpoint. Yet, it plays an important role in optimizing the overall design.

8.2 Histogram

Creating a histogram is a common function used in image processing, signal processing, data processing (e.g., databases) and many other domains. It makes a probability distribution by dividing the data under study into a number of bins, i.e., a set of intervals, and calculating the frequency of values that fall into each of these intervals.

Figure 8.5 provides a simple example that explains the histogram operation. Given a set of input data, a histogram counts the occurrence of each element (Bin=elements,

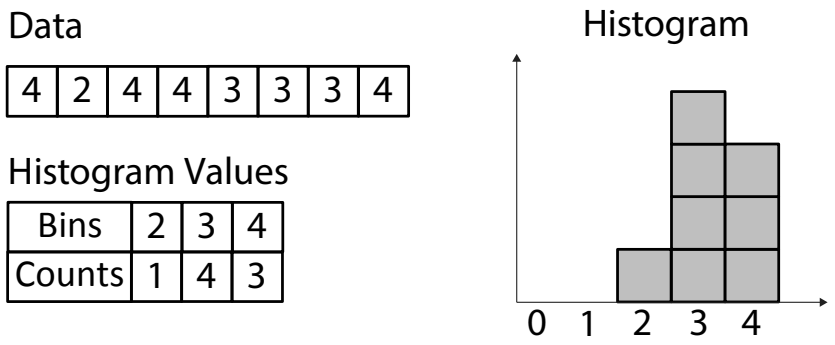


Figure 8.5: An example of the histogram operation. The data set provides a list of values. Those are summed into a set of intervals; in this case the intervals are integer numbers between [0,4]. The histogram calculates the number of values in each of these intervals.

```
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    int val;
    for(int i = 0; i < INPUT_SIZE; i++) {
        #pragma HLS PIPELINE
        val = in[i];
        hist[val] = hist[val] + 1;
    }
}
```

Figure 8.6: Original code for calculating the histogram value. The `for` loop iterates across the `input` array, use each of those values to index into the `hist` array, and increment that current value.

Counts=occurrence). The result is plotted in a graph. Figure 8.6 provides an initial code for the histogram function.

To optimize this code for a hardware implementation, we can direct the Vivado® HLS tool to exploit instruction level parallelism by applying the `pipeline` pragma to the body of the `for` loop. Ideally, we reach an initiation interval of one cycle ($II = 1$). Unfortunately, we are unable to achieve this due to the read-after-write (RAW) dependency. This is due to the fact that we are reading from the `hist` array and writing to the same array in every iteration of the loop.

Figure 8.7 shows the hardware architecture for the code in Figure 8.6. You can see that the `hist` array has a read and write operation. The `val` variable is used as the index into the `hist` array, and the variable at that index is read out, incremented, and written back into the same location.

Eliminating the read after write dependency allows the `pipeline` directive to be more effective. This is done in the code in Figure 8.8. The architecture uses an accumulator register (`accu`), comparator, and the old value of current address. This architecture updates the accumulator if the current address and old address are the same. Otherwise, it writes the

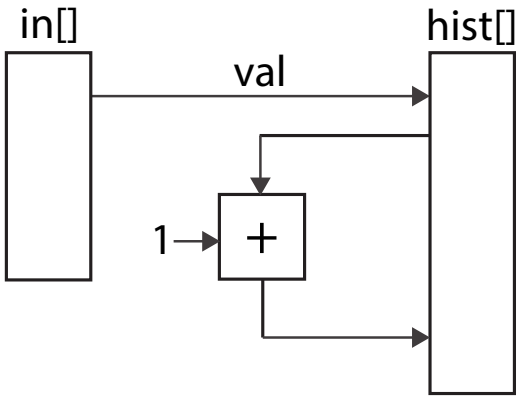


Figure 8.7: An architectural sketch corresponding the the code in Figure 8.6. The `val` data from the `in` array is used to index into the `hist` array. This data is incremented and stored back into the same location.

accumulator value to the old address and reads a new accumulator value from the current address. In this case, the code reads from and writes to a different address. Thus, the RAW dependence can be ignored. However, the Vivado® HLS tool must be explicitly told that there is no such dependency. This is done through the `dependence` directive. This restructured code enables a lower initiation interval.

Synthesize the code from Figure 8.6 and Figure 8.8. What is the initiation interval (II) in each case? What happens when you remove the `dependence` directive from the code in Figure 8.8? How does the loop interval change in both cases? What about the resource usage?

A pictorial description of the restructured code from Figure 8.8 is shown in Figure 8.9. Not all of the operations are shown here, but the major idea of the function is there. You can see the two separate `if` and `else` regions (denoted by dotted lines). The `acc` variable is replicated twice in order to make the drawing more readable; the actual design will only have one register for that variable. The figure shows the two separate datapaths for the `if` and the `else` clause with the computation corresponding to the `if` clause on the top and the `else` clause datapath on the bottom.

We can further optimized the histogram kernel by running several of the architectures in parallel to boost the throughput. We start with the optimized architecture as described in Figures 8.8 and 8.9. The goal is to create multiple versions of this architecture, which we call a processing element (PE), split the input data into different streams, and the split input data independently into each processing element. Then we merge the results from all of the processing elements. This idea is shown in Figure 8.10.

We can view each processing element as a task. The tasks can be run in parallel since the computation does not depend upon each other. This is an example of task level parallelism.

```
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    int acc = 0;
    int i, val;
    int old = in[0];
    #pragma HLS DEPENDENCE variable=hist array intra RAW false
    for(i = 0; i < INPUT_SIZE; i++) {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val)
            acc = acc + 1;
        else {
            hist[old] = acc;
            acc = hist[val] + 1;
        }
        old = val;
    }
    hist[old] = acc;
}
```

Figure 8.8: Removing the read after write dependency from the `for` loop. This requires an `if/else` structure that may seem like it is adding unnecessary complexity to the design. However, it allows for more effective pipelining despite the fact that the datapath is more complicated.

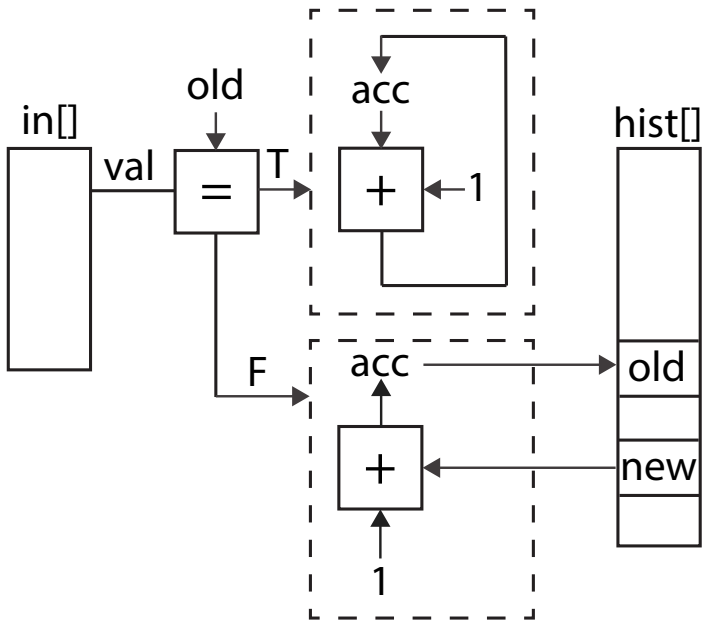


Figure 8.9: A depiction of the datapath corresponding to the code in Figure 8.8. There are two separate portions corresponding to the `if` and `else` clauses. The figure shows the important portions of the computation, and leaves out some minor details.

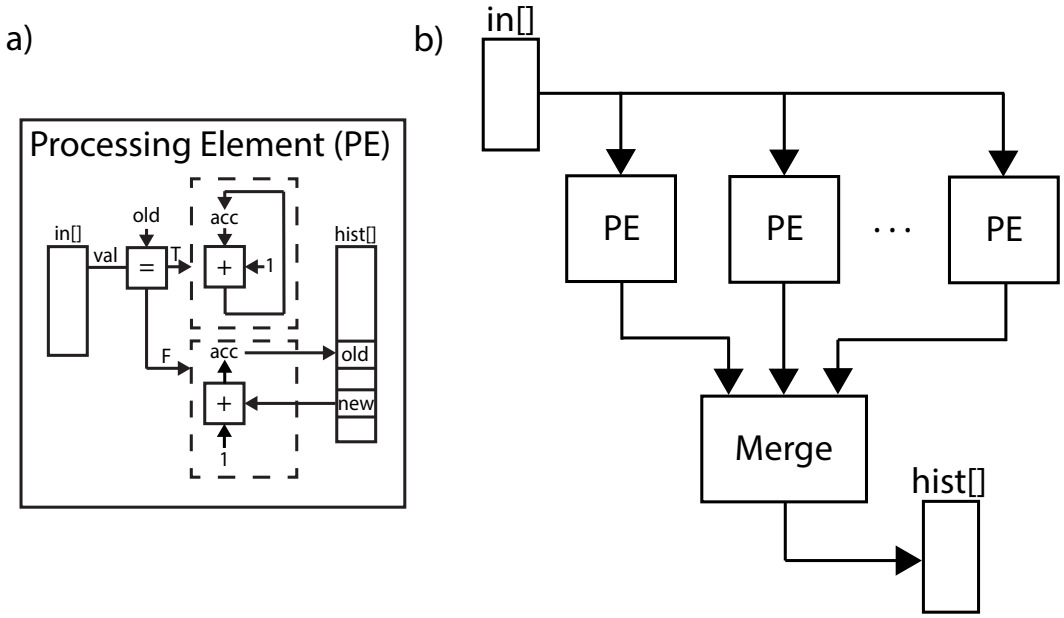


Figure 8.10: The processing element (PE) in Part a) is the same architecture as shown in Figure 8.9. We can replicate that architecture many times, and split the input data from the `in` array across these processing elements. Then we merge the data to create the final histogram.

This is accomplished in a relatively simple manner using functions. The final `merge` task is a separate function that takes as input the output of all of the processing elements. Each of the `PE` tasks creates a histogram, but only on the data that it received. Thus, the `merge` task goes through each of these generated histograms, and accumulates them into one final histogram. This is done by stepping through each element of each of the histograms, and summing the elements into the final output `hist` array.

The input data should be divided evenly across these processing elements. One way to accomplish this is to divide the data into separate arrays and have each processing element only read from those arrays. If possible, this is the most efficient technique as it eliminates the contention on the memory ports for the memory that holds the array. This can be done using directives or manually in the code. We choose to do the latter.

The code for implementing this architecture is shown in Figure 8.11. The function `PE` serves the role of the processing element. The computation here is equivalent to that done in the previous architecture (Figures 8.8 and 8.9 where the function is called `histogram`; we rename it here since it is only performing part of the overall histogram operation). We add some additional code to initialize the local storage in each of the processing elements. Each `PE` function has an input array `in`, which is the subset of the data that this processing element should perform a histogram upon, and the output array `hist`, which is the number of elements of each value in `in`, i.e., the histogram of `in`.

The `merge` function takes as input each of the histogram outputs from the processing elements. In our code example in Figure 8.11, we have only two processing elements, thus the merge has two input arrays `hist1` and `hist2`. This can easily be extended to handle more processing elements. The `merge` function iterates through each output from the processing elements, and sums them into a separate array `final`, which is final histogram output for the entire function.

The final function in the code is the top level function `histogram`. This assumes that the data that we wish to perform the histogram upon is already divided into two arrays `inputA` and inputB. The function creates two separate processing element (`PE`) functions, and feeds `inputA` and inputB into each of them. The output of the two `PE` functions are stored in the arrays `hist1` and `hist2`. These are feed into the `merge` function which combines them and stores the result in the array `hist`, which is the final output of the top level function `histogram`.

How would you change the code in order to have more than two processing elements? What happens to the throughput and task interval?

We use the `dataflow` directive in the `histogram` function in order to perform task level pipelining. Here the tasks are the three functions: two instances of the `PE` functions, and the `merge` function. The two `PE` functions can be performed in parallel since they work on independent data. The `merge` function uses the results of from these two `PE` functions. Thus, it must start after these two functions complete. Thus, the `dataflow` directive essentially creates a two stage task pipeline with the `PE` functions in the first stage, and the `merge` function in the second stage.

```
void PE(int in[INPUT_SIZE], int hist[VALUE_SIZE]){
    int i, val;
    unsigned int acc=0;
    for(i = 0; i < VALUE_SIZE; i++){
        hist[i] = 0; }

    #pragma HLS DEPENDENCE variable=hist array intra RAW false
    int old = in[0];
    for(i = 0; i < INPUT_SIZE; i++) {
    #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val) {
            acc = acc + 1; }
        else {
            hist[old] = acc;
            acc = hist[val] + 1; }
            old = val;
        }
    hist[old] = acc;
}

void merge(int hist1[VALUE_SIZE], int hist2[VALUE_SIZE], int final[VALUE_SIZE]) {
    for(int i = 0; i < VALUE_SIZE; i++){
        final[i] = hist1[i] + hist2[i]; }
}

//Top level function
void histogram(int inputA[INPUT_SIZE], int inputB[INPUT_SIZE], int hist[VALUE_SIZE]){
    #pragma HLS DATAFLOW
    int hist1[VALUE_SIZE];
    int hist2[VALUE_SIZE];

    PE(inputA, hist1);
    PE(inputB, hist2);
    merge(hist1, hist2, hist);
}
```

Figure 8.11: Another implementation of histogram that uses task level parallelism and pipelining. The histogram operation is split into two sub tasks, which are executed in the two `PE` functions. These results are combined in the final histogram result using the `merge` function. The `histogram` function is the top level function that connects these three functions together.

The initiation interval of the entire `histogram` function depends upon the maximum initiation interval of the two stages. The two `PE` functions are in the first stage, and they are equivalent functions, thus they will have the same initiation interval, call it II_{PE}. The `merge` function will have another initiation interval (II_{merge}). The initiation interval of the `histogram` function $II_{histogram} \geq \max(II_{PE}, II_{merge})$.

> What happens when you add or change the locations of the `pipeline` directives? For example, is it beneficial to add a `pipeline` directive to the `for` loop in the `merge` function? What is the result of moving the `pipeline` directive into the `PE` function, i.e., hoisting it outside of the `for` loop where it currently resides?

The goal of this section was to walk through the optimization of another small, but important operation, i.e., the histogram, which is important atomic operation for many applications. The key takeaways are that it is important to understand the architecture that is generated. The implementation that works well in software (e.g., that shown in Figures 8.6 and 8.7) does not necessarily map well to an FPGA. The first set of optimizations shown in Figures 8.8 and 8.9 are almost counterintuitive; they add in a `if/else` control structure into the `for` loop, which generally is a bad idea. But here, it allows us to break a read after write (RAW) dependency, which ultimately leads to better performance. The final set of optimizations break the histogram into smaller subtasks, which allow for task level parallelism and pipelining (see Figures 8.10 and 8.11).

Chapter 9

Video Systems

9.1 Basics of Video Processing

Video Processing is a common application for FPGAs. One reason is that common video data rates match well the clock frequencies that can achieved with modern FPGAs. For instance, the common High-Definition TV format known as FullHD or 1080P60 video requires $1920\frac{\texttt{pixels}}{\texttt{line}} * 1080\frac{\texttt{lines}}{\texttt{frame}} * 60\frac{\texttt{frames}}{\texttt{second}} = 124,416,000\frac{\texttt{pixels}}{\texttt{second}}$.

When encoded in a digital video stream, these pixels are transmitted along with some blank pixels at 148.5 MHz, and can be processed in a pipelined FPGA circuit at that frequency. Higher data rates can also be achieved by processing multiple samples per clock cycle. Details on how digital video is transmitted will come in Section 9.1.2. Another reason is that video is mostly processed in *scanline order* line-by-line from the top left pixel to the lower right pixel, as shown in Figure 9.1. This predictable order allows highly specialized memory architectures to be constructed in FPGA circuits to efficiently process video without excess storage. Details on these architectures will come in Section 9.1.3

Video processing is also a good target application for HLS. Firstly, video processing is typically tolerant to processing latency. Many applications can tolerate several frames of processing delay, although some applications may limit the overall delay to less than one frame. As a result, highly pipelined implementations can be generated from throughput and

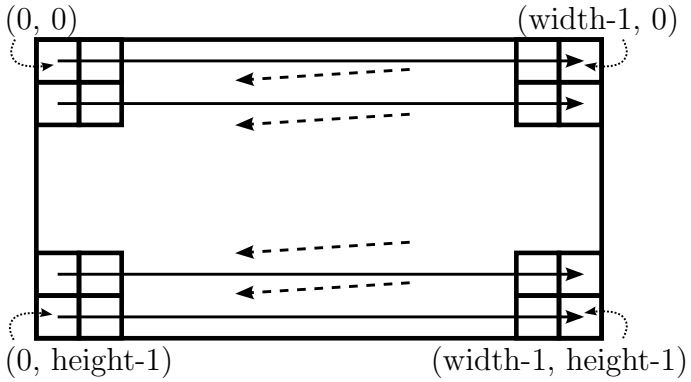


Figure 9.1: Scanline processing order for video frames.

clock constraints in HLS with little concern for processing latency. Secondly, video algorithms are often highly non-standardized and developed based on the personal taste or intuition of an algorithm expert. This leads them to be developed in a high-level language where they can be quickly developed and simulated on sequences of interest. It is not uncommon for FullHD video processing algorithms to run at 60 frames per second in an FPGA system, one frame per second in synthesizable C/C++ code running on a development laptop, but only one frame per hour (or slower) in an RTL simulator. Lastly, video processing algorithms are often easily expressed in a nested-loop programming style that is amenable to HLS. This means that many video algorithms can be synthesized into an FPGA circuit directly from the C/C++ code that an algorithm developer would write for prototyping purposes anyway

9.1.1 Representing Video Pixels

Many video input and output systems are optimized around the way that the human vision system perceives light. One aspect of this is that the cones in the eye, which sense color, are sensitive primarily to red, green, and blue light. Other colors are perceived as combinations of red, green, and blue light. As a result, video cameras and displays mimic the capabilities of the human vision system and are primarily sensitive or capable of displaying red, green, and blue light and pixels are often represented in the RGB colorspace as a combination of red, green, and blue components. Most commonly each component is represented with 8 bits for a total of 24 bits per pixel, although other combinations are possible, such as 10 or even 12 bits per pixel in high-end systems.

A second aspect is that the human visual system interprets brightness with somewhat higher resolution than color. Hence, within a video processing system it is common to convert from the RGB colorspace to the YUV colorspace, which describes pixels as a combination of Luminance (Y) and Chrominance (U and V). This allows the color information contained in the U and V components to be represented independently of the brightness information in the Y component. One common video format, known as YUV422, represents two horizontally adjacent pixels with two Y values, one U value and one V value. This format essentially includes a simple form of video compression called *chroma subsampling*. Another common video format, YUV420, represents four pixels in a square with 4 Y values, one U value and one V value, further reducing the amount of data required. Video compression is commonly performed on data in the YUV colorspace.

A third aspect is that the rods and codes in eye are more sensitive to green light than red or blue light and that the brain primarily interprets brightness primarily from green light. As a result, solid-state sensors and displays commonly use a mosaic pattern, such as the *Bayer* pattern[4] which consists of 2 green pixels for every red or blue pixel. The end result is that higher resolution images can be produced for the same number of pixel elements, reducing the manufacturing cost of sensors and displays.

Video systems have been engineered around the human visual system for many years. The earliest black and white video cameras were primarily sensitive to blue-green light to match the eye's sensitivity to brightness in that color range. However, they were unfortunately not very sensitive to red light, as a result red colors (such as in makeup)

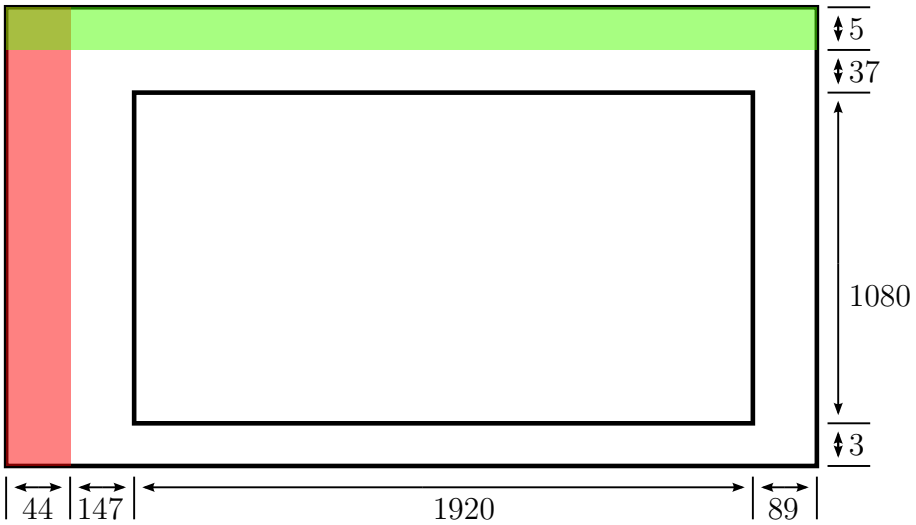


Figure 9.2: Typical synchronization signals in a 1080P60 high definition video signal.

didn't look right on camera. The solution was decidedly low-tech: actors wore garish green and blue makeup.

9.1.2 Digital Video Formats

In addition to representing individual pixels, digital video formats must also encode the organization of pixels into video frames. In many cases, this is done with synchronization or *sync* signals that indicate the start and stop of the video frame in an otherwise continuous sequence of pixels. In some standards (such as the Digital Video Interface or *DVI*) sync signals are represented as physically separate wires. In other standards (such as the Digital Television Standard BTIR 601/656) the start and stop of the sync signal is represented by special pixel values that don't otherwise occur in the video signal.

Each line of video (scanned from left to right) is separated by a *Horizontal Sync Pulse*. The horizontal sync is active for a number of cycles between each video line. In addition, there are a small number of pixels around the pulse where the horizontal sync is not active, but there are not active video pixels. These regions before and after the horizontal sync pulse are called the *Horizontal Front Porch* and *Horizontal Back Porch*, respectively. Similarly, each frame of video (scanned from top to bottom) is separated by a *Vertical Sync Pulse*. The vertical sync is active for a number of lines between each video frame. Note that the vertical sync signal only changes at the start of the horizontal sync signal. There are also usually corresponding *Vertical Front Porch* and *Vertical Back Porch* areas consisting of video lines where the vertical sync is not active, but there are not active video pixels either. In addition, most digital video formats include a Data Enable signal that indicates the active video pixels. Together, all of the video pixels that aren't active are called the *Horizontal Blanking Interval* and *Vertical Blanking Interval*. These signals are shown graphically in Figure 9.2.

The format of digital video signals is, in many ways, an artifact of the original analog television standards, such as NTSC in the United States and PAL in many European countries. Since the hardware for analog scanning of Cathode Ray Tubes contained circuits with limited slew rates, the horizontal and vertical sync intervals allowed time for the scan recover to the beginning of the next line. These sync signals were represented by a large negative value in the video signal. In addition, since televisions were not able to effectively display pixels close to the strong sync signal, the front porches and the back porches were introduced to increase the amount of the picture that could be shown. Even then, many televisions were designed with *overscan*, where up to 20% of the pixels at the edge the frame were not visible.

The typical 1080P60 video frame shown in Figure 9.2 contains a total of $2200 * 1125$ data samples. At 60 frames per second, this corresponds to an overall sample rate of 148.5 Million samples per second. Most modern FPGAs can comfortably process at this clock rate, often leading to 1 sample-per-clock cycle architectures. Systems that use higher resolutions, such as 4K by 2K for digital cinema, or higher frame rates, such as 120 or even 240 frames per second often require more the one sample to be processed per clock cycle. Remember that such architectures can often be generated by unrolling loops in HLS (see Section 1.4.2). Similarly, when processing lower resolutions or frame rates, processing each sample over multiple clocks may be preferable, enabling operator sharing. Such architectures can often be generated by increasing the II of loops.

For instance, the code shown in Figure 9.3 illustrates a simple video processing application that processes one sample per clock cycle with the loop implemented at II=1. The code is written with a nested loop over the pixels in the image, following the scanline order shown in 9.1. An II=3 design could share the rescale function computed for each component, enabling reduced area usage. Unrolling the inner loop by a factor of 2 and partitioning the input and the output arrays by an appropriate factor of 2 could enable processing 2 pixels every clock cycle in an II=1 design. This case is relatively straightforward, since the processing of each component and of individual pixels is independent. More complicated functions might not benefit from resource sharing, or might not be able to process more than one pixel simultaneously.

A high-speed computer vision application processes small video frames of 200 * 180 pixels at 10000 frames per second. This application uses a high speed sensor interfaced directly to the FPGA and requires no sync signals. How many samples per clock cycle would you attempt to process? Is this a good FPGA application? Write the nested loop structure to implement this structure using HLS.

9.1.3 Line Buffers and Frame Buffers

Video processing algorithms typically compute an output pixel or value from a nearby region of input pixels, often called a *window*. Conceptually, the window scans across the input

```
#include "video_common.h"

unsigned char rescale(unsigned char val, unsigned char offset, unsigned char scale) {
  return ((val − offset) ∗ scale) >> 4;
}

rgb_pixel rescale_pixel(rgb_pixel p, unsigned char offset, unsigned char scale) {
#pragma HLS pipeline
        p.R = rescale(p.R, offset, scale);
        p.G = rescale(p.G, offset, scale);
        p.B = rescale(p.B, offset, scale);
        return p;
}

void video_filter_rescale(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
                          rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH],
                          unsigned char min, unsigned char max) {
#pragma HLS interface ap_hs port = pixel_out
#pragma HLS interface ap_hs port = pixel_in
row_loop:
  for (int row = 0; row < MAX_WIDTH; row++) {
  col_loop:
    for (int col = 0; col < MAX_HEIGHT; col++) {
#pragma HLS pipeline
      rgb_pixel p = pixel_in[row][col];
                    p = rescale_pixel(p,min,max);
      pixel_out[row][col] = p;
    }
  }
}
```

Figure 9.3: Code implementing a simple video filter.

image, selecting a region of pixels that can be used to compute the corresponding output pixel. For instance, Figure 9.4 shows code that implements a 2-dimensional filter on a video frame. This code reads a window of data from the input video frame (stored in an array) before computing each output pixel.

In Figure 9.4, there is the code `int wi = row+i-1; int wj = col+j-1;`. Explain why these expressions include a '-1'. Hint: Would the number change if the filter were 7x7 instead of 3x3?

A key observation about adjacent windows is that they often overlap, implying a high locality of reference. This means that pixels from the input image can be buffered locally or cached and accessed multiple times. Unfortunately, the above code typically results in a poor synthesis result. If the input array is implemented in a memory, then repeated reads of the array result in a communication bottleneck. The input array also cannot be implemented as a stream, since accesses to the array are not in sequential order. However, by refactoring the code to read each input pixel exactly once and store the result in a local memory, a better result can be achieved. In video systems, the local buffer is also called a *line buffer*, since it typically stores several lines of video around the window. Line buffers are typically implemented in FPGA Block RAM, while window buffers are implemented using Flip Flops. Refactored code using a line buffer is shown in Figure 9.5. Note that for an NxN image filter, only N-1 lines need to be stored in line buffers.

The line buffer and window buffer memories implemented from the code in Figure 9.5 are shown in Figure 9.6. Each time through the loop, the window is shifted and filled with one pixel coming from the input and two pixels coming from the line buffer. Additionally, the input pixel is shifted into the line buffer in preparation to repeat the process on the next line. Note that in order to process one pixel each clock cycle, most elements of the window buffer must be read from and written to every clock cycle. In addition, after the 'i' loop is unrolled, each array index to the window array is a constant. In this case, Vivado® HLS will convert each element of the array into a scalar variable (a process called *scalarization*). Most of the elements of the window array will be subsequently implemented as Flip Flops. Similarly, each row of the line_buffer is accessed twice (being read once and written once). The code explicitly directs each row of the line_buffer array to be partitioned into a separate memory. For most interesting values of MAX_WIDTH the resulting memories will be implemented as one or more Block RAMs. Note that each Block RAM can support two independent accesses per clock cycle.

Line buffers are a special case of a more general concept known as a *reuse buffer*, which is often used in stencil-style computations[**?**]. High-level synthesis of reuse buffers and line buffers from code like Figure 9.4 is an area of active research. See, for instance [5].

```
#include "video_common.h"

rgb_pixel filter(rgb_pixel window[3][3]) {
  const char h[3][3] = {{1, 2, 1}, {2, 4, 2}, {1, 2, 1}};
  int r = 0, b = 0, g = 0;
  i_loop:
  for (int i = 0; i < 3; i++) {
        j_loop:
    for (int j = 0; j < 3; j++) {
      r += window[i][j].R * h[i][j];
      g += window[i][j].G * h[i][j];
      b += window[i][j].B * h[i][j];
    }
  }
  rgb_pixel output;
  output.R = r / 16;
  output.G = g / 16;
  output.B = b / 16;
  return output;
}

void video_2dfilter(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
                                rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface ap_hs port = pixel_out
#pragma HLS interface ap_hs port = pixel_in
  rgb_pixel window[3][3];
  row_loop:
  for (int row = 0; row < MAX_HEIGHT; row++) {
        col_loop:
    for (int col = 0; col < MAX_WIDTH; col++) {
#pragma HLS pipeline
      for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
          int wi = row + i - 1;
          int wj = col + j - 1;
          if (wi < 0 || wi >= MAX_HEIGHT || wj < 0 || wj >= MAX_WIDTH) {
                                              window[i][j].R = 0;
                                              window[i][j].G = 0;
                                              window[i][j].B = 0;
                                        } else
            window[i][j] = pixel_in[wi][wj];
        }
      }

      pixel_out[row][col] = filter(window);
    }
  }
}
```

```
void video_2dfilter_linebuffer(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
                               rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface ap_hs port=pixel_out
#pragma HLS interface ap_hs port=pixel_in
  rgb_pixel window[3][3];
  rgb_pixel line_buffer[3][MAX_WIDTH];
#pragma HLS array_partition variable=line_buffer complete dim=1
  row_loop:
  for (int row = 0; row < MAX_HEIGHT; row++) {
        col_loop:
    for (int col = 0; col < MAX_WIDTH; col++) {
#pragma HLS pipeline
      for (int i = 0; i < 3; i++) {
        window[i][2] = window[i][1];
        window[i][1] = window[i][0];
      }

      window[2][0] = (line_buffer[2][col] = line_buffer[1][col]);
      window[1][0] = (line_buffer[1][col] = line_buffer[0][col]);
      window[0][0] = (line_buffer[0][col] = pixel_in[row][col]);

      if (row == 0 || col == 0 || row == (MAX_HEIGHT - 1) ||
          col == (MAX_WIDTH - 1)) {
        pixel_out[row][col].R = 0;
        pixel_out[row][col].G = 0;
        pixel_out[row][col].B = 0;
      } else {
        pixel_out[row][col] = filter(window);
      }
    }
  }
}
```

Figure 9.5: Code implementing a 2D filter with an explicit line buffer.

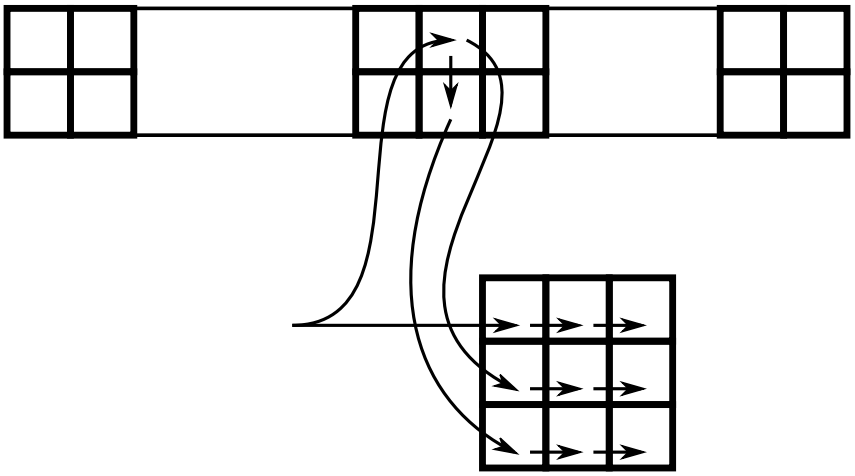


Figure 9.6: Memories implemented in Figure 9.5.

Vivado® HLS includes hls::line_buffer<> and hls::window_buffer<> classes that simplify the management of window buffers and line buffers.

For a 3x3 image filter, operating on 1920x1080 images with 4 bytes per pixel, How many FPGA Block RAMs are necessary to store each video line?

9.1.4 Causal Filters

The filter implemented in Figure 9.5 reads a single input pixel and produces a single output pixel each clock cycle. However, the output is computed from the window of previously read pixels, which is 'up and to the left' of the pixel being produced. As a result, the output image is shifted 'down and to the right' relative to the input image. The situation is analogous to the concept of causal and acausal filters in signal processing. Most signal processing theory focuses on causal filters because only causal filters are practical for time sampled signals (e.g. where x[n] = x(n*T) and y[n] = y(n*T)).

A *causal* filter $h[n]$ is a filter where $\forall k < 0, h[k] = 0$. A finite filter $h[n]$ which is not causal can be converted to a causal filter $\hat{h}[n]$ by delaying the taps of the filter so that $\hat{h}[n] = h[n - D]$. The output of the new filter $\hat{y} = x \otimes \hat{h}$ is the same as a delayed output of the old filter $y = x \otimes h$. Specifically, $\hat{y}[n] = y[n - D]$.

Figure 9.7: Video filter timelines

> Prove the fact in the previous aside using the definition of the convolution for $y = x \otimes h$: y[n] = $\sum_{k=-\infty}^{\infty} x[k] * h[n-k]$

For the purposes of this book, most variables aren't time-sampled signals and the times that individual inputs and outputs are created may be determined during the synthesis process. For systems involving time-sampled signals, we treat timing constraints as a constraint during the HLS implementation process. As long as the required task latency is achieved, then the design is correct.

9.1.5 Extending the iteration domain

In many video processing algorithms, the spatial shift introduced in the code above is undesirable and needs to be eliminated. Although there are many ways to write code that solves this problem, a common way is known as *extending the iteration space*. In this technique, the loop bounds are increased by a small amount so that the first input pixel is read on the first loop iteration, but the first output pixel is not written until later in the iteration space. A modified version of the filter code is shown in Figure 9.8.

9.1.6 Boundary conditions

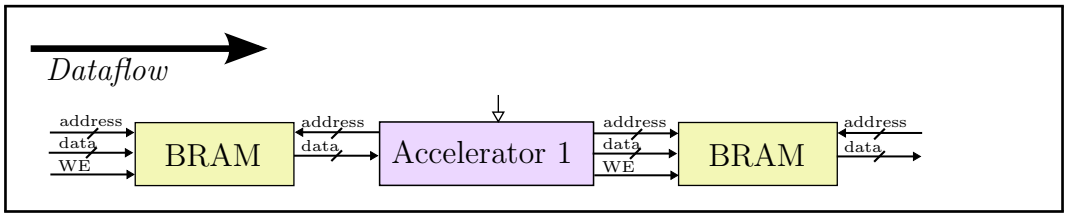
In most cases, the processing window contains a region of the input image. However, near the boundary of the input image, the filter may extend beyond the boundary of the input image. Depending on the requirements of different applications, there are many different ways of accounting for the behavior of the filter near the boundary. Perhaps the simplest way to account for the boundary condition is to compute a smaller output image that avoids requiring the values of input pixels outside of the input image. However, in applications where the output image size is fixed, such as Digital Television, this approach is generally unacceptable. In addition, if a sequence of filters is required, dealing with a large number images with slightly different sizes can be somewhat cumbersome. The code in Figure 9.5 creates an output with the same size as the input by padding the smaller output image with a known value (in this case, the color black). Alternatively, the missing values can be synthesized, typically in one of several ways.

- Missing input values can be filled with a constant
- Missing input values can be filled from the boundary pixel of the input image.
- Missing input values can be reconstructed by reflecting pixels from the interior of the input image.

```
void video_2dfilter_linebuffer_extended(
  rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
  rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface ap_hs port=pixel_out
#pragma HLS interface ap_hs port=pixel_in
  rgb_pixel window[3][3];
  rgb_pixel line_buffer[3][MAX_WIDTH];
#pragma HLS array_partition variable=line_buffer complete dim=1
  row_loop: for(int row = 0; row < MAX_HEIGHT+1; row++) {
    col_loop: for(int col = 0; col < MAX_WIDTH+1; col++) {
#pragma HLS pipeline II=1
      rgb_pixel pixel;
      if(row < MAX_HEIGHT && col < MAX_WIDTH) {
        pixel = pixel_in[row][col];
      }

      for(int i = 0; i < 3; i++) {
        window[i][2] = window[i][1];
        window[i][1] = window[i][0];
      }

      window[2][0] = (line_buffer[2][col] = line_buffer[1][col]);
      window[1][0] = (line_buffer[1][col] = line_buffer[0][col]);
      window[0][0] = (line_buffer[0][col] = pixel);

      if(row >= 1 && col >= 1) {
        int outrow = row-1;
        int outcol = col-1;
        if(outrow == 0 || outcol == 0 ||
          outrow == (MAX_HEIGHT-1) || outcol == (MAX_WIDTH-1)) {
          pixel_out[outrow][outcol].R = 0;
          pixel_out[outrow][outcol].G = 0;
          pixel_out[outrow][outcol].B = 0;
        } else {
          pixel_out[outrow][outcol] = filter(window);
        }
      }
    }
  }
}
```

Figure 9.8: Code implementing a 2D filter with an explicit line buffer. The iteration space is extended by 1 to allow the filter to be implemented without a spatial shift.

Figure 9.9: Examples of the effect of different kinds of boundary conditions.




```
void video_filter(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
        rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface ap_memory port = pixel_out // The default
#pragma HLS interface ap_memory port = pixel_in // The default
```

Figure 9.10: Integration of a video design with BRAM interfaces.

Of course, more complicated and typically more computationally intensive schemes are also used.

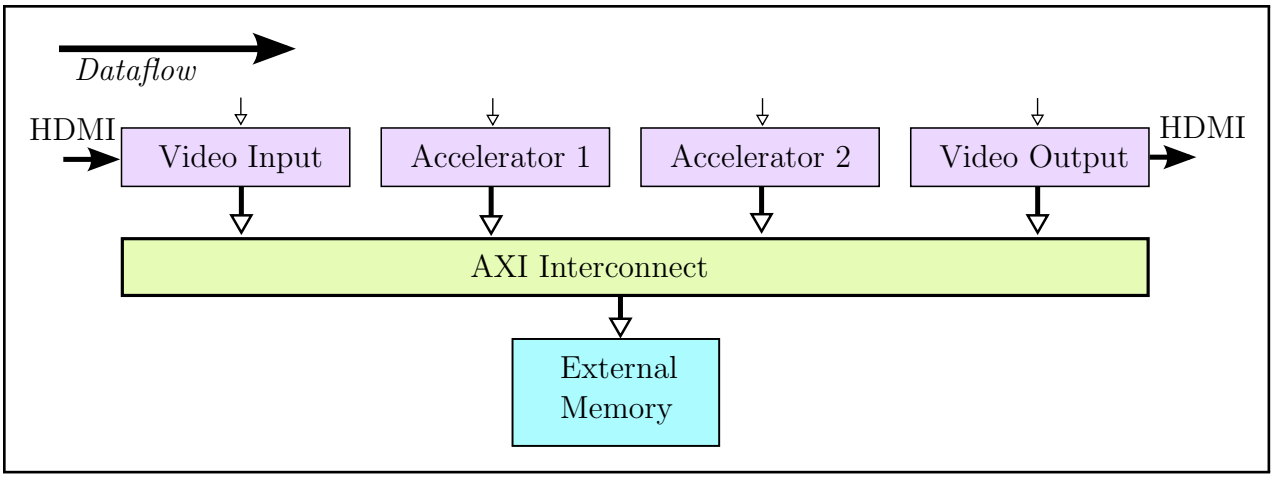
A simple way of writing code to handle boundary conditions is shown in Figure **??**. In this code, values input to the filter are conditionally read from the window buffer, resulting in a multiplexer for each tap of the filter. Similarly, the code in Figure **??** also results in multiplexers, since each read from the window buffer is at a variable address. However, the multiplexers in this case have more inputs and typically require more hardware resources to implement.

An alternative technique with reduced resource usage is shown in Figure **??**. In this code, the boundary condition is handled when data is written into the window buffer and the window buffer is shifted in a regular pattern. In this case, there are only N multiplexers, instead of N*N, resulting in significantly lower resource usage.

9.2 Video processing system architectures

Up to this point, we have focused on building video processing applications without concern for how they are integrated into a system. In each of the examples, such as Figure 9.8, the bulk of the processing occurs within a loop over the pixels and can process one pixel per clock when the loop is active. In this section we will discuss some possibilities for system integration.

By default, Vivado® HLS generates a simple memory interface for interface arrays. This interface consists of address and data signals and a write enable signal in the case of a write interface. Each read or write of data is associated with a new address and the expected latency through the memory is fixed. It is simple to integrate such an interface with on-chip memories created from Block RAM resources as shown in Figure 9.10. However Block RAM resources are generally a poor choice for storing video data because of the large size of each frame, which would quickly exhaust the Block RAM resources even in large expensive devices.



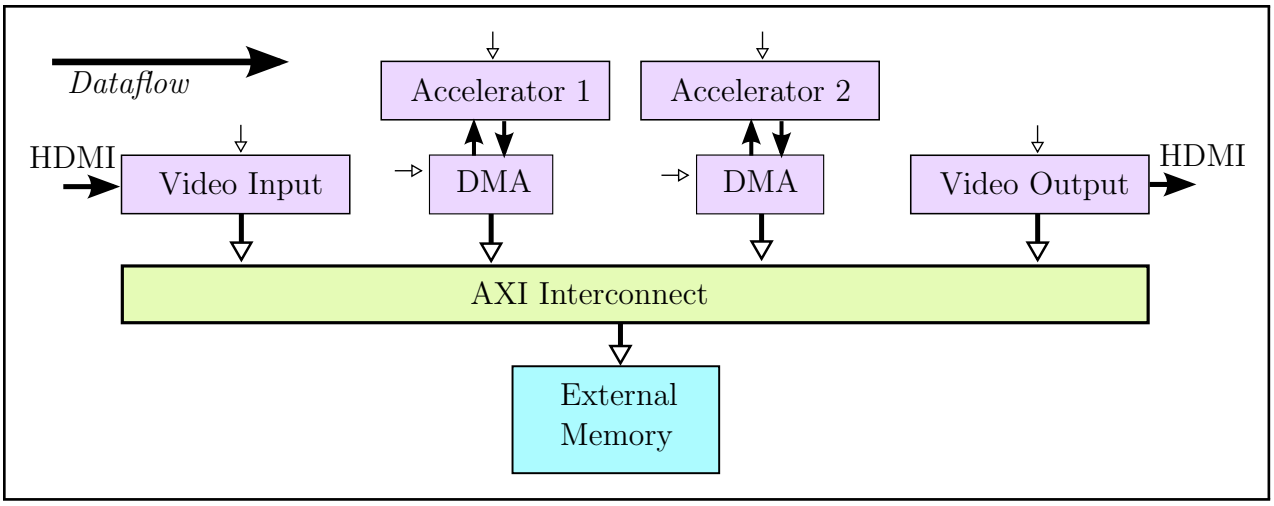

```
void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH],
      pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface m_axi port = pixel_out
#pragma HLS interface m_axi port = pixel_in
```

Figure 9.11: Integration of a video design with external memory interfaces.

For 1920x1080 frames with 24 bits per pixel, how many Block RAM resources are required to store each video frame? How many frames can be stored in the Block RAM of the FPGA you have available?

A better choice for most video systems is to store video frames in external memory, typically some form of double-data-rate (DDR) memory. Typical system integration with external memory is shown in Figure 9.11. An FPGA component known as *external memory controller* implements the external DDR interface and provides a standardized interface to other FPGA components through a common interface, such as the ARM AXI4 slave interface [**?**]. FPGA components typically implement a complementary master interface which can be directly connected to the slave interface of the external memory controller or connected through specialized *AXI4 interconnect* components. The AXI4 interconnect allows multiple multiple master components to access a number of slave components. This architecture abstracts the details of the external memory, allowing different external memory components and standards to be used interchangeably without modifying other FPGA components.

Although most processor systems are built with caches and require them for high performance processing, it is typical to implement FPGA-based video processing systems as shown in 9.11 without on-chip caches. In a processor system, the cache provides low-latency access to previously accessed data and improves the bandwidth of access to external memory by always reading or writing complete cache lines. Some processors also use more complex mechanisms, such as prefetching and speculative reads in order to reduce external memory latency and increase external memory bandwidth. For most FPGA-based video processing systems simpler techniques leveraging line buffers and window buffers are sufficient to avoid




```
void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH],
      pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface s_axi port = pixel_out
#pragma HLS interface s_axi port = pixel_in
```

Figure 9.12: Integration of a video design with external memory interfaces through a DMA component.

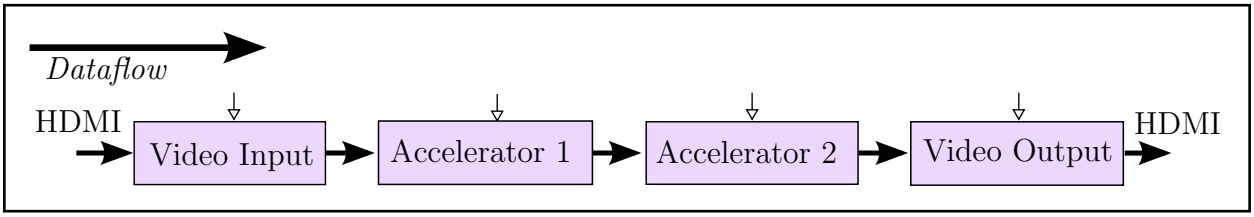
fetching any data from external memory more than once, due to the predictable access patterns of most video algorithms. Additionally, Vivado® HLS is capable of scheduling address transactions sufficiently early to avoid stalling computation due to external memory latency and is capable of statically inferring burst accesses from consecutive memory accesses.

An alternative external memory architecture is shown in Figure 9.12. In this architecture, an accelerator is connected to an external Direct Memory Access (DMA) component that performs the details of generating addresses to the memory controller. The DMA provides a stream of data to the accelerator for processing and consumes the data produced by the accelerator and writes it back to memory. In Vivado® HLS, there are multiple coding styles that can generate a streaming interface, as shown in Figure 9.13. One possibility is to model the streaming interfaces as arrays. In this case, the C code is very similar to the code seen previously, but different interface pragmas are used. An alternative is to model the streaming interface explicitly, using the `hls::stream<>` class. In either case, some care must be taken that the order of data generated by the DMA engine is the same as the order in which the data is accessed in the C code.

One advantage of streaming interfaces is that they allow multiple accelerators to be composed in a design without the need to store intermediate values in external memory. In some cases, FPGA systems can be built without external memory at all, by processing pixels as they are received on an input interface (such as HDMI) and sending them directly to an output interfac, as shown in Figure 9.14. Such designs typically have accelerator throughput requirements that must achieved in order to meet the strict real-time constraints at the external interfaces. Having at least one frame buffer in the system provides more flexibility to build complex algorithms that may be hard to construct with guaraunteed throughput.

```
void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH],
                  pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface ap_hs port = pixel_out
#pragma HLS interface ap_hs port = pixel_in

void video_filter(hls::stream<pixel_t> &pixel_in,
                  hls::stream<pixel_t> &pixel_out) {
```

Figure 9.13: Coding styles for modelling streaming interfaces in HLS.




```
void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH],
       pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
#pragma HLS interface ap_hs port = pixel_out
#pragma HLS interface ap_hs port = pixel_in
```

Figure 9.14: Integration of a video design with streaming interfaces.

A frame buffer can also simplify building systems where the input and output pixel rates are different or potentially unrelated (such as a system that receives an arbitrary input video format and outputs an different arbitrary format).

Chapter 10

Sorting Algorithms

10.1 Introduction

Notation: it seems that the standard notation is that the array to sort is denoted as A. We will follow that notation here.

There are basic elements. Compare and swap (sorting networks), insertion cell, merge, select value. Should there be a discussion about this here? Or after presenting all the algorithms? or not at all?

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 iterations selects the next unsorted element. It searches across the sorted (sub) array, and places it in the appropriate spot. 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 sub array (since it is only one element). The next step is to consider element `A[1]`, and place it into the sorted sub array such that the new sub array (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 sub array (i.e., all the previously considered elements of `A`) remains sorted.

Equation 10.1 gives a step by step view of the array `A[] = {3, 2, 5, 4, 1}`. In each step, the element under consideration is underlined. The starred (*) element is the one that

was previously moved into place.

$$
\begin{array}{llllll}
\texttt{A[]} = \{\underline{3}, & 2, & 5, & 4, & 1\} \\
\texttt{A[]} = \{3*, & \underline{2}, & 5, & 4, & 1\} \\
\texttt{A[]} = \{2*, & 3, & \underline{5}, & 4, & 1\} \\
\texttt{A[]} = \{2, & 3, & 5*, & \underline{4}, & 1\} \\
\texttt{A[]} = \{2, & 3, & 4*, & 5, & \underline{1}\} \\
\texttt{A[]} = \{1*, & 2, & 3, & 4, & 5\}
\end{array}
\tag{10.1}
$$

The first line is trivial. We consider only the first value `3` which makes a sub array with one element. Any sub array with one element is in sorted order. The second line places the second value `2` into the sorted sub array. The end result is that the value `2` is placed into the first element of the sorted sub array, 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 sub array. In this case, since `A[2] = 5`, it is already in its correct location. Thus, nothing needs to happen. The fourth line considers the value `4`. This is moved into its appropriate place, which shifts one element 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.

Despite it relative simplicity, insertion sort has a number of advantages.

- **Stability:** This refers to the property that if two items in the input data are equal on the element being sorted that they will appear in the same order on the output. For example, assume that we are sorting based upon age using a set of records with both names and ages. In the input data, John appears before Jane, and both are 25 years old. Stability states that John will appear on the output before Jane.
- **Online:** The algorithm allows for data to be sorted as it is received. This is particularly important for hardware implementations that are pipelined or streaming.
- **In-place:** A list with `n` elements can be sorted using `n` memory elements.
- **Adaptive:** It is efficient for data that is already relatively sorted. For example, if the data is already sorted, it will run in linear time.

Because of these properties, insertion sort is often used as a base case for sorting small arrays. For example, a set of data will be recursively broken down into a number of smaller arrays, and then these small arrays will be sorted using insertion sort. Then they will be combined using another algorithm (e.g., merge sort).

10.2.1 Basic Insertion Sort Implementation

Figure 10.1 shows the C code for insertion sort, which is a fairly optimized C implementation. The outer `for` loop 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 this `for` loop starts by copying the current element that we wish to insert into the sorted subarray (i.e., `A[i]`) into the `index`

```
void insertion_sort(DTYPE A[SIZE])
{
    //L1:
    int i, j, index;

    for(i = 1; i < SIZE; i++)
    {
      //L2:
      index = A[i];
      j = i;

      while(j > 0 && A[j-1] > index)
      {
        //L3:
        A[j] = A[j-1];
        j--;
      }
      A[j] = index;
    }
}
```

Figure 10.1: 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.

variable. Then, it goes through a nested `while` loop. This loop walks down the lower part of array `A` (which is the sorted sub array) looking for the appropriate location to place the value `index`. We continue to iterate down the list looking for the location for element `index` while `j > 0` (which insures that we do not run off the bottom of the array) and `A[j-1] > index` (which indicates that `index` – the item we want to insert – is still smaller than current element in the sorted sub array). As this condition holds, we shift the elements of the sorted sub array. This will make room for the insertion of `index` when we eventually find its correct location. The shifting is done in the statement `A[j] = A[j-1]`. Once we have found the correct location for `index`, we copy its value into the proper location (`A[j] = index`) and continue to the next iteration of the outer `for` loop. After the completion of iteration `i`, the elements from `A[0]` to `A[i]` are in sorted order.

The code in Figure 10.1 is an straightforward implementation without any optimizations. We can optimize it using different Vivado® HLS directives. Some common optimization directives are `PIPELINE` which exploits instruction level parallelism, `UNROLL` which vectorizes loops, and `PARTITION` which divides the specified array into multiple memories (BRAMs). We denote three potential locations for these directives: `L1`, `L2`, and `L3`. For example, we can direct the HLS tool to exploit instruction level parallelism by applying the `PIPELINE` directive to the body of the inner for loop at point `L3`; similarly, we can apply other optimizations at the labels `L1`, `L2`, and `L3`. Unfortunately, as we will shortly see, designers cannot rely on these directives alone, and must often write special code, which we call restructured code, to generate the best results. This restructured code requires substantial hardware design expertise [22, 34]. 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 10.1.

Table 10.1: Optimization of the basic `insertion_sort` function implementation through directives. Each line inidicates the directives used at the different labels in the code from Figure 10.1. The exact performance and area numbers are left as an exercise to the reader.

	Directives	Category	II	Period	Slices
1	`L3: PIPELINE II=1`	slow/small	?	?	?
2	`L3: UNROLL factor=2` `ARRAY_PARTITION cyclic factor=2`	slow/small	?	?	?
3	`L2: PIPELINE II=1`	slow/small	?	?	?
4	`L2: UNROLL FACTOR=2` `ARRAY_PARTITION cyclic factor=2`	slow/small	?	?	?
5	`L1: PIPELINE II=1` `ARRAY_PARTITION complete`	faster/large	?	?	?

Table 10.1 shows five different ways to optimize the code from Figure 10.1. In each case, we add directives at the specified labels (`L1, L2, L3`). We categorize the performance and area results from into two groups: 1) slow/small and 2) fast/large. Ideally, we want to generate faster and smaller designs from HLS. The first four designs are very slow and

have small area. Design 5 achieves higher performance with unrealistically large area due to aggressive HLS optimizations. The area and performance results are not specified; we leave the implementation of these directives as an exercise to the reader.

> Table 10.1 provides five different ways to optimize the code from Figure 10.1. In each case, the provided directives should be inserted at appropriate locations as specified by the labels. Synthesize each of these designs and determine the initiation interval (II), clock period, and required number of slices. Do the results match the expected categories (slow/small and fast/large)? Why or why not? What would happen if you combined the directives from multiple rows into one design?

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 – high performance and low-area – it is typically required to rewrite code in a manner that will create an efficient hardware architecture. This is often very different than code that will create efficient software.

10.2.2 Insertion Cells

One efficient method to perform insertion sorting in hardware uses a linear array of insertion cells [37, 6, 31, 3]. Each cell has one input and one output, and it holds one element of the array that is being sorted. The input to the cell is compared to the local element of the cell, and the smaller of the two is passed to the output of the cell. The number of cells is equal to the number of items that should be sorted. This is effective for hardware since each cell can work in a parallel, pipelined manner.

Figure 10.2 shows the hardware architecture for an insertion cell sorting implementation. Each cell (e.g., the ith `cell`) compares its input (`in`) with the value in current register (`local`). The smaller of data from `local` and `in` is given to the output `out`. In other words, `out = min(in,local)`. This output is then provided as input to the next insertion cell in the linear array (`cell` $i+1$). In turn, this data is compared to the local data in `cell` $i+1$. Each insertion cell in the linear array performs this same operation, and ultimately the smallest data of all of the elements in the array is provided as output of the last insertion cell `cell` $N-1$. The process continues as the next element is feed into the first insertion cell `cell` 0. This is propagated into its appropriate place as a `local` element in an insertion cell. And the next smallest data from the array is output from the last insertion cell. This means that the data from the linear array is output from smallest to largest. This can be changed with modifications to the `cell` code.

The code for one insertion cell is shown in Figure 10.3. The code uses a streaming interface by declaring the input and output variables as an `hls::stream` type. `DTYPE` is a parameterizable data type; you can consider it to be an `unsigned int` for these examples. The `local` variable is the register that holds the value for the cell. It is `static` because we

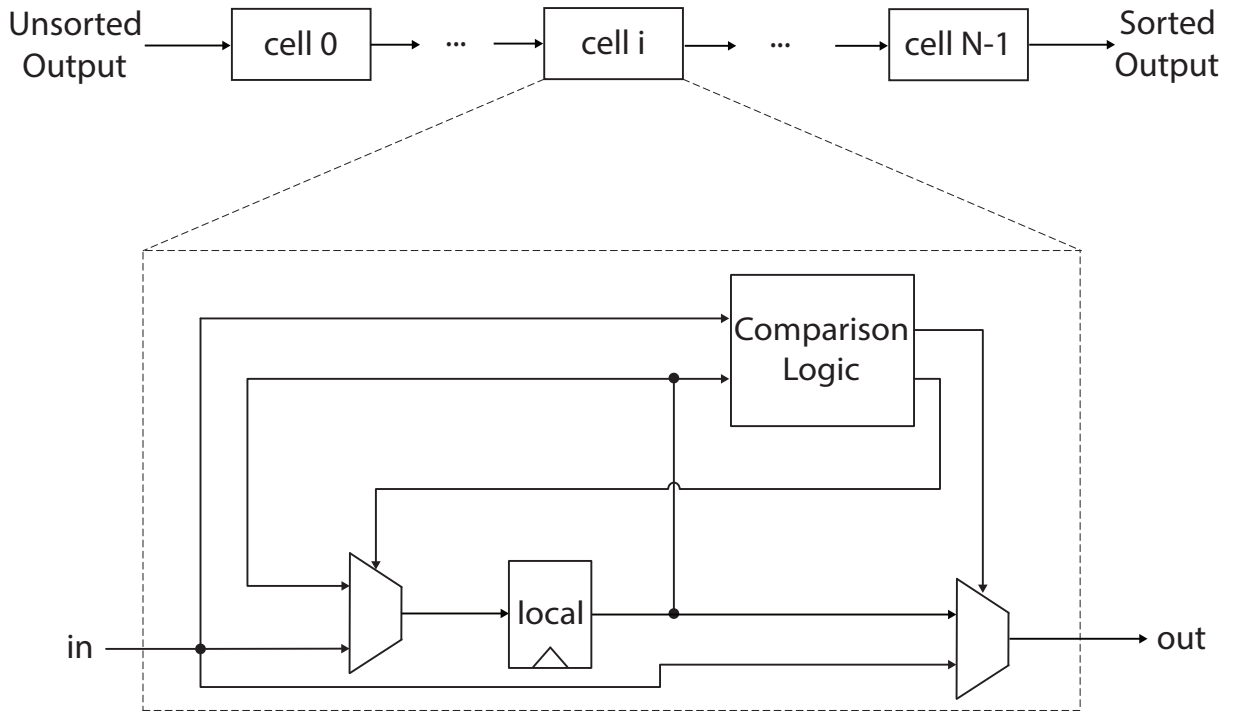


Figure 10.2: The architecture of one insertion cell. Each cell holds exactly one value in the register `local`. On each execution it receives an input value `in`, compares that to its `local` value, and writes the smaller of the two to `output`. Sorting `N` values requires `N` cells.

```
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 10.3: 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 10.2. 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 10.4: 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.

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., instantiating the same `cell` function N times) would cause a problem since each cell must have a separate static variable, i.e., one static variable cannot be shared across N functions.

The variable `in_copy` holds the result of a read from the `in` stream. This value is compared to the `local` value held in the cell's register. If `in_copy > local`, then we keep the value from `in_copy` in our `local` register, and write the previous value from the `local` register to the output. Otherwise, we write the `in_copy` to the `out` stream and leave the value in the `local` register the same.

Linking the insertion cells together is straightforward. The function `insertion_cell_sort` in Figure 10.4 shows the code for sorting eight elements. Expanding this to larger number of elements simply requires replicating more `cell` functions, additional `hls::stream out` 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)?

We perform functional pipelining across each of the eight `cell` functions using the `DATAFLOW` directive. This enables us to overlaps the execution of the sorting. In essence, it creates an eight stage pipeline where each stage performs one `cell` function.

How many cycles is required to necessary 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 10.5 shows the code for the testbench. The following provides a high level overview of the testbench code. It starts by initializing some random data. Then the testbench feeds the data into the insertion cell array by calling the `insertion_cell_sort` function multiple

times. After that, the testbench provides the same data to the `insertion_sort` function. The code for the `insertion_sort` function is not shown in this testbench, but it is the same as shown in Chapter 10.2.1 in Figure 10.1. Finally, the testbench compare the results of these two insertion sort implementations. The testbench passes if every element in both of the output arrays is equal; otherwise it fails.

Now we fill in some of the important details for the testbench. First, the `SIZE` constant is set to the number of elements that we wish to sort. 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.

We define an array for the initial data `input`. The input data is generated randomly using the `rand()` function. You can change the data by modifying the argument in the `srand(20)`; the argument is the seed. Change it to any other integer and the data will be different.

The testbench first runs the data through the insertion cell sorting. This is done by calling `insertion_cell_sorting` function `SIZE*2` times. The first `SIZE` times feeds in the data into the insertion cell arrays. The next `SIZE` calls are done to get the data out of the cell arrays. They come out one at a time starting with the smallest elements. This data is placed into the array `cell_output`.

After sorting the data using insertion cell sort, we use the basic insertion sort algorithm described in Chapter 10.2.1. While it is not shown here, it is the exact same code as presented in Figure 10.1. The input data is sorted in-place. That is, the sorted results are in the `input` array after the function completes.

The final portion of the testbench compares the data in the `input` and `cell_output` arrays element by element. If any of the elements are not equal, the `fail` variable is set to `1`. This `fail` variable is the return value; it indicates whether the testbench passes or not.

10.3 Merge Sort

Merge sort is a stable, divide and conquer algorithm invented by John von Neumann in 1945 [27]. The algorithm has two parts. The first part divides each element into its own sublist or subarray. The second part repeatedly merges these subarrays into a new sorted subarray until there is only one list.

Merging is the most important and time consuming part of the algorithm, and thus we focus on its implementation. Dividing the elements into individual arrays is a straightforward operation that simply requires treating each element as an individual array. So while we will create a complete merge sort algorithm in hardware, you will see that the divide portion is implicit in the way that we treat the incoming unsorted data.

Many implementations of the merge sort algorithm use recursion. However, most high-level synthesis tools do not support recursion or it is supported is a limited manner. Thus, we must refactor the code in order to eliminate the recursive function calls. And while the core idea for the FPGA implementation remains, the code will look substantially different from a typical "software" implementation.

The merging process combines two sorted arrays into one larger sorted array. The process

```
int main ()
{
  int fail = 0;
  DTYPE input[SIZE];
  DTYPE cell_output[SIZE] = {0};

  hls::stream<DTYPE> in;
  hls::stream<DTYPE> out;

  //generate random data to sort
  if(DEBUG)
    printf("==============Begin Initial 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)
      printf("%d\t", input[i]);
  }
  if(DEBUG)
    printf("\n==============End Initial Data==============\n");

  //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();
      if(DEBUG)
        printf("HW[%d]= %d\t", i, cell_output[i]);
    }
    else //then send in dummy data to flush pipeline
    {
      in.write(MAX_NUMBER);
      insertion_cell_sort(in, out);
      cell_output[i-SIZE] = out.read();
      if(DEBUG)
        printf("HW[%d]=%d\t", i, cell_output[i-SIZE]);
    }
  }

  //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)
  {
    printf("\n==========Begin Hardware Output==========\n");
    for(int i = 0; i < SIZE; i++)
    {
      printf("%d\t", cell_output[i]);
    }
    printf("\n==========End Hardware Output==========\n");
    for(int i = 0; i < SIZE; i++)
    {
      if(input[i] != cell_output[i])
        fail = 1;
      printf("golden=%d, hw=%d\t", input[i], cell_output[i]);
    }
    printf("\n");

    if(fail == 0)
      printf("PASS\n");
    else
      printf("FAIL\n");
  }

  return fail;
}
```

Figure 10.5: The testbench for the `insertion_cell_sort` function. It creates random data, sorts the data using two different implementation of insertion sorting (`insertion_cell_sort` and `insertion_sort` from the previous section), compares the output of these two sorting implementations, and returns whether these output results exactly match.

of merging is sometimes called the "two finger algorithm". Equation 10.2 describes the process. Their are two input arrays `in1` and `in2`. These are merged into the output array `out`. Both of the input arrays are in sorted order.

We start with a "finger" on the first element of each array - elements `3` and `1` in arrays `in1` and `in2`, respectively. We underline these elements in order to show where the fingers are placed, but they are always on the first element of each array unless the array is empty.

$$
\begin{array}{lll}
\texttt{in1[]} = \{\underline{3}, 4, 6, 7\} & \texttt{in2[]} = \{\underline{1}, 2, 5, 8\} & \texttt{out[]} = \{\ \} \\
\texttt{in1[]} = \{\underline{3}, 4, 6, 7\} & \texttt{in2[]} = \{\underline{2}, 5, 8\} & \texttt{out[]} = \{1\} \\
\texttt{in1[]} = \{\underline{3}, 4, 6, 7\} & \texttt{in2[]} = \{\underline{5}, 8\} & \texttt{out[]} = \{1, 2\} \\
\texttt{in1[]} = \{\underline{4}, 6, 7\} & \texttt{in2[]} = \{\underline{5}, 8\} & \texttt{out[]} = \{1, 2, 3\} \\
\texttt{in1[]} = \{\underline{6}, 7\} & \texttt{in2[]} = \{\underline{5}, 8\} & \texttt{out[]} = \{1, 2, 3, 4\} \\
\texttt{in1[]} = \{\underline{6}, 7\} & \texttt{in2[]} = \{\underline{8}\} & \texttt{out[]} = \{1, 2, 3, 4, 5\} \\
\texttt{in1[]} = \{\underline{7}\} & \texttt{in2[]} = \{\underline{8}\} & \texttt{out[]} = \{1, 2, 3, 4, 5, 6\} \\
\texttt{in1[]} = \{\ \} & \texttt{in2[]} = \{\underline{8}\} & \texttt{out[]} = \{1, 2, 3, 4, 5, 6, 7\} \\
\texttt{in1[]} = \{\ \} & \texttt{in2[]} = \{\ \} & \texttt{out[]} = \{1, 2, 3, 4, 5, 6, 7, 8\}
\end{array}
\tag{10.2}
$$

The first line of Equation 10.2 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.

10.3.1 Basic Merge Sort

We start with a straightforward implementation of merge sort. It is simple. straightforward, and will synthesize using the Vivado® HLS tool. Yet, it is not highly optimized. We found the code on the Wikipedia Merge Sort page, and modified it slightly so that it would synthesize and would fit our coding style (e.g., changed the variable names).

Figure 10.6 shows this "bottom-up" merge sort implementation. It is called bottom-up since it iteratively merges smaller subarrays into one final sorted array. The function takes as input an array to sort. Then it breaks down this array into a set of subarrays. This starts with subarrays of length equal to one, i.e., if the input array `A` has `SIZE` elements, there are initially `SIZE` subarrays each holding one element. Then it merges adjacent subarrays. This creates subarrays with two times the number of elements. This process continues until all of the elements have been merged into one array.

The sorting process starts in the `bottomup_merge_sort` function. The function takes as input the `A[]` array and provides the sorted output in the `B[]` array. Both use a custom type `DTYPE` which you can assume is an `int`. And both have `SIZE` elements with is a predefined constant.

```
//Array A[] has the items to sort; array B[] is a work array and the output
void bottomup_merge_sort(DTYPE A[SIZE], DTYPE B[SIZE])
{
    //Each 1−element run in A is already "sorted"
    //Make successively longer sorted runs of length 2, 4, 8, 16, ... until whole array is sorted
    for (int width = 1; width < SIZE; width = 2 * width)
    {
        //Array A is full of runs of length width
        for (int i = 0; i < SIZE; i = i + 2 * width)
        {
            // Merge two runs: A[i:i+width−1] and A[i+width:i+2*width−1] to B[]
            // or copy A[i:n−1] to B[] if( i + width >= n)
            int left_index = i + width < SIZE ? i + width : SIZE;
            int right_index = i + 2*width < SIZE ? i + 2*width : SIZE;
            bottomup_merge(A, i, left_index, right_index, B);
        }
        //Now work array B is full of runs of length 2*width

        //Copy array B to array A for next iteration
        copy_array(B, A);
        //Now array A is full of runs of length 2*width.
    }
}

//Left run is A[iLeft : iRight−1]; Right run is A[iRight : iEnd−1]
void bottomup_merge(DTYPE A[SIZE], int iLeft, int iRight, int iEnd, DTYPE B[SIZE])
{
    int i = iLeft, j = iRight;
    //While there are elements in the left or right runs...
    for (int k = iLeft; k < iEnd; k++)
    {
        //If left run head exists and is <= existing right run head
        if (i < iRight && (j >= iEnd || A[i] <= A[j]))
        {
            B[k] = A[i];
            i++;
        }
        else
        {
            B[k] = A[j];
            j++;
        }
    }
}
```

```
void copy_array(DTYPE B[SIZE], DTYPE A[SIZE])
{
    for(int i = 0; i < SIZE; i++)
    {
        A[i] = B[i];
    }
}
```

Figure 10.6: A bottom-up implementation of merge sort. The `bottomup_merge_sort` function iteratively merges subarrays until the entire array has been sorted.

The computation of the function consists of two nested `for` loops. The outer `for` loop keeps track of the `width` variable, which keeps track of the number of elements in each of the subarrays that are currently being merged. The function starts by putting each element into its own subarray; hence the `width` variable is initialized as `1`. Every iteration of this outer `for` loop merges two consequetive subarrays into a larger one. These new subarrays have twice as many elements which is why the `width` variable doubles on each iteration of this outer `for` loop. The loop terminates when `width` is greater than or equal to `SIZE`, i.e., the subarray size contains all the elements of the input array `A[]`.

The inner `for` loop performs the merging of consecutive subarrays. Each iteration of this loop performs the merging of two subarrays. In each iteration, it calls the function `bottomup_merge` providing the appropriate references to the beginning of the two subarrays (passed in the variables `left_index` and `right_index`). There are two ternary conditional operators (`?`). These perform a minimum function, e.g., the `left_index = min(i + width, SIZE)`. This is necessary for the boundary condition, i.e., to deal with cases at the end of the array with the width is larger than the number of elements of the subarray.

> How does the ternary conditional operator synthesize to hardware? Is this operation always necessary, i.e., are there any cases when we can replace this operation by a simple assignment or remove it altogether? What are the values of `SIZE` that allow us to eliminate the ternary conditional operator?

The `bottomup_merge` function performs the "two finger" algorithm on two subarrays within the `A[]` array. The function takes as arguments the `A[]` array (which holds the input data) and `B[]` array (where the merged result data resides). It also has three references `iLeft`, `iRight`, and `iEnd` that provide the first element of the "left" subarray, the first element of the "right" subarray, and the last element of the "right" subarray. We assume that the two subarrays are next to each other in the `A[]` array thus we do not need to pass the last element of the "left" subarray.

> Can we reduce the complexity of the `bottomup_merge` function when `SIZE` is a power of two? What will change? How does this effect the area and performance?

The variables `i` and `j` within the function correspond to the position of the left and right fingers, respectively. The `if` portion of the conditional clause corresponds to the case when the value at the left finger is smaller than the value at the right finger. Thus we copy that value from `A[i]` to `B[k]` and we move the finger to the next element by incrementing `i` (`i++`). The `else` condition is the case when the right finger points to the smaller element. In this case we copy that element to the `B[]` array and increment `j`. The actual condition is a bit more complex as it must check that we have not already copied all the elements from the left subarray (`i < iRight`) and that we not already copied all the elements from the right subarray (`j >= iEnd`).

The `bottomup_merge` function returns and then the next iteration of the inner `for` loop starts. The subsequent iterations perform merge operations on the remaining subarrays residing in the `A[]` array. The inner `for` loop completes once it has merged across all the elements of the `A[]` array. At this point, the outer `for` loop doubles the size of `width`, and the inner `for` loop performs merging again, this time on larger subarrays.

After we complete the execution of the inner `for` loop in the top level `bottomup_merge_sort` function, we call the `copy_array` function. This simply moves the results of the merged consecutive subarrays that was stored in the `B[]` array into the `A[]` array. This completes one iteration of the outer `for` loop. This continues until there is only one subarray, i.e., the entire `A[]` array has been sorted into the `B[]` array.

> What happens to the `A[]` array over the course of the computation? Describe the state of the elements in the `A[]` array after each iteration of the outer `for` loop. What is the final order of the elements in the `A[]` array when `bottomup_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 optimized at all for a hardware implementation. The first and easiest way to perform optimizations is by adding directives. Given that we have a large number of `for` loops, and that these commonly comprise the largest part of the computation time, they are the natural place to begin optimization. The common loop directives are `pipeline` and `unroll`.

> Perform different optimizations using the `pipeline` and `unroll` directions on the `for` loops. What provides the best performance? Which gives the best tradeoff between resource utilization and performance?

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 seems 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. That is, we must rewrite the code such that is synthesizes in a manner that is amenable to a hardware implementation. This is the trick in using high-level synthesis in an optimum manner. The designer should carefully consider the best way to implement the design in hardware, and then write the high-level synthesis code in a manner that creates that design. Unfortunately, there is no automatic process for this. It typically takes an expert hardware designer with a deep understanding of the Vivado® HLS tool. Like with any skill worth learning, practice makes perfect. In the next section we describe one way to significantly restructure the merge sort code in order to create highly optimized design.

10.3.2 Restructured Merge Sort

The restructured version of merge sort breaks apart the algorithm into a number of functional pieces. This is done primarily in order to perform functional pipelining. Figure 10.7 provides an overview of how we will subdivide the merge sort algorithm into different functions.

The input `A[]` array is given to a `split` function. This function divides the array into two streams `merge1a` and `merge1b`. Each of the two streams contain subarrays with two elements. These two elements are sorted.

These two streams are then provided as input to the `subset_merge` function. This function takes the two element subarrays and merges them into one sorted four element subarray. This is output onto the stream `merge2a`. The function then takes the next two two element subarrays, merges them into a four element subarray and places that onto the second output stream `merge2b`.

The next `subset_merge` function merges two four element subarrays into one eight element subarray that is output on the variable stream `merge3a`. It does this again and puts another sorted 8 element subarra y onto output variable `merge3b`. This process continues by merging two sets of two subarrays from the input streams and placing them on the two streams. The `subset_merge` function is written in a parameterizable manner that allows it to sort any arbitrary power of two, and output two arrays of twice the size on the output streams. We provide the code for this function, and discuss it in more detail later.

This `subset_merge` function continues as necessary, that is until we have merged the data enough into two large subsets. Each `subset_merge` doubles the size of the merged arrays from input to output. At the end of the final `subset_merge` function, the variables `merge7a` and `merge7b` each have two 128 element arrays.

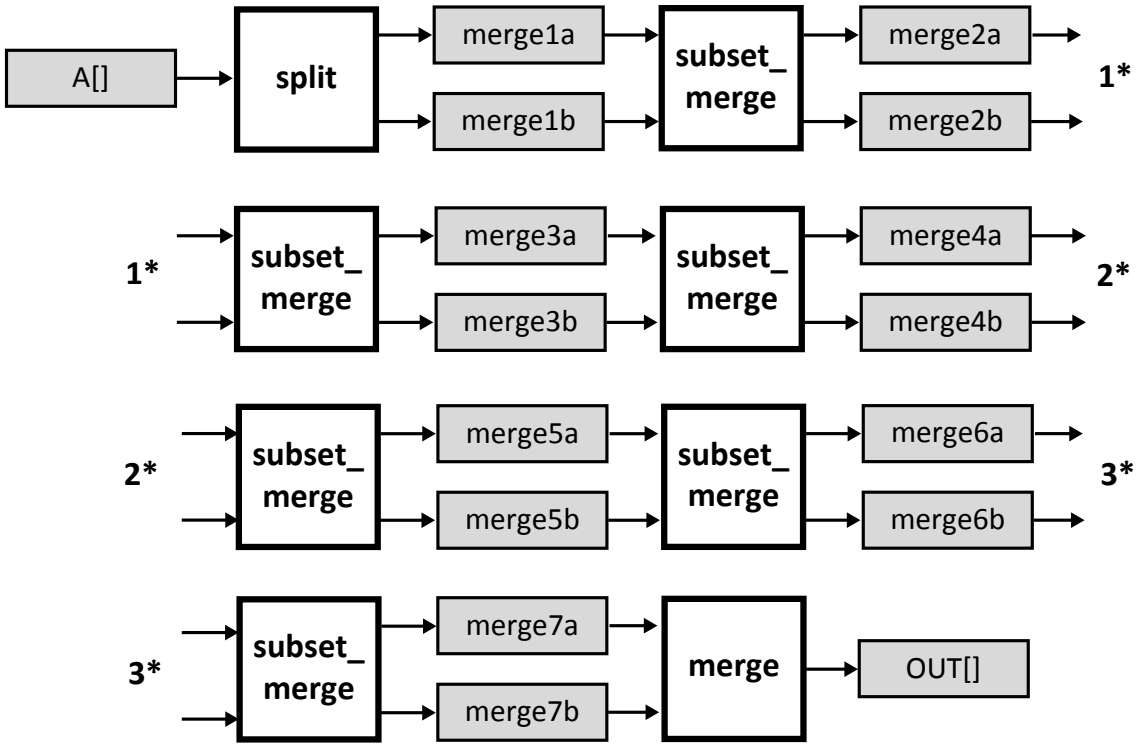


Figure 10.7: **add me**

The final `merge` function performs the merge of two input streams into one output stream. This is the final sorted array. In the case of Figure 10.7, we perform sorting of 256 elements which is put into the

Mention that we will implement radix sort in Chapter 11.

Need a summary

```
void merge_sort(DTYPE IN[SIZE], DTYPE OUT[SIZE])
{
  #pragma HLS INTERFACE axis port=IN
  #pragma HLS INTERFACE axis port=OUT
  #pragma HLS DATAFLOW

  hls::stream<DTYPE> temp1a, temp1b, temp2a, temp2b, temp3a, temp3b, temp4a, temp4b;
  hls::stream<DTYPE> temp5a, temp5b, temp6a, temp6b, temp7a, temp7b;

  //how are the depth values calculated?
  #pragma HLS stream depth=20 variable=temp1a
  #pragma HLS stream depth=20 variable=temp1b
  #pragma HLS stream depth=22 variable=temp2a
  #pragma HLS stream depth=22 variable=temp2b
  #pragma HLS stream depth=26 variable=temp3a
  #pragma HLS stream depth=26 variable=temp3b
  #pragma HLS stream depth=34 variable=temp4a
  #pragma HLS stream depth=34 variable=temp4b
  #pragma HLS stream depth=50 variable=temp5a
  #pragma HLS stream depth=50 variable=temp5b
  #pragma HLS stream depth=82 variable=temp6a
  #pragma HLS stream depth=82 variable=temp6b
  #pragma HLS stream depth=146 variable=temp7a
  #pragma HLS stream depth=146 variable=temp7b

  split(IN, temp1a, temp1b);
  subset_merge(temp1a, temp1b, temp2a, temp2b, 4);
  subset_merge(temp2a, temp2b, temp3a, temp3b, 8);
  subset_merge(temp3a, temp3b, temp4a, temp4b, 16);
  subset_merge(temp4a, temp4b, temp5a, temp5b, 32);
  subset_merge(temp5a, temp5b, temp6a, temp6b, 64);
  subset_merge(temp6a, temp6b, temp7a, temp7b, 128);
  merge(temp7a, temp7b, OUT);
}
```

Figure 10.8: **Add me**

Chapter 11

Huffman Encoding

11.1 Introduction

Lossless data compression is a key ingredient for efficient data storage, and Huffman coding is amongst the most popular algorithm for variable length coding [25]. Given a set of data symbols and their frequencies of occurrence, Huffman coding generates codewords in a way that assigns shorter codes to more frequent symbols to minimize the average code length. Since it guarantees optimality, Huffman coding has been widely adopted for various applications [19]. In modern multi-stage compression designs, it often functions as a back-end of the system to boost compression performance after a domain-specific front-end as in GZIP [17], JPEG [39], and MP3 [40]. Although arithmetic encoding [41] (a generalized version of Huffman encoding which translates an entire message into a single number) can achieve better compression for most scenarios, Huffman coding is typically the algorithm of choice for production systems since developers do not have to deal with the patent issues surrounding arithmetic encoding [28].

Canonical Huffman coding has two main benefits over traditional Huffman coding. In basic Huffman coding, the encoder passes the complete Huffman tree structure to the decoder. Therefore, the decoder must traverse the tree to decode every encoded symbol. On the other hand, canonical Huffman coding only transfers the number of bits for each symbol to the decoder, and the decoder reconstructs the codeword for each symbol. This makes the decoder more efficient both in memory usage and computation requirements. Thus, we focus on canonical Huffman coding.

11.2 Background

In basic Huffman coding, the decoder decompresses the data by traversing the Huffman tree from the root until it hits the leaf node. This has two major drawbacks: it requires storing the entire Huffman tree which increases memory usage. Furthermore, traversing the tree for each symbol is computationally expensive. Canonical Huffman encoding addresses these two issues by creating codes using a standardized canonical format.

Figure 11.1 shows the canonical Huffman encoding process. The `filter` module only passes symbols with non-zero frequencies. The `sort` module rearranges the symbols in

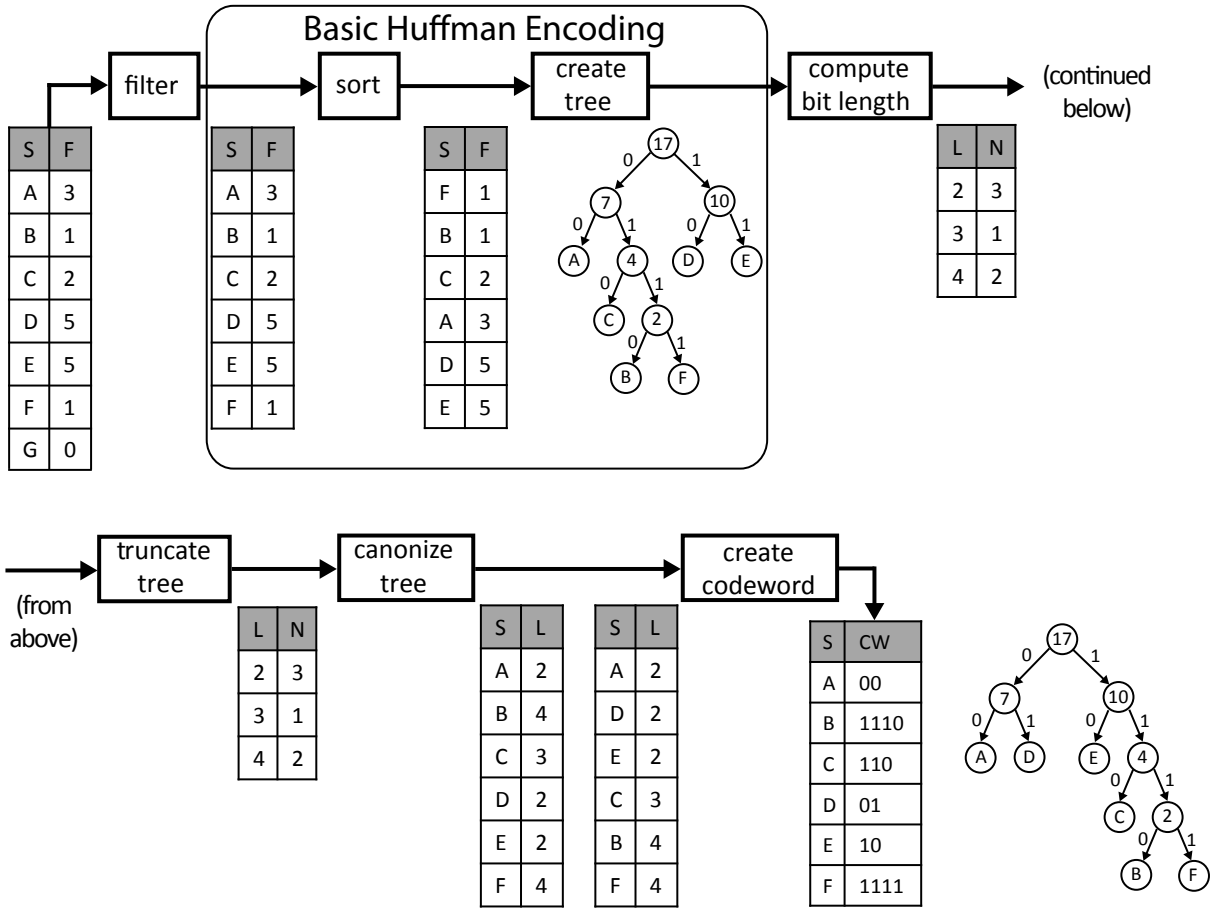


Figure 11.1: The Canonical Huffman Encoding process. The symbols are filtered and sorted, and used to build a Huffman tree. Instead of passing the entire tree to the decoder (as is done in "basic" Huffman coding), the encoding is done such that only the length of the symbols in the tree is required by the decoder. Note that the final canonical tree is different from the initial tree created near the beginning of the process.

ascending order based upon their frequencies. Next, the `create tree` module builds the Huffman tree using three steps: 1) it uses the two minimum frequency nodes as an initial sub-tree and generates a new parent node by summing their frequencies; 2) it adds the new intermediate node to the list and sorts them again; and 3) it selects the two minimum elements from the list and repeat these steps until one element remains. As a result, we get a Huffman tree, and by labelling each left and right edge to 0 and 1, we create codewords for symbols. For example, the codeword for A is 00 and codeword for B is 1110. This completes the basic Huffman encoding process. The `create tree` module does not necessarily create the canonical Huffman tree, e.g., the output of this function Figure 11.1 is not canonical. We perform a number of additional steps to make it canonical. The benefit of using a canonical encoding is that we only need to transmit the length of each Huffman codeword.

The `compute bit len` module calculates the bit lengths of each codeword. It saves this information to a list where the key is length and value is the number of codewords with that length. In the example case, we have three symbols (A,D,E) with the code length of 2. Therefore, the output list contains L=2 and N=3. The `truncate tree` module rebalances the Huffman tree when its height is too large (i.e., unbalanced). This improves decoder speed at the cost of a slight increase in encoding time. This is not necessary in the example in Figure 11.1. We set the maximum height of the tree to 27.

Using output from the `truncate tree` module, the `canonize tree` module creates two sorted lists. The first list contains symbols and lengths sorted by symbol. The second list contains symbols and lengths sorted by lengths. These lists are used for faster creation of the canonical Huffman codewords.

The `create codeword` module creates uniquely decodable codewords based on the following rules: 1) Longer length codes have a higher numeric value than the same length prefix of shorter codes; 2) Codes with the same length increase by one as the symbol value increases. This means if we know the starting symbol for each code length, we can construct the canonical Huffman code in one pass. The method to calculate the starting canonical code for each code length is as follows. We start from the smallest bit length l and assign the first symbol (in sorted order) as 0. We continue to increment the symbol with that bit length by 1 until we have done all of the symbols with the smallest bit length. Then we move to the next bit length $l+1$. In our example, symbols `A`, `D`, and `E` have bit length $l = 2$. Thus, `A = 00`, `D = 01`, and `E = 10`. Note that we consider them in alphabetical order. This is necessary to make the tree canonical. Then we move to the next bit length $l = 3$. Here we add one and shift left by 1. Thus, symbol `C = (10 + 1) << 1 = 11 << 1 = 110`. This is the only symbol with bit length $l == 3$, thus we move to $l = 4$. Once again we add one and shift by one. Thus the codeword for `B = (110 + 1) << 1 = 1110`. The final codeword for symbol `F = 1110 + 1 = 1111`. We explain this in more detail in Chapter 11.3.7

The canonical Huffman encoding pipeline includes many complex and inherently sequential computations. For example, the `create tree` module needs to track the correct order of the created sub trees, requiring careful memory management. Additionally, there is very limited parallelism that can be exploited. In the following, we discuss the hardware architecture and the implementation of the canonical Huffman encoding design using Vivado® HLS.

Figure 11.2 shows the entire "top" `huffman_encoding` function. This sets up the arrays and other variables that are passed between the various subfunctions. And it instantiates

```
void huffman_encoding(
  Symbol in[INPUT_SYMBOL_SIZE],
  sdap_uint<32> encoding[INPUT_SYMBOL_SIZE],
  int *num_nonzero_symbols
)
{
  #pragma HLS DATAFLOW

  Symbol filtered[INPUT_SYMBOL_SIZE];
  Symbol sorted[INPUT_SYMBOL_SIZE];
  Symbol sorted_copy1[INPUT_SYMBOL_SIZE];
  Symbol sorted_copy2[INPUT_SYMBOL_SIZE];
  ap_uint<10> parent[INPUT_SYMBOL_SIZE-1];
  ap_uint<10> left[INPUT_SYMBOL_SIZE-1];
  ap_uint<10> right[INPUT_SYMBOL_SIZE-1];
  int n = INPUT_SYMBOL_SIZE;

  filter(in, filtered, &n);
  sort(filtered, sorted, n);

  ap_uint<10> partial_bit_length1[HUFFMAN_CODEWORD_LENGTH_LIMIT];
  ap_uint<10> partial_bit_length2[HUFFMAN_CODEWORD_LENGTH_LIMIT];
  ap_uint<16> parent_copy1[INPUT_SYMBOL_SIZE-1];
  ap_uint<16> left_copy1[INPUT_SYMBOL_SIZE-1];
  ap_uint<16> right_copy1[INPUT_SYMBOL_SIZE-1];
  ap_uint<16> parent_copy2[INPUT_SYMBOL_SIZE-1];
  ap_uint<16> left_copy2[INPUT_SYMBOL_SIZE-1];
  ap_uint<16> right_copy2[INPUT_SYMBOL_SIZE-1];
  ap_uint<10> initial_bit_length[HUFFMAN_CODEWORD_LENGTH_LIMIT];
  ap_uint<10> truncated_bit_length1[HUFFMAN_CODEWORD_LENGTH_LIMIT];
  ap_uint<10> truncated_bit_length2[HUFFMAN_CODEWORD_LENGTH_LIMIT];
  ap_uint<32> symbol_bits[INPUT_SYMBOL_SIZE];
  ap_uint<32> encoding_copy[INPUT_SYMBOL_SIZE];

  for(int i = 0; i < INPUT_SYMBOL_SIZE; i++)
  {
    sorted_copy1[i].value = sorted[i].value;
    sorted_copy1[i].frequency = sorted[i].frequency;
    sorted_copy2[i].value = sorted[i].value;
    sorted_copy2[i].frequency = sorted[i].frequency;
  }

  create_tree(sorted_copy1, n, parent, left, right);
```

```
  for(int i = 0; i < INPUT_SYMBOL_SIZE−1; i++){
    #pragma HLS PIPELINE II=1
    parent_copy1[i] = parent[i];
    left_copy1[i] = left[i];
    right_copy1[i] = right[i];
    parent_copy2[i] = parent[i];
    left_copy2[i] = left[i];
    right_copy2[i] = right[i];
  }

  ap_uint<10> num_symbols = n;
  ap_uint<10> a1 = 128;
  ap_uint<10> loop_bound1 = (num_symbols > a1) ? a1:num_symbols;
  ap_uint<10> a2 = 256;
  ap_uint<10> loop_bound2 = (num_symbols < a2) ? num_symbols:a2;

  //parallelize the bit length computation across two modules
  compute_bit_length(parent_copy1, left_copy1, right_copy1, 0,
                     loop_bound1, partial_bit_length1);
  compute_bit_length(parent_copy2, left_copy2, right_copy2, 128,
                     loop_bound2−1, partial_bit_length2);

  //merge the results from two parallel bit length calculations
  for(int i = 0; i < HUFFMAN_CODEWORD_LENGTH_LIMIT; i++)
  {
    #pragma HLS PIPELINE II=1
    initial_bit_length[i] = partial_bit_length1[i] + partial_bit_length2[i];
  }

  truncate_tree(initial_bit_length, truncated_bit_length1, truncated_bit_length2);
  canonize_tree(sorted_copy2, num_symbols, truncated_bit_length1, symbol_bits);
  create_codeword(symbol_bits, truncated_bit_length2, encoding_copy);

  for(int i = 0; i < INPUT_SYMBOL_SIZE; i++)
    {
      encoding[i] = encoding_copy[i];
    }

  *num_nonzero_symbols = num_symbols;
}
```

Figure 11.2: The "top" `huffman_encoding` function. It defines the arrays and variables between the various subfunctions. These are described graphically in Figures 11.1 and 11.4.

these functions.

There is some additional copying of data that may seem unnecessary. This is due to our use of the `DATAFLOW` directive. This imparts some restrictions on the flow of the variables between the subfunctions. In particular, there are some strict rules on producer and consumer relationships of data between the parts of the function. This requires that we replicate some of the data. For example, we create two copies of the arrays `parent`, `left` and `right`. We also do the same with the array `truncated_bit_length`. The former is done in a `for` loop in the top `huffman_encoding` function; the latter is done inside of the `canonize_tree` function.

> The `DATAFLOW` directive imposes restrictions on the flow of information in the function. Many of the restrictions enforce a strict producer and consumer relationship between the subfunctions. One such restriction is that an array should be written to by only one function and it should be read by only one function. i.e., it should only serve as an output from one function and an input to another function. For example, if multiple functions read from the same array, the code will not synthesize using the `DATAFLOW` directive (it will synthesize, but not using functional pipelining; it will through a warning). The same is true when a function attempts to read from and write to the same array (assuming that that array is later uses as input or output of another function). Thus, this often requires replicating the data into multiple arrays. Another trick is to create internal copies of the input data inside of a subfunction, and then read and write to that internal array, so that we eliminate reading to and writing to an external array (which would violate the `DATAFLOW` directive requirements). We perform both of these tricks. We will discuss these requirements and how to adhere to them as we go through the code in the remainder of this chapter.

11.3 Implementation

The canonical Huffman encoding process is naturally divided into subfunctions. Thus, we can work on each of these subfunctions on a one-by-one basis. Before we do that, we should consider the interface for each of these functions.

Figure 11.4 shows the functions and their input and output data. For the sake of simplicity, it only shows the interfaces with arrays, which, since they are large, we can assume are stored in block rams (BRAMs). Before we describe the functions and their inputs and outputs, we need to discuss the constants, custom data types, and the function interface that are defined in `huffman.h`. Figure 11.3 shows the contents of this file.

The constant `INPUT_SYMBOL_SIZE` is the maximum number of symbols that will be given as input for encoding. This is set at 256 which is a common symbol length, e.g., for encoding ASCII data. The constant `HUFFMAN_CODEWORD_LENGTH_LIMIT` is the specified upper bound for the codeword length during the initial Huffman tree generation. This is set to 64; this is the maximum path in the generated tree. The constant `HUFFMAN_CODEWORD_LENGTH` is the target tree height for the tree rebalancing which is done in the function `truncate_tree`. Finally, the constant `HUFFMAN_CODEWORD_LENGTH_BITS` is the number of bits to encode the Huffman symbols. This is equal to $\log_2\lceil$`HUFFMAN_CODEWORD_LENGTH`$\rceil$, which in this case is 5.

```
#include "ap_int.h"

// input number of symbols
#define INPUT_SYMBOL_SIZE 256

// upper bound on codeword length during tree construction
#define HUFFMAN_CODEWORD_LENGTH_LIMIT 64

// maximum codeword tree length after rebalancing
#define HUFFMAN_MAX_CODEWORD_LENGTH 27

// number of bits needed to record HUFFMAN_MAX_CODEWORD_LENGTH value
#define HUFFMAN_CODEWORD_LENGTH_BITS 5

struct Symbol
{
    ap_uint<10> value;
    ap_uint<32> frequency;
};

void huffman_encoding (
  Symbol in[INPUT_SYMBOL_SIZE],
  sdap_uint<32> encoding[INPUT_SYMBOL_SIZE],
  int *num_nonzero_symbols
);
```

Figure 11.3: The constants, custom data type, and function interface for the top level function `huffman_encoding`.

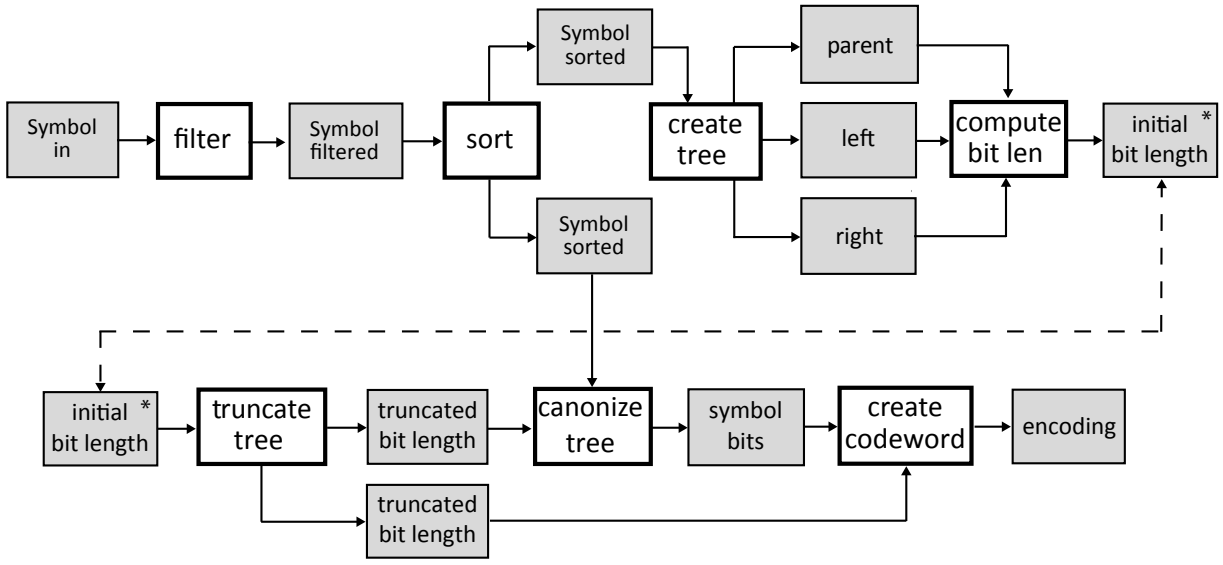


Figure 11.4: The block diagram for our hardware implementation of canonical Huffman encoding. The gray blocks represent the significant input and output data that is generated and consumed by the different subfunctions. The white blocks correspond to the functions (computational cores). *Note that the array initial bit length has been redrawn to allow the figure to be more clear.

We create a custom data type `Symbol` to hold the data corresponding the input values and their frequencies. These are used in the `filter`, `sort`, and other functions in the encoding process that require access to such information. The data type has two fields `value` and `frequency`.

Finally, the `huffman.h` file has the `huffman_encoding` function interface. This is the specified top level function for the Vivado® HLS tool. It has three arguments. The first argument is an array of `Symbols` of size `INPUT_SYMBOL_SIZE`. This is the input data to encode. The next two arguments are outputs. The first is the encoded data for each of the possible input symbols. The third and final argument is the number of non zero symbols from the input data. This is the same as the number of symbols that remain after the `filter` operation.

The input to the system is an array of `Symbols`. This holds the symbol value and frequencies in the array `in`. Each symbol holds a 10 bit `value` and a 32 bit `frequency`. The size of this array is set as the constant `INPUT_SYMBOL_SIZE` which is 256 in our example. The `filter` module reads from the `in` array and writes its output to the `filtered` array. This is an array of `Symbols` which holds the number of non-zero elements which is the input to the `sort` module. The `sort` module writes the symbols sorted by frequency into two different arrays – one is used for the `create tree` module and the other for the `canonize tree` module. The `create tree` module creates a Huffman tree from the sorted array and stores it into three arrays (`parent`, `left`, and `right`); these arrays hold all the info for each node of the Huffman tree. Using the Huffman tree information, the `compute bit len`

```
void filter(Symbol in[INPUT_SYMBOL_SIZE], Symbol out[INPUT_SYMBOL_SIZE], int *n)
{
    #pragma HLS INLINE off
    ap_uint<10> j = 0;
    for(int i = 0; i < INPUT_SYMBOL_SIZE; i++)
    {
        if(in[i].frequency != 0)
        {
            out[j].frequency = in[i].frequency;
            out[j].value = in[i].value;
            j++;
        }
    }
    *n = j;
}
```

Figure 11.5: The `filter` function iterates across the input array `in` and add any `Symbol` entry with a non-zero `frequency` field to the output array `out`. Additionally, it records the number of non-zero frequency elements and passes that in the output argument `n`.

module calculates the bit length of each symbol and stores this information to a `initial bit len` array. We set the maximum number of entries to 64, covering up to maximum 64-bit frequency number, which is sufficient for most applications given that our Huffman tree creation rebalances its height. The `truncate tree` module rebalances the tree height and copies the bit length information of each codeword into two separate `truncated bit length` arrays. They each have the exact same information, but they must be copied to ensure that the Vivado® HLS tool can perform functional pipelining; we will talk about that in more detail later. The `canonize tree` module walks through each symbol from the `sort` module and assigns the appropriate bit length using the `truncated bit length` array. The output of the `canonize` module is an array that contains the bit lengths for the codeword of each symbol. Finally, the `create codeword` module generates the canonical codewords for each symbol.

11.3.1 Filter

The first function for the Huffman encoding process is `filter`, which is shown in Figure 11.5. This function takes as input a `Symbol` array. The output is another `Symbol` array that is a subset of the input array `in`. It removes any entry with a frequency equal to 0. The function itself has a straightforward implementation. It consists of a `for` loop that iterates across the `in` array. It checks the current entry `in[i]`, and if its `frequency` field is non-zero, it stores those entries into the output array `out`. It also keeps track of the number of non-zero entries, i.e., the size of the output array. This is passed as the output argument `n`.

Vivado® HLS often decides to automatically inline functions with a small amount of code. The directive `INLINE` allows the user to specify whether or not to have the tool inline the function. Using the argument `off` will insure that this function does not get inlined. This can be useful in cases when a function is inlined, but you want to determine its performance and resource usage.

11.3.2 Sort

The `sort` (see Figure 11.6) function arranges the symbols in sorted order based upon the `frequency` values. The input to this function is the `in Symbol` array, and the number of elements in that array `n`. The output of the function is the `out Symbol` array, which is the same size as `in` with the same elements but in sorted order.

The function itself has three `for` loops. The first `copy_in_to_sorting for` loop does a copy of the input symbols into a temporary array called `sorting`. The second `radix_sort for` loop does the bulk of the work; it implements a radix sort algorithm to reorder the input symbols. The third `final_copy for` loop moves the sorted symbols into the output array `out`.

The first block of code in the `sort` function defines the variables for the function. The variables `i` and `j` are used for iterating over `for` loops. The variable `radix_size` defines the radix value for sorting the 32 bit `Symbol.frequency` variable; this partitions the `Symbol.frequency` variable into digits, and radix sort performs a counting sort one by one on each of these digits. The `radix_size` is a constant set to 16. The `bin` and `prefix` arrays are used in the `radix_sort for` loop; they both have size of `radix_size`. The two `ARRAY_PARTITION` directives declares that these two arrays should be completely partitioned, i.e., they should be put into registers. They are small, and used frequently, thus this does not use many resources, and can provide performance benefits. Finally, the array `masked_symbol_value` stores the current radix value of the frequency that is the subject of the counting sort for every iteration of the radix sort.

The first `copy_in_to_sorting for` loop moves the input `in` array into the local `sorting` array. It is useful to create a local version of the input data when performing functional pipelining, i.e., the `DATAFLOW` directive, at a higher level. The `sorting` function reads and writes to the `sorting` array throughout its execution. Doing this on the input array would disallow the use of the `DATAFLOW` directive since it has restrictions on the types of access patterns on array. We use the `PIPELINE` directive with an `II=1` to maximize the throughput. Finally, we use the `LOOP_TRIPCOUNT` directive setting a minimum and maximum value of 0 and 256, respectively. This allows the Vivado® HLS tool to provide better synthesis estimates on the number of cycles. The maximum value of 256 is accurate since this is the maximum size of the `in` array. It may be smaller depending upon the result of the `filter` operation. The minimum value is also accurate though it does not provide a very tight lower bound. We could also provide an average value, which depends upon the input data.

The next portion of the `sorting` function sorts the input array `in` using a radix sorting algorithm. This is comprise of the outer `radix_sort for` loop, which contains three internal `for` loops: `init_bin`, `histogram`, and `re_sort`.

```
void sort(Symbol in[INPUT_SYMBOL_SIZE], Symbol out[INPUT_SYMBOL_SIZE], int n)
{
  int i, j, shift, value;
  Symbol previous_sorting[INPUT_SYMBOL_SIZE], sorting[INPUT_SYMBOL_SIZE];
  const unsigned int radix_size = 16;
  ap_uint<10> bin[radix_size], prefix[radix_size];
  #pragma HLS ARRAY_PARTITION variable=prefix complete dim=1
  #pragma HLS ARRAY_PARTITION variable=bin complete dim=1
  ap_uint<10> masked_symbol_value[INPUT_SYMBOL_SIZE];

  copy_in_to_sorting:for(j = 0; j < n; j++)
  {
    #pragma HLS LOOP_TRIPCOUNT min=0 max=256
    #pragma HLS PIPELINE II=1
    sorting[j] = in[j];
  }

  radix_sort:for(shift = 0; shift < 32; shift += 4)
  {
    init_bin:for(i = 0; i < radix_size; i++)
    {
      #pragma HLS PIPELINE II=1
      bin[i] = 0;
    }

    histogram:for(j = 0; j < n; j++)
    {
      #pragma HLS LOOP_TRIPCOUNT min=0 max=256
      #pragma HLS PIPELINE II=1
      value = (sorting[j].frequency >> shift) & (radix_size - 1); //get the correct 4 bits to sort
      masked_symbol_value[j] = value; //stores the masked value for each symbol on each sort
      bin[value]++;
      previous_sorting[j] = sorting[j]; //store current sorted order of symbols
    }

    prefix[0] = 0;
    prefix_sum: for(i = 1; i < radix_size; i++)
      prefix[i] = prefix[i-1] + bin[i-1];
```

```
    re_sort:for(j = 0; j < n; j++)
    {
      #pragma HLS LOOP_TRIPCOUNT min=0 max=256
      #pragma HLS PIPELINE II=1
      i = masked_symbol_value[j]; //grab masked value
      sorting[prefix[i]] = previous_sorting[j]; //move symbol to new sorted location
      prefix[i]++; //update prefix sum for duplicates
    }
  }

  final_copy:for(j = 0; j < n; j++)
  {
    #pragma HLS PIPELINE II=1
    #pragma HLS LOOP_TRIPCOUNT min=0 max=256
    out[j] = sorting[j];
  }
}
```

Figure 11.6: The `sort` function employs a radix sort on the input symbols based upon their frequency values.

The general radix sorting algorithm works on the input data by each radix from left to right (least significant digit) or right to left (most significant digit) — our algorithm works from least significant to most significant. In a decimal system, the radix takes values from 0 to 9. Our algorithm sorts on `Symbol.frequency` variable, which are represented using a 32-bit number. We divide the 32-bit number into 4-bit portions, i.e., we are using radix $r = 2^4 = 16$. We perform counting sort for each 4-bit number. There are $32/4 = 8$ counting sort operations – one for each of the 8 digits. The `radix_sort for` loop performs these 8 counting sort operations. It iterates to 32 by increments of 4.

What would happen if we increased or decreased the radix? How would this effect the number of counting sort operations that are performed? How would this change the resource usage, e.g., the size of the arrays?

This implementation of the radix sort algorithm has three parts: the first performs a histogram on the current digit; the second part does a prefix sum on the resulting histogram values; and the third part reorders the symbols based upon these results. These three parts are in the `for` loops `histogram`, `prefix_sum`, and `re_sort`, respectively.

Radix sort works by working on one digit at a time. The algorithm sorts the number corresponding solely to this digit. We use the counting sort algorithm to perform the sorting of the digits; we discuss the counting sort implementation shortly. We use a least significant digit radix sort, which means that we start our sorting with rightmost digit. Our digit is a 4-bit number, so we first look at the least significant four bits of the `Symbol.frequency`. This

is the first iteration of the `radix_sort for` loop. The histogram operation determines the number of each of the 16 values. Then the `prefix_sum for` loop determines the new sorted location for each of these values. We are essentially counting the number of each occurrence of the digit, and then using the prefix sum to figure out its sorted location. The `re_sort for` loop uses this information in order to reorder the symbol values into the `sorting` array. It goes through the `previous_sorting` array, which holds the symbol values corresponding to reordering from previous iterations of the `radix_sort for` loops, and determines the new location where to store the symbol in the `sorting` array based upon the histogram and prefix sum in this iteration of radix sort. It does this by first getting the masked value that was stored in the `masked_symbol_value` array in the `histogram for` loop, and then using that to index into the `prefix` array. This provides the new index for the location of this symbol sorted in the correct manner according to this digit under consideration. The final statement (incrementing the value in the `prefix` array) insures that we put any future symbols with the same digit into a new index in the sorting array.

We have previous discussed two parts of this algorithm – the histogram and prefix sum algorithms in Chapter 8.2 and 8.1. We will not go into any more detail here about optimizing these `for` loops.

What happens to the performance and utilization results when you perform the optimizations on the prefix sum and histogram loops as specified in Chapter 8.2 and 8.1?

The optimization of the `re_sort for` loop is straightforward. The `LOOP_TRIPCOUNT` directive informs the Vivado® HLS compiler of the potential bounds in order to yield more accurate performance estimates (the same as previously discussed for the `copy_in_to_sorting` for loop). And we inform the tool to pipeline the loop with an initiation interval of one cycle using the appropriate directive.

Is the `re_sort for` loop able to achieve the specified initiation interval of one cycle? Why or why not?

The final part of the `sorting` function is the `final_copy for` loop. This moves the sorted symbol values from the `sorting` array into the `out` array that is specified in the function interface. This may seem redundant, e.g., why not just eliminate the `sorting` array and directly write to the `out` array throughout the function. However, it is necessary to perform these local copies so that we obey the read and write rules required to arguments between functions that will be pipelined using the `DATAFLOW` directive.

The `DATAFLOW` directive has several requirements in order to perform the task level pipelining optimization. One of them is the need for single producer and consumer of data between the tasks. Since we would like to perform task level pipelining for the Huffman encoding process as shown in Figure 11.4, we must insure that each of these

tasks follow this requirement. In the case of this `sort` function, which is one of the tasks, it must only consume (read from) the input argument data and produce (writes to) the output argument data. In order to met this requirement, we create the internal array `sorting`, which is read from and written to throughout the function. We copy the input data from the argument `in` at the beginning of the function. And then we transfer the final results to the output argument `out` at the end of the function. This insures that we follow the producer/consumer requirements for the `DATAFLOW` directive.

11.3.3 Create Tree

The next function in the Huffman encoding process forms the binary tree. To do this, the algorithm selects the two symbols with the smallest frequency and uses them as the left and right nodes of a new intermediate node. That intermediate node has a frequency that is the sum of these two child nodes. This process continues by iteratively creating intermediate nodes from the two nodes with the smallest frequencies. Except now, the intermediate nodes are now eligible for being selected as one or both of the left or right child nodes. The tree building process continues until all the symbol nodes are selected, and all of the intermediate nodes have been incorporated into the binary tree.

The `create_tree` function is shown in Figure 11.8. There are two inputs: the `Symbol` array `in` and the number of elements in that array `n`. The function uses those symbols to create the binary tree. The data for the binary tree is held into three output arrays `parent`, `left`, and `right`.

This is an interesting function from an high-level synthesis standpoint since we simplify the data structures in order to make the resulting hardware less complex. A typical implementation would use a queue or similar data structure in order to resort the nodes. That is, it would create a new intermediate node, and then add that back into the queue. And then take two elements from the queue, and continue this process until the queue is empty. However, this involves a lot of pointer chasing and/or array reordering. Instead, we use two arrays – one that has a sorted list of the input symbols and one that keeps a sorted list of the intermediate nodes. This requires that we check each array for the lowest sorted values. Furthermore, we store the resulting tree into three arrays `parent`, `left`, and `right`. These are used to store the nodes in the binary tree. Each index from the three arrays corresponds to one node. For example, `parent[i]`, `left[i]`, and `right[i]` hold all of the information for the ith intermediate node of the binary tree (the parent of that node, and the left and right children respectively). These are stored in the order that they are created. Correspondingly, they are also stored in sorted order (based upon the frequency of their children) since the $i + 1$ intermediate node is guaranteed to have a frequency that is equal to or larger than node i. The frequency value for the node is stored at `intermediate[i]`.

Figure 11.7 shows the running input example. The six symbols sorted by their frequencies are stored in the `in` array. The resulting Huffman tree is shown along with the values for the four arrays used to create the tree: `intermediate`, `left`, `right`, and `parent`. We directly denote the node numbers for the `left` and `right` arrays (e.g., `n0`, `n1`, etc.) for the sake of illustration. These will hold a special internal node value in reality.

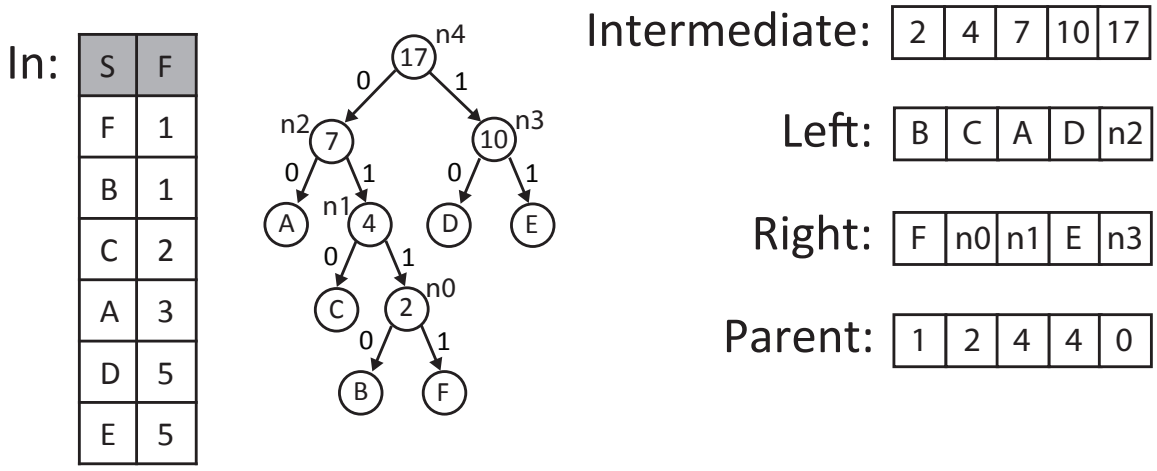


Figure 11.7: The `Symbol` array `in` is used to create the Huffman tree. The tree is shown graphically along with the corresponding values for the four arrays used to represent the tree (`intermediate`, `left`, `right`, and `parent`).

The first block of code defines the local variables that we use in the function. We have an 32 bit unsigned integer array `intermediate` that is used to store the frequencies for the intermediate nodes. The variable `node_freq` holds the running sum of the frequencies for the two child nodes which we ultimately store into the `intermediate` array. The variables `left_index` and `right_index` are used to store the index of the last used values in both of these output arrays. These are incremented by one for every new intermediate node that is created. The last two variables `unlinked_index` and `intermediate_index` are both used on the `intermediate` array. The variable `intermediate_index` is similar to `left_index` and `right_index`; it holds the last used frequency value corresponding to the most recently created intermediate node. It is incremented every time a new intermediate node is created. The variable `unlinked_index` holds the index of the smallest intermediate node (both in terms of frequency and index number) that has not yet been added into the binary tree. That is, it has not been selected as a left or right child node, and it does not yet have a parent. Note that all of the nodes between `unlinked_index` and `intermediate_index` have not yet been added into the tree.

The next block of code creates the first intermediate node. It makes the assumption that there are at least two symbols in the input `in` array i.e., `n` $\geq$ `2`. And it takes those first two symbols, in indices 0 and 1 of the `in` array, and uses them as the left and right child of the intermediate node that is being created. The intermediate nodes of the Huffman binary tree are stored across three arrays `left`, `right`, and `parent`. The `left` and `right` arrays hold the symbol values of the left and right children. You can see here that the code stores these values from the first two symbols in the `in` array. The `parent` array holds the index of the parent to this intermediate node. We do not know this yet, and it will get filled in later. The final array that we use throughout the `create_tree` function is `intermediate`. This holds the frequency value of the intermediate nodes. Here we store the sum of the two frequencies of the input symbols. The code block ends by setting the `in_index = 2` which

```
void create_tree (
        Symbol in[INPUT_SYMBOL_SIZE],
        int n,
        ap_uint<10> parent[INPUT_SYMBOL_SIZE-1],
        ap_uint<10> left[INPUT_SYMBOL_SIZE-1],
        ap_uint<10> right[INPUT_SYMBOL_SIZE-1])
{
    ap_uint<32> intermediate[INPUT_SYMBOL_SIZE-1];
    ap_uint<32> node_freq = 0;
    ap_uint<10> left_index = 0; //keeps track of lowest empty position in each array
    ap_uint<10> right_index = 0;
    ap_uint<10> unlinked_index = 0; //smallest "unlinked" node
    ap_uint<10> intermediate_index = 0; //total number of intermediate nodes created

    // assume two leaves are available; form first intermediate node by merging two leafs
    left[0] = in[0].value;
    node_freq = in[0].frequency;
    right[0] = in[1].value;
    node_freq += in[1].frequency;
    intermediate[0] = node_freq;
    node_freq = 0;
    ap_uint<10> in_index = 2;

    while(in_index < n)
    {
        #pragma HLS LOOP_TRIPCOUNT min=1 max=256
        #pragma HLS PIPELINE II=5
        node_freq = 0;
        intermediate_index++;

        if (in[in_index].frequency <= intermediate[unlinked_index])
        {
            left_index++; //point to the new left entry
            left[left_index] = in[in_index].value; //set input symbol as left node
            node_freq = in[in_index].frequency; //add symbol frequency to total node frequency
            in_index++; // move to the next input symbol

            if (in_index == n) //special case handling: no more leaf nodes
            {
                goto GetRightNode;
            }
        }
```

```
    else
    {
        left_index++; //point to the new left entry
        left[left_index] = INTERNAL_NODE; //Set symbol to indicate an internal node
        node_freq = intermediate[unlinked_index]; //add child node frequency
        parent[unlinked_index] = left_index; //set this node as child's parent
        unlinked_index++; //go to next parentless internal node

        if(unlinked_index == intermediate_index)
        {
            //no internal nodes remain. Use input symbol for right child
            goto GetRightLeaf;
        }
    }

    //this if/else part finds the second node
    //if part makes the right node a symbol; else makes it an internal node
    if (in[in_index].frequency <= intermediate[unlinked_index])
    {
        GetRightLeaf:
        right_index++;
        right[right_index] = in[in_index].value;
        node_freq = node_freq + in[in_index].frequency;
        in_index++;
        intermediate[intermediate_index] = node_freq;
    }
    else
    {
        GetRightNode:
        right_index++;
        right[right_index] = INTERNAL_NODE;
        intermediate[intermediate_index] = node_freq + intermediate[unlinked_index];
        parent[unlinked_index] = right_index;
        unlinked_index++;
    }
}

//all input symbols added to tree. Connect up all internal nodes.
while(unlinked_index < intermediate_index)
{
    #pragma HLS LOOP_TRIPCOUNT min=1 max=256
    #pragma HLS PIPELINE II=1
    intermediate_index++;
```

```
        //add left subtree
        left_index++;
        left[left_index] = INTERNAL_NODE;
        node_freq = intermediate[unlinked_index];
        parent[unlinked_index] = left_index;
        unlinked_index++;

        //add right subtree
        right_index++;
        right[right_index] = INTERNAL_NODE;
        node_freq = intermediate[unlinked_index];
        parent[unlinked_index] = right_index;
        unlinked_index++;
    }
    parent[unlinked_index] = 0; //Set parent of last node (root) to 0
}
```

Figure 11.8: The complete code for Huffman tree creation. The code takes as input the sorted `Symbol` array `in`, the number of elements in that array `n`, and outputs the Huffman tree in the three arrays `left`, `right`, and `parent`.

indicates that the next unused symbol is at index 2.

The code continues with the first of two `while` loops. This loops iterates until we have added all of the input symbols from the `in` array into intermediate nodes. We use the variable `in_index` to hold the index of the `in` array that has not yet been added to the tree. Thus when this variable is equal to `n`, then we have considered all of the symbols from the `in` array.

There are two directives. The first `LOOP_TRIPCOUNT` directive one defines the number of times that this loop will iterate. The second `PIPELINE` directive informs the Vivado® HLS tool that we would like to pipeline this `while` loop with a target $II = 5$.

> In what situations would this `while` loop iterate only one time? What conditions would make it iterate 256 times? The `LOOP_TRIPCOUNT` directive can also accept an `avg` argument. What would be an appropriate value for this?

The initial block of code continues with two statements. It sets the `node_freq = 0`. This will store the sum of the frequencies of left and right child nodes that we will select shortly. And it increments the `intermediate_index`. This makes room for us to store the frequency value for this node that we are about to create.

The code continues with an `if/else` clause which finds the left child node. The `if` portion selects the left child node from one of the input symbols, i.e., from the `in` array. The `in` array is sorted by frequency. Thus, we check the smallest (lowest) index that we have not already considered. Similarly, the `intermediate` array is in sorted order due to the fact

that we continually add nodes to that are guaranteed to have the smallest value. We look at the `unlinked_index` entry in the `intermediate` node. This is the smallest entry that has not yet been added to the binary tree. If the `if` clause it taken (i.e., the condition is true), then we are adding from the input symbol array `in`. We make space in the `left` array and add the `Symbol.value` which indicates this symbol is the left child. We store its frequency temporarily into the `node_freq` array and increment the `in_index` to denote that we have now used this symbol. If at this point, we have used all of the input symbols, i.e., `in_index == n`, then we skip directly to the code at label `GetRightNode` which adds an intermediate node as the right child.

The `else` portion adds an intermediate node as the left child node. This performs a similar set of actions as the code in the `if` clause. Since we are using an internal node, and not a symbol, we store a special value `INTERNAL_NODE` in the `left` array. And since we are selecting an internal node as a child, we can set this node as the parent of that child. Finally, we increment the `unlinked_index` to denote that we have now added that internal node to the tree. The final `if` clause here checks to see if `unlinked_index == intermediate_index`, that is if all of the internal nodes have now been added to the tree. If that is the case, then the right child must be a symbol, so we jump to the code that performs this action.

The next part of the `while` loop is another `if/else` clause. This performs similar activities to the previous `if/else` clause except here we are determining the symbol or internal node for the right child. The code is almost exactly the same in both cases (except replacing "right" for "left"). The only additional code is related to the fact that we now know the frequency values for both the left and right child, so we calculate that and store that into the `intermediate` array.

This first `while` loop continues to create intermediate nodes until we have added all of the input symbols into intermediate nodes. The second `while` loop then takes over. Since there are no symbols to consider, it continues to create new intermediate nodes from existing intermediate nodes. Essentially this is performing the same code from the previous `else` clauses. This continues until all of the intermediate nodes have been added into the tree. That condition is met when `unlinked_index == intermediate_index`. Once this `while` loop completes, we set the parent of the last created intermediate node – the root node – to a special value 0. This completes the building of the Huffman tree.

11.3.4 Compute Bit Length

The `compute_bit_length` function determines the number of symbols at each depth value. The depth corresponds to the number of bits for the encoding of that symbol. The input arguments to the function are the Huffman tree created in the `create_tree` function. These are the arrays `parent`, `left`, and `right`. The other two inputs are `lower_node_number` and `upper_node_number`. These two arguments are used to set the upper and lower bounds for the nodes in the Huffman tree to iterate across. These are used if we want to instantiate multiple versions of this function in order to create parallel versions to speed up the computation. The output is the array `bit_length`; each element of that array store the number of symbols that have that depth value (or equivalently the number of bits). Thus, if there are five symbols that have depth of three, then `bit_length[3] = 5`.

The variables `curr_index`, `next_index` are used to keep track of the nodes as we traverse

```
void compute_bit_length (
    ap_uint<16> parent[INPUT_SYMBOL_SIZE-1],
    ap_uint<16> left[INPUT_SYMBOL_SIZE-1],
    ap_uint<16> right[INPUT_SYMBOL_SIZE-1],
    ap_uint<10> lower_node_number,
    ap_uint<10> upper_node_number,
    ap_uint<10> bit_length[HUFFMAN_CODEWORD_LENGTH_LIMIT]
)
{
    ap_uint<10> curr_index, next_index, length;

    //Temporary storage to insure dataflow synthesizes.
    //Copies to bit_length at end of function
    ap_uint<10> internal_bit_length[HUFFMAN_CODEWORD_LENGTH_LIMIT];

    //initialize arrays
    init_array:for(int i = 0; i < HUFFMAN_CODEWORD_LENGTH_LIMIT; i++)
    {
        #pragma HLS PIPELINE II=1
        internal_bit_length[i] = 0;
    }

    //go through each of the nodes in the loop bounds one at a time
    calculate_lengths:for(int i = lower_node_number; i < upper_node_number; i++)
    {
        #pragma HLS LOOP_TRIPCOUNT min=1 max=256
        if(left[i] != INTERNAL_NODE || right[i] != INTERNAL_NODE){
            //this node has at least one symbol as a child
            //thus we should trace it and see its length
            curr_index = i;
            length = 1;
```

```
            traverse_tree:for(int j = 0; j < HUFFMAN_CODEWORD_LENGTH_LIMIT; j++)
            {
                next_index = parent[curr_index]; //what node do we look at next?
                curr_index = next_index;

                if(next_index == 0){
                    //Reached the root node
                    if(left[i] != INTERNAL_NODE && right[i] != INTERNAL_NODE)
                    {
                        //Special case: both the children of the original node were symbols
                        //therefore, increment by 2
                        internal_bit_length[length] = internal_bit_length[length] + 2;
                    }
                    else
                    {
                        //at least one child is internal node
                        internal_bit_length[length] = internal_bit_length[length] + 1;
                    }
                    break;
                }
                else
                {
                    //Not at root yet. Increment bit length and continue...
                    length++;
                }
            }
        }
    }

    //copy the internal array into the output array.
    //Need to adhere to dataflow properties (only write to output array)
    copy_bit_length:for(int i = 0; i < HUFFMAN_CODEWORD_LENGTH_LIMIT; i++)
    {
        #pragma HLS PIPELINE II=1
        bit_length[i] = internal_bit_length[i];
    }
}
```

Figure 11.9: The complete code for determining the number of symbols at each bit length. The code takes as input the Huffman tree (in the three array `left`, `right`, and `parent`) and a range of nodes to iterate across (denoted in `lower_node_number` and `upper_node_number`). The output is an array `bit_length` where each element denotes the number of symbols that have that bit length.

the tree to calculate the depth, which is stored in the variable `length`. We define an array `internal_bit_length` which stores the resulting depths for each of the nodes. These will be written into the `bit_length` array at the end of the function. We have a separate internal array in order to insure that our function adheres the requirements for the `dataflow` directive. That is, the input variables to the function can only be read from, and the output variables can only be written to. The `internal_bit_length` array is written to and read from throughout the function, so we could not substitute it with the output argument `bit_length`. We used the same technique in the `sort` function.

The `init_array for` loop sets the values '0'. This is necessary when we use parallel versions of this function due to the way that we later merge the results into a single array that holds all of the bit lengths.

> The `init_array for` loop has a `pipeline` directive with `II = 1`. Is it possible to meet this II? What happens if we increase the II to something larger? What happens if we do not apply this directive?

The `calculate_lengths for` loop performs the majority of the computation in this function. It iterates across (a subset of) the nodes in the Huffman tree and calculates the depths of the nodes that are associated with symbols, i.e., those that are not internal nodes. This `for` loop has bounds [`lower_node_number`, `upper_node_number`] that are passed into the function. This allows the function to be parallelized, i.e., we can create multiple instances operating in parallel, each of which iterates across a subset of the nodes in the Huffman tree. Remember that the information about each node is distributed across the arrays `left`, `right`, and `parent`.

> The `calculate_lengths for` loop uses the `LOOP_TRIPCOUNT` directive and sets the `min=1` and `max=256`. What happens if this loop is parallelized? What are the appropriate `min`, `max`, and `avg` arguments?

For each node, we first check to determine whether or not is has one or two symbols as the child, i.e., we check to determine that at least one of the values in the array `left` or `right` does not have the value `INTERNAL_NODE`. If both children are internal nodes, then we do not need to do anything for this node. If one or both of the child nodes are symbols, then we should start traversing up the tree until we reach the root node.

The nested `traverse_tree for` loop moves up through the tree by continually looking at the parent node. It stops when the parent node is the root node. We keep track of the current depth in the `length` variable. This is set to '1' before we enter the nested `for` loop. The `curr_index` variable holds the node that we are considering at this iteration of the `for` loop. This is set to `i` before the start of the `traverse_tree for` loop.

The first statements in the `for` loop set the `next_index` as the parent node of the `curr_index`, i.e., we are moving "up" the Huffman tree. Then we check to see if the `next_index` is the root node. Recall that we marked the root node using the value `0`. If it is the root node, then we do one of two things. The `if` clause is when the initial node that we

are considering (at index `i`) has both of its children as symbols. In this case, we increment by two. The other case in the `else` clause is when only one of the children were a symbol. Thus we increment by one. At this point, we can break out of the inner `traverse_tree for` loop since we have reached the root node.

When we have not reached the root node, then we increment `length` by `1`. This is done in the outer `else` clause. And then we continue to the next iteration of the loop, which considers the parent node. This continues until we reach the root node.

> There are no directives used in the `traverse_tree for` loop. What happens when we apply the `PIPELINE` directive? What about using the `LOOP_TRIPCOUNT` directive? What are the appropriate arguments?

After we have considered all of the nodes in our range, we conclude the function by copying the depths or bit lengths from the `internal_bit_length` into the output array `bit_length`. This is done in the final `copy_bit_length for` loop.

Figure 11.10 shows an example of how we could divide the computation into two parts. Here we split the nodes in half, and we would have two `compute_bit_length` functions. The first would traverse the first three nodes, and the second would handle the last two nodes. The workload in this case would be uneven since left side of the tree, performed in Process 1, would have more traversals up the tree than the right side. A balanced tree would even out the workloads, but in general it would be difficult to precisely determine the number of cycles for each process since this would depend on the runtime characteristics of the data.

> What is the approximate number of cycles required for Process 1 and Process 2 in Figure 11.10? Is there a better way to partition the nodes to even out the computation of the processes?

11.3.5 Truncate Tree

The next part of the Huffman encoding process reorganizes nodes that have a depth that is larger than that specified in `HUFFMAN_MAX_CODEWORD_LENGTH`. This is done by finding any symbols with a greater depth, and moving them to a level that is smaller than that specified target.

We perform all of these operations on the array `input_bit_length`, which was derived from compute bit length process described in the previous section. We no longer need the data corresponding to the Huffman tree, thus we only operate on this one input array. There are two identical output arrays `truncated_bit_length1` and `truncated_bit_length2`. These arrays are feed into two separate functions later in the process (`canonize_tree` and `create_codewords`), and thus we must have two arrays to adhere to the constraints of the `DATAFLOW` directive.

The code is shown in Figure 11.11. The first `for` loop copies the data from the input array `input_bit_length` into one of the output arrays `truncated_bit_length1`. This is done so that we only read from this input array to insure that the `DATAFLOW` directive will

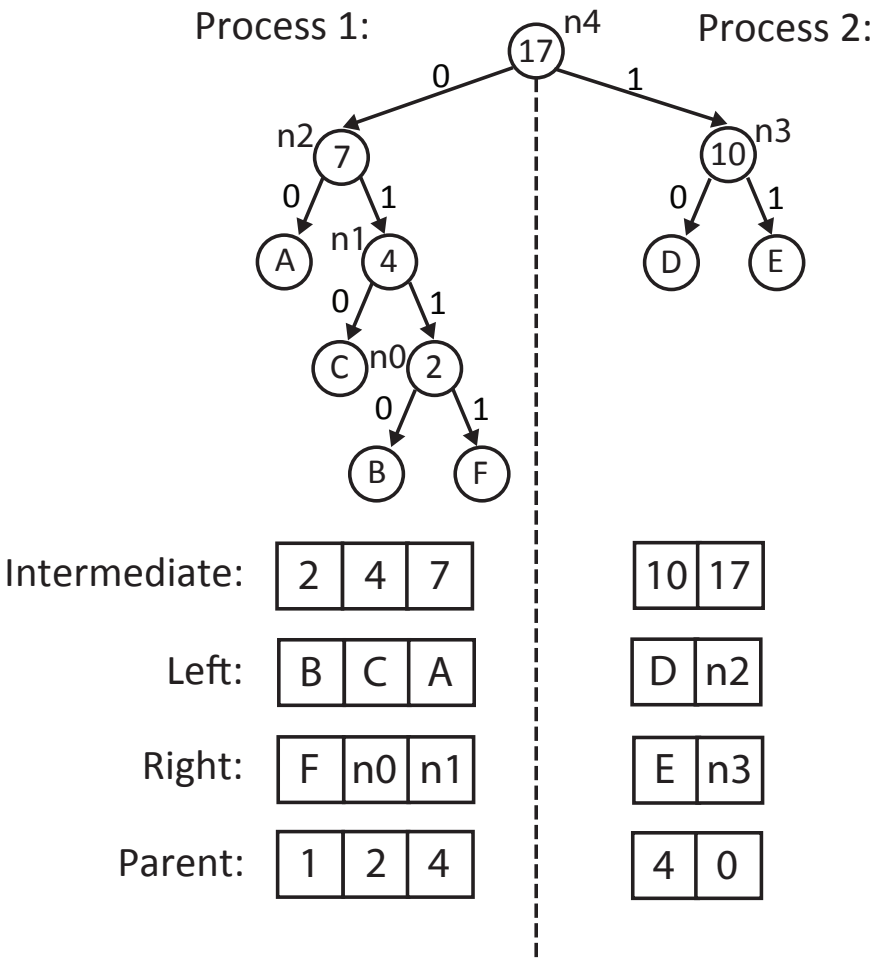


Figure 11.10: We can parallelize the computation by performing multiple `compute_bit_length` functions in parallel. Here we divide the node array in half and have two processes running in parallel.

```
void truncate_tree(
  ap_uint<10> input_bit_length[HUFFMAN_CODEWORD_LENGTH_LIMIT],
  ap_uint<10> truncated_bit_length1[HUFFMAN_CODEWORD_LENGTH_LIMIT],
  ap_uint<10> truncated_bit_length2[HUFFMAN_CODEWORD_LENGTH_LIMIT]
)
{
  ap_uint<10> i;
  ap_uint<10> j;

  //copy into temporary storage to maintain dataflow properties
  copy_in:for(i = 0; i < HUFFMAN_CODEWORD_LENGTH_LIMIT; i++)
  {
    truncated_bit_length1[i] = input_bit_length[i];
  }

  //Set j to the target depth for rebalancing the tree
  j = HUFFMAN_MAX_CODEWORD_LENGTH;

  //Rebalance the tree until it is under the target depth
  move_nodes:for(i = HUFFMAN_CODEWORD_LENGTH_LIMIT - 1;
       i > HUFFMAN_MAX_CODEWORD_LENGTH; i--)
  {
    //Look to see if there is any nodes at lengths greater than target depth
    reorder: while(truncated_bit_length1[i] != 0)
    {
      #pragma HLS LOOP_TRIPCOUNT min=3 max=3 avg=3
      if (j == HUFFMAN_MAX_CODEWORD_LENGTH)
      {
        //Find deepest leaf with codeword length < target depth
        do
        {
          #pragma HLS LOOP_TRIPCOUNT min=1 max=1 avg=1
          j--;
        }
        while(truncated_bit_length1[j] == 0);
      }
```

```
      //Convert leaf at level j into node
      //Attach to it one leaf from level i and one leaf from level j;
      //Remaining leaf at level i moves one level up
      truncated_bit_length1[j] -= 1; // The node at level j is no longer as leaf
      truncated_bit_length1[j+1] += 2; // Two new leaf nodes are attached at level j+1.
      truncated_bit_length1[i-1] += 1; // The leaf node at level i+1 gets attached here.
      truncated_bit_length1[i] -= 2; // Two leaf nodes have been lost from level i.

      // now deepest leaf with codeword length < target length
      //is at level (j+1) unless j+1 == target length
      j++;
    }
  }

  copy_out:for(i = 0; i < HUFFMAN_CODEWORD_LENGTH_LIMIT; i++)
  {
    truncated_bit_length2[i] = truncated_bit_length1[i];
  }
}
```

Figure 11.11: The complete code for rearranging the Huffman tree such that the depth of any node is under the target specified by the constant `HUFFMAN_MAX_CODEWORD_LENGTH`. The code takes as input the bit lengths of the initial symbols (in the arrays `input_bit_length`). The output is two identical arrays `truncated_bit_length1` and `truncated_bit_length2`, which will have no symbol nodes larger than the target depth. There are two output arrays since they are feed to two separate subsequent functions. They must be independent to adhere to the constraints imposed by the `DATAFLOW` directive.

synthesize. We will operate on the truncated_bit_length1 throughout this function, and the copy its results into the other output array truncated_bit_length2 at the end of the function.

> The copy_in for loop is not optimized. What happens to the latency and initiation interval of the truncate_tree function if we use a PIPELINE or UNROLL directive on this loop. What happens to the overall latency and initiation interval of the design (i.e., the huffman_encoding function)?

The function continues in the second move_nodes for loop, which performs the bulk of the computation. This for loop starts by iterating through the truncated_bit_length array from the largest index (HUFFMAN_CODEWORD_LENGTH_LIMIT - the specified maximum depth for a tree). This continues down through the array until there is a non-zero element or i reaches the HUFFMAN_MAX_CODEWORD_LENGTH. If we do not find a non-zero element, that means the initial input Huffman tree does not have any nodes with a depth larger than the target depth. In other words, we can exit this function without performing any truncation. If there is a value larger than the target depth, then the function continues by reorganizing the tree so that all of the nodes have depth smaller than the target depth. This is done by the operations in the reorder while loop. When there are nodes to move, the move_nodes for loop goes through them from those with the largest depth, and continues to smaller depths until all nodes are rearranged with a depth smaller than the target. Each iteration of this move_nodes for loops works on moving nodes from one depth at a time.

The reorder while loop moves one node in each iteration. The first if statement is used to find the leaf node with the largest depth. We will then alter this node by making it an intermediate node, and adding it and the leaf node with a depth larger than than target as children. This if clause has a do/while loop that iterates downward from the target looking for a non-zero entry in the truncated_bit_length array. It works in a similar manner as the beginning of the move_nodes for loop. When it has found the deepest leaf node less than the target depth, it stops. The depth of this node is stored in j.

Now we have a node with a depth i larger than the target, and a node with a depth smaller than the target stored in j. We move the node from depth i and a node from j into child nodes at depth j + 1. Therefore, we add two symbols to truncated_bit_length[j+1]. We are making a new intermediate node a depth j thus, we subtract a symbol from that level. We move the other leaf node from depth i to depth i - 1. And we subtract two from truncated_bit_length[i] since one of the nodes went to level j + 1 and the other when to level i - 1. These operations are performed in the four statements on the array truncated_bit_length. Since we added a symbol to level j + 1, we update j, which holds the highest level under the target level, and then we repeat the procedure. This is done until there are no additional symbols with a depth larger than the target.

The function completes by creating an additional copy of the new bit lengths. This is done by storing the updated bit lengths in the array truncated_bit_length1 into the array truncated_bit_length2. We will pass these two arrays to the final two functions in the huffman_encoding top function; we need two arrays to insure that the constraints of the DATAFLOW directive are met.

11.3.6 Canonize Tree

The next step in the encoding process is to determine the number of bits for each of the symbols. We do this in the `canonoize_tree` function shown in Figure 11.12. The function takes as input the symbols in sorted order (`sorted` array), the total number of symbols (`num_symbols`), and the number of symbols at each bit length value (`bit_length` array). The output is an array (`num_bits`) where each value corresponds to the number of encoded bits used for the symbol at that index. Thus, if the symbol with value `0x0A` is encoded in 4 bits, then `num_bits[10] = 4`.

The canonization process starts by initializing the values of the `symbol_bits` array to `0`. It then continues into a second `for` loop. This loop iterates across the symbol in sorted order - from smallest frequency to largest frequency. The code inside of the `for` loop starts with an `if` clause with a nested `do/while` loop. This is used to find the largest non-zero bit length that has not yet been considered. The body of the `do/while` loop decrements `i`, and uses that as an index into the `bit_length` array. This continues to decrement until that value in `bit_length[i]` is non-zero. That is, we have found the next largest bit length.

When this `do/while` loop is complete, we have the bit length stored in the variable `i`, and the number of symbols with that bit length stored in the variable `n`. At this point, we go through the numbers from smallest frequency to largest frequency (this is done in the outer `for` loop) and assign the next `n` of them to have the bit length `i`. This is done by indexing into the `symbol_bits` array using the value of the symbol.

The process is complete when we have gone through all of the symbols. And the output is an array where each index corresponds to a symbol value. And the value of that index is the number of bits for that symbol.

11.3.7 Create Codeword

The final step in the encoding process is to create the codeword for each symbol. The code for this is shown in Figure 11.13. To compute the codewords, we use two pieces of information. The first is the number of bits required for the encoding of each of the symbols. This is the information that we computed in the `canonize_tree` function from the previous section. It is provided as input in the `symbol_bits` array. Additionally, we need the number of symbols for each bit length. This can be derived from the number of bits for each encoding, but since it was already computed in the `truncate_tree` function from Chapter 11.3.5, we can simply use that. This information in passed as input in the array `bit_length`. The output is the encoding for each symbol, which is stored in the output array `encoding`. The encoding has two parts. It holds the codeword, and it holds the number of bits for that codeword. The maximum number of bits required for any codeword is given in the constant `HUFFMAN_CODEWORD_LENGTH_BITS`. Thus, the `HUFFMAN_CODEWORD_LENGTH_BITS` least significant bits of each element in the `encoding` array holds the data from the input array `symbol_bits`. The remainder of that `encoding` element is the codeword.

The canonical encoding process works by first calculating the starting symbol value corresponding to each bit length. The "first" symbol from that bit length takes the starting symbol value. Then subsequent symbols with that same bit length take the next value (computed by incrementing by one). The symbols are considered in order, thus the encoding will

```
void canonize_tree(
  Symbol sorted[INPUT_SYMBOL_SIZE],
  ap_uint<10> num_symbols,
  ap_uint<10> bit_length[HUFFMAN_CODEWORD_LENGTH_LIMIT],
  ap_uint<32> symbol_bits[INPUT_SYMBOL_SIZE]
)
{
  ap_uint<10> i;
  ap_uint<10> n;

  for(int i = 0; i < INPUT_SYMBOL_SIZE; i++)
    symbol_bits[i] = 0;

  //i starts at largest number and decrements until there is a non-zero bit length
  i = HUFFMAN_CODEWORD_LENGTH_LIMIT;
  n = 0;

  //Iterate across the symbols from lowest frequency to highest
  //Assign them largest bit length to smallest
  for(int k = 0; k < num_symbols; k++)
  {
    #pragma HLS LOOP_TRIPCOUNT min=256 max=256
    if (n == 0)
    {
      //find the next non-zero bit length
      do
      {
        #pragma HLS LOOP_TRIPCOUNT min=2 max=2
        i--;
        // n is the number of symbols with encoded length i
        n = (ap_uint<10>) bit_length[i];
      }
      while (n == 0);
    }
    symbol_bits[sorted[k].value] = i; //assign symbol k to have i bits
    n--; //keep assigning i bits until we have counted off n symbols
  }
}
```

Figure 11.12: The complete code for canonizing the Huffman tree. The goal is to determine the number of bits for each symbol. This is stored in the output array `symbol_bits`. The function takes as input an array of symbols sorted by their frequency (`sorted`), the number of symbols (`num_symbols`), and the number of symbols at each bit length (`bit_length`).

```
void create_codeword(
    ap_uint<32> symbol_bits[INPUT_SYMBOL_SIZE],
    ap_uint<10> bit_length[HUFFMAN_CODEWORD_LENGTH_LIMIT],
    ap_uint<32> encoding[INPUT_SYMBOL_SIZE]
)
{
    unsigned int first_codeword[HUFFMAN_MAX_CODEWORD_LENGTH+1];
    unsigned int out_reversed;

    //Computes the initial codeword value for a symbol with bit length i
    first_codeword[0] = 0;
    first_codewords:for(int i = 1; i < HUFFMAN_MAX_CODEWORD_LENGTH+1; i++)
    {
        first_codeword[i] = (first_codeword[i−1] + bit_length[i−1]) << 1;
    }

    unsigned int bits;
    assign_codewords:for (int i = 0; i < INPUT_SYMBOL_SIZE; ++i)
    {
        #pragma HLS PIPELINE II=5
        bits = symbol_bits[i];
        //if symbol has 0 bits, it doesn't need to be encoded
        make_codeword:if(bits != 0){
            out_reversed = bit_reverse32(first_codeword[bits]);
            out_reversed = out_reversed >> (32 − bits);
            encoding[i] = (out_reversed << HUFFMAN_CODEWORD_LENGTH_BITS) + bits;
            first_codeword[bits]++;
        }
    }
}
```

Figure 11.13: The complete code for generating the canonical Huffman codewords for each of the symbols. The codewords can be computed with knowledge of number of bits that each symbol uses (stored in the input array `bit_length`). Additionally, we have another input array`symbol_bits` which stores at each entry the number of symbols with codewords at that bit length. The output is the code word for each symbol stored in the `encoding` array.

be canonical.

To start, we must first determine the appropriate starting symbols for each of the bit lengths. The initial code word for each of the bit lengths is stored in the `first_codeword` array. It is derived from the `bit_length` array. The recurrence relation is:

```
first_codeword[i] = (bit_length[i-1] + first_codeword[i-1]) << 1
```
(11.1)

The base case is `first_codeword[0] = 0`.

Let us now go through our running example and show how this is used to derive the initial codewords. In the example, the symbols `A`, `D`, and `E` have two bits for their encoding; symbol `C` has three bits; and symbol `B` and `F` have four bits. Thus, we have:

$$\begin{aligned} \texttt{bit\_length[0]} &= 0 \\ \texttt{bit\_length[1]} &= 0 \\ \texttt{bit\_length[2]} &= 3 \\ \texttt{bit\_length[3]} &= 1 \\ \texttt{bit\_length[4]} &= 2 \end{aligned} \tag{11.2}$$

We can use Equation 11.1 and the base case to calculate the values of `first_codeword`. Their values are:

$$\begin{aligned} \texttt{first\_codeword[0]} &= 0 \\ \texttt{first\_codeword[1]} &= (0+0) << 1 = 0 \\ \texttt{first\_codeword[2]} &= (0+0) << 1 = 0 \\ \texttt{first\_codeword[3]} &= (3+0) << 1 = 6 \\ \texttt{first\_codeword[4]} &= (1+6) << 1 = 14 \end{aligned} \tag{11.3}$$

This process is done in the code in the `first_codewords for` loop.

Once we have determined these values, then we go through the symbols in order from smallest symbol value to largest symbol value. For each symbol, we determine the number of bits that it uses through a lookup in the in the input `symbol_bits` array, which holds the number of bits required for each symbol codeword.

After that, we perform a lookup into the array `first_codeword` at the index corresponding to the number of bits for this symbol (`bits`). For now, ignore the bit reversal and other code; we will come back to that later. The value at `first_codeword[bits]` is the encoding for that symbol. We set the codeword for the symbol to this value, and increment that value in the `first_codeword` array to prepare it as the codeword of the next symbol with the same number of codeword bits. This process continues to iterate until we have completed all of the symbols.

In the running example, we go through the symbols `A`, `B`, `C`, `D`, `E`, and `F` in alphabetical order. The symbol `A` has two bits for its encoding. We perform a lookup into `first_codeword[2] = 0`. Thus we assign the codeword for `A` to 00. We increment the value at `first_codeword[2]` to 1. The symbol `B` has four bits. Since `first_codeword[4] = 14 = 0b1110`, it gets assigned the codeword 1110. Symbol `C` has three bits. The value of `first_codeword[3] = 6 = 0b110`, thus it gets the codeword 110. Symbol `D` has two bits so it gets `first_codeword[2] = 1 = 0b01`; remember that we incremented this value after we assigned the codeword to symbol `A`. Symbol `E` has two bits so it gets the codeword $01 + 1 = 10$. And F has four bits so it gets the codeword $1110 + 1 = 1111$.

The final codewords for all of the symbols are:

$$\begin{array}{lcl} \texttt{A} & = & 00 \\ \texttt{B} & = & 1110 \\ \texttt{C} & = & 110 \\ \texttt{D} & = & 01 \\ \texttt{E} & = & 10 \\ \texttt{F} & = & 1111 \end{array} \tag{11.4}$$

Some systems that use Huffman encoding have a convention that the codewords are provided in a bit reversed order. This may make the decoding process easier since it requires that we traverse the tree from leaf node to root node. We add the bit reversal process here to demonstrate how it could be done. The first part of this performs a bit reversal on the code word using the `bit_reverse32` function. This is done in the first statement in the `make_codeword if` clause. We have talked about this function previously in the FFT chapter (see Chapter 5.3), so we will not discuss it here again. The next statement removes the least significant '0' bit, i.e., it leaves only the bit reversed codeword. The third statement shifts the codeword and adds in the number of bits required for encoding this code word. The least significant `CODEWORD_LENGTH_BITS` hold the length of the symbol. The remaining upper bits hold the symbol code word. This may or may not be a requirement depending upon the application. After that, the fourth statement increments the code word to set it up for the next symbol with that same number of bits, as we have previously discussed.

This completes the canonical Huffman encoding process. The outputs are stored in the `encoding` array. Each entry has the reversed code word and the number of bit required for that code word.

11.3.8 Testbench

The final part of the code is the testbench. This is shown in Figure 11.14. The general structure is to read the input frequency values from a file, put them into the `in Symbol` array, call the `huffman_encoding` function, and compare the encoded outputs to an existing golden reference that is stored in a file.

The `main()` function starts by setting up the variables required to read the frequencies from a file (in this case the file is `input\random256.txt`) and puts them into the `in Symbol` array. This is done in the `file_to_array` function, which takes as input the `filename` for the input data and the length of the data (`array_length`), and stores the entries in that file into the `array` variable. This file should store the frequencies of the symbols. These frequency labels are stored in order of the appearance of the file, thus the first value of the file has the frequency of label '0', and so on.

The `main()` function continues by initializing the `in Symbol` array using the frequencies from the file. It then calls the top `huffman_encoding` function using that `in` array. This function returns the encoded symbol values in the `encoding` array.

The next step is to compare this encoding values to a golden reference, which is provided in the file `ShortSymbolOut.Gold.dat`. We do this by performing a `diff` operation on that file with a file that contains the output result from our top `huffman_encoding` function. Before we perform that `diff` operation, we must store the output data from the `encoding`

```
#include "huffman.h"
#include <stdio.h>
#include <stdlib.h>

void file_to_array(const char *filename, ap_uint<16> *&array, int array_length)
{
  printf("Start reading file [%s]\n", filename);
  FILE *file = fopen(filename, "r");
  if(file == NULL)
  {
    printf("Cannot find the input file\n");
    exit(1);
  }

  int eof_check = 0;
  int file_value = 0;
  int count = 0;
  array = (ap_uint<16> *) malloc(array_length*sizeof(ap_uint<16>));

  while(1)
  {
    eof_check = fscanf(file, "%x", &file_value);
    if(eof_check == EOF)
      break;
    else
    {
      array[count++] = (ap_uint<16>) file_value ;
    }
  }
  fclose(file);

  if(count != array_length)
    exit(1);
}

int main ()
{
  printf("Starting canonical Huffman encoding testbench\n");
  FILE *output_file;
  int return_val = 0;
  ap_uint<16> *frequencies = NULL;
  file_to_array("input\\random256.txt", frequencies, INPUT_SYMBOL_SIZE);
```

```
  Symbol in[INPUT_SYMBOL_SIZE];
  for (int i = 0 ; i < INPUT_SYMBOL_SIZE; i++)
  {
    in[i].frequency = (ap_uint<32>) frequencies[i];
    in[i].value = i;
  }

  int num_nonzero_symbols;
  ap_uint<32> encoding[INPUT_SYMBOL_SIZE];
  huffman_encoding(in, encoding, &num_nonzero_symbols);

  output_file = fopen("ShortSymbolOut.dat", "w");
  for(int i = 0; i < INPUT_SYMBOL_SIZE; i++)
    fprintf(output_file, "%d, %x\n", i, (unsigned int) encoding[i]);
  fclose(output_file);

  printf ("\n***************Comparing against output data*************** \n\n");
  if (system("diff ShortSymbolOut.dat ShortSymbolOut.Gold.dat"))
  {
    fprintf(stdout, "*******************************************\n");
    fprintf(stdout, "FAIL: Output DOES NOT match the golden output\n");
    fprintf(stdout, "*******************************************\n");
    return_val = 1;
  }
  else
  {
    fprintf(stdout, "*******************************************\n");
    fprintf(stdout, " PASS: The output matches the golden output\n");
    fprintf(stdout, "*******************************************\n");
    return_val = 0;
  }

  printf("Ending canonical Huffman encoding testbench\n");
  return return_val;
}
```

Figure 11.14: The complete code for the canonical Huffman encoding testbench. The code initializes the `in` array with data from an input file. It passes that into the top `huffman_encoding` function. Then it stores the resulting codewords into a file, and compares that with another golden reference file. It prints out the results of the comparison, and returns the appropriate value.

array into a file. We set the name of this file as `ShortSymbol.Out.dat`. If the `diff` operation returns '0', then the files are identical. It will be a non-zero value if the files are not identical. Thus, the `if` condition occurs when the files are different, and the `else` condition is executed when the files are the same. In both cases, we print out a message, and we set the `return_val` to the appropriate value.

The `main` function concludes by printing out a message, and returning `return_value`. This is important since this is what is expected by the Vivado® HLS tool for cosimulation. The return value should be '0' if it passes, and non-zero if it does not pass.

Part I

Appendix: Wireless Systems Projects

Chapter 12

Basics of Wireless Communication

Wireless systems transmit data over the air using electromagnetic waves. These electromagnetic waves are converted to/from an electrical signal using an antenna. To transmit data, The antenna itself converts data encoded as an electrical signal into an electromagnetic wave which carries the information through the air. On the receive side, the antenna does the opposite; it converts the electromagnetic waves into a voltage across the antenna's terminals.

The transmitted/received electrical signals are naturally real valued where the value represents the current provided to the antenna's terminals when transmitting or a voltage across the terminal when the systems is in receive mode. The signal can be varied over time in order to transmit information. The exact process used to convey this information is called the modulation. More specifically, modulation changes one or more properties of a high frequency signal which is called the *carrier signal*. This involves changing the signal's frequency, amplitude and/or phase. The carrier signal is at a much higher frequency than the data rate, e.g., WiFi uses 2.4 GHz.

Before we go any further on exactly how data is modulated, let us first provide some basic terminology. At its most simplistic form, a *signal* is something that conveys information. Mathematically, it is a function that maps a domain into a range. In the case of wireless communication, it is a mapping of time into a voltage or current. A *continuous signal* is a function defined of a continuous time interval. A *discrete signal* has values defined at only certain points of time. A continuous signal can be converted into a discrete time signal by *sampling* where the *sampling interval* defines the time between two consecutive samples. The *sampling frequency* is the reciprocal of the sampling interval. For example, a 1 GHz sampling frequency has a sampling interval of 1 ns. Wireless communication typically employs an *analog to digital converter (ADC)* to produce a discrete signal from the received continuous signal. This allows for digital processing of the signal in a general purpose microprocessor, digital signal processor, or FPGA, the latter of which is of course the focus of this book. A *digital to analog converter (DAC)* does the reverse; it converts a discrete signal into an continuous signal, e.g., to give to provide a current to the antenna to transmit a signal.

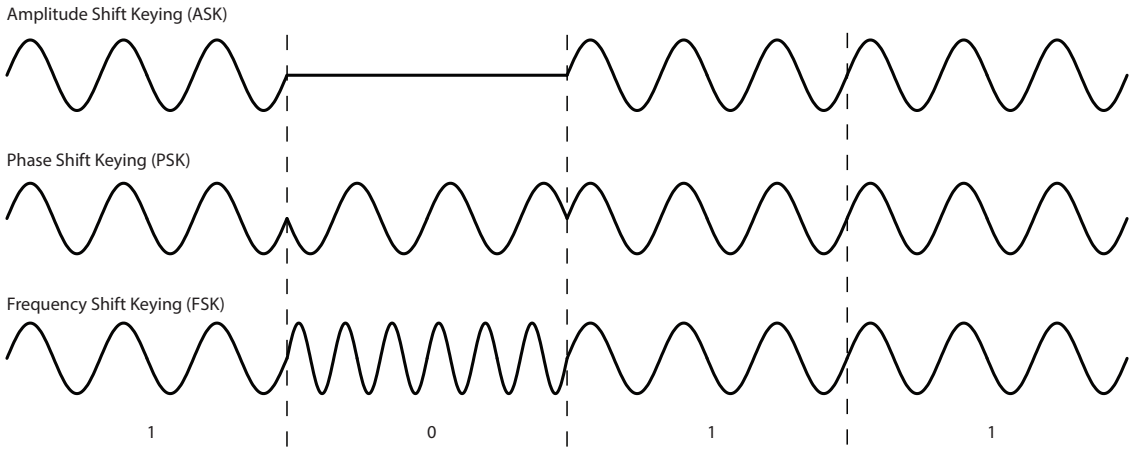


Figure 12.1: The ASK, PSK and FSK waveforms representing the binary data '1011'.

12.1 Modulation

To convey information using a continuous carrier signal $A_c \cos(2\pi f_c t + \phi)$, we can change the amplitude A_c, the phase ϕ, and/or the frequency f_c. The basic techniques for the transmission of digital information include amplitude shift keying, phase shift keying, and frequency shift keying. Each of these techniques changes the specified portion of the signal in order to transmit a set of 'ones' and 'zeros'. The "key" is a discrete value encoded into the signal; in essence it is a transform between the digital (ones and zeros) to analog (carrier signal) domains.

> The term "key" is derived from Morse code key which is a device that turns something on or off. The straight key is probably the most familiar; it generates an electrical signal when the knob is pressed and is off otherwise.

Amplitude Shift Keying (ASK) encodes data using the amplitude of the carrier signal. The simplest form of ASK is called *on-off keying* which uses the presence or absence of the carrier signal to represent a "one" or "zero", respectively. More generally speaking, ASK uses a finite number of amplitude to encode digital data, e.g., using four amplitudes would encode two binary bits. *Phase Shift Keying (PSK)* uses the phase of the carrier signal to encode data. For example, the carrier's phase can be modulated between 0° and 180° in order to represent "one" and "zero". Additional phases can be used to encode more data, e.g., 0°, 90°, 180°, 270° would represent two binary bits. *Frequency Shift Keying (FSK)* modulates the frequency in order to encode data. Binary FSK would use two different frequencies to encode "zero" and "one". Once again, additional frequencies can be used to encode more data. Figure 12.1 shows how four bits of binary data is encoded into the carrier signal using each of these modulation techniques. *Quadrature Amplitude Modulation (QAM)* modulates both the amplitude and the phase. As such it can be viewed as simultaneously employing ASK and PSK to encode data.

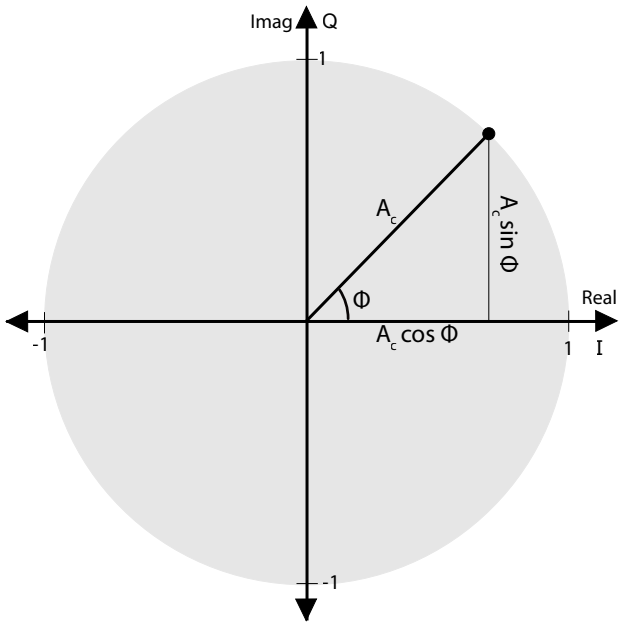


Figure 12.2: A polar coordinate representation of a sinusoid. The amplitude of the sinusoid is the length of the vector and the phase is the angle. The x axis is often referred to as the real or in-phase (I) value; similarly the y axis is the imaginary or quadrature (Q) value.

The job of the wireless receiver is to demodulate the received signal into a digital data. To do this, we must determine the key parameters of the carrier signal. Consider again the carrier signal $A_c \cos(2\pi f_c t + \phi)$. Depending on the type of modulation, we need to extract the amplitude A_c, the frequency f_c, and/or the phase ϕ. The frequency is the rate of change of the phase, i.e., f_c is the first derivative of ϕ with respect to time ($f_c = \frac{\partial \phi}{\partial t}$). Therefore, if we know the amplitude A_c and the phase ϕ at every instance of time, we can fully recreate the carrier signal. Because of this sinusoids are often represented using a polar coordinate system as in Figure 12.2. The amplitude of the sinusoid (A_c) is the length of the vector while the phase ϕ is the angle. The conversion between the polar and Cartesian coordinate system can be done using the trigonometric functions $x = A_c \cos \phi$ and $y = A_c \sin \phi$. The x axis is called the in-phase (I) or real axis portion of the signal while the y axis is called the quadrature (Q) or imaginary portion.

Since we modulate data based upon the amplitude, phase, and frequency, using polar coordinates is a natural way to represent and reasonable about wireless signals. Much of the mathematical foundations behind wireless communications utilize a polar representation. However, the discrete signal provided from/to the ADC and DAC is in terms of Cartesian coordinates, i.e., and I and Q or real and imaginary data. This is due to the fact that it is difficult (and therefore expensive) to change the phase of a high frequency carrier signal using an analog circuit. And it is relatively easy to create an analog circuit that varies the amplitude and phase of signal using I and Q data.

To do this, consider the following trigonometric identity:

$$\cos(u + v) = \cos(u)\cos(v) - \sin(u)\sin(v) \tag{12.1}$$

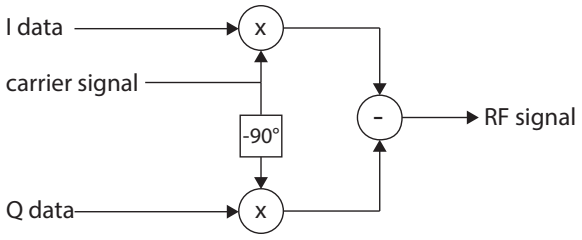


Figure 12.3: Circuit for generating an RF signal using I and Q data.

If we transform the carrier signal using this identity we get:

$$A_c \cos(2\pi f_c t + \phi) = A_c \cos(2\pi f_c t) \cos(\phi) - A_c \sin(2\pi f_c t) \sin(\phi) \tag{12.2}$$

Noting that the real or in-phase (I) portion of the carrier signal at any instance of time is $A_c \cos\phi$ while the imaginary or quadrature portion is $y = A_c \sin\phi$ (see Figure 12.2), we can rewrite Equation 12.2 as:

$$A_c \cos(2\pi f_c t + \phi) = I \cos(2\pi f_c t) - Q \sin(2\pi f_c t) \tag{12.3}$$

The implications of this are that we can modulate the phase of the carrier signal by changing the amplitude of a sine and cosine wave of the same frequency. And therefore we can modify the amplitude, phase, and frequency of a carrier signal by changing the amplitude of a two signals of the same frequency separated by $-90°$ phase offset. Or equivalently we can modulate the amplitude, phase, and frequency of a carrier signal using I and Q signals. It is much easier to create a circuit that performs this type of modulation as shown in Figure 12.3. The circuit requires two mixers (multipliers), a subtractor, and a phase change by 90°. Each of these components are relatively straightforward to design. As such, wireless data is very often represented as in-phase (I) and quadrature (Q).

A basic understanding of I\Q data and the complex plane allows for a better representation of the different modulation techniques. For example, we can denote binary PSK on the complex plane as shown in Figure 12.4. A carrier signal with a phase of 0° encodes a value of '1' while a phase of 180° denotes a value of '0'. This is called a constellation diagram. This can be extended to more complex modulation techniques. For example, we can modulate data using both the amplitude and phase. This is called quadrature amplitude modulation (QAM), an example of which is shown in Figure 12.5. Here four bits are encoded into unique amplitudes and phases using a Gray code. The amplitude and phases are arranged into a rectangular fashion, which is common because they are easier to demodulate than other QAM constellations (e.g., circular). Note that BPSK can be considered as a special case of QAM, as can other ASK, PSK and FSK modulations. QAM is a popular modulation technique for higher data rate applications. For example, it is used in digital TV.

ADD MORE HERE AS NECESSARY.

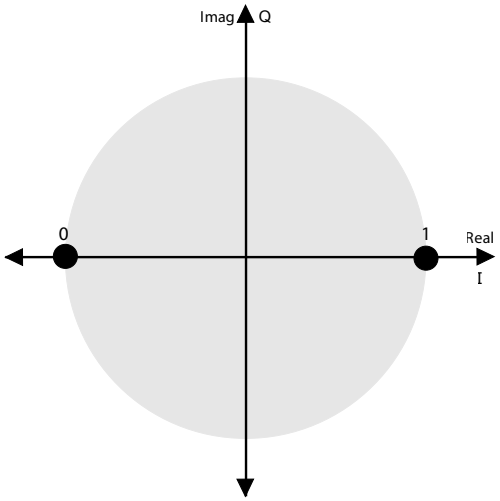


Figure 12.4: Binary Phase Shift Keying (BPSK) represented on the I\Q plane using a constellation diagram.

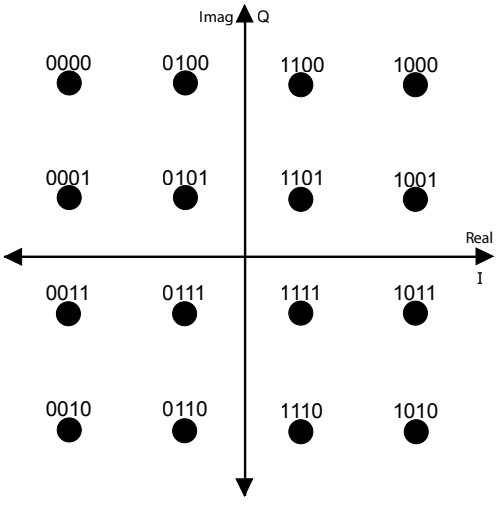


Figure 12.5: The constellation diagram of a 16 QAM modulation. Four bits are encoded using different phases and magnitudes across the I\Q plane.

Chapter 13

Finite Impulse Response (FIR) Filter Project

13.1 Introduction

In this project, you will do three tasks using Vivado® HLS tool and the Zedboard:

- Design 11 tap FIR filter
- Demonstrate the 11 tap filter on the Zedboard
- Design and optimize 128 tap matched filter.

We provide baseline code for the designs and demo. For the 11 tap FIR filter and demo, you must acquire necessary knowledge from the resources provided with this document. For the matched filter, you must perform necessary optimizations to increase the throughput, decrease the latency, and decrease the area. In your report, you must provide the synthesizable code, describe the optimizations and their architectures, and discuss the impact of each optimization on the throughput, latency, and area.

13.2 Preparation

Before you start, we strongly suggest that you go through these high level synthesis tutorials: Lab 1, Lab 2 and Lab 3 in this document: ug871-vivado-high-level-synthesis-tutorial.pdf. You can find this document in the given Tutorial folder. **Fix this. Add this document somewhere and point to it.**

13.3 Materials

Need to add a location for this. You can find the following files in project1-fir.zip:

- `fir11` folder: 11 tap fir filter
  - `fir.cpp`: Implements top level function
  - `fir.h`: Header file

  - `fir_test.cpp`: Test bench
  - `input.dat`: Input chirp signal
  - `out.gold.dat`: "Golden" output. When the testbench (from fir_test.cpp) is run through the file fir.cpp it should generate this result. If it does not, you did something wrong.

- `fir128` folder: 128 tap matched filter
  - `fir.cpp`: Implements top level function
  - `fir.h`: Header file
  - `fir_test.cpp`: Test bench
  - `input.dat`: Input chirp signal
  - `out.gold.dat`: "Golden" output. When the testbench (from fir_test.cpp) is run through the file fir.cpp it should generate this result. If it does not, you did something wrong.

- `Demo` folder: Demo folder for 11 tap filter
  - `host_fir.c`: Unfinished host program
  - `exe_script.sh`: Execution script
  - `plot_script.p`: Plotting script
  - `input.dat`: Input chirp signal

- Tutorial folder
  - `ug871-vivado-high-level-synthesis-tutorial.pdf`: Various tutorials for Vivado HLS

13.4 Project Goal

The first goal of this project is to generate a design from HLS (FIR11) and implement it on the Zedboard. Also you should start to gain an understanding of different HLS optimizations. For FIR128, you should modify the code to generate at least four additional architectures. Your goal is to create architectures that provide tradeoffs between area and execution time. This will require you to rewrite the code and/or insert pragmas. Your final assignment must contain followings:

- Design a 11 tap fir filter with HLS. In the rest of this document, we use the term *FIR11* to refer this task.
- Demo for FIR11: In the rest of this document, we use the term *Demo* to refer this task.
- Design 128 tap fir filter with HLS and optimize it. In the rest of this document, we use the term *FIR128* to refer this subtask.

13.5 FIR11

The first step for the project is to get a functionally correct design working for an 11 tap FIR filter. For this, you will need to use the Vivado® HLS tool, and finish the function body of `void fir()` in the file `fir.cpp` to implement the filter. You can test the correctness of your code by using the provided testbench. This code does not need to be highly optimized; you will work on creating optimized code later. It just needs to work correctly. This is the code that you will use to program the programmable logic portion of the Zynq chip on the Zedboard. The next step requires that you set up the Zedboard to run Linux on the ARM processor on the Zynq chip, and communicate data back and forth between the ARM processor and your FIR11 core running on the programmable logic. Further details on this are explained in next section.

13.6 Demo

Following are steps to implement your FIR11 HLS design on the Zedboard. You will provide the input data (chirp signal) from the ARM, and get the output from the Zedboard to the ARM. To do that, you must write a `host_fir.c` program. We provided a template of `host_fir.c`. Find the comment `//Add your code here:` and insert your code after that. You also need to write the gnuplot script to plot the output on the GUI. To do this demo you need: a Zedboard, Mouse, Keyboard, Monitor, and USB Hub.

The specific things you must do in this section are:

- Download two files from:
  - `http://xillybus.com/downloads/xillinux-eval-zedboard-1.3a.zip`
  - `http://xillybus.com/downloads/xillinux-1.3.img.gz`
- Read Sections 1 - 4 in the Xillybus tutorial (`http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf`). We will use ISE (not Vivado) in this lab since Xillybus does not play well with the current versions of Vivado. In other words, follow the instructions in Section 3.5.1 in the Xillybus tutorial to generate your bit stream.
- Read Part I - IV in the Xillybus HLS integration tutorial (`http://xillybus.com/tutorials/vivado-hls-c-fpga-howto-1`), integrate your 11-tap FIR HLS design with Xillybus, and implement your FIR HLS design on the Zedboard. Finish the given host program `host_fir.c`. Again, the functions you need are described in this tutorial. Compile your host program (using `gcc`). Run `exe_script.sh` to get this input/output signal graph. The expected output is as shown in Figure 13.1.

You should submit the graph with your final write up as proof that you got it to work. We also reserve the right to see the demo in person.

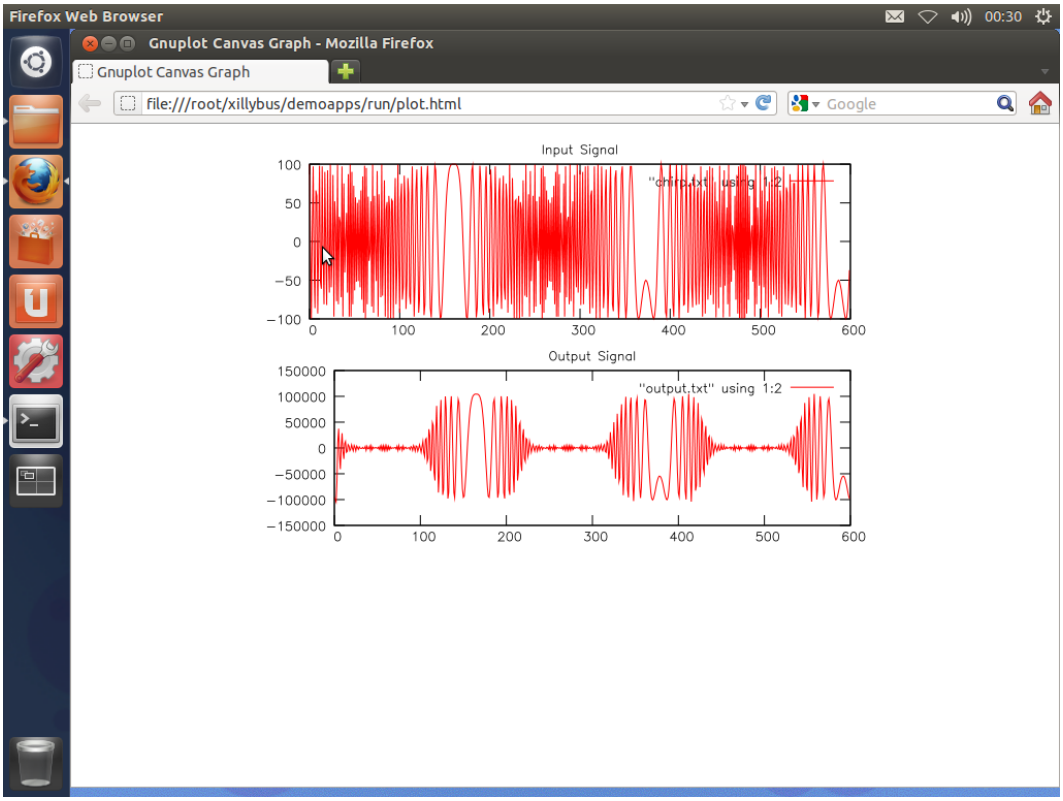


Figure 13.1: The expected output for the `FIR11` demo.

13.7 FIR128 Optimization Hints

You must generate at least one architecture that utilizes all of these optimization hints. This architecture does not have to use all of the hints, e.g., you could generate five different architectures where each one performs one of these optimizations. You are free to use any additional optimizations. The only restriction is that your resulting code is correct, i.e., it matches the golden output. Some possibilities for optimization include:

- **Variable Bitwidths:** The bitwidth of the variables provides a tradeoff between precision, area, and performance. It is possible to specify the exact size of each variable to the tools. Make sure that you do not affect the results, i.e., the output still matches the golden output.
- **Removing Conditional Statements:** If/else statements and other conditionals limit the possible parallelism and create additional area. If the code can be rewritten to remove them, it typically makes the resulting architecture smaller and faster.
- **Pipelining:** This increases the throughput at the cost of adding area (registers). The area and performance can be varied by changing the initiation interval (II).
- **Loop Fission:** Dividing the loop into two or more separate loops may allow for each of those loops to be executed in parallel. This will increase the performance and the area.

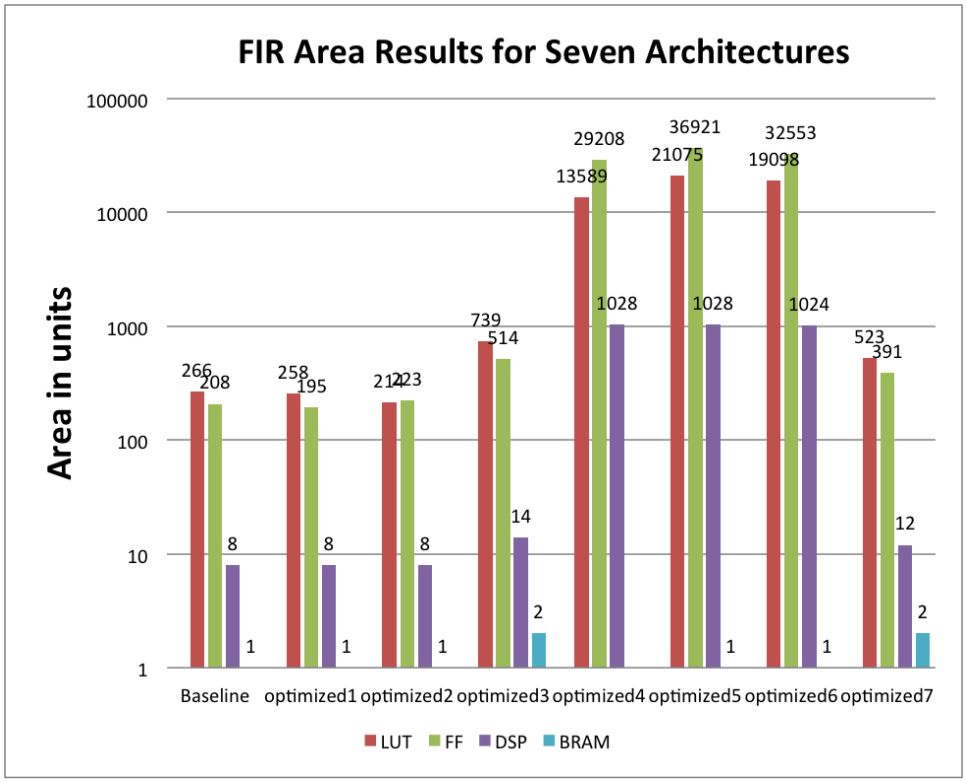


Figure 13.2: The resource usage of eight different FIR architectures.

- **Memory Partitioning:** The storage of the arrays in memory plays an important role in area and performance. On one hand, you could put an array entirely in one memory. And this memory can have a different number of ports (e.g., one or two ports for FPGA block RAM). Or you can divide the array into two or more memories to increase the number of ports. Or you could instantiate each of the variables as its own register, which allows simultaneous access to all of the variables at every clock cycle, but has high area costs.

For your designs, you should report performance and area results. Figure 13.2 and Figure 13.3 provide an example of how to plot the area and performance results. Figure 13.2 plots the number of LUTs, FFs, DSP, and BRAMs used for 8 different designs. Figure 13.3 shows the performance in terms of number of FIR outputs/second for these 8 designs. We describe how we calculate the throughput below. Your graphs do not necessarily have to look exactly like these. These are provided as examples. There are many other ways to present the results. However, it is important to make sure to present the results in the most coherent and organized manner as possible.

We calculated the throughput of the architectures in terms of number of FIR outputs per second. This requires that we calculate the total time that one call to the `fir()` function. The throughput is reported in Hz using the formula from Equation 2. We use the "Estimated Clock Period (ns)" from HLS report instead of the specified clock period.

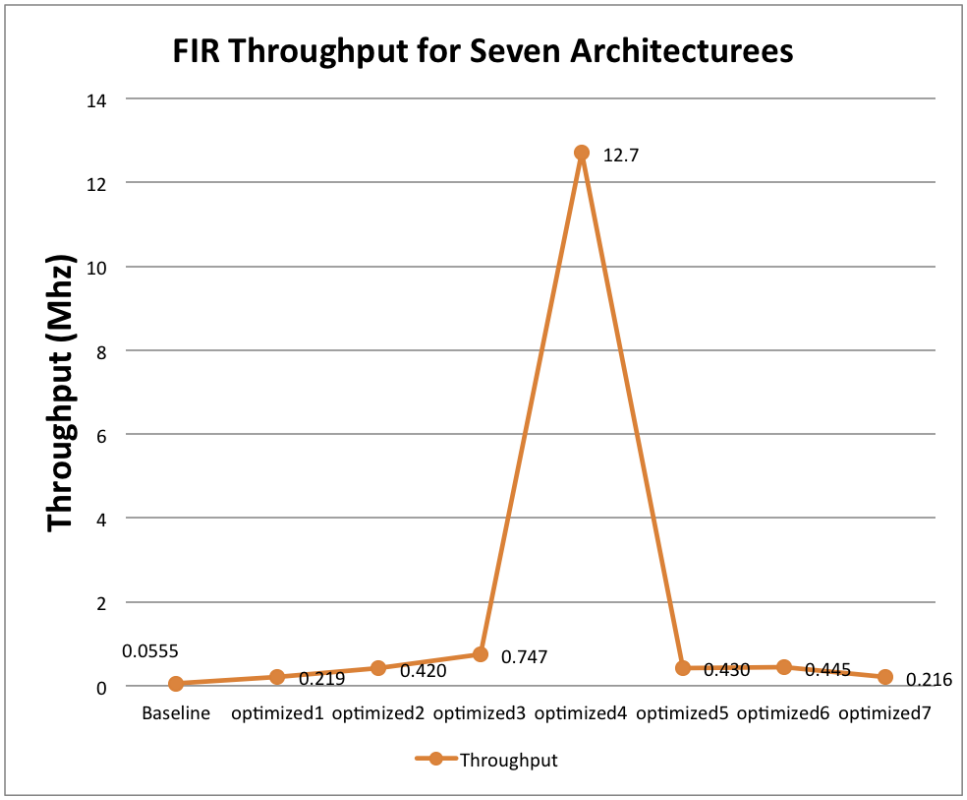


Figure 13.3: The throughput in terms of number of FIR outputs/second of eight different FIR architectures.

The throughput in Hertz can be calculated as:

$$Throughput(Hz) = \frac{1}{Clock\_Period(s) \cdot Num\_Clock\_Cycles} \tag{13.1}$$

The throughput in MHz can be calculated as:

$$\begin{aligned} Throughput(Mhz) &= \frac{1}{Clock\_Period(s) \cdot Num\_Clock\_Cycles} \cdot \frac{1}{10^6} \\ &= \frac{1000}{Clock\_Period(ns) \cdot Num\_Clock\_Cycles} \end{aligned}$$

You should always present your results using units (Hz, KHz, MHz, etc.) that make "sense". For example, you should not do 10000 Hz rather 10 KHz, and 200 KHz instead of .2 MHz.

Remove the following two sections?

13.8 Submission Procedure

You should submit a report that must include your results for the demo and each of the architecture that you generated for FIR128. For the FIR128, this report should plot the results, e.g., resource utilization in terms of BRAMs, DSP48, FFs, LUTs, etc., and performance in terms of number of throughput (number of FIR operations/second), clock cycles, clock frequency (which is fixed to 10 ns). Your report should describe each of the architectures that you generated. You should discuss the exact optimization and its resulting architecture. Figures describing the architectures are highly encouraged.

You must also submit your code (and only your code, not other files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. Or you can insert the optimizations into your TCL script. We must be able to have all of the knowledge available to easily synthesize any and all of your designs.

The folder should be zipped with a similar file structure as below: `FIR_lab_LASTNAME1_LASTNAME2.zip`, where `LASTNAME2` is included if you are submitting a joint report with another person.

Contents:

- `report.pdf`
- `demo_picture`: A screenshot or pdf version of your plot graph from the demo.
- Folder `baseline`: original unmodified `fir.cpp`.
- Folder `optimized1` first optimization with `fir.cpp`.
- Folder `optimized2`: second optimized design with `fir.cpp`
- ...
- Folder `optimizedN`: nth optimized design with `fir.cpp`.

Each folder also must contain `script.tcl` file that can be used to create the project

13.9 Grading Rubric

The working demo of your code on the Zedboard will account for 60% of your grade. You can document this in your report with a screenshot as well as some discussion as to any difficulties that you had during this implementation. Of course, we reserve the right to have you demo this for us in person. The remaining 40% requires that you generate at least one architecture that uses each of the five listed optimizations (Variable Bitwidths, Removing Conditional Statements, Pipelining, Loop Fission, Memory Partitioning).

The report should be well written and presented. Additional points (up to 20) will be subtracted for poor reports. Examples of issues include any spelling errors, multiple/egregious grammar, poor presentation of results, lack of written comparison of the results, etc. I expect there to be insights into the different architectures. Simply stating the optimizations that you did and presenting the results is not sufficient. You should attempt to answer questions as to why did this optimization produce better results. Why is one architecture better than the other? What sort of architecture is being generated? Etc. This will help you for future assignments so it is worthwhile spending time understanding the tools now.

Chapter 14

Phase Detector Project

14.1 Introduction

In this lab, you will design the phase detection part of a receiver by implementing a complex FIR filter and a COordinate Rotation DIgital Computer (CORDIC) module. You are going to build a complex FIR filter utilizing four "real" FIR filters similar to what you developed in Project 1 (see Chapter 13).

The complex FIR filter is used to correlate to a known complex signal. We use Golay codes which have some great properties related to orthogonality and auto-correlation. This is not important to this lab, but is some really amazing math. I hope you look into it.

The CORDIC function is built from scratch and will likely take a large amount of your time for this assignment. A CORDIC is an efficient method for calculating trigonometric and hyperbolic functions. CORDIC can do a lot of different functions; we will use it to convert Cartesian coordinates (x, y) to the polar coordinates $(r, theta)$.

In the end, you will combine all of these modules into something that is the beginning of a digital communication receiver. The goal is to do simple synchronization and discover the phase of the signal. The output of the CORDIC $(r, theta)$ gives you these results. It is not everything that you need to do, but it is a good start.

The provided Simulink file gives the transmitter, channel, and receiver. You are building an equivalent receiver in HLS in this project. We used this Simulink project to create the various testbenches for the project.

Finally, you will demonstrate a working system of the complete phase detector using Xilinux on the Zedboard.

14.2 Materials

The provided folder has a number of subfolders and files corresponding to the different parts of the phase detector. This contains the documents necessary to build the project. In this folder you will see two folders. One is HLS and other is demo. You will start from HLS folder to design your phase detector using Vivado® HLS.

- `HLS/fir` folder: This folder contains `*.cpp`, `*.h`, and script files for a complex FIR filter. The complex is matched filter. You are matching the complex I and Q Golay

codes. You can to design the complex FIR filter using your `fir()` function Project 1. In the `fir.cpp` file, there are four sub functions `firI1`, `firI2`, `firQ1`, and `firQ2`. These functions are real FIR filters i.e., the same that you designed in Project 1. You can use your favorite code from Project 1 in these four sub functions. In the complex fir filter, four of these functions are used in the `fir` function. In that function, you need to connect the four FIR filters `firI1`, `firI2`, `firQ1`, and `firQ2` to an adder and subtractor to create the complex matched filter. This structure is demonstrated in the Simulink file.

- `HLS/cordic` folder:
  - `cordiccart2pol.cpp`: The place where you write your synthesizable code. Currently, it only contains the function prototype.
  - `cordiccart2pol.h`: Header file with various definitions that may be useful for developing your code.
  - `cordiccart2pol_test.cpp`: Testbench
  - `script.tcl` and `directive.tcl`: Use this to create your project
- `HLS/phasedetector` folder: After you design the CORDIC and the complex FIR, you will use them to design the phase detector.
  - `fir.cpp`: Real FIR filter that you designed in Project 1.
  - `cordiccart2pol.cpp`: `cordiccart2pol` function you will design in this project.
  - `phasedetector.cpp`: Top level skeleton for the phase detector. You use `fir.cpp` and `cordiccart2pol.cpp` to design the `phasedetector` function.
  - `phasedetector_test`: Testbench
  - `phasedetector.h`: Header file
  - `script.tcl` and `directive.tcl`: Use this to create your project
- `Demo` folder: Demo folder for 11 tap filter
  - `host.c`: Unfinished host program
  - `exe_script.sh`: Execution script
  - `plot_script.p`: Plotting script
  - `input_i.dat`:Input I signal
  - `input_q.data`: Input Q signal

14.3 Tasks

In this project, you will build a phase detector to process the given I and Q signals shown in Figure 14.1. The final goal is to implement this phase detector on the Zedboard. To achieve this goal, you will need to finish the following tasks:

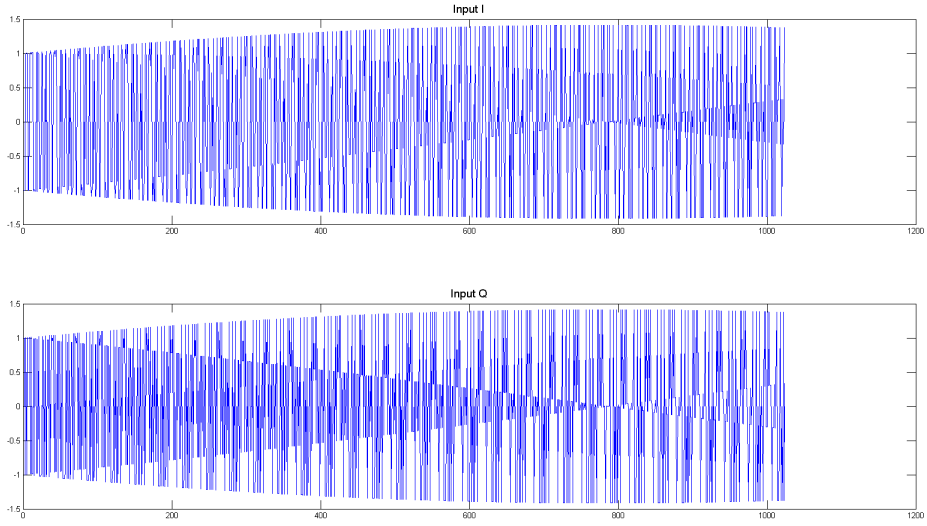


Figure 14.1: Input I and Q values to your complex FIR filter.

1. Implement the complex matched filter and verify it with the given testbench. This matched filter consists of four FIR filter modules which are similar to the ones in your Project 1. For the purpose of debugging, if you plot the outputs of this step, you should expect the waveforms as shown in Figure 14.2.

2. Implement the CORDIC HLS code and verify it with the given testbench.

3. Connect the complex matched filter and CORDIC modules to implement the receiver. Verify this overall design with the given testbench. For the purpose of debugging, if you plot the outputs of this step, you should expect the waveforms as shown in the figure below.

4. Implement the receiver design on the Zedboard. This process is similar to the demo in your Project 1. But this time, we will generate the bit file using Vivado instead of ISE. Please read the "Hints" section for some tips about this. The expected output is as shown in Figure 14.3.

The thetas at the R peaks are `0.015529, 0.047509, 0.079485, 0.111526, 0.143491`, etc.. This is the rotated phases that have been detected by your design.

14.4 Hints

- Start building the subcomponents and get working versions, i.e., start with `HLS/fir` and `HLS/cordic`. Verify them independently.

- Once you design the `HLS/fir` and `HLS/cordic`, move the `HLS/phasedetector`.

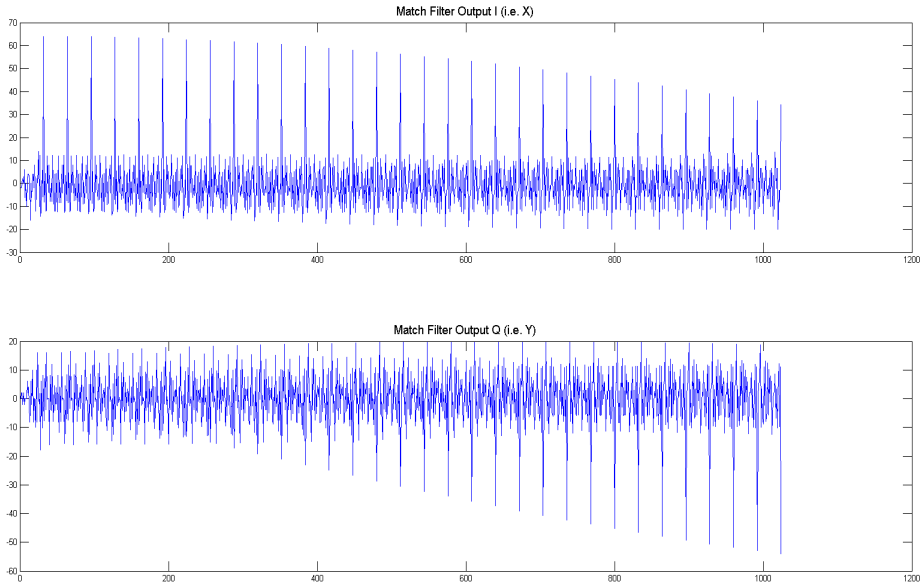


Figure 14.2: Output values from the complex FIR filter corresponding to the inputs from Figure 14.1.

- Once you have final functions verified, using the same code start to make the Zedboard demo.
- Due to Xillybus restrictions, you must use Vivado 2014.1 for the demo. You can use 2014.1 for the entire project.
- When you synthesize your `xillybus_wrapper` HLS project, make sure you set the RTL tool to Vivado (see Figure 14.4).
- After you synthesize your `xillybus_wrapper` HLS project, you will need to export the RTL. Make sure you follow the settings as specified in Figure 14.5 when exporting the RTL.
- To create a Vivado project: In `xillinux-eval-zedboard-1.3a` → `verilog`, you will find `xillydemo-vivado.tcl`. In Vivado, click `Tools` → `Run Tcl Script` to run this tcl file of xillydemo. It will create the Vivado project which is the replacement of the ISE project we used in Project 1.
- Now you need to import the RTL IP core you exported from your `xillybus_wrapper` HLS project to the Vivado design suite. In `Project Manager`, click `IP catalog`, then click `IP Settings`. In `IP Settings`, click `Add IP`. You need to locate the `xilinx_com_hls_xillybus_` file generated by your `xillybus_wrapper` HLS project. It is in your `HLS project folder` → `solution1` → `impl` → `IP`.
- You will see a folder called `VIVADO HLS IP` in your IP Catalog view. There is your `xillybus_wrapper` IP. Double click it to generate a core of this IP. Keep the default

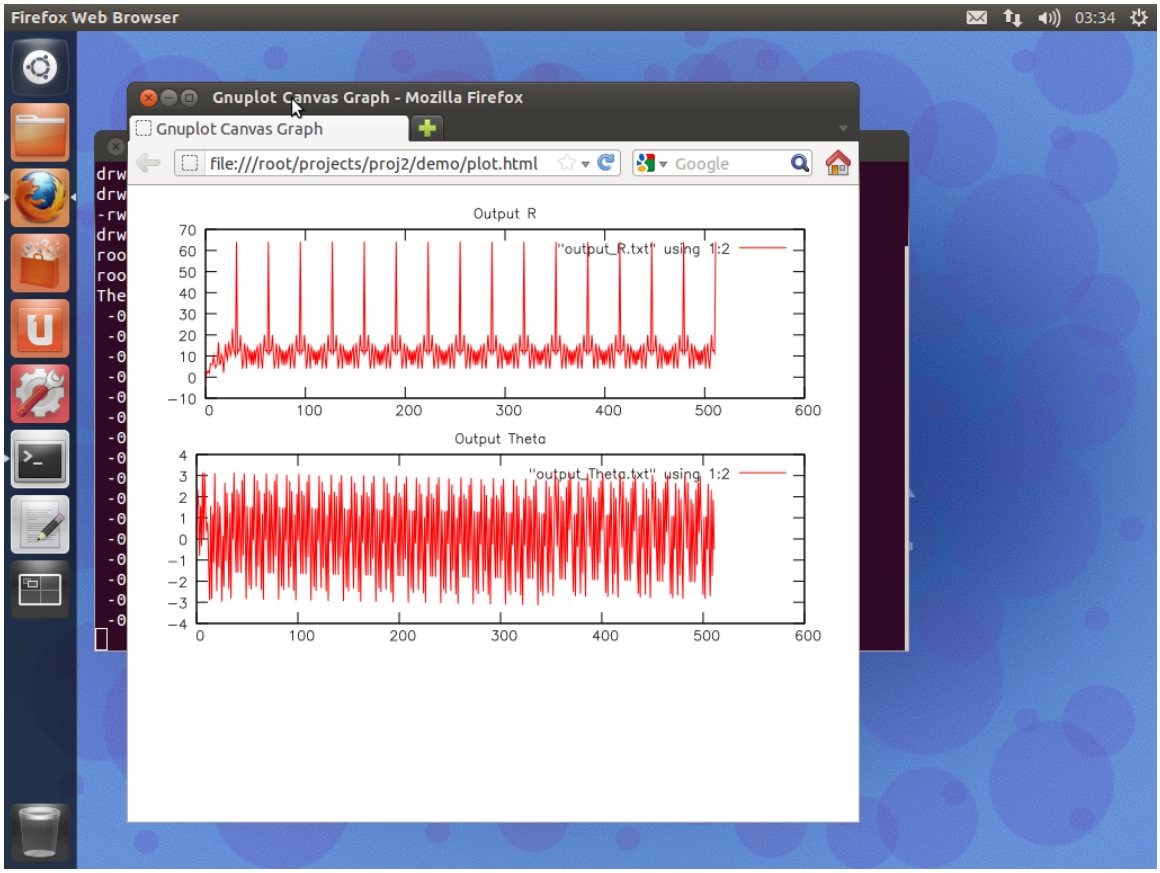


Figure 14.3: Example output plot for the phase detector demo.

setting in every step of this process. Remember the component name of your IP core, which should be `xillybus_wrapper_0` by default. You can change this component name to anything as long as you use the same name in your `xillydemo.v` Verilog code.

- Now, modify `xillydemo.v` code. This is similar to what you did in Project 1. Again, make sure you use the consistent name where you instantiate your `xillybus_wrapper` IP core.
- Vivado is very picky about unconnected wires. Therefore, make sure you do not have any unconnected components. Hint: if you do not use xillybus debug, do not instantiate `fifo_8x2048` in `xillydemo.v`.
- If everything is connected correctly, you should be able to generate the bit file from the Vivado project.

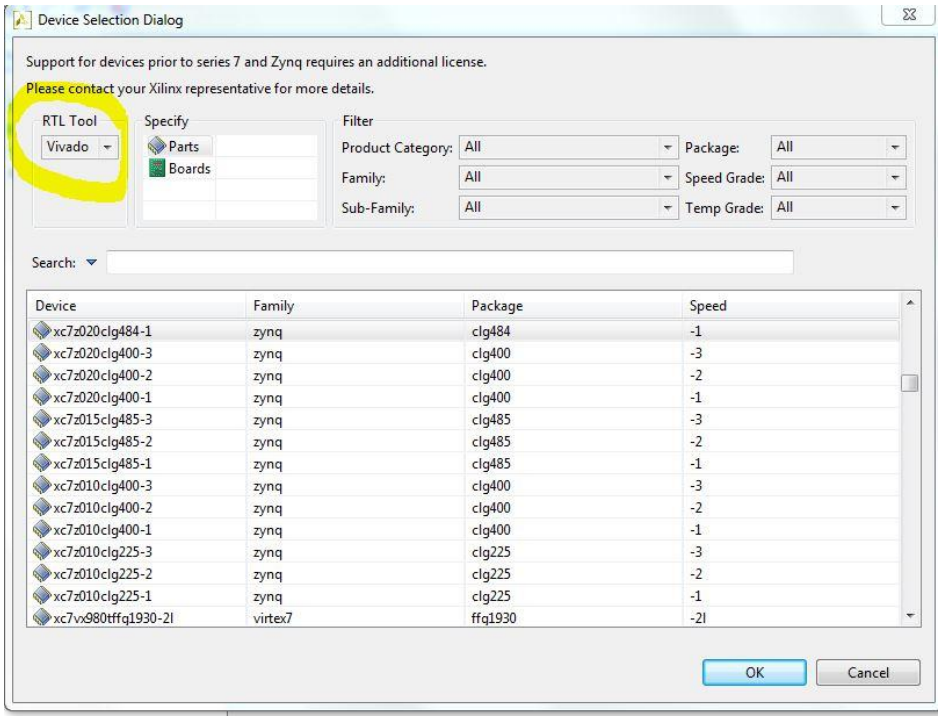


Figure 14.4: Make sure to select Vivado as the design tool.

14.5 Report

Your report should mostly focus on the design of your CORDIC core. For this, I want you to use understand how the precision of the CORDIC core affects the performance and area of your designs. The precision is affected by the number of rotations, and also the bitwidths of the data types, both the data types for the input and output arguments, and the data types of the internal variables.

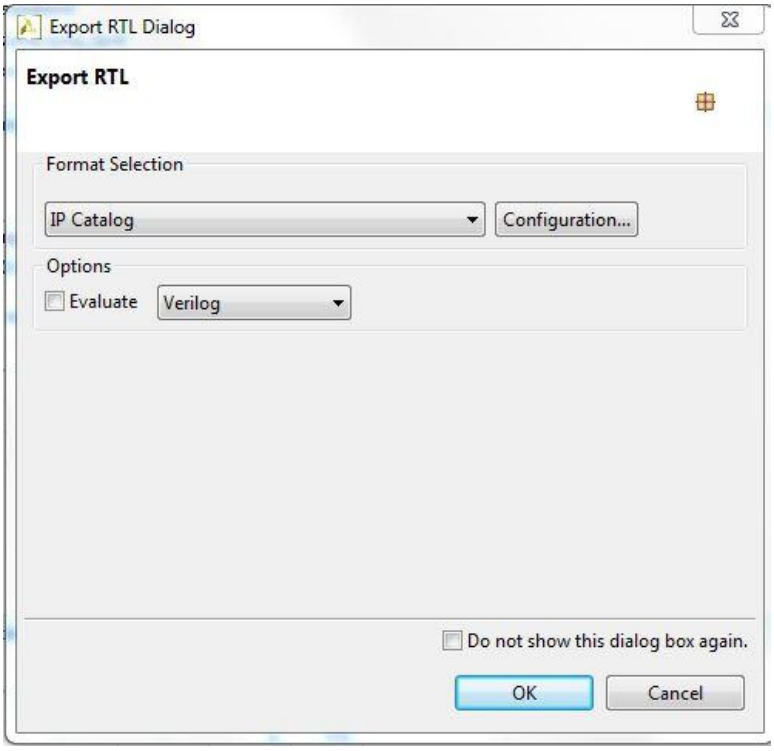


Figure 14.5: Required settings for exporting RTL into XIllybus.

You should investigate both of these sources of imprecision by generating a number of different designs. One should plot how the number of rotations affects the performance (throughput/latency) and the area (DSP48s, BRAMS, LUTs, FFs).

The second source of imprecision is how the number of bits for the fixed point variables that you use. Again, you should investigate how changing the fixed point bits for the variables affects the precision, area, and performance.

In both cases, you should plot some graphs. Talk about the trends. Give insights into why these trends make sense. Talk about trends or data points that do not make sense.

Some ideas to discuss in your report:

1. What is the total throughput of your Phase Detector? How does that relate to the individual components (FIR, CORDIC, etc.)? How can you make it better?

2. I am looking for deep insight into the design. Why does the accuracy stop improving after so many iterations? What is the minimal amount of bits required for each variable? Does this depend on the input data? If so, can you characterize the input data to help you restrict the number of required bits? Do different variables require different number of bits?

3. The ultimate goal of the CORDIC is not to use only simple operations, i.e., add and shift. You should not be using divide, multiply, etc. in your CORDIC core.

4. How does the ternary operator `?` synthesize? Is it useful in this project?

Chapter 15

Discrete Fourier Transform (DFT) Project

15.1 Introduction

The goal of this project is design architectures that implement the Discrete Fourier Transform (DFT). The DFT is a common operation in signal processing which generates a frequency domain representation of the discrete input signal. We start with the direct implementation of the DFT which is a matrix-vector multiplication. The input signal is a vector of samples and the matrix is composed of a set of basis functions corresponding to discrete cosine and sine waveforms of different frequencies. The multiplication of the input signal with these basis functions describes how well the input signal correlates with those waveforms, which is the value of the Fourier series at that frequency.

15.2 Materials

You can find the following files in `project2-dft.zip`. These are divided into four folders: `dft_8_precomputed`, `dft_32_precomputed`, `dft_256_precomputed`, and `dft_1024_precomputed`. Each of these folders has its own testbench, `out.gold.dat` file, and `coefficients.h` file.

Each folder contains following files:

- `dft.cpp`: The baseline implementation for the dft function.
- `dft.h`: Header file holding important constants and other variables.
- `dft_test.cpp`: Testbench
- `coefficientsX.h`: A file containing the values of corresponding to one sine/cosine period sampled based upon the DFT points. For example, an 8 point DFT has 8 samples across both the sine and cosine function evenly spaced across one period. This is equivalent to dividing one rotation in the complex plane equally by the number of points in the DFT.
- `out.gold.dat`: "Golden" output. The testbench (dft_test.cpp) generates a sample input and calls the function `dft` in `dft.cpp` with that sample input. This output of the function is compared to the expected output. This will indicate PASS or FAIL. If

it fails, then the code in `dft` is incorrect. There are four different versions of depending on the DFT size and way in which the DFT coefficients were generated.

- `script.tcl` and `directives.tcl`: Used to create the project.

15.3 Project Goal

You should modify the code to create a number of different architectures that perform tradeoffs between performance and area. For `dft_256_precomputed` and `dft_1024_precomputed` designs, you need to use precomputed values from `coefficients256.h` and `coefficients1024.h`.

For 256-point and 1024-point DFTs, you will create a report describing how you generated these different architectures (code restructuring, pragmas utilized, etc.). For each architecture you should provide its results including the resource utilization (BRAMs, DSP48, LUT, FF), and performance in terms of throughput (number of FFT operations/second), latency, clock cycles, clock frequency (which is fixed to 10 ns). You can do most of the design space exploration on the 256 point DFT. You should pick your "best" 256 architecture and synthesize that as a 1024 DFT.

The 8 and 32 point folders are provided for your convenience. If you would like, you can do some of your initial design space optimization on these smaller architectures. But it is not necessary to use these at all.

The key in this project is to understand the tradeoffs between loop optimizations (unrolling and pipelining) and data partitioning. Therefore you should focus on these optimizations.

15.4 Optimization Hints and Guidelines

- You should use a clock period of 10 ns.
- The output of your architecture must closely match the golden output. Be sure to generate a working function before performing any optimizations. If the result does not match exactly, but is close, please explain why in the report.
- You should use float for all data types. You do not need to perform bitwidth optimization of this project.
- The current design is set up to do the DFT in-place, i.e., you put the output results back into the same arrays that give you the input results. You can change this if you think it will give you better results.
- There are many different ways to generate the DFT coefficients. These are constants when the DFT size is fixed. We have given you the coefficients for both 256 (in `coefficients256.h`) and 1024 (in `coefficients1024.h`). They each have two constant arrays, `sin_table` and `cos_table`. You can use these coefficient arrays directly as memories in your architectures. You are also free to create your own arrays using a different structure (e.g., 2D array, reordering of elements in the given 1D arrays, etc.). Or you could dynamically generate the coefficients.

- There is significant amount of parallelism that can be exploited by (partially) unrolling the for loops. Pipelining these (partially) unrolled for loops should lead to higher throughputs.
- There are more efficient methods for performing the DFT that exploit the symmetries of the Fourier constants, e.g., the fast Fourier transform (FFT). Do not use these symmetries. In other words, treat this like a matrix-vector multiply with unknown matrix values. Do not worry, we will implement FFT architectures in a following project that will fully take advantage of these symmetries.
- You do not need to report your optimizations for your 8 point and 32 point DFT; these folders are provided for your convenience. Since these will very likely synthesize much faster than larger point DFT functions, it may be useful to use these to debug your code or in your initial design space exploration.
- Your report must explicitly state how you calculated the throughput results. Note that this is often not simply a function of the latency and the clock period, and involves using the initiation interval.

Chapter 16

Orthogonal Frequency-Division Multiplexing (OFDM) Project

16.1 Introduction

In this project, you will learn the very basic idea behind an OFDM system, and implement a simple OFDM receiver in programmable logic. The project is divided into three parts. The first provides an OFDM transmitter and receiver in Simulink. This gives you insight into the ideas behind OFDM. In the second part, you develop a basic OFDM receiver using HLS. The receiver consists of a Fast Fourier Transform (FFT) module and a QPSK symbol decoder. The final part is to integrate the receiver onto the Zedboard using Xillybus to transmit data to the OFDM receiver, and receive the decoded data back from your hardware implementation in the programmable logic. You should use Gplot to show the transmitted and received data as you have done in previous projects.

16.2 Part I: Simulink

We provided a Simulink file with an OFDM transmitter, simulated channel, and receiver design. The goal of this part of the project is to gain an initial understanding of how a very simple OFDM system works. Furthermore, you will use the Simulink file to extract data as a testbench for Parts II and III. In this part, we walk you through a series of steps to familiarize you with Simulink and OFDM. This part is optional in the sense you do not have to turn anything. Though you will need to have at least a decent understand of the Simulink in order to complete the assignments from the next two parts of this project.

Because the HDL-compatible FFT block inherently bit-reverses, there are two different Simulink models.

- Version a: No bit reversal in TX, bit reversal in both HDL and behavioral/algorithmic RX, and thus would provide suitable incoming data for a non-bit-reversing Vivado FFT receiver.
- Version b: Bit reversal in TX and in both RX thus be used with a bit-reversing Vivado FFT receiver.

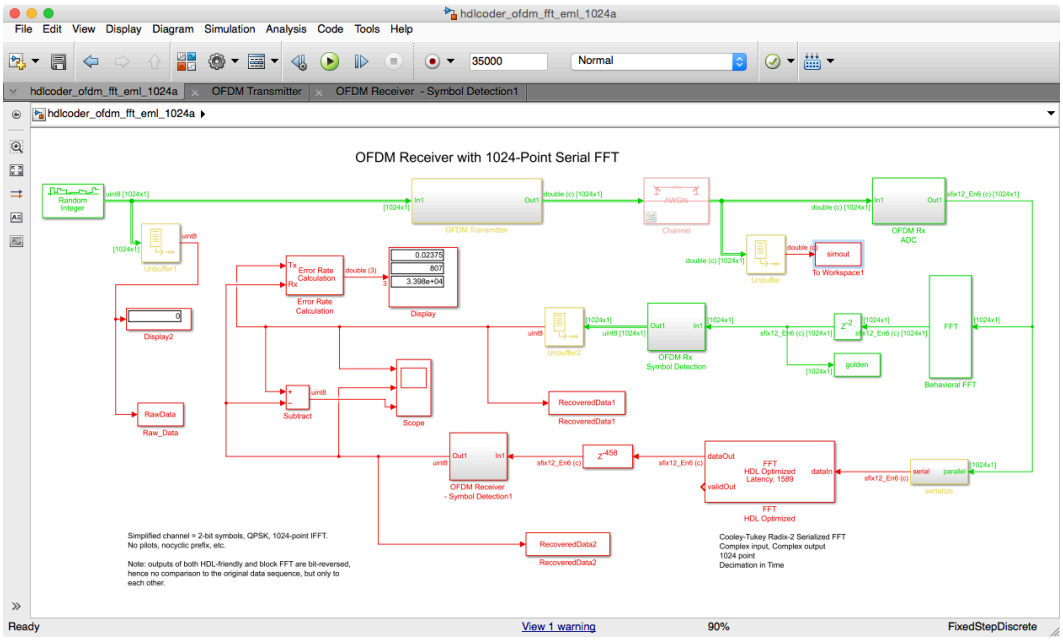


Figure 16.1: Simulink test bench for a simple OFDM transceiver. This is found in the Simulink file `hdlcoder_ofdm_fft_eml_1024b.slx`.

The simple OFDM transmitter encodes 2-bit data symbol (either values of 0 through 3 or random) onto a digital QPSK modulation pattern. 1024 of these symbols go through a 1024-point blockwise IFFT and are then converted to floating point and sent over a simulated AWGN channel. Two receiver architectures are provided in parallel: HDL-optimized FFT and block FFT. You can synthesize the HDL FFT into Verilog and then run that through Vivado (not HLS) to obtain FPGA resource use estimates.

Figure 16.1 shows the top level of the provided `hdlcoder_ofdm_fft_eml_1024b.slx` file. The 2-bit/symbol data source is at upper left, the transmitter (QPSK + IFFT) is top center, then the AWGN simulated channel and the OFDM A/D converter. We do everything except the channel math in fixed point, since we are trying to build real hardware. The channel, is not synthesizable into Verilog or hardware, is kept in floating point, in deference to the computational limits of Matlab, which needs to add Gaussian-distributed random numbers to what we transmit into the channel.

Run the simulation and note the resulting bit error rate and scope traces. Note that the transmitter input and the receiver output have been manually time-aligned for you. In a real system this function will be handled by the synchronization correlators (matched filters), such as those we studied in an earlier projects (e.g., Chapter 14). The z-2 block delays the `BehavioralFFT` output by two frames = 2048 sample times, to compensate for the 1589 delays through the HDL Optimized FFT, the 458 delays after it, and one delay for serializer in front of it. You can see from the scope that the two have been aligned by the external delays.

The third scope channel is feed from subtractor that compares the two parallel receiver constructs (see Figure 16.2). We did something like this in earlier labs, to provide convenient visual verification that two data streams matched. **Where is the 2% error coming from?**

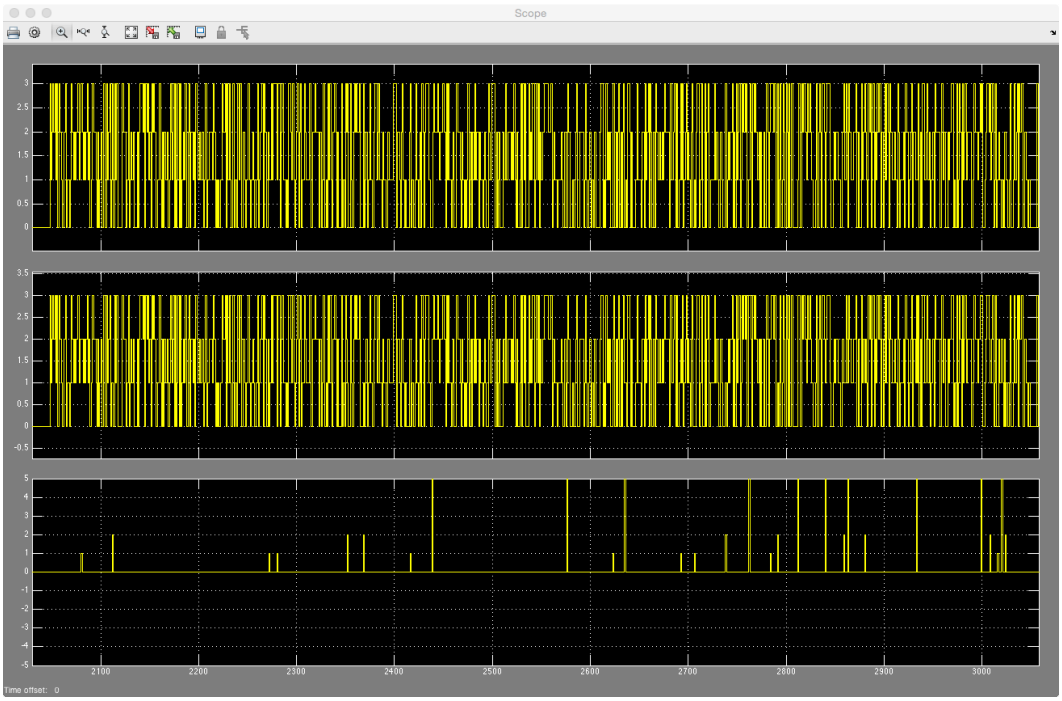


Figure 16.2: The first two plots shows signal outputs from two different FFT paths in `hdlcoder_ofdm_fft_eml_1024a`. The third scope plots the difference between these two outputs.

Hint: try scoping the respective outputs of the BehavioralFFT and FFT HDL Optimized blocks – if the FFTs match, then what is causing the disparity?

Click on `Analysis` at the top of your screen, then `Fixed Point Tool`, then make sure `hdlcoder_ofdm_fft_eml_1024a` is selected and change `Data type override:` to `[Floating Point] Double [precision]`. What happens to the bit error rate? What is likely going on here?

Important note: Since the hardware-optimized FFT generates results in bit-reversed order, we have set the block FFT to do the same. Thus, we compare the (bit-reversed) results from the two receivers, instead of comparing one receiver's output against the original data sequence that was transmitted. We need to add a bit reversal block to the design to do a proper TX-RX bit error rate comparison, but for the moment take the block FFT as your golden standard.

Now lump the `FFT HDL Optimized`, `458-clock delay`, and `OFDM Receiver Symbol Detection` blocks into a single subsystem (a series of left clicks, one per block, with Shift held down will do nicely), right click on it, and start HDL advisor.

Run the resulting Verilog code through Vivado and report projected FPGA resource usage. Do not worry yet about putting this into a Xillybus environment.

Now open `hdlcoder_ofdm_fft_eml_1024b.slx`. This is similar, except that we are now checking true symbol error rate of recovered data versus originally transmitted data. How were we able to do this, given that the two FFTs both output bit-reversed data? Also, why are the two error/difference traces on the scope so different? One runs as expected between -3 and +3, but the other has huge spikes of about 255.

Describe the two data sources, and try each one in turn by clicking on the `Manual Switch`. Feel free to invent your own data source generator and to swap it in.

Important points from this lab:

1. Simplified 1024-point OFDM system with data on every channel.
2. AWGN channel simulation available – will inject Gaussian noise at a user-selected SNR.
3. Quantization – we chose 12-bit precision for the receivers' input – was this enough? What is our practical upper limit of bits?
4. Bit reversal – the data emerge from the receiver in bit-reverse order, but we can compensate if we tell the transmitter its incoming data are also in bit-reverse order.

16.3 Part II: High Level Synthesis

The major portion of the OFDM receiver is a 1024-point FFT. The FFT is a more efficient version of the Discrete Fourier Transform (DFT). The FFT utilizes symmetry in the DFT coefficients to provide a recursive implementation that reduces the runtime from O(N2) to O(N log N) where N is the number of samples in the input signal.

Your tasks for this part of the lab are:

1. Implement a working FFT module that passes the testbench in HLS.
2. Optimize the FFT module (create and explore multiple architectures)
3. Implement and optimize the QPSK decoder.
4. Integrate the FFT and decoder into a complete OFDM receiver.

16.3.1 Materials

You are largely responsible for generating the testbenches for this project. This can be done by extracted data from the provided Simulink file. We do however provide you with testbenches for the FFT. This is the major part of the receiver.

You are given a zip file with three folders `0_Initial`, `1_Subcomponents`, and `2_Skeleton_Restructured`. Folder `0_Initial` contains the files corresponding to the "software" version of the FFT. Folder `2_Skeleton_Restructured` provides a framework for a more optimized FFT implementation. And folder `1_Subcomponents` has a number of subfolders that allow you to create projects for individual functions that you will develop over the project. This is largely for your convenience for testing and development. All of the code developed here will eventually be placed in to `0_Initial` and `2_Skeleton_Restructured`. The structure of each of these folders is largely the same.

- `*.cpp`: The place where you write your synthesizable code.
- `*.h` header file with various definitions that may be useful for developing your code.

- `*_test.cpp`: Testbench
- `out.gold.dat`: "Golden" output. The testbench (`*_test.cpp`) generates a sample input and calls the corresponding function in `*.cpp` with that sample input. This output of the function is compared to the expected output. This will indicate PASS or FAIL. If it fails, then the code in `*.cpp` is incorrect.
- `script.tcl` and `directive.tcl`: These allow you to easily create a project. To do this, open the Vivado® HLS Command Prompt tool (`Start` → `Xilinx Design Tools` → `Vivado HLS` → `Vivado HLS Command Prompt`) and go to the directory where source files and script files reside by using `cd` command. Then type `vivado_hls script.tcl`. This will create a HLS project automatically, so you do not have to add source files, set the top function, select the target device, etc. This will create a folder called `hls` in that directory with the project. It will also synthesize the project.
- There is a `README.txt` which contains a link to the project instructions (this file). **probably remove this.**

16.3.2 Goals

Part II of the project is divided into stages. The first part of the project is to perform the bit reversal of the input data. Then you optimize a "software" version of the code which we have given you (minus the bit reversal portion). After that, you will create a more hardware friendly FFT architecture. We have provided a testbenches for the individual functions in addition to the testbenches for the overall FFT. Finally, you must design and implement at QPSK decoder, and integrate it with the FFT to complete the receiver.

While the major goal of this project is create a functional core, you will also perform optimizations on the code. You should modify the code to create a number of different architectures that tradeoff between performance and area. You will create a report describing how you generated these different architectures (code restructuring, pragmas utilized, etc.). For each architecture you should provide its results including the resource utilization (BRAMs, DSP48, LUT, FF), and performance in terms of throughput (number of FFT operations/second), latency, clock cycles, clock frequency (which is fixed to 10 ns).

16.3.3 FFT Bit Reversal

The first step in most optimized FFT implementation is to reorder the input data by performing "bit reversed" swapping. This allows for in-place computation of the FFT, i.e., the resulting "frequency domain" data (as well as the intermediate results) can be stored into the same locations as the input "time domain" data. In addition, the output frequency domain data will be in the "correct order" at the end of the computation.

An example of the bit reversed data for an 8 point FFT is shown in Table 16.1.

In other words, the input data that was initially stored in the array at location 1 is stored in location 4 after the bit reversal is completed. The input data stored in the array at location 4 will be put in array location 1. The input data stored in locations 0, 2, 5 and 7 stay in those locations. Note that this is only true for an 8 point FFT. Other sizes of FFT

Table 16.1: The data for an FFT must be swapped in a "bit reverse" manner. The table shows the decimal value, corresponding binary value, and the reversed binary and decimal values for an 8-point FFT.

Input Decimal Address	Input Binary Address	Reversed Binary Address	Reversed Decimal Address
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

will have different reordering of the data though it is still based on the bit reversed pattern. For example, in a 16 point FFT, the input data stored in location 1 (binary 0001) will be relocated into location 8 (binary 1000).

You should create an architecture that, efficiently as possible, transforms the input data into a bit reversed order. Note that there are many "software" implementations of this that will not effectively map to "hardware". While the first goal is to get a working function, you should also consider the performance of the architecture.

We have given you a set of files that allows you to develop and test this bit reversal code in isolation. This includes a simple testbench that exercises this function directly. You should develop and optimized your bit reversed code here. You will later copy this code into the FFT code.

This code is in subfolder `1_bit_reverse` in the folder `1_Subcomponents`. You should develop your code here to insure that it matches the expected result. Note that this testbench is exercising only one input/output result. In other words, even if it passes this, it may not pass all results. Feel free to add additional testbenches to insure your code is correct.

The bit reverse function has the following prototype: `void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE])`

You should perform the swapping "in place" on the data in both of the real and imaginary portions of the data. That is the input data in both `X_R` and `X_I` will be reordered when the function completes.

Your report should describe your code and its performance and area results. Focus on how you modified your code in order to make it more "hardware friendly".

Hint: Logical operations map well to hardware. Calculating the indices of the arrays that should be swapped can be done with logical operations.

16.3.4 Optimizing the "Software" Version of the FFT

The next portion of this lab performs optimization on a typical software implementation of the FFT. You are given typical three nested loop implementation of the FFT in the folder `0_Initial`. First, you should understand in detail what this code is doing. It is worth spending time on this now as you will have to rewrite the FFT in a more hardware friendly manner in the next steps. You can reuse some of this code in those steps.

You should optimize this code as much as possible and detail these optimizations and their performance area results in your report. The results of the code will be poor; it will likely have greater than 250 million cycles. The throughput here is likely much worse than running this in software on a microprocessor. This often happens when we put the initial software versions of an application into a high level synthesis tool. And it should not be all that surprising. The code is optimized to run quickly in software, which runs largely in a sequential model of computation. The code must typically be carefully optimized with the final hardware architecture in mind to get good results. This involves exploiting parallelism and pipelining.

You will also notice that the first loop has function calls to sine and cosine. This code will synthesize quickly with these function calls. However, you may wish to replace these function calls (which will synthesize into CORDIC cores), into table lookups. We have provided two tables in the `.h` file, `W_real` and `W_imag` which contain the precomputed twiddle factors for our 1024 FFT, i.e., `W_real[i] = cos(2.0*pi*i/SIZE)` and `W_imag[i] = sin(2.0*pi*i/SIZE)` where $i = [0, 512)$.

Some potential optimizations include:

- Using the `W_real` and `W_imag` tables.
- Pipelining
- Loop unrolling
- Memory partitioning

In the report, you should discuss the performance and area numbers of the various optimizations that you apply. This must describe the effect of these optimizations on the architecture, and not just state the optimization that you used.

16.3.5 Hardware Friendly FFT Implementation

A good architecture will selectively expose and take advantage of parallelism, and allow for pipelining. Your final FFT architecture will restructure the code such that each stage is computed in a separate function or module. There will be one module for bit reversal that you have already developed, and then $\log N$ stages (10 in our case) for the butterfly computations corresponding to the 2-point, 4-point, 8-point, 16-point, ... FFT stages.

The skeleton code for this final FFT implementation can be found in the `2_Skeleton_Restructured` folder. This creates code connects a number of functions in a staged fashion with arrays acting as buffers between the stages. Figure 16.3 provides a graphical depiction of this process.

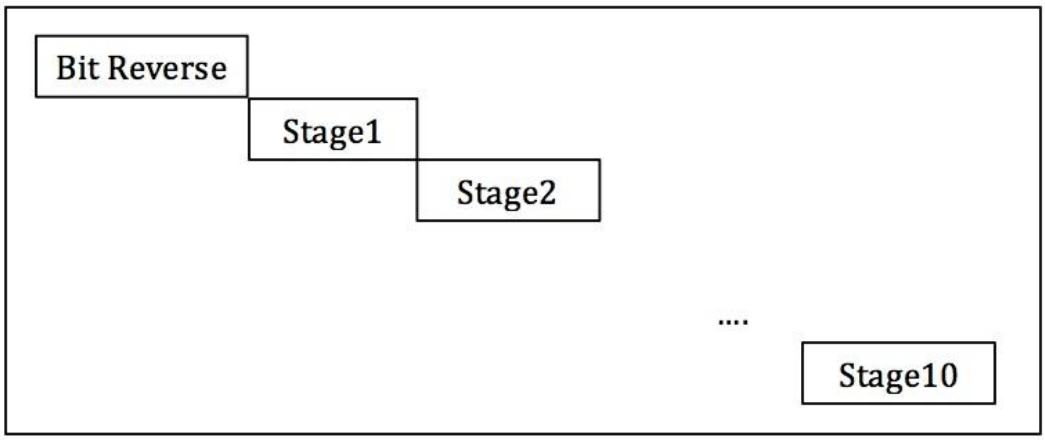


Figure 16.3: A staged implementation of a 1024 FFT. Bit reversal is followed by 10 stages of butterfly computations. This architecture is capable of pipeline both within the stages and across the stages.

The first step in this process is to create code that computes the first and last stages of the FFT. The hope is that this will allow you to get a better understanding of exactly how memory accesses and the butterfly computations are performed in a general case. You can develop these two functions `fft_stage_first` and `fft_stage_last` in isolation. They both have subfolders in the `1_Subcomponents` folder. Once these are working correctly, you can copy and paste the code directly in the same functions in the `2_Skeleton_Restructured` project.

The next task is to create code that can implement "generic" function, i.e., one that can compute any stage of the FFT. This is the function `fft_stages` which also has its own project in the `1_Subcomponents` folder. Note that this function prototype is similar to `fft_stage_first` and `fft_stage_last` with one major difference: it has a stage argument. This code will used to implement stages 2 through 9 in the `2_Skeleton_Restructured` project.

Hints:

1. These stages are performing the same calculation as one iteration of the outer for loop in the `0_Initial` project.

2. The major difference between the stages is what data elements you are performing the butterfly functions on, i.e., in what order do you pull data from `X_R` and `X_I`.

3. Test each of the functions in isolation with the provided projects. Make sure that the code compiles and passes the testbench before attempting any optimizations.

Once you have a correctly functioning set of functions, you should copy and paste them in the `2_Skeleton_Restructured` project and make sure that it passes the testbench. Since our testbenches perform one check, which is far from comprehensive, it is possible, though hopefully unlikely, that you have some error that the `2_Skeleton_Restructured` testbench exposes and was not exercised in the individual testbench. If your code passes the `2_Skeleton_Restructured` project you can assume it is correct (though again since it

is only one test, it may be wrong; you should perform significantly more testing if you want this to be anywhere close to production quality).

Now on to the final part of the project, optimizing of this restructured code. You should perform the typical tricks here: pipelining, memory partitioning, unrolling, etc. Some of these may not make sense depending on how you wrote your code. This final architecture should be orders of magnitude better than the `0_Initial` project. Highly optimized FFT architectures can easily have less than 10000 cycles.

You should describe in your report the baseline results and the effects of any optimizations that you performed. By this point, you should be starting to understand what optimizations will work and which will not work, and most importantly why. Describe these insights in your report.

16.3.6 QPSK Decoder

The decoder takes the 1024 I/Q outputs from the FFT and decodes them into the corresponding binary data. Essentially you have to reverse the encoding that was done in the transmitter. The encoder went from 0 and 1 to -1 and 1. Because you have a perfect channel, the output of your FFT will be -1 and 1. You must make those into 0 and 1. You can look at the Simulink file to determine the exact encoding. It used a Gray code, and the constellation taken directly from the Simulink file is shown below.

The output of your encoder should be the exact data that was given to the OFDM receiver in the Simulink file. You can create your testbench to reflect this.

16.3.7 Receiver Integration

You should connect the FFT and the QPSK decoder together to form the complete OFDM receiver. The input to the receiver is the data from the channel. The output of the receiver should match the transmitted data.

16.3.8 Optimization Hints and Report Guidelines

- You should use a clock period of 10 ns. While varying the clock period can drastically change the results, this is not a focus of this project. Though feel free to modify this and observe the changes.

- The output of the various architectures that you generate must match the golden output. We have broken down the project into subcomponents to allow you to develop and test them individually. You would be wise to do it in such a manner.

- You should not change the data types as given to you. You do not need to perform bitwidth optimization of this project.

- There are some variable defined in the `.h` files for you convenience. These include `SIZE = 1024`, `SIZE2 = 512`, and `M = 10` (i.e., log SIZE). Feel free to use these in your code. They are defined in every `.h` file across all of the different folders.

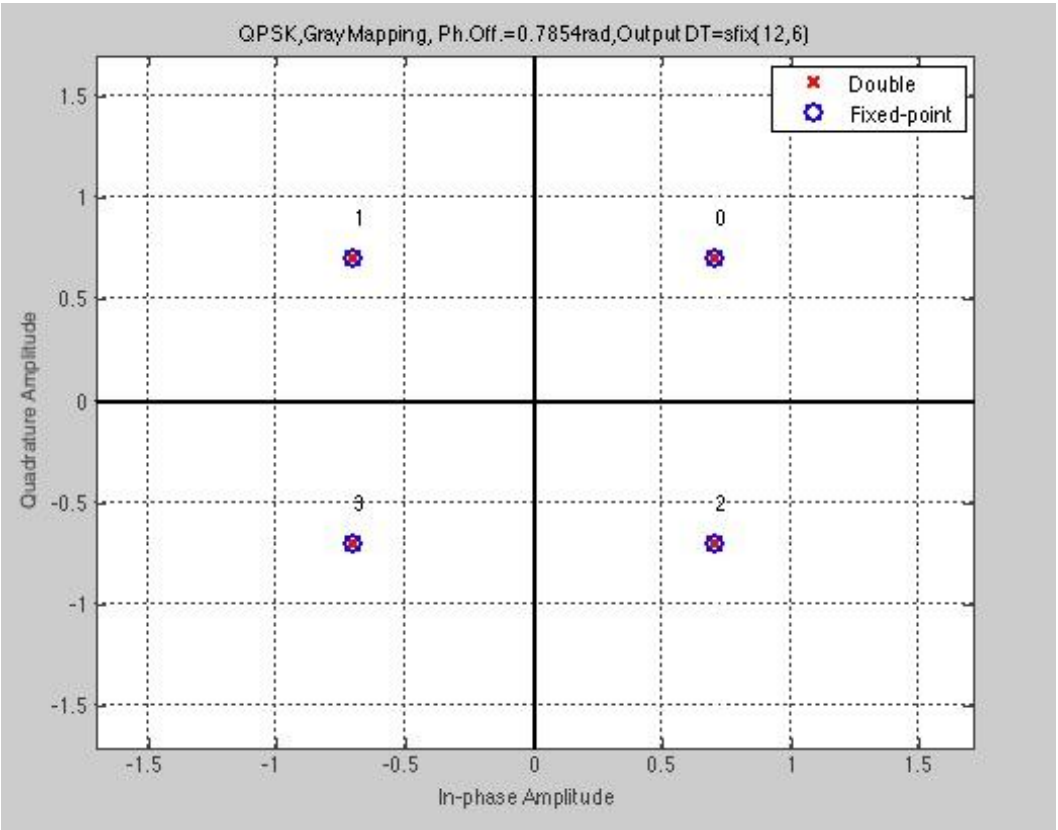


Figure 16.4: A QPSK constellation using a Gray mapping for translating into the I and Q values.

- The major goal of this project is to get fully functional versions of all the code. You should focus more of your time on getting everything to work rather than getting some but not all of the components to work and optimizing them.
- Your report should describe your code at a high level. Since each of you will likely write different code anyone that reads your report should be able to fully understand what each line of your code does. However, this does not mean that you need to explain each line of code in excruciating detail. It is possible to be succinct and thorough at the same time.
- The software version has a nested for loop structure that does not allow Vivado HLS to provide an exact number of cycles. The `tripcount` directive can help with this. You should be able to understand the reported results. For example, while Vivado may give you a best, worst and average case numbers, your algorithm for a fixed size FFT should be a fixed number of cycles. You should report the exact number of cycles for your throughput numbers. You must describe in your report exactly how you derived your throughput results.
- You must provide a comparison between the best architecture that you derived using the "software" version and the best one using the "hardware friendly" framework.
- It is ok to rewrite the code if it helps you with optimizations. For example, you can change the function interfaces.
- Your code should be written such that the hardware implementation is optimal. For example, there are many ways to do a bit reversal that are efficient in software but not necessarily efficient in hardware.
- Comment your code.

16.4 Part III: Zedboard Demo

You must implement the entire OFDM receiver on the Zedboard in the programmable logic. You should use Xillybus to transmit data to the receiver, the receiver operates on the data, and you will use Xillybus to get the data off of the programmable logic and back onto the ARM core. You must use Gplot to show that transmitted and the received data.

You are responsible for extracting the transmitted data from the Simulink testbench. You can use "To Workspace" data from any signal in the Simulink file. This will then appear as a variable in the Matlab workspace. There are already several "To Workspace" blocks in your Simulink file that you can either use directly or modify to extract the necessary data. You only need to show one "frame", i.e., the data from the output of one 1024 FFT.

We provided the general framework for Gplot, Xillybus, etc. in previous labs. You should use that to create this demo. We will not be providing you with anything more than what was given in previous labs. You should provide a plot similar to Figure 16.5 in your report to prove your design is working. Please plot the entire 1024-point output. Please also plot a zoomed-in version of the output. We would like to see the "0 1 2 3 0 1 2 3 0 1 2 3 . . . " sequence clearly.

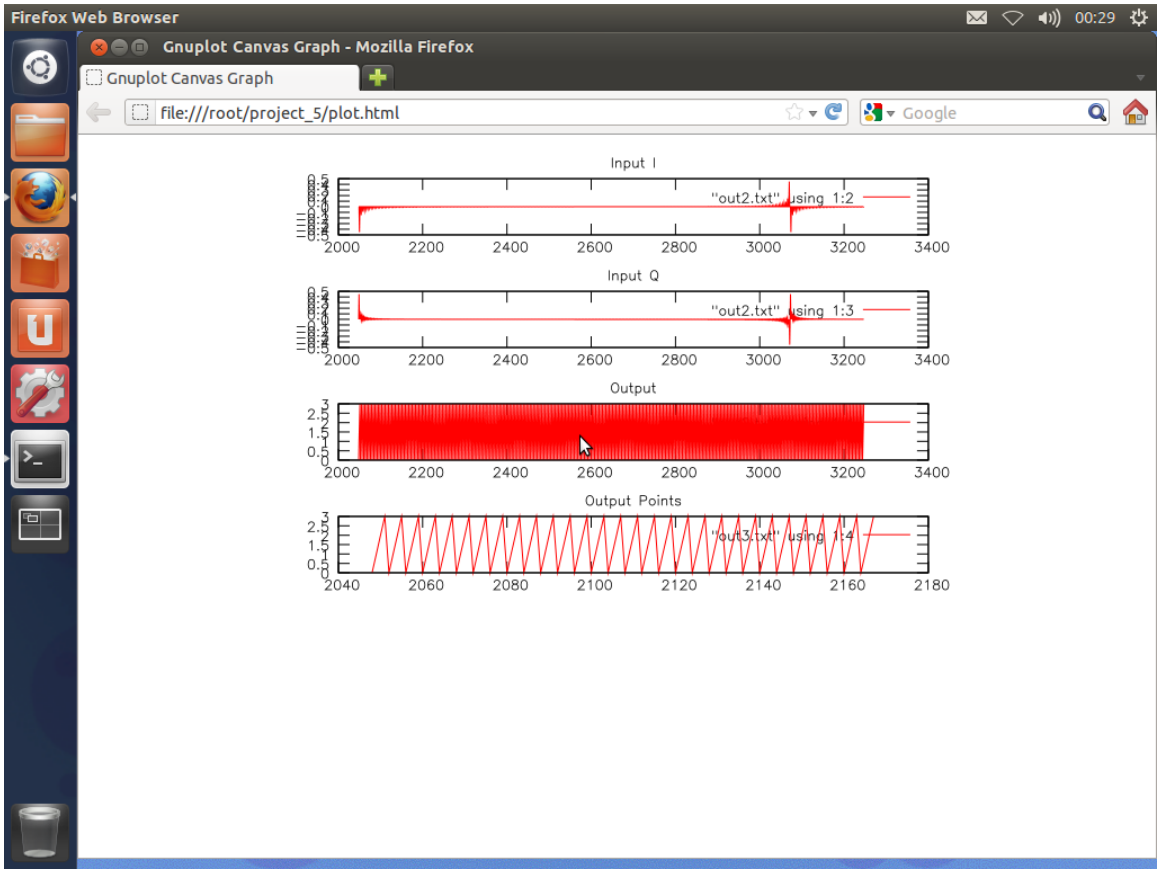


Figure 16.5: An example plot showing the input and output values to the simple OFDM receiver.

Bibliography

[1] Hassan M Ahmed, Jean-Marc Delosme, and Martin Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *Computer*, 15(1):65–82, 1982.

[2] Raymond J Andraka. Building a high performance bit-serial processor in an fpga. In *Proceedings of Design SuperCon*, volume 96, pages 1–5, 1996.

[3] Oriol Arcas-Abella et al. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *International Conference on Field Programmable Logic and Applications*, pages 1–8. IEEE, 2014.

[4] Bryce E. Bayer. Color imaging array, July 1976. US 3971065.

[5] Samuel Bayliss and George A. Constantinides. Optimizing SDRAM bandwidth for custom FPGA loop accelerators. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 195–204. ACM, February 2012.

[6] Marcus Bednara et al. Tradeoff analysis and architecture design of a hybrid hardware/-software sorter. In *International Conference on Application-Specific Systems, Architectures, and Processors*, pages 299–308. IEEE, 2000.

[7] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[8] Guy E Blelloch. Prefix sums and their applications. 1990.

[9] Stephen Brown and Jonathan Rose. Fpga and cpld architectures: A tutorial. *IEEE design & test of computers*, 13(2):42–57, 1996.

[10] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[11] Jianwen Chen, Jason Cong, Ming Yan, and Yi Zou. Fpga-accelerated 3d reconstruction using compressive sensing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 163–166. ACM, 2012.

[12] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[13] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[15] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.

[16] Alvin M Despain. Fourier transform computers using cordic iterations. *IEEE Transactions on Computers*, 100(10):993–1001, 1974.

[17] L Peter Deutsch. Deflate compressed data format specification version 1.3. 1996.

[18] Jean Duprat and Jean-Michel Muller. The cordic algorithm: new results for fast vlsi implementation. *Computers, IEEE Transactions on*, 42(2):168–178, 1993.

[19] Brian P Flannery, William H Press, Saul A Teukolsky, and William Vetterling. Numerical recipes in c. *Press Syndicate of the University of Cambridge, New York*, 1992.

[20] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[21] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578. ACM, 1966.

[22] Nivia George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[23] Scott Hauck and Andre DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*, volume 1. Morgan Kaufmann, 2010.

[24] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, October 1984.

[25] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[26] Open SystemC Initiative. SystemC. `http://www.systemc.org/`, last accessed: 2008-01-08.

[27] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[28] Glen G Langdon Jr, Joan L Mitchell, William B Pennebaker, and Jorma J Rissanen. Arithmetic coding encoder and decoder system, February 27 1990. US Patent 4,905,297.

[29] Dajung Lee, Janarbek Matai, Brad Weals, and Ryan Kastner. High throughput channel tracking for jtrs wireless channel emulation. In *24th International Conference on Field Programmable Logic and Applications*. IEEE, 2014.

[30] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference On VLSI*, 1993.

[31] Rui Marcelino et al. Sorting units for fpga-based embedded systems. In *Distributed Embedded Systems: Design, Middleware and Resources*, pages 11–22. Springer, 2008.

[32] Janarbek Matai, Joo-Young Kim, and Ryan Kastner. Energy efficient canonical huffman encoding. In *25th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2014.

[33] Janarbek Matai, Pingfan Meng, Lingjuan Wu, Brad T Weals, and Ryan Kastner. Designing a hardware in the loop wireless digital channel emulator for software defined radio. In *2012 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2012.

[34] Janarbek Matai, Dustin Richmond, Dajung Lee, and Ryan Kastner. Enabling fpgas for the masses. *arXiv preprint arXiv:1408.5870*, 2014.

[35] Carver Mead and Lynn Conway. *Introduction to VLSI systems*, volume 1080. Addison-Wesley Reading, MA, 1980.

[36] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[37] Jorge Ortiz et al. A streaming high-throughput linear sorter system with contention buffering. *International Journal of Reconfigurable Computing*, 2011.

[38] Marios C. Papaefthymiou. Understanding retiming through maximum average-weight cycles. In *SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 338–348, 1991.

[39] William B Pennebaker. *JPEG: Still image data compression standard*. Springer, 1992.

[40] Bhaskar Sherigar and K RamanujanValmiki. Huffman decoder used for decoding both advanced audio coding (aac) and mp3 audio, June 29 2004. US Patent App. 10/880,695.

[41] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

Glossary

bitstream The configuration data used to program the functionality of an FPGA. 11

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. 16

EDA Electronic design automation (EDA) are a set of software tools used to aid the hardware design process. . 11

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.. 11

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. 18

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

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. 9

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.. . 13, 272

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 FPGAs.. 11

routing channel A routing channel provides a flexible set of connections between the FPGA programmable logic elements. . 12, 13, 272

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. 9

slice A (typically small) set of LUTs, FFs and multiplexors. These are often reported in FPGA resource utilization reports. . 11, 12, 13

switchbox A switchbox connects routing channels to provide a flesible routing structure for data routed between the programmable logic and I/O block. . 12, 13

task A fundamental atomic unit of behavior or high-level synthesis computation; this corresponds to a function invocation in high-level synthesis. 16

task interval The time between when one task starts and the next starts or the difference between the start times of two consecutive tasks. 16, 18

task latency The time between when a task starts and when it finishes. 16, 18

Acronyms

ASIC application-specific integrated circuit. *Glossary:* ASIC

EDA electronic design automation. *Glossary:* EDA, 11

FF flip-flop. *Glossary:* FF, 11, 12, 16, 272

FPGA field-programmable gate array. *Glossary:* FPGA, 11, 12, 13, 271, 272

HLS high-level synthesis. *Glossary:* HLS, 9, 10, 11, 12, 13

LUT lookup table. *Glossary:* LUT, 11, 12, 16, 272

RTL register-transfer level. *Glossary:* RTL, 9, 10, 11, 16