Abstract

This report details the development of heterogeneous field programmable gate array (FPGA) based hardware to perform data compression in as few clock cycles as possible. The goal of the project was a full implementation of the core DEFLATE algorithms, however, as detailed later, during the course of the project it became clear that effort should be focused on the implementation of the Lempel–Ziv–Storer–Szymanski (LZSS) algorithm. The primary topic of this report will be the implementation of that algorithm in hardware, and it will include an overview of FPGA technology and design techniques, AXI protocol interface design and hardware accelerator performance assessment.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This report is broken into *sections*, with each section generally covering a topic and building on the previous sections. These sections are dispersed across various *chapters*, which act as grouping operators on sections. The trait the sections are grouped by is *assumed level of domain familiarity*, for example, the sections in chapter 2 try to assume little or no reader familiarity with the technology involved. The topics are

- an overview of data compression generally and DEFLATE in particular
- an overview of general FPGA technology
- a detailed look at the FPGA used for this project
- a detailed description of the accelerator core as designed and implemented
- an exploration of the interface protocol used between the on-chip cpu and the accelerator core
- an assessment of the hardware accelerator, including performance and resource utilization

The principle goal of this report is to aid the reader in developing an understanding of the core technologies and techniques involved, and to provide an in depth description of the particular hardware solution developed.

Chapter 2

Technical Background

The sections in this chapter will each provide an introduction to the general topics of the project. The aim is to make it easy for the reader to rapidly gain sufficient understanding of the relevant fundamentals to be able to meaningfully reason about the implementation details discussed in later sections. As such, this chapter contains no real discussion of the particulars of the project, and serves mainly to contextualize the later sections.

2.1 Data Compression and DEFLATE

2.1.1 Data Compression

Data compression is a fundamental technique in computer science. The goal of data compression is to reduce the number of bits required to *meaningfully represent* some piece of data. This definition is important, as different domains have different definitions for meaningful representation. As an example, a meaningfully compressed representation of an executable set of instructions (a binary) is one which, when the compression process is reversed (the data is *decompressed*), produces an executable which behaves identically to the original executable. This is very different from a meaningful representation of a novel, which requires perfect reconstruction of the input data stream, or an audio file, which can tolerate significant differences between the original data stream and decompressed stream, due to the limits of human hearing.

The above examples hit upon a key dichotomy in the field of data compression, between *lossy* and *lossless* data compression. A lossy data compression algorithm is one which does not perfectly reproduce it's input when decompressed, generally by leveraging domain-specific traits about the input data to reason about segments which are "safe" to remove, as they are not required for whatever process the data is used for. One could consider an optimizing compiler a type of lossy compressor, taking the executable example above, albeit a uni-directional one, as it is not possible to reconstruct the input stream. Alternatively, lossless data compression is data compression which perfectly encodes everything about the input

data, ensuring perfect reproducibility at the cost (generally) of some reduction in data size post compression. Lossless compression algorithms are more generally applicable than lossy ones, as they make no assumptions about the data they are compressing.

The core focus of this project is two lossless compression algorithms, Lempel-Ziv-Storer-Szymansk (LZSS)[1] and Huffman coding. The core operation of LZSS is to replace sequences of bytes in some stream of data with a reference to another location in the data stream at which the same sequence of bytes occurs[1]. It achieves this by maintaining a lookahead buffer, `B`, into the input stream of bytes, and a sliding window, `W`, of the N previous bytes seen in the input stream. Generally, `W` is significantly larger than `B`, on the order of 4096 bytes to 15 bytes. As each byte in the input stream is pushed into the encoder, it will iterate through all positions in `W` searching for the longest string of bytes which prefix `B`, referred to as the *longest prefix position*. Once the longest prefix position is found, the encoder will output either the literal byte which was just pushed, or, if the longest prefix is sufficiently long, the number of steps backwards from the current position of the encoder in the input stream to the longest prefix position, and the length of the prefix. Using the numbers above, a `position/length` pair could be encoded in 16 bits or 2 bytes, so any substitution longer than 2 bytes would be worth doing. Algorithm 2.1, modified from [2], provides a high-level pseudo-code description of the LZSS sliding-window concept.

Algorithm 2.1: LZSS Algorithm

```
1  while input is not empty do
2      prefix := longest prefix of input that begins in window
3
4      if prefix exists then
5          i := distance to start of prefix
6          l := length of prefix
7          c := char following prefix in input
8      else
9          i := 0
10         l := 0
11         c := first char of input
12     end if
13
14     output (i, l, c)
15
16     s := pop l + 1 chars from front of input
17     discard l + 1 chars from front of window
18     append s to back of window
19 repeat
```

The Huffman coding algorithm is a variable-length encoding algorithm[3]. This means it encodes *symbols*, where a symbol is some word-length, i.e. a `position/length` pair from LZSS or a byte, with a variable number of bits. The core principle of operation of Huffman coding is to encode frequently occurring symbols with as few bits as possible, at the cost of

[1] LZSS is a variation on the fundamental algorithm LZ77, adding replacement length checking and a literal/reference marker bit at the start of each byte. The two are sufficiently similar that for our purposes they will be treated as interchangeable.

encoding less frequently occurring symbols with more bits than the original encoding scheme required. The implicit assumption of this algorithm is that frequently occurring symbols occur with such frequency that the compression gains made by encoding them more concisely more than offset the losses suffered by encoding the less frequent symbols more verbosely, and this assumption has been found to hold true for most data commonly found in digital systems.

Huffman coding is specifically a method for finding the *optimal* encoding scheme which satisfies the above goals. To achieve this, the algorithm takes a 3 phase approach (discounting the actual encoding of the data). The phases of Huffman coding are

1. calculate frequency weights for all symbols in the data set
2. derive an encoding for all symbols seen, based on frequency weights
3. read the data set as a stream, emitting a code for each input symbol in the data stream

The first phase is described below in algorithm 2.2. Note that the frequency weight of each symbol can be calculated as the absolute number of times that symbol appears in the input data or as the number of times it appears in the input data divided by input data size, either method provides valid input for the next phase of the algorithm. In the pseudo code below, the absolute frequency is used for convenience.

Algorithm 2.2: Huffman Encoding Frequency Weight Calculation

```
1  while input not empty do
2      if input not in symbol_table
3          add symbol to symbol_table
4          symbol_table[symbol] := 1
5      else
6          symbol_table[symbol] := symbol_table[symbol] + 1
7      end if
8  repeat
9  output symbol_table
```

Once the symbol table mapping a symbol to a weight is constructed, a list of all the symbols in the symbol table, where the first entry is the most common list and the last entry the least common, is constructed[2]. To simplify the pseudo code of algorithm 2.3, we will treat this list as a *priority queue*, where `pop` returns the least frequent item in the queue and `push` inserts a new item into the queue in a sorted manner, that is to say, in a position such that the item above it is less frequent and the item below it is more frequent.

Algorithm 2.3: Huffman Encoding Symbol Tree Formation

```
1  datatype symbol is
2      frequency is number 0
3       left_child  is symbol reference empty
4       right_child  is symbol reference empty
5      value is text empty
6      bit is boolean 0
7  end datatype
8
9  do
```

[2]This ordered list is referred to in the original paper as the symbol *ensemble*.

```
10      if size symbol_queue is 1
11          break
12      end if
13      L := pop symbol_queue
14      R := pop symbol_queue
15      L.bit := 1
16      R.bit := 0
17      N is new symbol
18      N.frequency := L.frequency + R.frequency
19      N. left_child := L
20      N. right_child := R
21      push N to symbol_queue
22  repeat
23  R := pop symbol_queue
24  output R
```

Astute readers will note that algorithm 2.3 is creating a *binary tree*[3]. In order to produce an encoding for a given input symbol *S*, one must merely walk the tree in a depth-first manner searching for the symbol *S*, and storing all the `bit` values one has seen so far once *S* is found. To make this easier, the above algorithm 2.3 can be modified by adding `N.value := L.value + R.value` to the `do repeat` block, which would allow the code construction to always take the shortest path to the symbol *S*, by checking if the left or right branch of the current node contained `S.value` in it's `value` member. Algorithm 2.4 assumes the above modification and produces the code for a given symbol (assumed to be a member of the set of symbols in the input data used in algorithm 2.2).

Algorithm 2.4: Symbol Derivation Given Huffman Tree

```
1
2   S is symbol input
3   N is symbol tree root
4   C is text empty
5
6   while S is not N do
7       L := N. left_child
8       R := N. right_child
9       C := C + N.bit
10      if S.value in L.value
11          N := L
12      else
13          N := R
14  repeat
15  output C
```

The final phase of the Huffman encoding algorithm is the encoding of the data stream itself, using the symbol tree built above[4]. The encoder reads the input data stream a symbol at a time, emitting a code for each symbol read. The emitted code stream is the compressed data representation. Reversing Huffman code based compression is as simple as reading the

[3]The original paper extends this method to allow for N-child trees for any value of N, but as we are attempting to produces a minimum length binary encoding, this is merely an interesting aside, not particularly useful.

[4]Or more commonly a lookup table built by flattening the symbol tree to increase encoding performance, as generally the symbol set size *S* if significantly smaller than the input data length *L*.

compressed data stream a bit at a time, storing the bits read in the order they're read until they match one of the codes in the symbol-code lookup table. This is guaranteed to validly decompress the data as no shorter symbol code prefixes a longer one[3].

2.1.2 DEFLATE

DEFLATE is a file format and compression method standard[4], used in ubiquitous software such as GZIP[5]. DEFLATE operates by applying the two compression methods described above in order, first compressing an input data stream with LZSS and then encoding the resulting compressed stream using Huffman coding. The DEFLATE standard also includes a set of file format standards and options, as well as a series of formatting requirements on the output data stream such as a standard layout for storing the Huffman symbol table described above. The standards, while important for interoperability between different implementations of DEFLATE and GZIP, were ignored for this project, as their implementation is not relevant to any performance criteria measured and they increase the implementation complexity.

2.2 Field Programmable Gate Arrays

Field programmable gate arrays are a digital logic device whose internal configuration can be changed through some method. The "field" term refers to the fact it is possible to reconfigure these devices after production. The majority of these devices do not store their internal state between power cycles, and as such, must be re-programmed on each power up. During development this is generally done through a JTAG cable connected to a development board, while during "deployment" this is often achieved through on-chip self-configuration logic which can read the chip configuration bit stream from some non-volatile storage medium, such as EEPROM.

The structure of an FPGA is that of a repeating grid of configurable logic blocks (CLB's), whose inputs and outputs are connected via configurable interconnect blocks. Each CLB will house some combination of *combinatorial* and *sequential* logic elements. The primary basic elements required to understand FPGA design are *lookup tables* (LUT's) and *flop-flops* ("flops").

2.2.1 Look Up Tables

A lookup up table can be conceptualised as a segment of read-only asynchronous-read memory (ROM), with an address width of 6 and a single boolean stored in each address location. This means a single LUT can implement any boolean logic function with up to 6 input coefficients[5]. See table 2.1 for an example of a binary function implemented using a lookup

[5]More advanced techniques, such as packing multiple fewer-input operations into a single 6-LUT in order to reduce resource requirements, are also common

table. This should be relatively intuitive for any readers familiar with the use of truth tables

Table 2.1: LUT Based 3 Input AND Operation

A	B	C	Equivalent ROM Address	O
0	0	0	0x0	0
0	0	1	0x1	0
0	1	0	0x2	0
0	1	1	0x3	0
1	0	0	0x4	0
1	0	1	0x5	0
1	1	0	0x6	0
1	1	1	0x7	1

to describe a logic function. When configuring a 6-LUT to behave as a logic function with fewer than 6 inputs, the high address bits are simply treated as ”don’t care” signals in the LUT truth table, that is to say, the pattern of the active input bit truth table output column is just repeated. For an example, see table 2.2, which represents the implementation of a 3 operand logic function in a 4-LUT. As can be seen, the ”O” column is simply repeated to allow the ”Unused” input to be treated as a ”don’t care”.

Table 2.2: 4 Input LUT Based 3 Input AND Operation

Unused	A	B	C	Equivalent ROM Address	O
0	0	0	0	0x0	0
0	0	0	1	0x1	0
0	0	1	0	0x2	0
0	0	1	1	0x3	0
0	1	0	0	0x4	0
0	1	0	1	0x5	0
0	1	1	0	0x6	0
0	1	1	1	0x7	1
1	0	0	0	0x8	0
1	0	0	1	0x9	0
1	0	1	0	0xA	0
1	0	1	1	0xB	0
1	1	0	0	0xC	0
1	1	0	1	0xD	0
1	1	1	0	0xE	0
1	1	1	1	0xF	1

The metaphor of LUT-as-memory is an apt one, as a key feature of the FPGA used in this project is the ability of the LUT’s in the device used to be configured to act not as ROM but as true read-write random access memory (RAM), with a clock-synchronous write channel (a write address and data port, with a write occurring on the edge of a clock signal when a write enable signal is asserted) and an asynchronous read channel (such that the data output port changes on any change to the read address port, without waiting for a clock edge). This

allows for the design of very flexible memory architecture, which is leveraged in this project and discussed elsewhere in this report.

The key takeaway is that a LUT is a memory component whose contents can either be set at device configuration time or be cleared and then set during device operation, as per the design the device is being configured to run[6]. An important note about what is referred to as ”LUTRAM” (or in Xilinx parlance, ”DRAM” for *distributed* RAM, not to be confused with the *dynamic* RAM used in modern computers) is that it is often a quite limited Resource relative to other on-chip memory resources.

2.2.2 Flip-Flops

Flip-flops are the principle basic logic element used to store a value and update it each clock cycle. Flops can be configured to behave as edge-triggered flip-flops or level-triggered latches, however, for the purposes of this report, they will be treated exclusively as rising-edge triggered flip-flops with a synchronous reset line. To make this clearer, see figure 2.1. Note that the input signal *D* is only propagated to the output port *Q* on a rising edge of *clk*, and the flop is only cleared by reset signal *R* on a rising edge as well. The FPGA used in this project can support many different configurations of the register elements it uses, however, this is the configuration most commonly used for this project, and as such, understanding it is sufficient.

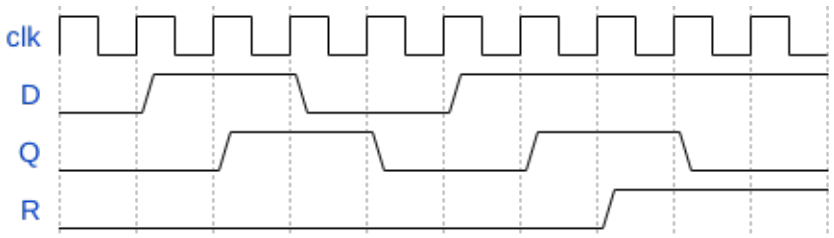

Figure 2.1: Characteristic Timing Diagram For Flops Used in this Report

Generally, the structure of a given configurable logic block is some number of LUT’s (of input width range 4 to 8), whose output passes through one or more parallel registers. Figure 2.2 illustrates the layout of the configurable logic blocks in the device used in this project — on the left can be seen a row of 4 LUTs with 6 inputs each, and on the right and be seen a pair of 2 registers for each LUT output. The LUT/Flop groups can be configured such that an input bypass the LUT to be fed directly into the flip-flops on the right hand side, and they may also be configured such that the output from the LUT bypasses the flip-flop entirely.

[6]”Run” here is a misnomer, it would be more accurate to say ”the design the FPGA is being configured to *be*”, however, that is not very intuitive. For the purposes of this report, a hardware design being said to be ”run” should be taken to mean an FPGA being configured to act according to the design’s logical function

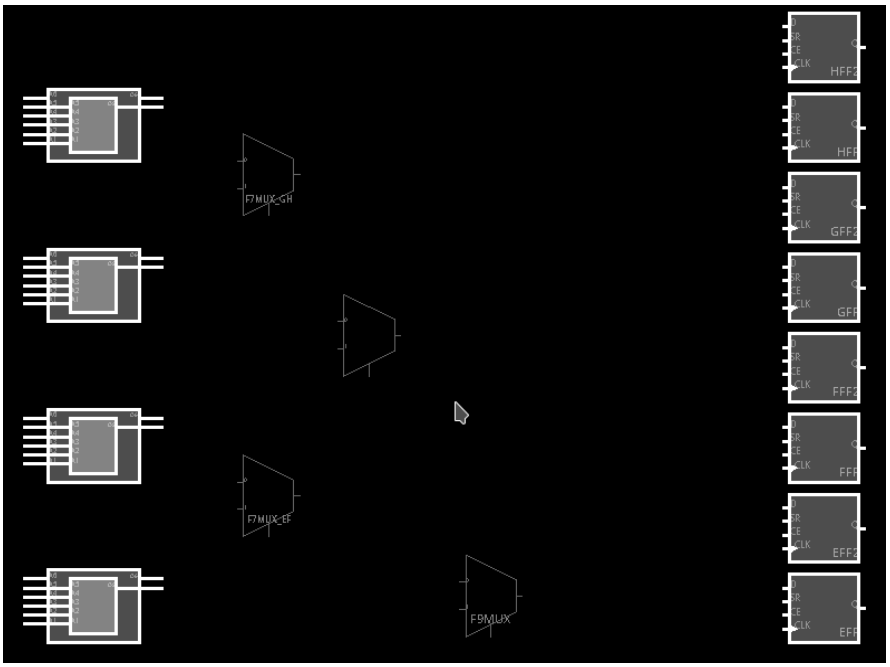

Figure 2.2: Configurable Logic Block Layout for Project Device

2.2.3 FPGA Interconnects

As described above, a configurable logic block may be considered a ”wrapper” around a set of LUT/Flop groups. The input and output ports of these CLBs are then tied in to a global mesh of wires, the connections between which are also configured on power up of the device. This global network of wires is called the *interconnect fabric*, and it's the second key functionality which makes FPGA's so flexible (the first being the reconfigurable logic itself). The ability to tie the output of any given LUT/Flop group to the input of another LUT/Flop group is what enables the FPGA to fulfil the role of almost any digital hardware, constrained only by maximum signal propagation delay requirements[7].

2.2.4 Hardware Design for FPGA's

In the previous section, FPGA's were described as digital devices whose function was configurable, via the loading of a *bitstream* onto the device. The process of generating these bitstreams follows this rough outline:

1. Write a file or set of files which describe the digital logic which the FPGA is to be configured as.
2. Pass this set of files into an *elaboration* engine, which converts the hardware as written into a graph-based representation. At this point, the graph-based representation of the logic designed can be inspected for obvious design errors[8].

[7] All the extra transistors required to do the dynamic reconfigurable signal routing add significant propagation delays to any CLB-to-CLB signals, and the flexibility of LUTs means they require significantly more individual logic gates to implement than any function they are emulating.

[8] This graph is sometimes referred to as the ”RTL” of a design, as the underlying representation is in what

3. Simulate how this digital logic graph behaves with a given set of stimuli (called a *test-bench*), verifying that it behaves in the desired manner based on design specifications[9].
4. Convert the generic logic graph into one which is composed of elements which represent the available basic logical elements on the device being targeted for this design[10]. This includes steps like converting abstract gates like `OR` into truth-table output columns for LUT configuration.
5. Choose the precise basic logic elements on the physical device which will contain the logic instantiated as part of the design (this is called *placing* the design).
6. Choose the exact path the signals in the design will take through the interconnect fabric on the device (this is called *routing* the design, which is often conflated with the previous step into a single stage known as *implementation*).
7. Parse the synthesised logic graph, constrained by the placement and routing stages, and convert it into a stream of bits which, when passed to the FPGA's configuration logic, configure the FPGA to implement the design (referred to as *bistream generation*).

The steps above which it is necessary to understand for the purposes of this report (and to begin working with FPGA's) have been described in earlier sections, bar 1; the process of describing the hardware one wishes to implement in a machine-readable format. This is done through the use of a *hardware description language* (HDL). Two main hardware description languages exist, VHDL and Verilog. A third, SystemVerilog, is an offshoot of Verilog which is mostly inter-operable, with some additional features which improve engineer productivity by simplifying common tasks. SystemVerilog is the language used to design the hardware developed for this project, and as such, the rest of this section will be given over to a brief introduction to the language, in order to make the snippets of SystemVerilog code included in later segments more readable.

The programming model of SystemVerilog is that of a collection of stateful and stateless wires, each wire being driven by and driving another, potentially through some logical operation. The only wires not driven by or driving another wire are those connected to device input and output ports[11]. Stateless wires immediately propagate any change on their driver to anything they are driving, stateful wires only propagate a change to the wires they are driving as a result of some change in another signal, usually the rising or falling edge of a clock. Stateless wires are used to model purely combinatorial logic, while stateful wires are used to model stateful logic, such as the registers described previously. Stateless wires are of type (intuitively) `wire`, stateful wires of type `reg`. See listing 2.5 for an example of a stateless piece of logic (in this case an OR operation, with operands A, B, and output Y). Readers

is called a "register transfer level" model, which describes digital logic as a set of registers and connections between registers. This report will conceptualise this as a signal graph for the sake of simple intuition.

[9] Various verification techniques exist. The exact verification steps undertook for this project are discussed elsewhere in this report.

[10] Please see the glossary for project device details.

[11] Or in some cases non-programmable on-device logic, such as the on-chip CPU embedded alongside the programmable logic in the Zynq device used in this project.

should note the use of `assign` to indicate the wire Y will always be driving the result of the operation `A | B`, and will immediately propagate any changes resulting from a change in A or B to any wire Y is driving[12].

Listing 2.5: SystemVerilog Stateless Logic Example

```
input wire A, B;
output wire Y;
assign Y = A | B;
```

The `always` block is necessary to write SystemVerilog code involving stateful logic. The `always` block defines a segment of logic whose assignments are evaluated every time a certain condition is met. This condition is called a *sensitivity list*, and the assignments within an always block occur whenever this sensitivity list is triggered, that is to say, the `reg` wires assigned to in an always block propagate their drivers to the wires they drive when the sensitivity list is triggered. See listing 2.6 for a demonstration of a sequential logic circuit being described in SystemVerilog, in this case implementing the same logic as the previous listing, but with an output whose updates are gated by a clock signal.

Listing 2.6: SystemVerilog Sequential Logic

```
input wire clock, A, B;
output reg Y;

always@(posedge clock) begin
//Y will only propagate changes in it's value once per clock cycle
    Y <= A | B;
end
```

Readers should note line 6 is a *non-blocking* assignment operator. Within `always` blocks, all non-blocking assignments are modeled to occur at the same time, whereas *blocking* assignments, described with a `=` rather than a `<=`, are modeled to occur sequentially, in the order written. See listing 2.7 for an example of the difference in behaviour between these assignment types.

Listing 2.7: Blocking vs. Non-Blocking Assignments

```
input clock, A, B;
output reg X, Y, Z;

always@(posedge clock) begin
    Y = X & B;
    X = A | B;
```

[12]This is an abstraction, in actual hardware propagation has some delay, which is the primary factor limiting the maximum frequency an FPGA based digital logic design can operate at.

```
    Z <= X & B;                                                    7
end                                                                8
```

Table 2.3 displays the input values, the values being driven by the output registers before the rising clock edge and the values being driven by the output registers after the rising clock edge. Note that the value of Y is derived from the value of X *before* it's value is derived from the value of the A and B input signals at the instant of the rising clock edge, while the value of Z is derived from the value of X *after* it's value is update, reflecting the *new* state of X. This is due to the assignments to X and Z being modeled as having occurred at the same instant, while the assignment to Y is modeled as having occurred before the assignment to X. Verilog enforces that all nonblocking assignments occur after all blocking assignments in a given block.

Table 2.3: SV Assignment Type Behavioural Differences

Wire	Initial State	Final State	Note
A	0	0	Input signal, value does not change
B	1	1	Input signal, value does not change
X	0	1	
Y	0	0	Y is driven to 1 as X is 1 at time of assignment
Z	0	1	Z is driven to 0 as X is 0 at time of assignment

SystemVerilog would not be very useful if every wire had to be declared individually, as working with logic to implement common requirements such as 32 bit integers would quickly become unmanageable. As such, SystemVerilog has the concept of *arrays*. Listing 2.8 has an example of a couple of different useful array declarations and uses, and their internal layouts. Note that both registers and wires can be declared as arrays, and declaring multi-dimensional register arrays is the most common memory generation technique[13].

Listing 2.8: SystemVerilog Array Types

```
1
//Some little endian bytes:                                          2
//A byte width wire, whose memory layout is:                         3
//[X, X, X, X, X, X, X, X]                                           4
wire[7:0] byte;                                                      5
//A pair of bytes, whose memory layout is:                           6
//[[X, X, X, X, X, X, X, X], //[X, X, X, X, X, X, X, X]]            7
wire[7:0] two_bytes[1:0];                                            8
//Note the bit addresses within the byte are as follows:             9
//[7, 6, 5, 4, 3, 2, 1, 0]                                           10
//The declaration can be read as "wire of width seven down to 0"     11
                                                                     12
//Similarly, arrays can be declared as big endian:                   13
wire[0:7] big_endian_byte;                                           14
//The above's address layout is:                                     15
//[0, 1, 2, 3, 4, 5, 6, 7]                                           16
//This is uncommon, and not used in this project                     17
                                                                     18
//Declaration of 4 kilobits of and 4 kilobytes of memory             19
reg four_kilobits[4095:0];                                           20
reg[7:0] four_kilobytes[4095:0];                                     21
```

Finally, SystemVerilog provides a method for "wrapping" a segment of logic into a *module*, which provides hierarchical abstraction and allows for multiple copies of a module to be instantiated as part of another module. SystemVerilog designs declare a *top* module, whose

[13] Using vendor-specific macro instantiation being the alternative.

input and output ports are then mapped to input and output pins on the device during implementation, and whom all other modules in the design are a *submodule* of. Listing 2.9 provides a simple example of a top module and a submodule, with the top module instantiating multiple copies of the submodule (readers should attempt to discern the logical function being implemented here in order to test their understanding so far, and then see footnote[14]).

Listing 2.9: SystemVerilog Modules

```
module sub_mod(
    input wire A, B,
    output wire Y
);
    assign Y = A & B;
endmodule

module top_mod(
    input wire clk, A, B, C, D,
    output reg Y
);

    wire internal_1, internal_2, internal_3;
//Module instantiation:
    sub_mod s1(A, B, internal_1);
    sub_mod s2(C, D, internal_2);
    sub_mod s3(internal_1, internal_2, internal_3);

    always@(posedge clk) begin
        Y <= internal_3;
    end
endmodule
```

2.3 The Zynq Heterogeneous FPGA

So far, the FPGA devices discussed have been *homogeneous*, that is to say, they have consisted only of the programmable logic region and some configuration logic to load the bitstream into the programmable logic region. Modern FPGAs are *significantly* more complex, containing many "hard blocks[15]", which are non-configurable regions which perform some function

[14] The logic designed is a 4 port AND gate, with a clock synchronous output.

[15] Called as such because they cannot be reconfigured. FPGA terminology commonly divides between "hard" and "soft" logic regions, with soft logic being configurable by the design engineer, and hard logic having a fixed function.

which is sufficiently common that it is worthwhile to design dedicated silicon for it, such as large regions of on-chip RAM or digital signal processors (hard blocks capable of performing mathematical operations with a far lower propagation delay than instantiated soft logic). These modern FPGAs are still considered purely programmable logic devices, however, due to the flexibility of the connections to and from any hard blocks they contain, granted by the hard blocks also being tied in to the interconnect fabric of the FPGA.

A common design pattern when using FPGAs was to create (or use an already existing) ”soft core”, which was a microprocessor instantiated in programmable logic, to handle the design tasks more suited to a software environment, such as user interfacing. The soft core would be used alongside a high-performance “accelerator” core, which performed some task which would rely on the advantages FPGAs offer over traditional CPUs; extreme parallelism and pipelining of multi-stage algorithms, as well as designing of hardware to achieve in a single clock cycle what would take a cpu many dozens of cycles. To address this common pattern, particularly in the context of rapidly increasing design complexity, driven by an increased focus on multi-role devices[16], the major FPGA vendors released what are termed *heterogeneous* FPGA devices, which include a full-fledged processor as a hard block within the device alongside the traditional programmable logic. The primary benefit of including the CPU, which is generally a single or multi-core ARM processor, in the same package as the FPGA, instead of designers including it as a second device within a full system, is that the physical proximity between the CPU and the programmable logic allows for extremely high-bandwidth interfacing between what is termed the *processor subsystem* (PS) and the programmable logic (PL).

The device used for this project is a Zynq UltraScale+ heterogeneous FPGA[17], developed and sold by Xilinx, Inc. The programmable logic region of the device includes 70,560 of the 6-input LUTs previously described, as well as 141,120 registers[6]. 28,800 of the 6-LUTs are able to be configured to act as random-access memory. Alongside the programmable logic region, the device includes an ARM processor subsystem, which includes 4 ARM Cortex-A53 processor cores, a pair of Cortex-R5 real-time processor cores, and a ”Platform Management Unit” to coordinate the 6 processors listed previously. It also incorporates a GPU to handle video processing. It is an *extremely* capable device.

[16]For example, an FPGA in a hardware design might be acting as an analog/digital bidirectional interface, a signal processor, a display driver and a CPU to run a lightweight operating system, all at once.

[17]The exact device code is “xczu3eg-sbva484-1-e”, and it is embedded in an Ultra96v2 development board

Chapter 3

Hardware Acceleration and Data Compression

The general goal of the sections in this chapter is to take a closer look at the algorithms, tools and concepts discussed in previous sections, and to think about how they relate to each other. If the previous sections focused on building *concept familiarity*, the following chapters will focus on pursuing *concept synthesis*. As such, the first section of this chapter will explore the opportunities the devices described in section 2.3 present. Its focus will be on the benefits and costs of the different hardware architectures, and therefore programming models, that both processors and programmable logic present. The section will finish with a brief introduction to the protocol used to communicated between PS and PL, and a short exploration of how these distinct models may complement each other.

The sections following the first of this chapter will apply the lessons learned in the first to the particular problem this project is focused on, and will start with a brief complexity and, as is often more relevant in multi-domain design, *structural* analysis of the DEFLATE algorithms. Following that, the implementation details of the hardware designed to address the principle goal of the project, high performance data compression using the algorithms leveraged by DEFLATE, will be outlined. Sections that were found to be particularly interesting or challenging will be highlighted and explored, in the hope that the reader may take from this chapter an insight into the hardware developed and the reasoning behind certain design decisions.

3.1 Hardware Acceleration and PS/PL Interfacing

3.1.1 Hardware Acceleration

The term hardware acceleration, in this report, refers to the use of digital hardware implemented in programmable logic to perform some algorithm or piece of an algorithm, which otherwise would have been carried out by software running on a processor, in order to improve

total system performance. The hardware, referred to as an *accelerator core*, is interfaced to the processor subsystem using some interface standard. In the case of this project, and Xilinx hardware in general, the interface standard is the Advanced Extensible Interface (AXI), which is part of the ARM Advanced Microcontroller Bus Architecture (AMBA)[7].

The goal of hardware acceleration is to leverage the strengths of the hardware domain to complement the weaknesses of traditional processor focused systems, similar to the goal of the Zynq project, which is to leverage the strengths of software design to complement the weaknesses of traditional hardware design workflows. In order for us to gain a strong understanding of hardware acceleration, it is necessary to understand the strengths and weaknesses of both hardware and software based problem solving.

3.1.2 Processor Architecture, Benefits and Costs

Modern processors are dizzyingly complex, a brief look at the current x86 instruction set makes that clear. Fortunately, some fundamental architectural principles remain as "fixed points", from which we may conduct a *very* generalised overview of the strengths and weaknesses of processors to solve a given design problem. Ignoring things like on-chip cache, instruction pipelines and speculative execution processors generally consist of

- an arithmetic and logic unit (ALU), with two inputs and an output, which performs some mathematical or logical operation on its inputs to produce an output
- a set of "register files", capable of storing a value of the same width as the processor (i.e. 64 bits in a 64 bit processor architecture), which can behave as inputs and outputs for the ALU
- A program counter, which points to the position in the instruction memory (which is not necessarily different from data memory) where the next instruction for the processor to execute is located
- some control logic, which performs operations like reading from and writing to memory, fetching the instruction at the program counter, and reading from and writing to the program counter

The general operation of a processor can be seen as a cycle of *fetch*$\rightarrow$ *decode*$\rightarrow$ *execute* $\rightarrow$ *write* $\rightarrow$ *repeat*. The processor fetches the instruction at the program counter, decodes it into an operation and the operands, performs the operation described, writes the operation to either a register or memory location (depending on the operation), and then starts again. The program counter generally simply increments each cycle, however, its value can be directly controlled by the processor using special instructions called "branches" or "jumps". These instructions are how structures like `if`, `while` and function calls are implemented in software. The key piece of information here is that a processor may only execute instructions *one at a time* and *one after the other*, that the processor is a *serial*, *sequential* device.

There are two main benefits modern processors bring, when compared to FPGA based

design solutions: they are *blazingly* fast, and they are easy to work with. The Cortex-A53 processor cores in the PS of the device used in this project run at 1,200 MegaHertz, while it is fairly challenging to design FPGA hardware that can handle a clock speed above 200 MHz, and approaching 300 MHz with a design of any reasonable complexity can become Sisyphean[1]. Humans are well equipped to understand sequential logic (so well equipped in fact that we assume sequential logic even where none exists[8]), and as such the sequential execution model of software development is fairly intuitive to us. Layer on top of this decades of free and open-source tools and libraries being developed and made publicly available, and the approachability of software based system design is obvious.

Something of a footnote which is critical in some applications is that most processors are not timing reliable, that is to say, the number of clock cycles required to execute a piece of software cannot be predicted perfectly. This is due to features such as transparent on-chip data caching, speculative execution (a processor will begin executing both branches of an `if` statement before the result of the predicate is known) and instruction pipelining making code path execution time unpredictable to the precise clock cycle. This is occasionally very important, such as in high-speed telecommunications or vehicular[2] guidance systems.

3.1.3 Programmable Logic, Benefits and Costs

The structure of programmable logic has been outlined in previous sections, so here we will focus only on how that structure can be used to complement an processor running some piece of performance critical software [3]. Since FPGAs provide near-infinite flexibility at the cost of maximum frequency, the question of how to complement a software system using hardware can best be answered by considering what a processor *cannot do*, rather than what an FPGA *can do*. A processor cannot *generally*

- execute multiple instructions at the same time
- execute the same instruction with multiple input and output groups
- execute instructions involving more than 2 operands
- execute instructions using values in memory, even on-chip cache, as the operands (save for the relevant load/store instructions to move values into and out of registers)

Exceptions exist to all of these, such as x86 vector instructions and using multi-core CPUs to do algorithm stage pipelining, but those techniques are device specific or impose significant overhead, and as such, are ignored here.

[1] Readers should note the lower bound on modern *laptop* central processing units (CPUs) is roughly 2200 MHz, although these CPUs scale their clock speed dynamically to save power. Most can also "boost" up to 3500 MHz, which is, in the author's opinion, *obscenely* fast

[2] "Vehicular" here meant in the "kill vehicle" sense, not the "road vehicle" sense, although the latter is becoming more common.

[3] Here, performance critical is taken to mean the software must complete the task it is designed to complete as rapidly as possible. This is distinct from the more stringent definition of performance criticality, which is software which must complete its task within a certain amount of time *or not complete it at all*.

It is necessary to take brief aside here to discuss algorithmic structure, and the representation of such. So far, pseudo code has been used to describe algorithms, however, another useful way to describe and reason about algorithms is as a graph of operations, each operations input being tied to another operations output, bar the first and last operation(s) in the graph. Readers should find this reminiscent of the Verilog programming model described in section 2.2.4. Figure 3.1 shows a simple data graph, with 3 inputs and 1 output, which implements the mathematical operation $Y = (A+B)*C$. Note the diamond-shaped objects are operations, and when execution order is being discussed, the green shaded operands are the ones being executed during a given cycle (clock cycle in the case of PL, fetch-decode-execute-store cycle in the case of PS).

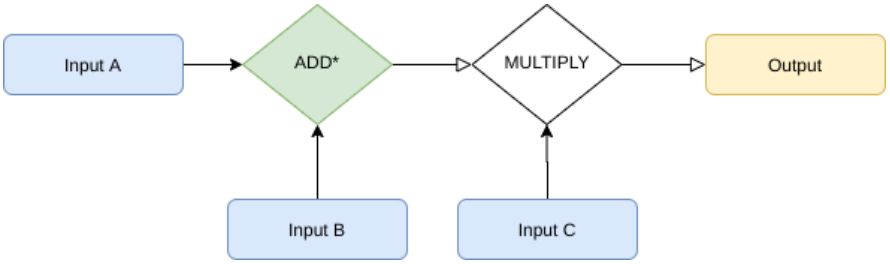

Figure 3.1: A Simple Data Graph

With this understanding of data-graph algorithm representation in hand, we can explore two ways in which FPGAs can be used to complement processors: pipelining and parallelism. Parallelism allows the evaluation of multiple input/output groups and operation sets at once, and is useful when an algorithm which is run inside some form of loop does not rely on the result of the previous loop to execute the next. The term used to describe this algorithm property is its data dependency. Algorithms 3.1 and 3.2 are examples of algorithms which are not and are data dependent, respectively.

Algorithm 3.1: "An Example of Data Independency"

```
1  i is number 0
2  while i is less than size input_array do
3      input_array[i] := input_array[i] + 1
4      i := i + 1
5  repeat
```

Algorithm 3.2: "An Example of Data Dependency"

```
1  i is number 1
2  while i is less than size input_array do
3      input_array[i] := input_array[i-1] + 1
4      i := i + 1
5  repeat
```

The two algorithms presented above may be viewed as having the data graphs (for the first and second loop iteration) seen in figures 3.2 and 3.3. From these diagrams, it is clear that the result of the second addition for algorithm 3.1 is completely independent from the result of

the first addition, whereas the result of the second addition for algorithm 3.2 *is* dependent on the result of the first. As such, if we were to execute both additions at the same time for both algorithms, the result stored to the input array would be valid for algorithm 3.1 but invalid for algorithm 3.2.

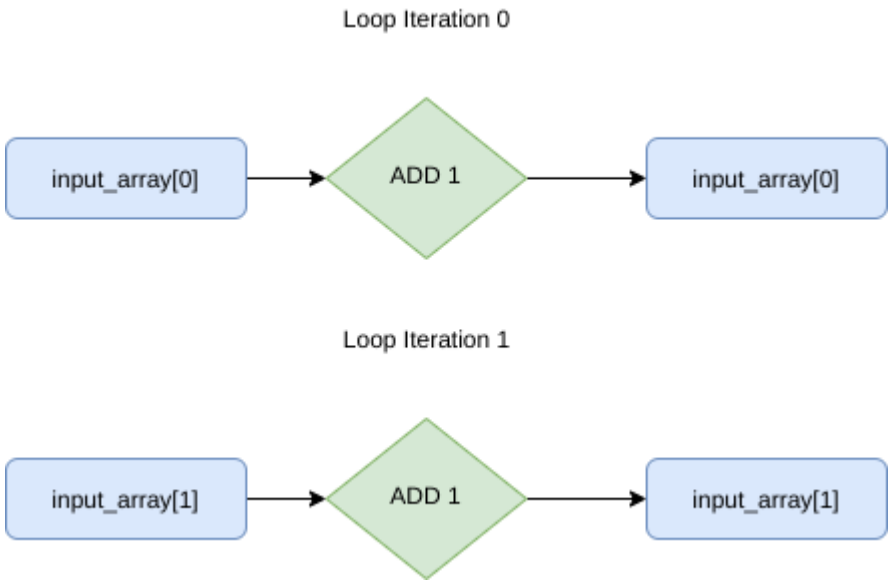

Figure 3.2: Data Independency

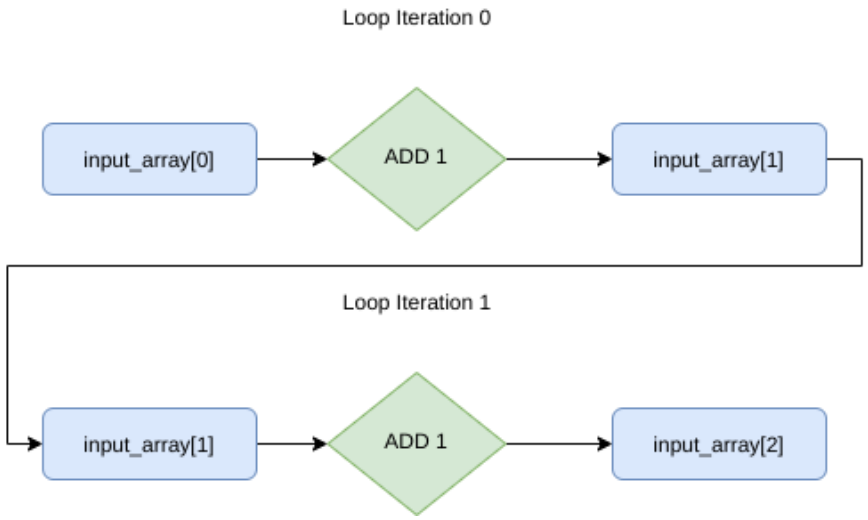

Figure 3.3: Data Dependency

FPGAs allow for this form of ”loop unrolling”, which is a huge advantage over processors in the case of algorithms which posses this data independency property. However, even for algorithms that do have some data dependency between loop iterations, FPGAs can still engage in pipelining, which is the execution of every stage within the algorithm loop at the same time. For example, given the algorithm 3.3

Algorithm 3.3: Simple Pipelineable Example

```
1
2  i is number 1
3  while i is < size input_array do
4        input_array [i] := ( input_array [i] + 1) * 2
5        i := i + 1;
6  repeat
```

Figure 3.4 present the above algorithm with each operation within the loop (addition, multiplication) being executed at the same time. This example brings to us the concepts of *throughput* and *latency*. Throughput is the number of results per cycle, and latency is the number of cycles between pushing an input into a system and seeing a result from the system. The throughput of the fully pipelined version of algorithm 3.3 is 1, as a result is received on the output once per cycle, bar the first cycle, while the latency is 2, as the system must carry out 2 operations on an input before presenting the respective output. An equivalent implementation of algorithm 3.3 in software would have a latency of 2 and a throughput of 0.5. Further, algorithm 3.3 could be executed in parallel as well as pipelined, for example, 4 copies of the data graph could be instantiated in hardware (called a loop unrolling factor of 4), and then the throughput would be increased to 4, 8 times higher than the software based solution.

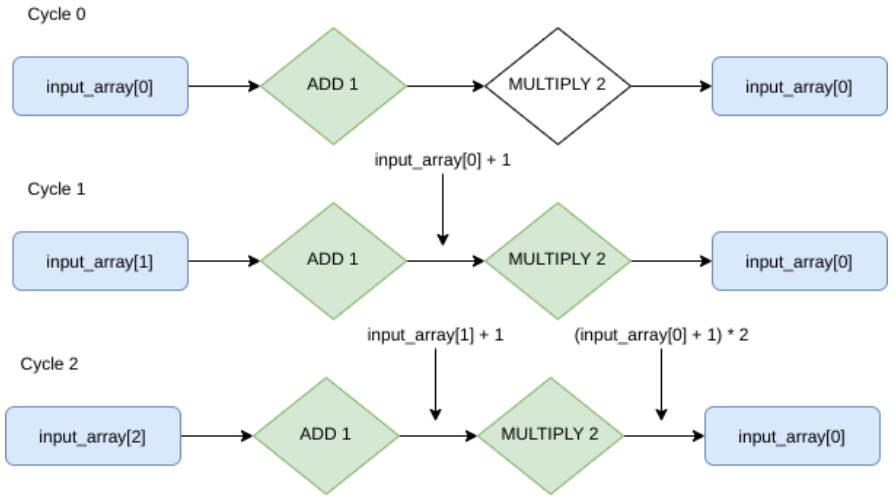

Figure 3.4: Simple Pipelining Example

3.1.4 AXI-Lite, a Brief Overview

AXI is the bus protocol used to interface between PS and PL. AXI takes a memory-mapping model for software/hardware communication, with peripherals instantiated in the PL being assigned an address region in the memory map of the PS processor cores. This means writing to a particular memory address in software transmits the data being written to the hardware peripheral, and reading from that address reads data back from the hardware peripheral. AXI uses the concept of channels, with 5 channels being defined, which are the

- write address channel

- write data channel
- write response channel
- read address channel
- read response channel

Table 3.1 presents the signals important to this project, and which end of the bus they are driven by. This project uses a reduced version of AXI, AXI-Lite, which greatly simplifies the implementation of the bus in hardware, at the cost of greatly reduced throughput due to lost support for multiple consecutive reads or writes after only a single transaction.

Table 3.1: AXI Signal Reference

Name	Driver
Write Address Valid	M
Write Address Ready	S
Write Address	M
Write Valid	M
Write Ready	S
Write Data	M
Write Response Valid	S
Write Response Ready	M
Write Response	S
Read Address Valid	M
Read Address Ready	S
Read Address	M
Read Valid	S
Read Ready	M
Read Data	S
Read Response	S

AXI deals in what are termed as *transactions*, which are always initiated by the master and, under the AXI-Lite specification, always involve a single transfer of data. A transaction is initiated by the bus master asserting a valid signal on the addressing channel, either read or write. Following this, the relevant address signal for that channel (Write Address or Read Address) is held constant until both the valid and ready signals for that address channel are asserted, at which point it is assumed the slave has begun processing the request of the master. Following this, the slave will assert a valid signal on the data channel (write valid or read valid), signalling that the write data or read data line is valid. This will then hold until both the data channel valid and ready signals are asserted, at which point it is assumed the master has read the request response, and these signal are no longer required to be constant. In the case of writes, the slave should assert write response valid, which implies the write response signal is valid, after both the write valid and write ready signals are asserted at the same time. The write and read channel group response signals are used to indicate a failed read or write, as defined by the AXI specification[7]. Figure 3.5 demonstrates a typical AXI write transaction,

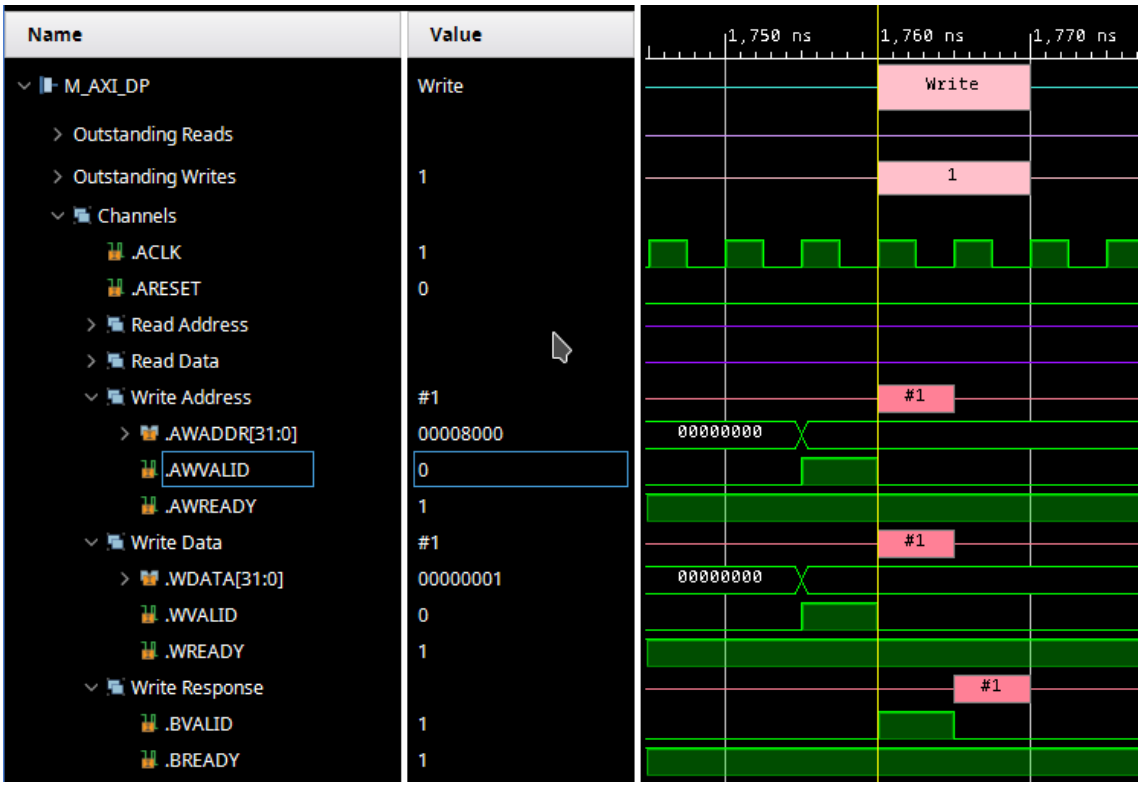

Figure 3.5: A Typical AXI-Lite Write Transaction

between a Xilinx Microblaze soft[4] processor core and the accelerator core developed for this project.

[4] "Soft" here to mean instantiated in programmable logic

It is *highly* recommended that readers interested in developing a further understanding of the AXI-Lite protocol read [9], which is a truly fantastic in depth introduction to the protocol used here, and was an invaluable resource to the author.

3.1.5 PS and PL Suitable Tasks

The strengths of processors and programmable logic may be summarized as follows; processors are good for rapid / low cost development and implementation of standard protocols due to the ease of use and plethora of open source implementation of almost every standard on the planet. Processors are also useful for any application which requires complex memory architecture (such as hash maps or linked lists), as implementing specialised data structures beyond simple arrays is complex and resource expensive in hardware. Programmable logic is useful for tasks which require consistent and reliable timing, and for improving the throughput of some expensive set of operations carried out in the inner loop of an algorithm. With that in mind, the next section will take a critical look at the DEFLATE algorithms, focusing on their suitability for hardware acceleration.

3.2 Algorithmic Analysis of DEFLATE

To decide which parts of a given algorithm or set of algorithms to accelerate in hardware, the following set of questions should be answered before investing resources in developing an accelerator:

1. Is performance critical for the application?
2. Have all the performance gains possible with a pure software approach been achieved?
3. How will a heterogeneous solution be deployed?
4. Will gains made using a heterogeneous solution be sufficient?
5. How suitable is the application for hardware acceleration?

Taking DEFLATE as our exemplary problem, we can most address these as follows:

1. The goal of this project is to improve algorithm performance as much as possible, so performance is critical by definition.
2. DEFLATE is an old and well known algorithm with a major canonical open source software implementation (GZIP), the code of which has been studied and optimized by countless engineers. It is unlikely any performance gains are available in the software domain.
3. Deployment is not relevant as this is an academic, not an industrial, project, although one can envision many solutions involving the gigabit per second Ethernet interfaces it is possible to instantiate on the device used.
4. As the goal of the project is just performance improvement, any gain made will be sufficient.

Question 5 of the suitability questions is somewhat trickier to answer, as we must look at the details of the algorithms in question in order to understand their suitability. Before we do so, a quick note about "big O" notation. Algorithmic time complexity can be expressed in terms of the number of operations carried out per item in the input data set, for example, algorithm 3.1, which was used to demonstrate a data dependency free algorithm, carries out an addition per input item, or n operations, while algorithm 3.3, which was used to demonstrate pipelining, carries out an addition and a multiplication per input item, or $2n$ operations. When talking about algorithmic complexity, it is often assumed n is so large that multiplying it by any value is meaningless, and algorithms are simply categorized in terms of their "order" relative to n. both algorithm 3.1 and 3.3 are as such considered $O(n)$ algorithms, which can be thought of as "each input item is 'looked at' effectively once". Algorithms on the order of $O(n)$, especially when they carry out a short, simple set of operations per input item[5], are said to be very inexpensive, beaten only by algorithms which achieve some work without considering the whole input data set (such as binary search).

3.2.1 Huffman Coding Acceleration Suitability

Huffman coding involves two primary loops: the calculation of frequency weights and the derivation of variable-length codes for each symbol. The finding of the symbol weights is extremely inexpensive, on the order of $O(n)$ with an n involving some hash function and an addition (to lookup the symbol in a symbol table and increment its frequency count). For smart implementations involving symbols with a bit width definitely less than some number, the hash function can even be optimized away in favour of treating the symbol as an unsigned integer and using it to index into an array. This is not exceptionally memory efficient, but for slender symbols, the trade off can often be worth it. Taking this into account, the finding of symbol weights is not suitable for hardware acceleration. The slowest part of this loop is often simple main memory reads and writes, which an FPGA can rarely improve, and taking into account that the second primary advantage of processors is their incredible speed, trying to beat a processor at this would be foolish.

The second primary loop of huffman coding appears at first to be a good candidate for hardware acceleration; it involves quite a few steps and operating on multiple operands at once, which implies it would be suitable for pipelining. Unfortunately, each iteration of the loop is also completely dependent on the result of the previous stage, making it unsuitable for parallel execution, and the concept of an ordered queue of complex object, while certainly very achievable in hardware, is tricky and error-prone. On top of this, the goal of that algorithm is to build a binary tree of values, which is another tricky data structure of indeterminate size whose principle function is based on memory references. These are not traits of an attractive acceleration target.

[5] "When n is small"

Huffman coding may be taken to have a third primary loop, the actual encoding of the symbol stream. This loop is similar to the first with a couple of exceptions; it uses a lookup table to convert a symbol into the variable length code representing the symbol, and it does not have the data dependency limitation of the first. Due to the already fairly low cost of this loop, it was not deemed suitable for acceleration. For the above reasons, it was decided that the Huffman coding algorithm was not suitable for acceleration as part of this project, and as such, the rest of this report shall discuss the acceleration of LZSS.

3.2.2 LZSS Acceleration Suitability

The LZSS algorithm is described at a high level in listing 2.1. From that listing, it can be seen that the algorithm has a primary outer loop, which moves across the input data set one byte at a time. The size of this input data is the N used for time complexity analysis of this algorithm. Within this outer loop, the algorithm requires finding the longest run of bytes in a sliding window of the last M bytes seen by the encoder which are a prefix of the next K bytes to be processed. Generally, relative to N, K is very small (15 bytes in the case of this project), which M can be quite large (4096 bytes in this project).

An algorithm which carries out this prefix location will require $K \times M$ byte comparisons, or 61,440 comparisons in this case of this project.LZSS requires that this prefix searching is done *for every byte pushed into the encoder*[6], which are not part of a substitution (following a substitution the number of bytes replaced by the substitution are pushed through the encoder into the sliding window without being searched for). This the worst case time complexity — appendix A.1 explores the relationship between N and the number of comparisons done in more details. Given the processor architecture outlined previously, it can be assumed that each comparison takes a load $\rightarrow$ execute $\rightarrow$ store cycle[7]. The complexity of the inner loop of LZSS makes it an excellent candidate for hardware acceleration, and so, it is the algorithm segment this project is focused on. Figure 3.6 illustrates the search process as described.

[6] Assuming a full sliding window and an input set which never has a 15 byte replacement candidate, Sensible encoder designs will stop searching the sliding window for a replacement early if it has checked as many positions as have been pushed into the sliding window or if it has found a max length substitution.

[7] "Vectorization" of this process would allow comparison of multiple bytes per cycle, and this can be used to improve performance somewhat, however, it is not a panacea for the time complexity of this algorithm.

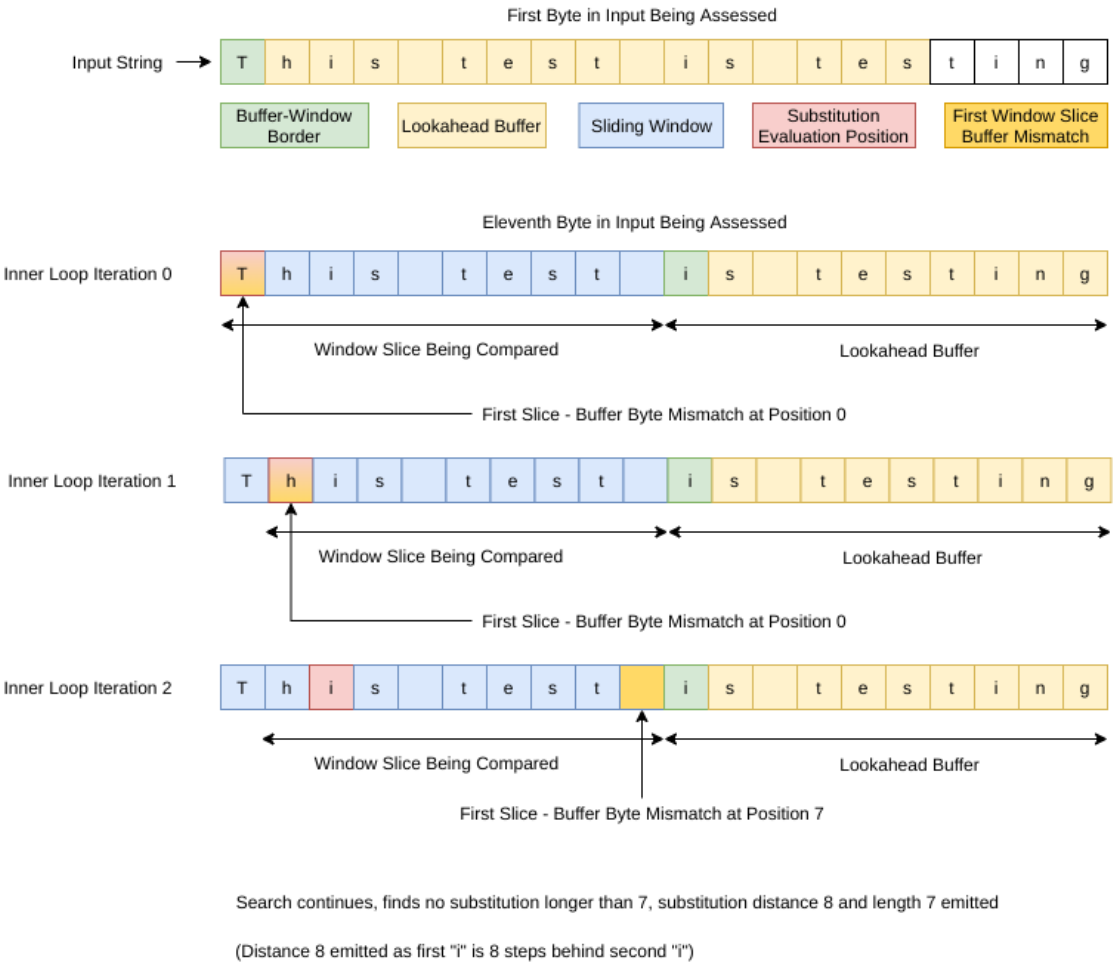

Figure 3.6: LZSS Substitution Searching

Chapter 4

Solution Design, Verification and Evaluation

With the selection of the inner loop of LZSS for hardware acceleration, certain key algorithm details become important to investigate; how to find the length of a shared prefix between two 15 byte strings, the memory structure of lookahead buffers and sliding windows, and the internal state machine of the inner loop. The following sections explore those topics as they were addressed in this project, and attempt to illustrates and illuminate the design thinking behind the accelerator architecture.

4.1 Finding Prefix Length

Finding the length of a shared prefix between two strings is a trivial problem by most standards, addressed in listing 4.1. The prefix length calculation algorithm presented in listing 4.1, however, takes at as many cycles (clock or load $\rightarrow$ execute $\rightarrow$ store) to complete as the length of the prefix found. If the upper bound of the prefix length is known, e.g. if the maximum length of one of the input strings is known, hardware can do significantly better.

Algorithm 4.1: Trivial Prefix Length Calculation

```
1  i is number 0
2  while input_str_1 [i] is input_str_2 [i] do
3      i := i + 1
4  repeat
5
6  output i
```

Table 4.1 outlines a lookup table based approach to prefix length finding, for an upper bound of 3. Given 2 strings of length 3, the input columns are boolean values representing whether bytes 1, 2 and 3 of the string matched. This would appear suitable to the LUT based architecture of modern FPGAs, however, this approach requires a memory space of 2^N addresses, where N is the string length upper bound. For this project, $N = 15$, which would require an address space of 32,768. This is not feasible for any FPGA based digital hardware,

and as such, a different approach must be taken.

Table 4.1: Prefix Finding as a LUT Operation

Byte 1 Match	Byte 2 Match	Byte 3 Match	Prefix Length
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	2
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	3

Table 4.1 illustrates how the problem of counting the length of a shared prefix of two strings may be converted into the problem of counting the number of leading 1's in a bit vector. Milenković et al. outline a hardware structure for the counting of leading 0's in a vector of 16 bits[10], in a purely combinatorial way (i.e. the output number changes immediately in response to a change in the input strings, without being gated by for a clock edge). This hardware structure is designed to reduce the "depth" of the combinatorial logic involved, which can be thought of as the number of operations a given change to a driving signal must propagate through before reaching the next register it will be stored in. The length of the register to register propagation path of a signal determines how long it takes for the driver of the output register to drive a valid signal. The clock driving the updating of that output register must have a period longer this propagation time, and as such, the longest propagation time in a design determines the maximum clock frequency that design supports.

It is critical to keep logic depth to a minimum to improve design performance, and as the primary computation done within the inner loop of the LZSS algorithm is this prefix length calculation, how this calculation is implemented is critical to design performance. The module used for prefix length finding in this project is shown in listing 4.2. The module is a derivation of the architecture outlined in [10]. Readers should note this and following modules' definitions use some "syntactic sugar", such as the use of `for` loops, array slices and ternary operators, the function of which is explained in listing A.2, in the appendices.

Listing 4.2: Prefix Length Calculator Module

```
module prefix_len_finder                                                    1
(                                                                           2
    input wire [7:0] str_1[14:0],                                           3
    input wire [7:0] str_2[14:0],                                           4
    output wire [3:0] prefix_len                                            5
);                                                                          6
//Bit vector of result of bitwise comparison between input strings          7
    wire [15:0] byte_matches;                                               8
```

```
//Reshaping of byte_matches into constituent nibbles
    wire [3:0] nibbles [3:0];
//vector testing whether each nibble of byte_matches consists only of 1's
    wire all_one_nibbles[3:0];
//16th bit of byte_matches is always 0 as input strings 15 bytes long
    assign byte_matches[15] = 1'b0;
//Perform assignments to nibbles and all_one_nibbles as described
    generate
        genvar i;
        for(i = 0; i < 15; i++) begin
            assign byte_matches[i] = (str_1[i] == str_2[i]);
        end
        for(i = 0; i < 4; i++) begin
            assign nibbles[i] = byte_matches[(i*4)+:4];
            assign all_one_nibbles[i] = &nibbles[i];
        end
    endgenerate


//"Loop" through all_one_nibbles, assigning output prefix length
//the index of the lowest nibble which does not contain only 1's,
//plus the number of continuous 1's that nibble does contain.
//This accurately calculates the length of the shared prefix between
//str_1 and str_2
    always_comb begin
        prefix_len = 15;
        for(int i = 3; i >= 0; i--) begin
            if(!all_one_nibbles[i]) begin
                prefix_len = (i*4) + (&nibbles[i][2:0] ? 3 :
                                      &nibbles[i][1:0] ? 2 :
                                      nibbles[i][0] ? 1 : 0);
            end
        end
    end
endmodule
```

The function of this module is to count the prefix match between two input strings, and it is the core combinatorial logic of the hardware accelerator developed for this project. It produces this count by first producing a bit vector of length 16, where each bit from positions $1 \rightarrow 15$ are driven by an equality check of the bytes in string 1 and 2 at that position. Position 16 is tied to 0, as the byte matches vector is forced to be of length 16 in order to be broken cleanly into nibbles, but the length of the prefix match will never be greater than 15.

Once this vector of byte matches has been created, a second vector of bits of length 4 is created. This vector is assigned a boolean value indicating whether each nibble[1] of the match vector contains only 1's. This vector of all one nibble test results will be referred to as the nibble test vector. The output of the module is assigned the lowest index of a 0 in this nibble test vector multiplied by four (as the vector is zero indexed, this accounts for the continuous run of 1's in the byte match vector up until this nibble), plus the number of 1's in the nibble that the index in the nibble test vector represents.

4.2 Lookahead Buffers and Sliding Windows

The LZSS algorithm calls for a sliding window of, in this project, 4096 bytes, and a lookahead buffer of 15 bytes. Given position in the input data array P, the lookahead buffer should contain the bytes in the input buffer as shown in table 4.2. This is implemented fairly trivially with a simple shift register, where new bytes are pushed into position 15, and registers $14 \rightarrow 0$ take the value of the register above them (i.e. position 0 reads from position 1, position 1 from position 2, etc.). Listing 4.3 demonstrates in a simplified form the implementation of this behaviour in this project. Readers should note carefully the use of blocking assignments to ensure the correct "trickle-down" behaviour.

Table 4.2: LZSS Sliding Window Structure

Lookahead Buffer Position	Input Data Position
0	P
1	$P+1$
2	$P+2$
3	$P+3$
4	$P+4$
5	$P+5$
6	$P+6$
7	$P+7$
8	$P+8$
9	$P+9$
10	$P+10$
11	$P+11$
12	$P+12$
13	$P+13$
14	$P+14$

Listing 4.3: Lookahead Buffer Implementation

```
input wire clock;                                    1
input wire [7:0] byte_in;                            2
reg [7:0] buffer[14:0];                              3
```

[1] A nibble is a bit vector of length 4, "half a bite", so to speak.

```
                                                                    4
always@(posedge clock) begin                                        5
    for(int i = 0; i < 14; i++) begin                               6
        buffer[i] = buffer[i+1];                                    7
    end                                                             8
    buffer[14] = byte_in;                                           9
end                                                                 10
```

The sliding window is a simple rolling buffer, implemented as an register file of 4096 bytes. The window maintains a read and a write pointer, and each time the window is written to, both read and write pointers are moved forward, wrapping to position 0 at position 4096. As such, walking backwards through the register file from the current write position and appending each byte found produces a string of the last 4096 bytes pushed into the window. This operation is illustrated in figure 4.1.

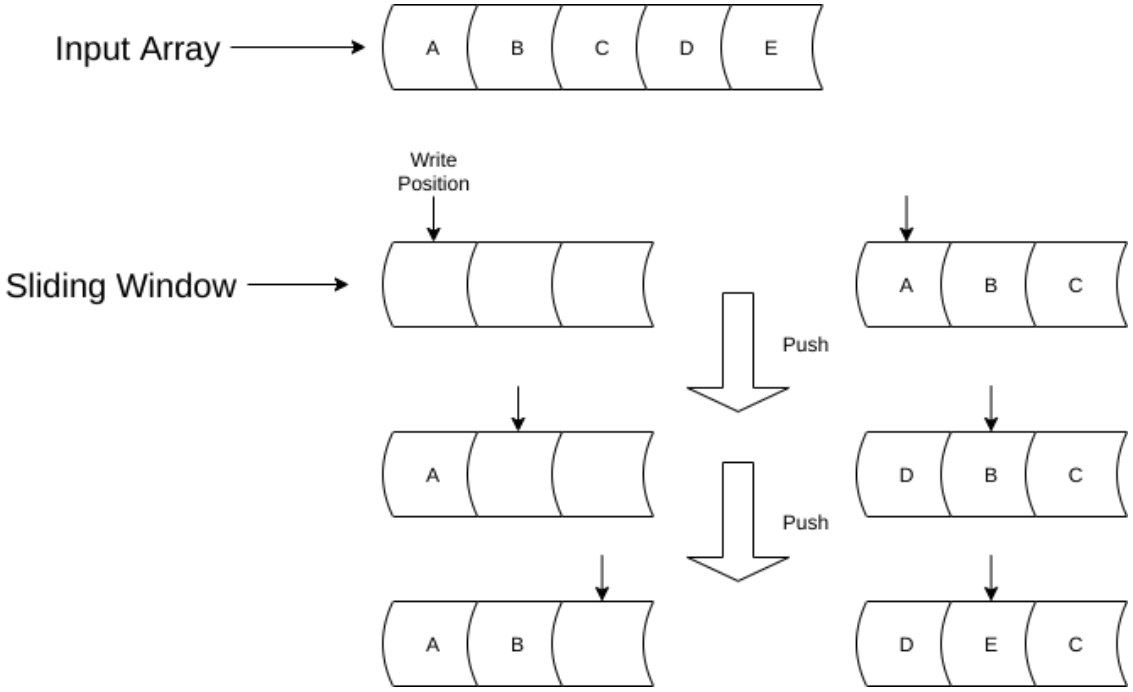

Figure 4.1: Operation of a Rolling Buffer

The requirements of the LZSS algorithm are such that while searching for a substitution position/length combination, a "slice" of bytes equal in length to the lookahead buffer must be extracted from the sliding window. Listing 4.4, taken from the accelerator implementation, displays how, given a read position, a string from the read position to 15 bytes earlier is constructed, such that the window slice and the current lookahead buffer may be used to drive the prefix length finding module detailed previously. Note the modulus operator implementing the wrapping behaviour of the rolling buffer.

Listing 4.4: Window Slice Implementation

```
reg [7:0] window[4095:0];
wire [7:0] window_slice[14:0];
reg [11:0] cur_read_pos;

generate
    genvar i;
    for(i = 0; i < 15; i++) begin
        assign window_slice[i] = window[(i + cur_read_pos)% 4096];
    end
endgenerate
```

4.3 State Machines and Reference Searching

The previous two sections describe how doing a 15 byte prefix length calculation is achieved in a single clock cycle while still maintaining a shallow enough logic level to allow a reasonable clocking frequency. This means the accelerator core can perform an iteration of the LZSS inner loop in a single clock cycle, a marked improvement on the 15 load → execute → store cycles required to perform the same inner loop in software. However, the accelerator core must now be able to store some internal state, which will encode what operation the accelerator should be doing on any given clock cycle. The fundamental states chosen for the accelerator core to occupy for this project are: idle, see arching for a reference, and clearing its internal memory.

To understand the accelerator core state machine, the core input and output signals should be understood. Listing 4.5 shows the input and output signals of the core. On the rising edge of the input clock, the core takes some set of steps and then evaluates the transitions available to it.

Listing 4.5: Accelerator Core Interface

```
module max_prefix(
//Inputs
    input wire clk,//Input clock signal
    input wire rst,//Core reset signal
    input wire push,//Control signal, forces byte_in to be pushed into the
        lookahead buffer if the core is idle
    input wire search,//Control signal used in state machine
    input wire[7:0] byte_in,//Input byte
//Outputs
    output wire waiting,//Indicates core is idle
    output wire busy,//Indicates core is busy, push/search ignored
```

```
    output reg res_valid,//Indicates the current output is valid, asserted      11
        when no push or reset has occurred since last search finished
//Substitution search results for this window and buffer combination:          12
    output reg[11:0] max_pos_out,                                               13
    output reg[3:0] max_len                                                     14
);                                                                              15
```

The core state machine is displayed in figure 4.2. The signals *done* and *clear* are internally generated. *Done* is asserted when the core has evaluated the prefix length at every position in the sliding window, when it has evaluated the same number of positions as bytes pushed into the sliding window or when it has found a substitution of length 15. *Clear* is asserted when the core has set all positions in the sliding window to 0 or when it has set the same number of positions in the sliding window to 0 as the number of bytes written to the sliding window (this is only reached before setting all positions when less than 4096 bytes have been processed).

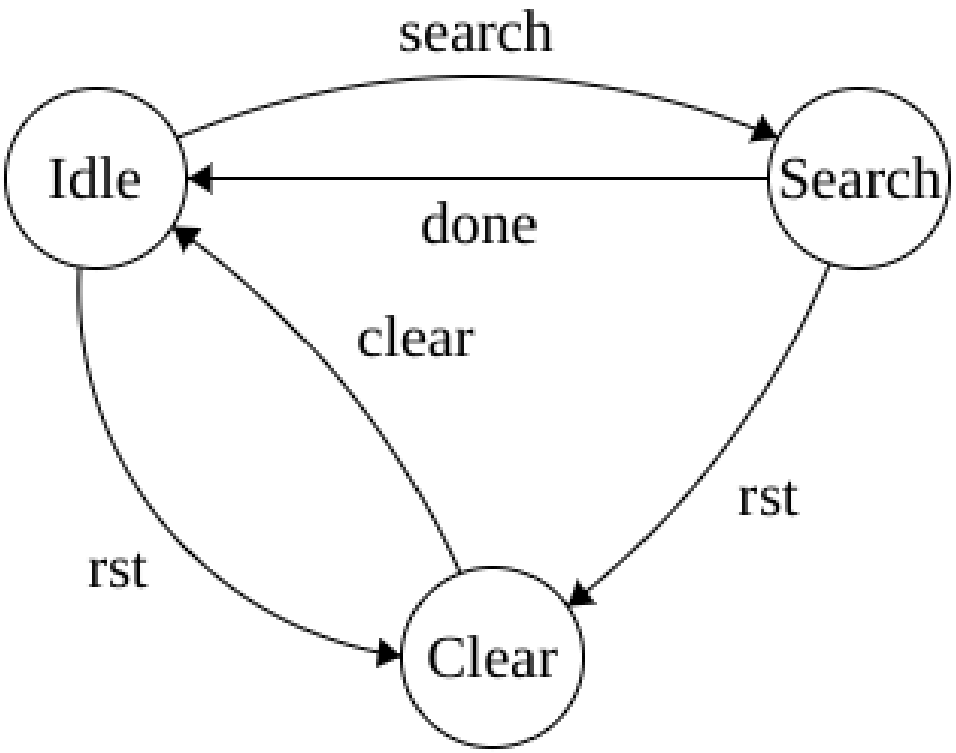

Figure 4.2: Accelerator Core State Machine

This state machine is implemented using a SystemVerilog `enum`, as shown in listing 4.6, in order to make it easier to reason about the code.

Listing 4.6: Accelerator Core State Space Declaration

```
typedef enum {
    ready,//Able to accept a new byte or start searching
    searching,//Busy, searching for match
    clearing//Busy, clearing sliding window
} State;

State s;//State tracking variable within core
```

4.4 Core Interfacing and Software Wrappers

As described previously, the core uses an AXI-Lite interface to connect to any processor in the design. The principles of AXI-Lite are outlined in section 3.1.4. As discussed there, the AXI-Lite interface maps the peripheral to a particular memory address or memory range (as some peripherals will have multiple addressable registers). From the embedded C or C++ application, interacting with the memory mapped peripheral is done through reading and writing to memory address in that range.

4.4.1 Interface Particulars

The accelerator core developed for this project operates using a 32 bit wide AXI bus, and only exposes a single register. This means that the write and read address channels are only used to initiate a transfer through the associated ready and valid signals, and the actual address sent is ignored. Further, this core never responds with a failure for reads or writes[2], and as such, the read and write response channels are simply tied to 0.

Of that single write channel exposed, the core only uses the lower 9 bits. On a write, bits 0 to 7 are used to drive the `byte_in` signal shown in listing 4.5, and bit 8 is used as an optional software reset. The core `rst` signal will be asserted on a bus reset, or when bit 8 in the write data signal is asserted. This is a design feature which allows developers to reset just the LZSS core, without resetting the entire AXI bus, which may also have other peripherals on it. A write will stall, meaning `write_valid` is not asserted by the slave, if a write is performed while the core is asserting `busy`, which means that if the core is reset in software and then immediately written to the processor will hang for as long as it takes the core to reset, a maximum of 4096 clock cycles. This is by design, in order to avoid the core dropping write transactions while busy.

When the bus master initiates a read transaction, and `res_valid` is not asserted (remember `res_valid` is only asserted in the case that the core is currently outputting valid

[2]There is not case where pushing a byte into the encoder will fail. It may hang while the encoder resets or conducts a search, but it will not fail.

substitution values for its current window and buffer state), `search` is asserted. The core will then conduct a search, and assert `res_valid` once the search is complete, which will in turn cause the AXI-Lite interface wrapper to drive the AXI read data channels with the core substitution data output, and assert a read valid signal, which it will hold until the processor asserts read ready as well. The lower 12 bytes of the read data is the position from the front of the sliding window, and bits 12 to 15 are the length of the substitution found.

The design of the core, with substitution searches being conducted on read, is for a couple of performance focused reasons: performing search on read allows the controlling software to push the first 15 bytes as fast as the processor can (as each write transaction takes 2 clock cycles, one of which is a response which can be sent as the next write is starting), and performing search on reads allows the processor to push as many bytes through the encoder as were substituted in the case of a valid substitution being found. This allows the accelerator core to be used as efficiently as possible.

4.4.2 Hardware Platforms

All of the hardware design carried out for this project was through the Vivado 2019.1 FPGA design tool, which is the developed by Xilinx, Inc. The flow used to develop this hardware was as follows:

1. Start a project to develop accelerator, without AXI-Lite interface wrapper, and verify basic functionality.
2. Develop the AXI-Lite interface wrapper in the same project as the core, package core as an "IP", Xilinx's term for a reusable module design for integrating into larger design.
3. Start a new project, and create a "block design", which is a high level design technique in which the design engineer visually instantiates and connects different "IP" or modules. This block design includes both the accelerator core "IP" and whatever processor system is being used, either the Zynq processor subsystem or a MicroBlaze soft processor.
4. Create a "HDL" wrapper around this block design, in order to make it possible for the Xilinx tools to synthesize and implement the design. The Vivado design suite can manage this wrapper automatically.
5. Produce a bitstream from this design, and then "export" that bitstream and the hardware project to the project workspace, in order to make it accessible to the Vivado Software Development Kit (SDK).
6. Launch the SDK from Vivado, noting that the hardware platform is present due to the previous export steps, and create a "Board Support Package" (BSP), selecting the standalone "operating system" option.
7. Create a new "application project", selecting the just-created board support package as the project BSP.
8. Open the "xparameters.h" file in the newly created BSP folder, and search for the ac-

celerator core name, in order to find the memory address it has been assigned[3]. There will be a preprocessor macro defined to map to the correct memory address.

9. Using the memory address macro, write software to interact with the core.

Section 4.5 discusses the process of simulating and verifying the hardware accelerator. This process is done with the block design displayed in figure 4.3 (accelerator core highlighted in orange), and the C software application shown in listing 4.7. The software shown was built and the produced executable linkable format (ELF) file loaded into the MicroBlaze program and data memory by associating the ELF file with the block design, using the option in Vivado. This allows the design engineer to verify core functionality and performance without access to physical hardware, and in an often preferable manner, as debugging is significantly easier when it is possible to probe every signal in the design.

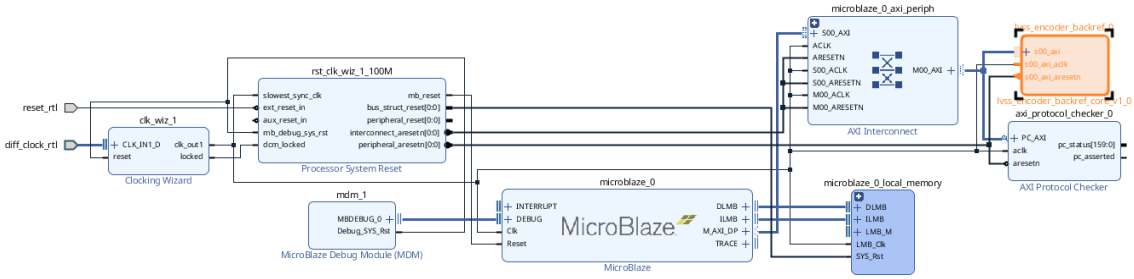

Figure 4.3: Core Simulation Hardware Platform Block Diagram

Listing 4.7: Core Verification Software

```
#include "xparameters.h"
#include "inttypes.h"

int main(){
  uint32_t *encoder = (uint32_t *)XPAR_LVSS_ENCODER_BACKREF_0_BASEADDR;
//Testing pushing multiple times without reading
  for(int j = 0; j < 2; j++){
    for(uint32_t i = 1; i <= 15; i++){
      *encoder = i;
    }
  }
  uint32_t res = *encoder;
  while(res > 0) res = *encoder;//Testing multiple reads without pushing
  return(0);
}
```

[3]Note that this is configurable as part of the block design.

4.5 Core Simulation and Verification

The application code in 4.7 pushes and identical 15 bytes of data to the LZSS accelerator core twice in a row, then reads back the substitution information from the accelerator core. This allows for the verification of multiple facets of the accelerator; the AXI-Lite interface, the buffer and window implementations, the prefix length counter and the state machine transitions. The block design pictured in figure 4.3 has two inputs, which are a reset signal and a differential clock[4]. The system is simulated by designing a Verilog testbench which simply drives the reset signal high for a 1000 nanoseconds, and then driving the differential clock at a frequency of 200 MHz. Listing 4.8 shows how the simulation features of Verilog make this possible. This testbench is sufficient to verify the core due to the preloading of the processor cores memory resources with the data from the software application ELF files, which causes the software application to immediately start executing on the core once the reset signal is de-asserted.

Listing 4.8: MicroBlaze Simulation Testbench

```
`timescale 1ns / 1ps

module mb_tb(

    );

    reg clk_p;
    reg rst_n;
    wire clk_n = ~clk_p;

    initial begin
        clk_p = 0;
        rst_n = 1;
        #1000 rst_n = 1;
    end

    always #2.5 clk_p <= ~clk_p;

//Instantiate the block design "HDL wrapper", and drive its input ports
//with the testbench simulation signals
    mb_bd_wrapper mb(clk_p, clk_n, rst_n);
endmodule
```

This approach allows the observation of the AXI transactions between the acceleration

[4]The differential clock requires 2 ports, a positive and negative, but is grouped here as a single input

core and the MicroBlaze processor. Additionally, the Xilinx provided "IP" module "AXI Protocol Checker" was used to verify the implementation of the AXI protocol interface wrapper developed for this project, and was invaluable for doing so[11]. This module monitors the AXI bus it is connected to for invalid signals, and diagnoses any invalid state it notices. By way of example, the initial interface implementation did not every drive the write response valid signal low, as it was reasoned that as writes to the accelerator cannot fail, the write response is always validly tied to 0. This is actually a breach of the AXI specification[7], and caused the processor to deadlock, as the processor tracks the number of "outstanding writes" on the AXI bus, and a continuously high write response valid and write response ready signal caused the that counter to overflow. The AXI Protocol Checker was used to diagnose this problem, and the interface wrapper updated to address it.

When the above described simulation is run, and the AXI bus monitored, a series of 30 write transactions followed by an indefinite number of read transactions can be observed, as would be expected given the software outlined in listing 4.7. Figures 4.4 through 4.6 show the AXI bus during simulation. Some details to note are the write transactions occurring in two clock cycles, the first read transaction taking 19 clock cycles (4.6), which is 2 on each end as bus overhead, 15 to complete the search, and the subsequent reads taking only 2 clock cycles (as no new data is pushed to the core and as such the core continues to assert `res_valid`). The read data signal is also useful as it is driven to `0xf00f` following the first read, which indicates an optimal substitution of length 15, 15 bytes behind the current read head. This is exactly what we would expect given the input data pushed to the accelerator by the control software.

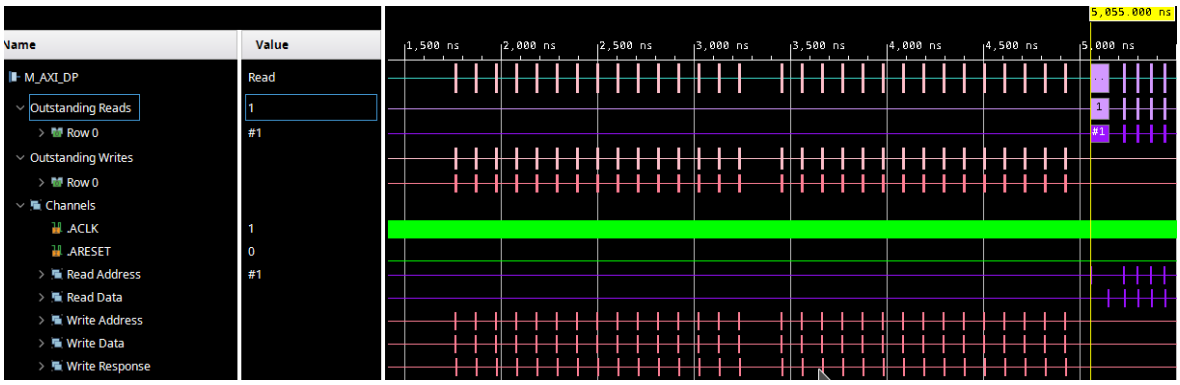

Figure 4.4: Core Simulation Overview

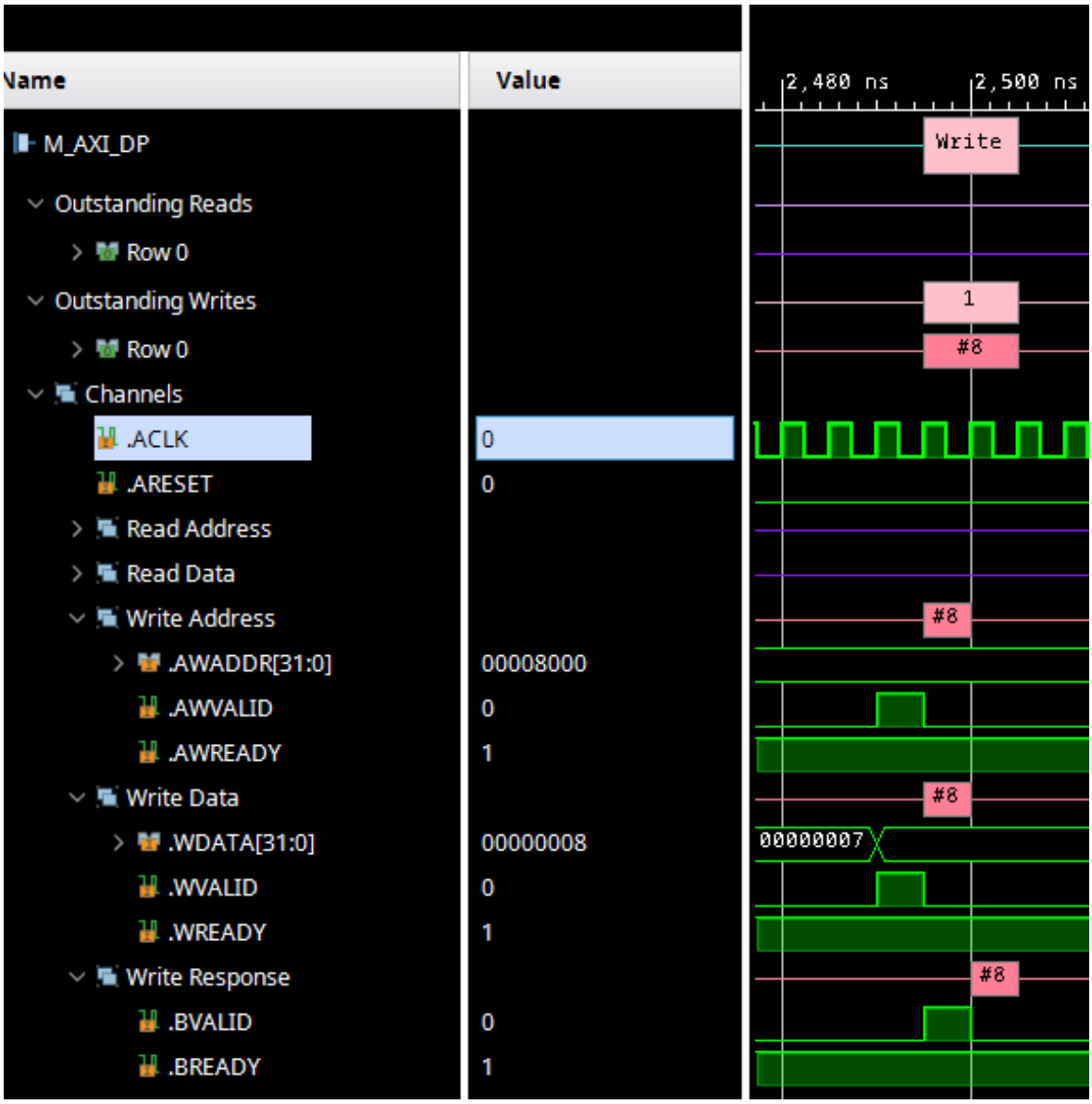

Figure 4.5: Core Simulation Write Transaction Detail

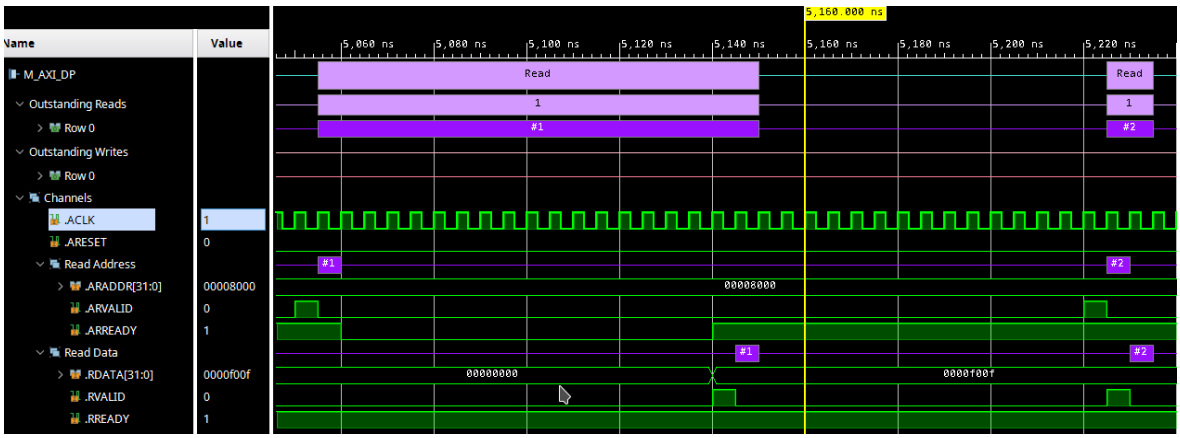

Figure 4.6: Core Simulation Read Transaction Detail

4.6 Accelerator Performance and Resource Utilization

Due to an unfortunate lack of foresight on the part of the author and a global pandemic at time of writing, it was not possible to test the accelerator core performance on hardware. Due to this misfortune, the accelerator will be assessed in terms of FPGA programmable logic resource utilization and performance versus the MicroBlaze soft core it was verified using. Some caveats apply to this evaluation; the soft core is running at the same clock frequency as the accelerator core, while the Zynq core on the target device for this project runs roughly five times faster than the accelerator core can achieve, and the MicroBlaze core has a different instruction set with different performance characteristics than the Zynq ARM A53 core. It would be a mistake to assume performance versus the MicroBlaze core is indicative of performance versus the Zynq ARM core.

4.6.1 Resource Utilization and Timing Slack

The Vivado design suite provides various reports of the utilization of a design during implementation. Taking the "IP" design, consisting of the accelerator core and the AXI-Lite interface wrapper, and bringing them through implementation, the utilization of resources of interest are presented in table 4.3. The design was implemented with clocking constraint with a period of 4.7 nanoseconds, which means the implementation software was trying to implement the design as if it was being driven by a clock running at 212 MHz.

Table 4.3: Accelerator Resource Utilization

Resource Name	Number Used	% Utilization
LUTs as Logic	3671	5.2
LUTs as RAM	9600	33.33
Flip Flops	512	0.36

The design, when implemented with all optimizations turned on, had a worst design slack of 0.058 nanoseconds, meaning the slowest signal propagation time was 0.058 nanoseconds faster than the clock period. The extremely low slack indicates 4.7 nanoseconds to be roughly the minimum period of a clock driving this design, and the safe operating clock speed of the design is estimated to be 200 MHz, to account for potential propagation delay fluctuations due to deployment environment.

The high utilization of available LUT as memory resources is due to the sliding window implementation being targeted at this resource, and the sliding window being quite large at 4096 kilobytes. This resource class was targeted for the sliding window instead of the RAM hard blocks on the device (block RAM and "Ultra" RAM), despite those resource being far more plentiful in terms of available storage space, as the flexibility of the distributed RAM allowed for the single byte write width and 15 byte read width architecture required to achieve the performance of the hardware accelerator. This is an unfortunate trade off, and a potential

avenue for future work is to design this somewhat specialised memory architecture in such a way that it can be implemented using the available hard RAM blocks. If this could be achieved, this core would use a very low percentage of available device resources, making it extremely suitable for integration into larger designs.

4.6.2 Performance Comparison

The goal of hardware acceleration is to produce a heterogeneous system which accomplishes the task some previously software-only system was designed for, faster or more efficiently than the software system did. As such, one of the most important evaluation metrics of the hardware system is a direct comparison between the software-only system and the hardware-software co-design. Table 4.4 presents a comparison between the runtimes of a software-only system and a heterogeneous system. Both systems were given an input array of length 36, and both systems produced identical results. Listings 4.9 and 4.10 are the software-only and the heterogeneous algorithm implementations, respectively. The test software, which ran one implementation after the other and then output the results to the AXI bus for easy comparison, was compiled using the GCC which is provided as part of the Vivado SDK. The software was compiled at optimization level 0, 1, 2 and 3.he internal structure of the test application is more complex, so the runtime is taken as the difference between simulation time when the first instruction in each test subroutine is executed and the simulation time when the result length of the subroutine is written to the axi bus. The design was running at a clock frequency of 200 MHz for all tests. All runtime in nanoseconds.

Table 4.4: Software vs. Heterogeneous Implementation Runtime

Optimization Level	HW Start	HW End	SW Start	SW End	Speedup
0	1,985	20,060	28,490	257,370	12.6x
1	1,900	11,075	15,025	92,920	8.5x
2	1,895	11,115	14,895	98,010	9x
3	1,885	10,720	13,365	96,480	9.4x

Table 4.4 shows that the accelerator significantly outperforms the MicroBlaze processor core at all optimization levels. 6 includes some discussion of further work which may be done to further increase heterogeneous system performance gains.

Listing 4.9: LZSS Software Implementation

```
uint16_t test_software(char *data_in, uint32_t in_len, char *data_out){
    uint32_t out_len = 0;
    uint32_t i = 0;
    while(i < in_len){
        uint16_t cur_skip_pos = i-1;
        uint8_t best_skip_len = 0;
        uint16_t best_skip_pos = 0;
```

```
        while(cur_skip_pos > 0 &&
                i - cur_skip_pos < 4096 &&
                best_skip_len < 15
        ) {
            uint8_t cur_skip_len = 0;
            for(uint32_t j = 0;
                j < 15 &&//15 byte lookahead buffer
                cur_skip_pos + j < i &&
                i + j < in_len;
                j++
            ){
                if(data_in[cur_skip_pos + j] != data_in[i+j]){
                    break;
                } else {
                    cur_skip_len += 1;
                }
            }
            if(cur_skip_len > best_skip_len){
                best_skip_len = cur_skip_len;
                best_skip_pos = cur_skip_pos;
            }
            cur_skip_pos--;
        }
        if(best_skip_len > 1){
            data_out[out_len++] = ((char)(best_skip_len<<4)) |
                ((char)(best_skip_pos>>8));
            data_out[out_len++] = (char)best_skip_pos;
            i += best_skip_len-1;
        } else {
            data_out[out_len++] = data_in[i];
        }
        i++;
    }
    return(out_len);
}
```

Listing 4.10: LZSS Hardware Test Driver

```
uint32_t test_core(char *in, uint32_t in_len, char *out){
    uint32_t out_len = 0;
    volatile uint32_t *encoder =
        (uint32_t*)XPAR_LVSS_ENCODER_BACKREF_0_BASEADDR;
```

```
    uint32_t padded_len = in_len + 14;
    int i;
    for(i = 0; i < 14; i++){
        *encoder = in[i];//Fill the encoder buffer with valid data
    }
    while(i < padded_len){
        *encoder = i < in_len ? i<<16 | (uint32_t)in[i] : 0x0000;//Push a
            byte in, padding the end of the string with null char
        uint32_t res = *encoder;//Read result from core for that byte
        uint32_t skip_len = res>>12;//bits 12 to 15 are length of substitution
        if(skip_len > 1){
            out[out_len++] = (char)(res>>8);
            out[out_len++] = (char)res;
            while(--skip_len > 0 && i < padded_len){
                i++;
                *encoder = i < in_len ? i<<16 | (uint32_t)in[i] : 0x0000;
            }
            i++;
        } else {
            out[out_len++] = in[i-14];
            i++;
        }
    }
    *encoder = 0x100;//Software reset the encoder
    return(out_len);
}
```

As *only* the inner loop of the LZSS algorithm which is being accelerated here, it is expected that as the input data size grows the difference in performance between the hardware and software implementations would also grow, up until the point at which the input data set is larger than the sliding window of 4096 bytes. This is because as the input data size grows, the amount of time spent executing the inner loop also grows, and as such the component of total runtime that is spent in that inner loop grows. This is to the accelerators advantage, is at can achieve in a single clock cycle within that inner loop what must take any processor atleast $3 \times K$ clock cycles[5], where K is the length of the prefix between the lookahead and a given position in the sliding window. Assuming a low K of 5, we would expect the processor to work through the input data roughly 15 times more slowly than the accelerator core. Some of the factors causing the core to underperform in this respect are

- AXI bus request overhead for reads and writes (2 cycles per transaction)

[5] A comparison, an increment of the index into the input data and a jump. This calculation ignores jump overhead and assumes loads are done as part of the comparison. The actual number of cycles is likely closer to 6 or 7.

- software-side data handling to emit the encoded data stream (multiple writes and bit shifts not seen in software implementation)
- overhead due to the pushing of 15 null characters into encoder after final byte of actual data[6]

A longer running test with an input data size in the tens of kilobytes would provide a more stringent assessment of the accelerator core performance, however, this is not possible due to the limitations of the simulation based approach this report is unfortunately constrained to.

[6]This is particularly expensive, and a direct result of the memory architecture. If the project was to continue, a primary goal would be the removal of this overhead.

Chapter 5

Ethics

According to [12], the requirements of this, mandatory, chapter are

- "All potential health and safety risks associated with the project [are] clearly identified and assessed" and "Relevant directives / standards / regulatory bodies cited"
- "Clearly identified the [ethical] principle, describes the ethical problem/s in detail having gathered pertinent facts & ascertains exactly what must be decided / what policy should include"
- "Determines who should be involved in the decision making process & thoroughly reflects on the viewpoints of the stakeholders" and "Clarifies a number of alternatives & evaluates each on the basis of whether or not there is interest & concern over the welfare of all the stakeholders"

The goal of this chapter is to meet these requirements. Please see 6.2.1 for the author's opinion on the ethical implications of this chapter. Readers interested only in the technical details of this report should skip this chapter, as it provides no value to them.

5.1 Identification of Health and Safety Risks

This project involved a relatively limited amount of equipment. The project work was done using a laptop, occasionally plugged into a desktop setup with a monitor, keyboard and mouse. The physical hardware uses a 12 Volt power supply, and as such, is unable to drive enough current to be dangerous to the operator. The only health and safety risk this project posed was the ergonomic risks posed by being a VDU user. The precautions taken to address this risk were as per health and safety authority recommendations[13]. As no other health and safety risks exist, this action sufficiently addresses all risks present as part of this project.

This section addresses the requirements: "All potential health and safety risks associated with the project [are] clearly identified and assessed" and "Relevant directives / standards / regulatory bodies [are] cited".

5.2 Environmental Impact

This project makes use of silicon-based microchips. The process of making these is involves many toxic chemicals, which is bad for the environment. This project was also conducted using a laptop, which consumes electricity to work. This electricity was generated by the Electrical Supply Board of Ireland (ESB), which does not use renewable source to generate the majority of its energy supply capacity. This is also bad for the environment.

There is an environmental benefit to be considered due to the decreased number of clock cycles required to complete the data compression. Most modern FPGAs can be put into a deep sleep mode, in order to conserve energy usage when not doing computation, and the ability to complete a task faster allows a system to return to its low power mode sooner, thereby saving energy (so called "race to sleep"). This is good for the environment, if the technology developed here is deployed at a significant scale.

5.3 Ethical Principles

This section aims to "Clearly identified the [ethical] principle, describes the ethical problem/s in detail having gathered pertinent facts & ascertains exactly what must be decided / what policy should include".

The ethical principle used to conduct an ethical analysis of this report is that of utilitarianism. The ethical quandry with this project, with respect to utilitarianism (particularly the utilitarian goal of maximising good for most people, interpreted in the hedonistic sense as maximising pleasure for the most people), is how well this project maximises pleasure. The project was enjoyable for the author to work on, and as such, maximised pleasure well during development. Unfortunately, it is possible readers will find this report displeasurable, and as such, the final output of the project may be more displeasurable than pleasurable. In order to avoid this, the policy around this project and report is that any reader who finds it displeasurable should stop reading immediately.

5.4 Stakeholder Determination and Alternative Evaluation

This section "Determines who should be involved in the decision making process & thoroughly reflects on the viewpoints of the stakeholders" and "Clarifies a number of alternatives & evaluates each on the basis of whether or not there is interest & concern over the welfare of all the stakeholders".

The stakeholders of this project are: the author. No individuals or organisations beyond the author are impacted by this project, and no individuals or organisations are impacted by this report except by choice. As such, the author should be involved in the decision making

process. Including others in the decision making process of this project is not necessary. What follows is an attempt to "thoroughly reflects on the viewpoints of the stakeholders".

The author's viewpoint is roughly 5 feet and 6 inches above the ground[1]. The author is not colourblind, and is not vision impaired in any known way, although some would argue the author lacks foresight. The author's viewpoint is that of an undergraduate electronic and computer engineering student, specialising in systems and devices. The author's view on this project at the time of writing is explored in more depth in section 6.2.

Meaningful alternatives to the steps taken in this project, which would acheive the same outcomes, do not exist.

Thus concludes this chapter, which is focused on the ethical implications of the work done during the course of this project. To reiterate, this chapter is written to meet the requirements:

- "All potential health and safety risks associated with the project [are] clearly identified and assessed" and "Relevant directives / standards / regulatory bodies cited"
- "Clearly identified the [ethical] principle, describes the ethical problem/s in detail having gathered pertinent facts & ascertains exactly what must be decided / what policy should include"
- "Determines who should be involved in the decision making process & thoroughly reflects on the viewpoints of the stakeholders" and "Clarifies a number of alternatives & evaluates each on the basis of whether or not there is interest & concern over the welfare of all the stakeholders"

Meeting these requirements, which this chapter does by addressing each individually and fulfilling them as described. This is chapter was written as it is mandatory.

[1] Assuming a forehead size of roughly 3 inches

Chapter 6

Conclusions and Further Research

This chapter is broken into two major sections, the first of which will provide an analysis of the project as it stands and identify pathways to further development of the accelerator core, while the second will be a brief informal commentary on the state of the project. The latter section will be written in the first person, to make it clear it is the opinion of the author being expressed, and that what is expressed is purely opinion.

6.1 Further Research Pathways

The project presented has a number of interesting threads on which one may pull:

1. The Huffman algorithm is not addressed in hardware as part of this report. While an analysis of this algorithm showed it should not be the principle target for acceleration, it still presents a potential avenue for improved total system performance.
2. The memory architecture of the sliding window was designed to target LUT based ram. This allowed for very high performance, at the cost of the design using a percentage of the available LUTRAM resource on the target device. It is possible some modifications to the memory architecture and access patterns would allow the sliding window to be implemented in a ram hard block on the device, massively reducing the footprint of the accelerator core.
3. The accelerator as designed implements only the inner loop of the LZSS algorithm. Implementing the full LZSS algorithm in hardware, including all of the algorithm features implemented in software in listing 4.10, and reducing the software driver to a minimum, would improve the performance further. This can be seen by the results in table 4.4, as the hardware implementation runtime is improved as a result of increased optimization, implying the software driver contributes significantly to algorithm runtme.
4. The AXI-Lite interface used in this project requires a minimum of 2 clock cycles per write and 3 cycles for a read. Following the finding of a valid substitution, the software driver pushes the number of bytes substituted into the core sequentially, without reading between pushes. Using an AXI-Full interface instead of the AXI-Lite one used would

allow this multi-byte push sequence to occur with the overhead of a single write.

5. With the use of AXI-Stream based direct memory accessing (DMA), the accelerator core could be designed in such way that it does not require communication with the processor subsystem in order to process the input data or emit the encoded dataset. The processor could simply send the accelerator the memory location of the input data, the length of the input data and a memory location to write the output to. Following this, the core could do all data fetching, processing and emitting itself, reducing the communication overhead significantly.
6. Additional signalling between the accelerator core and the processor would be useful, in order to allow the processor to send signals to the accelerator indicating the final byte of the input data has been pushed, for example, instead of the current implementation, which requires that the software driver pad the input data with a trail of 15 null bytes.

Pursuing any of the above avenues for investigation would almost certainly yield improvements in accelerator performance.

6.2 Project Retrospective

Hi, I'm Oliver. My voice is a little different to the one you've been reading for the last while, because this part is different. I'm going to talk about how I *feel* about certain aspect of the project, and what I would do differently, if I was to do all this again. I'll pause here, briefly, to acknowledge that this type of writing is discouraged in academia, and it also runs counter to my understanding of the Engineers Ireland Code of Ethics, which prizes objectivity from its members. However, I'm not an engineer yet, and I think how engineering feels is important, so as the one writing this report, I've decided to finish it like this[1]. If that bothers you, feel free to not read this section, I won't be offended. I understand that this is strange.

This project is the longest thing I have worked on continuously, and probably one of the most complicated things I've ever built. I understand that may seem absurd the you, if you're a hardware engineer used to this kind of work, but I am not, and this felt more difficult than anything I've done before. I think it's important to admit when things are hard. For reference, second place goes to a code coverage tool I once built, which had to work on a C++ codebase with more than a million lines and more than 100 dynamic libraries, and couldn't impose a runtime overhead greater than 30%. I'm rather proud of that project, as it was the first piece of engineering work I was paid to do, that I had design control of. I am also proud of it because it *worked*. I think taking honest pride in the work I do is healthy, and I think others should do the same.

I am not proud in an uncomplicated way of this project, although there are parts of it I *am* proud of. As you may have been able to tell by the general tone of certain sections of this report, I did not achieve all I had hoped to when I started working. I proposd this

[1] This is also likely the last piece of writing I will do as an academic, so this feels appropriate.

project[2] having worked for an FPGA vendor for the two summers prior to this one. I had been working on implementation software verification, and seen some of the interesting technology that existed in the FPGA space — technology so interesting that I wrote the project proposal to give myself the opportunity to "play" with some of it. Due, I suspect, to inexperience with digital design, I included in the proposal more than I now believe was possible for me to do.

The original goal I had for this project was a fully "hands-off" accelerator, which would be similar to the AXI-Stream based solution described in the previous section. I set out with the goal of building something I had internally been describing as "genuinely useful". As the college year went on, it became clear to me that I would not be able to dedicate the time necessary to achieve that goal, while still maintaining a social life and keeping up with my other academic responsibilities. When I realised this, and following some excellent advice from my supervisor, I chose building an accelerator for the most computationally expensive part of the DEFLATE algorithm pair as a "fallback" goal.

I have reached that goal, as previously described, and I am proud of the work done so far. Due to the global pandemic occurring at time of writing, it seems as though I will have significantly more idle time following my end of semester examinations than I had expected. I am considering spending some of that time taking the project as it currently stands and converting it into a set of tutorial documents, which I would publish online in some blog format. A resource like that would have been very beneficial to me when I was starting this project, and I think creating it would *feel* like doing something genuinely useful. That feeling of doing meaningful work is something I find important, more important often than the technology or problem being solved itself, and I expect many other engineers do too.

That being said, I do not view this project as finished, for any useful definition of the word[3]. There is still work to be done; I am unhappy with both the performance gained and limited benchmarking conducted thus far. The source files for the project have been released under an open source license[14], and I intend to continue to make improvements to the hardware and software drivers of the accelerator.

I do not expect this hardware to be useful in the broader world, although perhaps there is someone out there developing an embedded system which needs to compress data extremely quickly. If that's you, and you're able to share what it is you're building, please tell me, I'd love to know. I could paint a picture of some long-term deployment sensor system which wakes from sleep every second to sample something and every thousand samples compresses the previous thousand samples in order to save on storage space, which needs to compress that data as fast as possible in order to spend as long as it can in a low power sleep mode, but I find that use case unlikely. I would like to believe that this report, and the tutorial documents

[2]Both the original proposal, for a discrete fourier transform accelerator, and the final proposal, which is the title of this project.

[3]The project is "finished" insofar as this report is submitted and will be used as part of an academic assessment.

should I write them, *would* be useful to those wishing to learn how to build systems like this, but it would be dishonest to say I am confident they would be.

I suppose that's where I've ended up; I am pleased with the quality of what I have built but dissatisfied with it because I feel it is unfinished, and I am unconvinced of its direct utility and uncertain but hopeful of its indirect utility. I'll also so I am *extremely* pleased with how much I have learned while working on this project. I am now comfortable working with and debugging heterogeneous FPGA based systems, at least Xilinx flavoured ones running standalone applications. From the "vantage point" of the end of this report, I can see multiple areas of the heterogeneous systems field I would like to explore; embedded linux based systems, AXI-Stream and Full, and more complex hardware accelerator cores, to name a few. I am satisfied that I have fulfilled my original desire to spend time "playing" with the technology I found so interesting at the start of the project.

6.2.1 A Minor Rant

Please skip this subsection unless you would like to know why the ethics chapter of this report is, well, lacklustre at best.

I find the mandatory chapter on ethics in this report distasteful. Engineers are not ethicists, and my training thusfar has not equipped me to write about the ethical implications of a fairly banal optimization project on an algorithm from the 70s. I would remove it completely. I find the assessment framework used for that chapter encourages me to engage in dishonest writing, in which I place undue importance on tiny effects, e.g.:

> There is an environmental benefit to be considered due to the decreased number of clock cycles required to complete the data compression. Most modern FPGAs can be put into a deep sleep mode, in order to conserve energy usage when not doing computation, and the ability to complete a task faster allows a system to return to its low power mode sooner, thereby saving energy (so called "race to sleep").

The above is technically true, but practically dishonest, as the actual low-power system would not use an FPGA at all, due to their power consumption being significantly higher than dedicated hardware[4]. Additionally, the power saved by improved data compression performance could only be meaningful if the solution I developed was deployed at a *massive* scale, and the companies with the scale necessary prefer commodity hardware for economic reasons.

The honest answer to the question "What are the ethical implications of this work?", when asked to an undergraduate writing a report on a project to design hardware to implement an algorithm from the 70s, is "There are none.". Very few people will read this report, fewer still

[4]Due to all the increased transistors required to implement the hardware flexibility.

will read the source files for the hardware and software developed during the project. I would be willing to bet my degree that nobody will build anything incorporating them. Having a required ethics section is window dressing, and it undermines the importance of ethical reasoning in the field.

Bibliography

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, May 1977.

[2] "LZ77 and LZ78." `https://en.wikipedia.org/wiki/LZ77_and_LZ78`. Accessed: 2020-03-30.

[3] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[4] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, RFC Editor, May 1996.

[5] L. P. Deutsch, "GZIP file format specification version 4.3," RFC 1952, RFC Editor, May 1996.

[6] Xilinx, Inc., *UG1085: Zynq UltraScale+ Device Technical Reference Manual*, August 21, 2019.

[7] ARM, *AMBA® AXI™ and ACE™ Protocol Specification*, October 28, 2011.

[8] J. Aldrich, "Correlations genuine and spurious in pearson and yule," *Statistical Science*, vol. 10, no. 4, pp. 364–376, 1995.

[9] D. Gisselquist, "Building a custom yet functional axi-lite slave." `https://zipcpu.com/blog/2019/01/12/demoaxilite.html`. Accessed: 2020-03-30.

[10] N. Z. Milenković, V. V. Stanković, and M. L. Milić, "Modular design of fast leading zeros counting circuit," *ELECTRICAL ENGINEERING*, vol. 66, no. 6, pp. 329–333, 2015.

[11] Xilinx, Inc., *AXI Protocol Checker v2.0*, April 4, 2018.

[12] L. Fitzsimons, "Ethics in Engineering Presentation." Available Upon Request.

[13] "Display Screen Equipment - (DSE/VDU) - Frequently Asked Questions." `https://www.hsa.ie/eng/Workplace_Health/Manual_Handling_Display_Screen_Equipment/FAQs/Display_Screen_Equipment_FAQs/Display_Screen_Equipment1.html`. Accessed: 2020-03-30.

[14] "Pop FPGA GitHub Repository." `https://github.com/oliverb123/pop_fpga`.

Appendix A

Digressions and side notes

A.1 LZSS Comparisons Required for a Given N

The listing A.1 is a short python script which was used to generate table A.1, for varying values of N. As can be seen in figure A.1, the relationship between N and the number of comparisons required is exponential.

Listing A.1: LZSS Comparison Count Calculator

```
M = 4096
K = 15
N = 10

comp_count = 0
push_count = 0

for i in range(N):
        comp_count += K*push_count
        push_count += 1 if push_count < M else 0

print(comp_count)
```

Table A.1: LZSS *N* vs Comparison Count

N	**Comparisons Required**
1	0
2	15
3	45
4	90
5	150
6	225
7	315
8	420
9	540
10	675

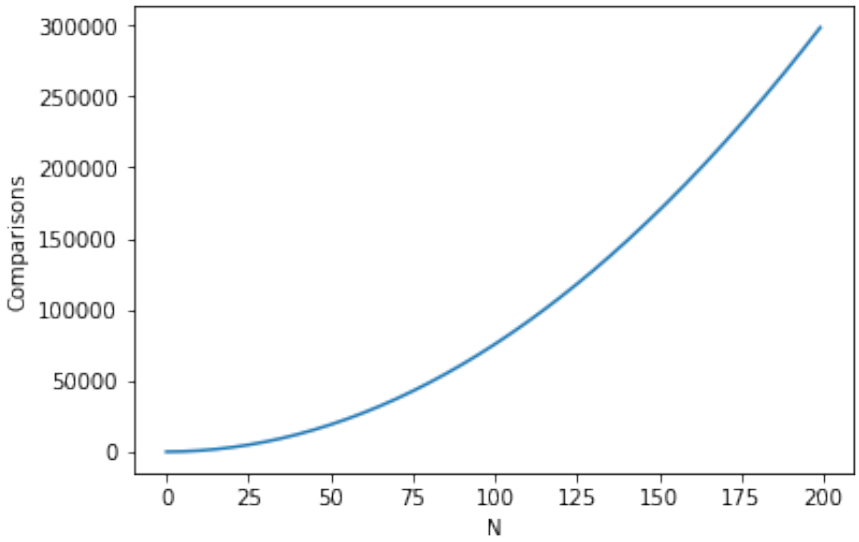

Figure A.1: LZSS *N* vs Comparison Count

A.2 Verilog Syntactic Sugar

Please see listing A.2 for examples and explanations of Verilog `for` loops, array slicing and ternary operators.

Listing A.2: Verilog Syntactic Sugar Examples

```
//For loops are a shorthand way of writing normal assignment expressions
//They may be placed in generate blocks or always blocks. Generally
//generate blocks are used for purely combinatorial logic, and always
//blocks for clocked logic.
wire [3:0] A, B;

generate
    genvar i;//a genvar is a variable which can only be used in a generate
```

```
        block.
    for(i = 0; i < 4; i++) begin
//Note the use of assign as this is a purely combinatorial statement
    assign B[i] = A[i];
    end
endgenerate
//The previous generate block is directly equivalent to this:
assign B[0] = A[0];
assign B[1] = A[1];
assign B[2] = A[2];
assign B[3] = A[3];
//For loops may be considered a way to "roll" up a series of assignments

//A ternary operator is shorthand for an if-else statement. This:
output wire A;
input wire B, C, D;
assign A = B ? C : D;
//Is equivalent to this:
always_comb begin
    if(B) begin
        A = C;
    end else begin
        A = D;
    end
end
//Note that the above required an "alwyas_comb" block to allow the use of
//if/else, which can normally only be used in an always block

//An array slice is shorthand for assigning single wires within an array
//to something. For example:

input wire [7:0] A;
input wire nibble_select;
output wire [3:0] B;

//This:
assign B[0] = nibble_select ? A[4] : A[0];
assign B[1] = nibble_select ? A[5] : A[1];
assign B[2] = nibble_select ? A[6] : A[2];
assign B[3] = nibble_select ? A[7] : A[3];

//is equivalent to this:
```

```
                                                                    50
assign B = nibble_select ? A[7:4] : A[3:0];                         51
```

A.3 Project Source Files

Please note that the source files for the hardware and software developed during the course of this project are distributed under the MIT open source license. See citation [14].

Glossary

For the sake of brevity, please note that the following applies to all sections of the report, unless otherwise stated:

- All discussions of *data* and *signals* assumes a binary representation
- All references to *integers* refers to unsigned 32 bit integers (`uint32_t` in C parlance)
- All references to *characters* or ”*chars*” refers to an 8 bit byte
- *Device* means the silicon chip in use, for this project a ”Zynq UltraScale+”, part number `xczu3eg-sbva484-1-e`
- *Board* refers to the ”Ultra96v2” development board
- All work was done using the Vivado design suite, version 2019.1, WebPack edition
- All programming of the board was done using a ThinkPad T430, running Kubuntu 19.10
- For the purposes of this report, discussion of FPGA fundamentals will be in line with Xilinx FPGA technology, including terminology such as ”Configurable Logic Blocks (CLBS’s)” and ”Basic Elements of Logic (BEL’s)”
- Numbers prefixed with ”0x” are present in hexadecimal format, those presented in ”0b” are in binary format.
- PS $\rightarrow$ Processor subsystem
- PL $\rightarrow$ Programmable logic