Master of Science Thesis in Computer Engineering
Department of Electrical Engineering, Linköping University, 2016

Time-Multiplexed Channel Switches for Dynamic Frequency Band Reallocation

Roland Stenholm

LINKÖPING
UNIVERSITY

Master of Science Thesis in Computer Engineering

Time-Multiplexed Channel Switches for Dynamic Frequency Band Reallocation

Roland Stenholm

LiTH-ISY-EX--16/4949--SE

Supervisor: **Oscar Gustafsson**
ISY, Linköpings universitet

Examiner: **Kent Palmkvist**
ISY, Linköpings universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Sammanfattning

En delvis parallel och delvis seriell kanalswitch för användning inom DFBR skapas. Dess permutation kan ändras medan den kör utan avbrott i dataströmmen. Tre alternativ undersöks: ett baserat ett sorteringsnätverk, ett baserat på minnen och multiplexrar och ett som baseras på Clos-nätverk. Versioner med mönsterdata sparad i skiftregister och i minnen prövas. De implementeras i automatiskt genererad Verilog och synthesiseras för en FPGA. Deras kostnad i areaanvändning, minnesanvändning och maximal klockfrekvens jämförs. Resultaten visar i princip att Clos-nätverken är bäst i alla avseenden och att mönsterdata ska sparas i RAM-minnen och inte i skiftregister. Arbetet är open source och kan laddas ner från `https://github.com/channelswitch/channelswitch`.

Abstract

A partially parallel reconfigurable channel switch is constructed for use in DFBR. Its permutation can be changed while running without any interruption in the streams of data. Three approaches are tried: one based on a sorting network, one based on memories and multiplexers and one based on an Clos network. Variants with the pattern stored in memories and in shift registers are tried. They are implemented in automatically generated Verilog and synthesized for an FPGA. Their cost in terms of area use, memory use and maximum clock frequency is compared and the results show that the Clos based approach is superior in all aspects and that pattern data should not be saved in shift registers. The work is open source and available for download at `https://github.com/channelswitch/channelswitch`.

Acknowledgments

I would like to thank my family for being supportive and my advisor for all of his help during these last few months.

Linköping,
Roland Stenholm

Contents

Notation

Abbreviations

Abbreviation	Meaning
FPGA	Field Programmable Gate Array
DFBR	Dynamic Frequency Band Reallocation
TST	Time Space Time
STS	Space Time Space
VPI	Verilog Procedural interface
ASIC	Application Specific Integrated Circuit
FFT	Fast Fourier Transform
RAM	Random Access Memory

Variables

Variable	Meaning
I	The number of inputs a switch takes in one clock cycle.
P	The number of clock cycles in one period
N	Total number of channels ($N = IP$)
D	Sample width in bits

1 Introduction

This chapter describes the problem and why it is interesting.

1.1 Motivation

This work explores the problem of partially parallel reordering of streams of data in digital electronics. Samples arrive in frames. Each sample in a frame corresponds to one channel. The interleaver reorders the samples within a frame arbitrarily, creating a new frame of the same size. The permutation of streams is defined by the user and can be changed while the hardware is running. Each sample in the input will appear exactly once in the output but there are no limitations on the order in which they appear.

The hardware runs at a higher clock frequency than the frequency at which frames arrive. There are several clock cycles between one frame and the next. There are not as many as there are samples in a frame, however. If there had been, the problem could have been solved simply by writing samples into a RAM and then reading them in the user-specified order. Instead, the hardware takes a few samples per clock cycle as input over a period of a few hundred clock cycles. How to accomplish this in an efficient manner is less obvious, and is the topic of this report.

The interleaver must be able to operate continuously. For example, there shall not be any clock cycles between frames where the interleaver does not take inputs. The interleaver must also support switching of permutations without pausing, but the time it takes is not important. There are no requirements on latency either. It will be at least the same time it takes to input a frame, as a delay shorter than that would be impossible if the permutation requires that a sample in the last clock cycle be output in the first.

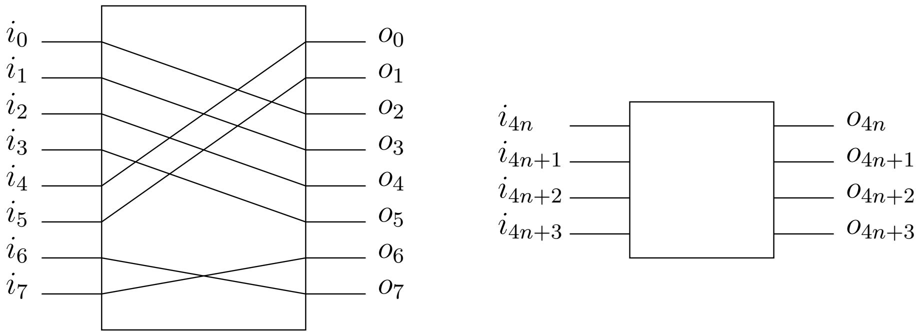

Figure 1.1: *Left: a purely parallel switch with 8 inputs and an example configuration. Right: a serial switch implementing the same function but over two clock cycles. How the switch on the right would implement this permutation is not shown.*

Figure 1.1 shows a black box model of a switch where $I = 8$ and $P = 1$ and one where $I = 4$ and $P = 2$. The same number of total samples is rearranged but the box on the right does it in two clock cycles. The index n is the number of clock cycles since the start of the period. This is either 0 or 1 since $P = 2$. That means that during the first clock cycle, i_0 to i_3 are read and during the second clock cycle, i_4 to i_7 are read.

The exact number of parallel inputs that are needed will depend on the clock frequency that the chip will run at. The maximum clock frequency is limited to the lowest of the maximum frequency set by any part of the chip. This can be either the interleaver or by another part of the hardware such as an FFT. The implementor must choose parameters that minimize the cost of the chip while still having the required throughput. For this to be possible, the maximum clock frequency of the chip as a function of its parameters must be known.

To find out the maximum clock frequency of the chip, the maximum frequency of all of its components must be known. Therefore, a model for the limit on maximum clock frequency imposed on the chip by the interleaver, depending on its parameters, must be developed. This will be done by synthesizing interleavers with different parameters and saving their timing information.

1.2 Purpose

DFBR, Digital Frequency Band Reallocation, is the practice of changing the frequency bands used by signals in commuication hardware dynamically in response to demand[4]. A DFBR capable satellite can have many input signals, each with its own bandwidth depending on the standard it uses. The signals will then be output at potentially different frequency bands chosen by the operator or automatically by the hardware, in the case of Cognitive Radio. DFBR also finds applications in digital TV distribution in cable networks[5].

In broad terms, DFBR works by splitting its full bandwidth into many smaller bands. Their bandwidth is chosen so that all expected input signals will occupy roughly an integer multiple of these bands. The input signal is split into its components. The samples from each band are rearranged. This is the function performed by the Channel Switch. They are then reassembled into the output signal. This can be done in such a way that signals which occupy more than one small band will not lose information. The theory behind it is described in [6].

Another reason that a satellite may have many input or output streams is the use of high-gain spot beam antennas. These only transmit or receive in one direction. A satellite can use many such antennas, with each one covering a smaller area of the earth. That way the satellite can distinguish signals in the same frequency band if the transmitters are located in different beams and can send signals only to receivers in specific areas. This means that each antenna has its own signal and in combination with DFBR, a very large number of channels need to be switched.

1.3 Background

The naive approach to the problem is to write the samples into a big RAM and then read them out in random order. If n samples arrive each clock cycle, then an n-port RAM is required. Any more than two ports is impractical. With one RAM per input, there would be enough bandwidth for all of the data, but more logic is required to put each sample into the right memory. It is far from obvious how to make this work. To get a better understanding of the problem, other solutions will be examined.

Sorting networks consist of compare-and swap elements and wires. They are usually drawn with all wires horizontal and swappers connecting two wires vertically. In such diagrams data arrives at the left, and wherever two lines are connected the values are swapped so that the larger one appears at the top. A valid sorting network can perform the task of rearranging its inputs in any arbitrary order. If a sorting network could be split across several clock cycles then it could be used to solve this problem. Unlike in a sorting network there is no need to actually perform comparisons every clock cycle, which could lead to simplifications.

Clos networks are used in telephone switching. A Clos network is a construct that creates the equivalent of a crossbar switch of a certain size using only smaller crossbar switches. They were first described by Charles Clos in 1952 [2]. Clos networks are not time-multiplexed, but there is a construct called a TST (Time Space Time) switch which is. Clos networks can also be made recursively to decrease complexity even more. These are called Benes networks.

Digital communication protocols such as Wifi use something called interleaving. Samples or bits of data are reordered within a frame before they are sent onto the physical link, and at the receiver they are reordered back again. The reason for this is that redundant codes are used for error correction. These codes do not work well when an error affects several consecutive bits, but that's the form real world errors usually take. The problem of interleaving is especially interesting because a lot has been written specifically about parallel interleaving, and things like clock cycles and power use are actually discussed.

1.4 Questions

Can sorting networks be adapted to this problem? Can Clos networks be used? What is the cost of these approaches in terms of area and maximum frequency when implemented on an FPGA?

1.5 Not Considered

This work concerns cases where the number of parallel inputs (I) is much smaller than the total number of samples (N) in a frame. ASIC implementations are not attempted due to the lack of a memory compiler. Latency is assumed to not be a problem.

2 Parallel Memories

This chapter describes the implementation of a channel switch based on memories and multiplexers. The number of inputs and the period length are variables. They are denoted I and P, respectively. The same notation is used in the chapters describing the other implementations.

2.1 Theory

This implementation is based on memories. Each output is given a memory. Each incoming sample is written into the memory of the output it is going to. Memories are written in sequential order and read out in random order.

The problem occurs when multiple incoming samples from the same clock cycle are going to the same output but in different clock cycles. Then, multiple samples must be written to one memory. The parallel memories implementation

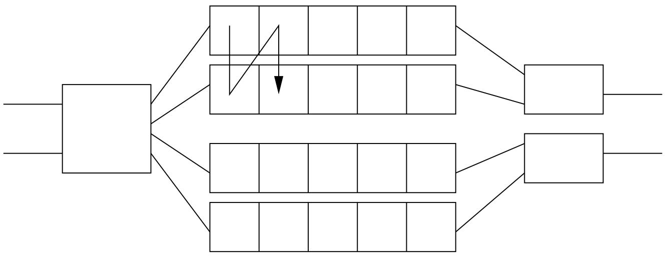

Figure 2.1: *A memory implementation with $I = 2$ and $P = 10$. The arrow shows the order in which the first incoming samples to be written to the upper output get written to the memories.*

solves this by simply splitting each memory into several smaller ones, as many as there are inputs. This lets multiple samples be written to the same output in one clock cycle. Each memory only has a size of P/I so the total memory capacity is $I^2 \times P/I = IP = N$ samples regardless of what P is. This is shown in figure 2.1.

In figure 2.1, there are three boxes. They represent the multiplexers and logic that routes samples in and out of memories. At the input, any input sample must be able to go to any memory. Therefore, there are I^2 multiplexers – one per memory – with I inputs each, one for each input to the switch. At the output, it is known that all samples destined for each output wire are in the memories corresponding to that output. This means that there is no need for multiplexers to be able to read from other memories. The boxes at the outputs represent the output multiplexers, and are separate to illustrate this.

Should the number of inputs be larger than the period, there do not need to be I memories per output but only P. In this case, there will at most be P samples going to one output in a single clock cycle and the reason for this is that there are exactly P samples going to one output over the whole period. This case is not investigated in detail in this chapter but it has been implemented properly.

2.2 Implementation

The parallel memories switch has been implemented as described. At each output is a multiplexer choosing between the memories corresponding to that output. That multiplexer is controlled by a value in the pattern. The read address of the memories are also controlled by the pattern.

At the input side, each of the I^2 memories has a multiplexer allowing it to be written with any of the inputs. These multiplexers are controlled by the pattern. Whether to write to that memory during a particular clock cycle is controlled by a bit in the pattern as well. The write address comes from a per-memory counter. These multiplexers quickly eat resources and decrease the maximum clock frequency when the number of inputs increases, but no better solution was readily apparent.

To enable continuous operation, there are two memory banks. When one is being written to, the other is being read. The control word stores the read address for each output, one bit for each of the I^2 memories telling us if it is to be written to this clock cycle and if so, from which input.

There is also a buffer on the output to deal with ready signals which go away at inopportune times. The buffer is implemented with registers and stores 7 clock cycles worth of samples. That means registers holding 7 samples for each input. The number 7 was chosen because the largest number of samples in the buffer seemed to be 5 in simulation, plus a safety margin of 2.

In the initial version of this implementation the control word was stored in two shift registers. This turned out to be bad in terms of chip area and synthesis time. The second version uses two RAM memories. Two are needed so that a new pattern can be used without interrupting the data stream.

2.3 Complexity

The switch will have $I \times \min(I, P)$ memories in total. It is assumed that P is greater than I, so the number of memories is I^2. Each of them has a multiplexer, giving I^2 multiplexers, each with I inputs. There is also one multiplexer per output. They have $\min(I, P)$ inputs, but it is assumed that $P > I$. In total, that makes

$$n_{multiplexers} = I^2 + I$$

multiplexers, each with I inputs.

There are $2I^2$ memories with a total memory capacity of $2I \times P$ samples. Per memory, that is P/I samples, with each sample having a width specified directly by the user with the −−datatype parameter. This does not take pattern memories into account.

There are two pattern memories. They store P control words each. A control word contains select bits for each of the I^2 multiplexers. One multiplexer with I inputs has $\log_2(I)$ select bits, giving

$$n_{inputselectbits} = I^2 \log_2(I)$$

select bits for the input multiplexers. Another

$$n_{outputselectbits} = I \log_2(I)$$

select bits come from the output multiplexers. Each input multiplexer needs one more bit to tell it whether it is to write anything during this clock cycle, adding

$$n_{writebits} = I^2$$

write enable bits. The read address for each output has a width of $\log_2(P)$. This adds another

$$n_{addrbits} = I \log_2(P)$$

read address bits to each control word. In total, each of the $2P$ control word consists of

$$n_{controlbits} = I^2 + \left(I^2 + I\right) \log_2(I) + I \log_2(P)$$

bits.

Expressed in terms of N, the number of multiplexers is

$$n_{multiplexers} = \left(\frac{N}{P}\right)^2 + \frac{N}{P},$$

the data memories hold

$$n_{samples} = 2N$$

samples and each control word consists of

$$n_{controlbits} = \frac{N^2}{P^2} \left(1 + \left(1 + \frac{P}{N}\right) \log_2(N) - \log_2(P)\right)$$

bits.

3 Bitonic Sorter

This chapter describes an implementation of the switch based on a sorting network.

3.1 Theory

A sorting network is a construct that sorts its inputs. It is constructed from compare and swap elements. Each one compares its two inpts and swaps them, depending on the result of the comparison.

Figure 3.1 shows a small sorting network. Inputs arrive from the left, and come out sorted on the right. Each arrow represents a compare and swap element between the two points that it connects. The smaller value goes in the direction that the arrow points in and the larger value in the other direction. In the sorting network from the figure, no matter which numbers are input, the output will always be sorted with the smallest number at the topmost wire and the largest one at the bottom.

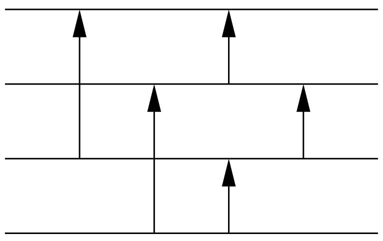

Figure 3.1: *A sorting network with 4 inputs.*

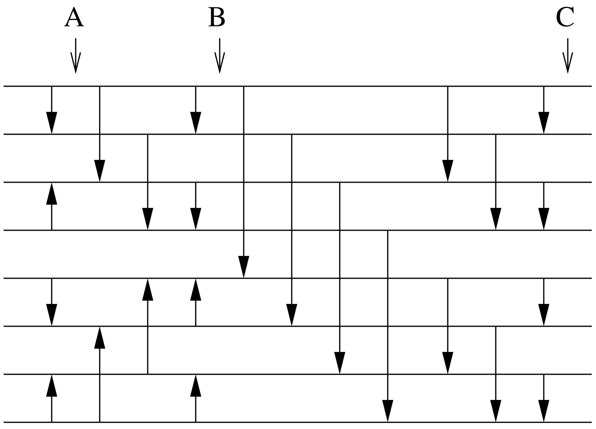

Figure 3.2: *A bitonic sorting network with 8 inputs.*

A sorting network therefore has the ability to reorder its inputs arbitrarily. If a sorting network could be efficiently modified to work over several clock cycles then it could be used for arbitrary interleaving.

The Bitonic sorting network is the sorting network equivalent of the merge sort algorithm. It consists of multiple stages, where each merges pairs of sorted intervals into sorted intervals with twice the length. The Bitonic sorting network in figure 3.2 has arrows marking the points between the stages.

At the input, the list is completely unsorted. The first stage merges the imaginary sorted intervals of length one into sorted lists of length two. At point A, the first two wires are sorted internally, the second two rows are sorted, and so on. Between A and B is the second stage. It merges the four two-length sorted lists into two four-length sorted lists. After B is the third stage. It merges the two four-length sorted lists into one eigth-length list, thereby completing the sorting.

Each stage contains comparators that cross twice the number of wires in the previous stage, then contains a copy of the previous stage. The diagram contains arrows dictating where the larger value should go. It is possible to construct a bitonic sorting network with all arrows pointing in the same direction, but that doesn't lend itself to serialization as well.

3.2 Implementation

The key insights that allow the bitonic sorting network to be easily serialized are as follows: The length of each arrow is a power of two, and that all arrows with a length of N are repeated for every multiple of $2N$ wires (so if an arrow connects wires 0 and 2, another will connect 4 and 6, for example).

Now, say we want to modify a bitonic sorting network with 8 inputs so that it runs over two clock cycles, and takes 4 inputs each clock cycle. Figure 3.2 shows the starting point. The upper half is executed during the first clock cycle and

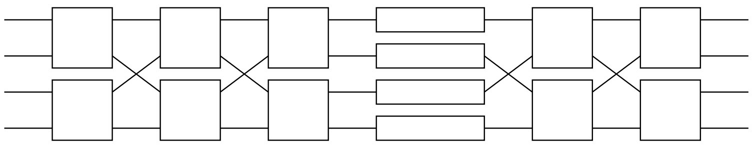

Figure 3.3: *This shows how the 8-input bitonic sorter in figure 3.2 would be converted to serial operation with 4 parallel inputs over a period of 2 clock cycles.*

the lower half would be executed during the second. Arrows with a length of 2 or less are duplicated across both clock cycles but only need to be implemented once, saving resources. Sometimes they want to sort in different directions, but this is taken into account when preparing patterns.

Consider the four arrows that are 4 wires long. They may need to swap a value that arrives in the first clock cycle with one that arrives the next. Obviously, this cannot be done without waiting for the next value to arrive. These swappers are thus implemented using memories. During the first clock cycle, they save the incoming data in the memory. During the second, they output the incoming data immediately and let the previously stored value remain in memory if they are supposed to swap the two values. If not, they output the previously saved value while saving the incoming data. During the last clock cycle, they write the value that was left in the memory. Everything that comes after a delay element must have its pattern delayed by the same number of clock cycles, of course.

Because all arrows in a bitonic sorter have lengths that are powers of two, any arrow longer than the number of parallel inputs (which must be a power of two), only connects to itself during different clock cycles. Their serialized realizations therefore only occupy one wire.

The bitonic sorter has been implemented in automatically generated Verilog. The first version simply stored the output address in registers and sent each sample along with its destination address through the sorter. This means that is contained many actual comparators. It was horrible in terms of resource use, and with large values for inputs and period, could take days to synthesize. The second version uses a control word, stored in a RAM, with one bit for each swapper.

The swap elements do not use any memories. Some of the delay elements use memories, but for really small memories the synthesizer chooses to use registers instead.

There are registers after each stage of swappers. This makes the Bitonic sorter entirely pipelined, which should increase the maximum clock frequency at which it can operate. The Bitonic sorter can be quite deep with large parameters, so this is neccesary.

3.3 Complexity

Now to figure out the complexity in number of swappers and number and sizes of memories. At the input end of the sorter, there are succesively larger stages without any delay elements whatsoever up to the point where the stage merges lists with the same length as then number of parallel inputs. Together, these contribute the same number of swappers as a standard Bitonic sorting network with I inputs, which is

$$n_{swappers1} = \frac{1}{2} I(\log_2^2(I) + \log_2(I))$$

of them.

Each subsequent stage consists of delay elements followed by a merge stage of size I. Such a merge stage has

$$n_{swappers2} = I(\log_2(I) + 1)$$

compare and swap elements, and there are $\log_2(P)$ of them.

The total number of swappers is the sum of the two:

$$n_{swappers} = I\left(\frac{1}{2}\left(\log_2^2(I) + \log_2(I)\right) + \log_2(P)(\log_2(I) + 1)\right)$$

The number of samples of memory capacity is I times

$$M(n) = \begin{cases} 0 & n = 0 \\ M(n-1) + 2^n - 1 & n \geq 1 \end{cases}$$

$$\Rightarrow$$

$$M(n) = 2^{n+1} - n - 2,$$

where n is the number of merging stages with at least one delay in them and is equal to $\log_2(P)$. This makes the total data memory capacity

$$n_{samples} = I(2P - \log_2(P) - 2)$$

samples split across

$$\frac{1}{2} \log_2(P)(\log_2(P) + 1)$$

memories.

The control word contains one bit for each swapper and one for each delayer, which is

$$n_{controlbits} = I \log_2(I)^2 + I \log_2(P) \log_2(I) + \frac{1}{2} \log_2(P) \left(\log_2(P) + 2I + 1\right)$$

bits. The length of the two memories is P as always.

Expressed in terms of N and P, the bitonic sorter has

$$n_{swappers} = \frac{N}{P}\left(\log_2\left(\frac{2}{N}\right)\log_2(P) + \log_2^2(N)\right)$$

2×2 switches,

$$n_{samples} = 2N - \frac{N}{P}\left(\log_2(P) - 2\right)$$

total samples of data memory capacity and 2 N-word pattern memories where each word is

$$n_{controlbits} = \frac{N}{P}\log_2^2(N) - \frac{N}{P}\log_2(N)\log_2(P) + \frac{1}{2}\log_2^2(P) + \left(\frac{N}{P} + \frac{1}{2}\right)\log_2(P)$$

bits wide.

4

Clos Network

4.1 Theory

The basic switching element in telephony is the crossbar switch. It can connect its inputs with its outputs in any configuration. The electrical conductors are arranged in a grid, where the lines in one direction are the inputs and the other one are outputs. At each intersection, an individually controllable switch can connect that input with that output. The crossbar switch has a complexity of $\mathcal{O}(n^2)$ where n in the number of inputs or outputs assuming they are the same.

A Clos network is a three stage switching network performing the same function as a crossbar switch but using fewer resources. Depending on its parameters, it can be capable of connecting any inputs to any outputs with varying difficulty. It is constructed from smaller crossbar switches. In the input and output stages, there are r crossbar switches, each with m connections on the side facing the middle stage and n connections on the other side. The middle stage consists of m $r \times r$ switches. Each middle switch has exactly one connection to each of the first stage

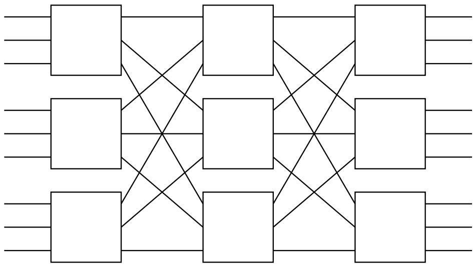

Figure 4.1: *A Clos network with $m = n = r = 3$.*

and last stage switches. Figure 4.1 shows a Clos network with all parameters equal to 3.

Charles Clos proved that when $m \geq 2n - 1$ there will always be a free path through a middle switch to make any connection immediately. Such networks are called strictly nonblocking. If $n = m$, the network is rearrangably nonblocking. Any set of connections can be realized, but to make a connection, existing connections may need to be routed through other middle switches first. Since this work is concerned with an application where patterns change rarely and all at once, a rearrangably nonblocking Clos network is used.

4.2 Routing

To make a connection in a rearrangably nonblocking Clos network, see if there is a middle switch whose connections to both the left crossbar switch of interest and the one to the right is free. If the answer is yes, the connection is trivially routed through that switch. This will always be the case if the network is strictly nonblocking. If the answer is no, then find one middle switch with a free connection to the left stage crossbar, and one with a free connection to the right one. This is always possible if $m \geq n$. Now, the task is to rearrange the connections between these two switches such that all of them can be made.

Disconnect everything through these two middle switches and remember the set of connections that were disconnected and the one that we want to make. Choose any connection. Connect it through any one of the two middle switches. Find the other connection that goes through the same right crossbar. There are always two or less since there are two middle switches and each outer stage switch has exactly one connection to each middle switch. Route that through the other middle switch. Now, find the other connection through the same left crossbar switch as the last connection and route that through the first middle crossbar again. This process is repeated. Every time a connection is made from the left to the right, the connection is made through the first middle crossbar and every time a connection is made in the other direction, the other middle crossbar.

The connections may form several disjoint cycles or sequences, so this process shall be repeated until all connections have been made.

4.3 Space-Time-Space Switch

Consider the case where m equals the number of parallel inputs to the interleaver, I and r equals the number of clock cycles in a period, P. During the first clock cycle, all of the inputs to the first crossbar switch arrive at the input wires. The crossbar switch would output one value to each of the middle switches. During the next clock cycle, the inputs to the next crossbar switch would arrive, and again it would send one sample to each of the middle switches. It is apparent that all that is needed is one crossbar switch at the input, which performs a different permutation each clock cycle. The same goes for the output. The middle

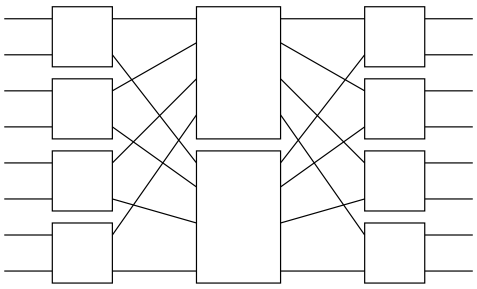

Figure 4.2: *How a Benes network is constructed.*

switches receive one sample per clock cycle and output one sample per clock cycle to the output stage. They are easily constructed using simple RAM memories that are written to sequentially and read in a random order. This approach is also decribed in [7].

The input and output stage crossbars are known as space switches because they only make connections in space – they never deal with memories or delays. The middle stages do not perform any routing of inputs to outputs and only reorder the samples that they receive. They are thus known as time switches. As a result, this arrangement is known as a Space Time Space, or STS switch.

4.4 Arbitrary Size Benes Networks

The input and output crossbar switches can be simplified further by using a Benes network. It is essentially a recursive Clos network where only 2×2 switches are used. Benes networks can only be constructed when the number of inputs is a power of two but this limitation can be overcome.

The paper Arbitrary Size Benes Networks[1] describes an approach to creating Benes networks with any number of inputs. When turning a crossbar switch with an even number of inputs to a Benes network, two middle switches, each with half that many inputs, are created. Every pair of two inputs and outputs are given one 2×2 switch, with one connection to each middle switch. The two middle switches are then turned into a Benes network recursively. This is the same as a regular Benes network. One step of this process is shown in figure 4.2.

The difference comes when turning a network with an odd number of inputs into a clos network. The bottom middle switch is made one input larger than the top one, and the extraneous input is connected directly to the bottom switch. Figure 4.3 shows this.

The process of routing a Benes network amounts to the same as routing any rearrangably nonblocking Clos network, but done recursively in the same manner that the network is constructed. Routing an arbitrarily sized Benes network is the same except that a connection which goes through the last input – the one without a 2×2 switch – obviously cannot go through the upper middle switch

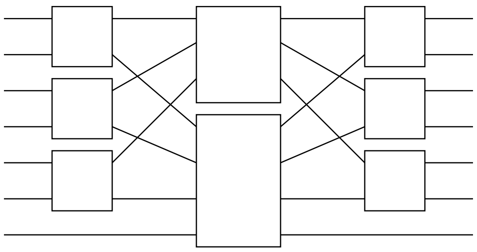

Figure 4.3: *One step in the construction of an Arbitrary Size Benes Network.*

but must go through the lower one instead. Any cycle or sequence which includes this connection must be routed so that this is possible.

4.5 Implementation

The interleaver was implemented as described. Two arbitrarily-sized Benes networks are constructed. One at the input and one at the output. They are pipelined, so that they will not limit the maximum clock frequency no matter their size. In pipelining them, delay elements must be inserted in paths that do not have crossbar switches were other paths do. Each 2×2 switch is controlled by one bit in the control word. The control word is individually set for each clock cycle in a period and is stored in a memory.

The middle switches were implemented from two single port memories each. When one is being written to, the other one is being read. When the first one is full, they switch. A delay of one frame is caused by this. The read address for each TSI is part of the control word.

Routing is done in the generate_pattern_data() function. The delay of each stage as a result of pipelining and the memories is compensated for by delaying the control bits for later stages by the right number of clock cycles.

A previous version of the Clos network based interleaver used a shift register for its pattern, but this used up a lot of resources.

An attempt to write this interleaver without using code generation was done. The ports were generated but everything else was done using for loops, functions and tasks.

Functions in Verilog 2001 do not have automatic arguments. They are essentially global variables. Benes networks are recursive, but a way to implement these without using recursion was found. The implementation did run in the simulator, but the synthesizer refused to synthesize it seemingly because it was unable to recognize that some function calls were in fact constant.

The attempt was abandoned and the Clos switch was implemented like all the other ones.

4.6 Complexity

The number of 2×2 switches in an arbitrarily sized Benes network with n inputs is found using the following recursive sequence:

$$S(n) = \begin{cases} 0 & n \leq 1 \\ 1 & n = 2 \\ 2\left\lfloor \frac{n}{2} \right\rfloor + S\left(\left\lceil \frac{n}{2} \right\rceil\right) + S\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & n \geq 3 \end{cases}.$$

This is roughly in the neighbourhood of

$$n \log_2(n) - \frac{n}{2},$$

which is the size of a Benes network when the number of inputs is a power of two. There are two Benes networks in one switch, each with I inputs.

There are exactly $2I$ data memories, each with a capacity of P samples. The control word contains one bit for each 2×2 switch in the Benes networks and read addresses for each of the I data memories that are being read. Those read addresses have

$$\lceil \log_2(P) \rceil$$

bits each, making approximately

$$n_{controlbits} = 2I \log_2(I) - \frac{I}{2} + I \log_2(P)$$

bits in a control word.

In terms of N and P the number of 2×2 switches is approximately

$$n_{switches} = 2N \log_2(N) - N,$$

the data memory capacity

$$n_{samples} = 2N$$

samples and the control word width

$$n_{controlbits} = \frac{N}{P}\left(2 \log_2(N) - \log_2 P - \frac{1}{2}\right)$$

bits.

5 Implementation

This chapter explains how the problem was solved in general terms without diving into any particular algorithm. It explains what programming languages were used, which HDL and the interface used by the hardware, since it is the same for all implementations.

5.1 Code Generation

The project takes the form of a command line program that generates C and Verilog files for an interleaver depending on its arguments. One argument, −−type, selects which type of interleaver to create. There are six types, because the three approaches were implemented in two different ways each. The argument −−inputs selects the number of parallel inputs (I). −−period selects the number of clock cycles in a period (P). The number given to −−data is the width of each sample in bits (D). Only powers of two can be used for −−inputs and −−period if the type is based on a bitonic sorting network.

The command creates five files: switch.v is the interleaver in Verilog. The C file generate_pattern_data.c and its header contain the C code to realize a particular permutation. The output will then be sent to the hardware's pattern_data input. The testbench is in testbench.v and it needs vpi.c so that it can call generate_pattern_data(). See your simulator's manual for how to use VPI.

The code has to be generated by a program because of limitations in Verilog 2001. There is a variable number of data inputs and outputs. The obvious way to implement this would be to use two-dimensional arrays with D in the packed dimension and I in the unpacked dimension. Unfortunately, Verilog 2001 doesn't support multidimensional ports, so several differently named input and output ports are automatically created instead.

An attempt to create an implementation that used functions and for loops instead of code generation ran into several other problems. Recursive functions are difficult, because their arguments are static by default. This was overcome but even though the code could eventually be simulated, it could not be synthesized. The synthesizer was unable to figure out some static function calls within loops and while it ran in the simulator, it was slower than the generated code versions. The attempt was eventually abandoned.

The code is generated using snprintf(). Long format strings are written into their own files. The makefile used to build the project uses the shell's printf command to null-terminate the file, and it is then turned into an object file using ld –b binary. The format string used to create the module specification may look like this:

Example 5.1

```
module switch(
  clk,
  reset,
  pattern_ready,
  pattern_valid,
  pattern_data,
  in_ready,
  in_valid,
  out_ready,
  out_valid,
  out_t%1$s
);
```

The %1$s is a printf positional parameter. It will be replaced with a generated string with the input and output ports. This is because the number of input and output ports depends on the arguments given to the program, as previously described. Most of the program is written in this way.

Anyone doing similar things is encouraged to look up the GNU libc function open_memstream(). It creates a FILE for use with fprintf () and friends that saves everything written to it in a dynamically allocated string.

5.2 Hardware Interface

For handshaking, the hardware uses ready and valid signals. When ready is asserted, that means that the corresponding input can be sent data the next clock cycle. When valid is asserted, it means that the data is valid. The user of the hardware starts by asserting out_ready. The interleaver will eventually assert in_ready. At that time, the user can begin to send data, which is done by putting the samples in in0 to in*n* and asserting in_valid. The interleaved stream is sent to out0 to out*n*, and out_valid is asserted every time new data is available. If the user can keep both out_ready and in_valid high without interruption then the interleaver is capable of operating continuously.

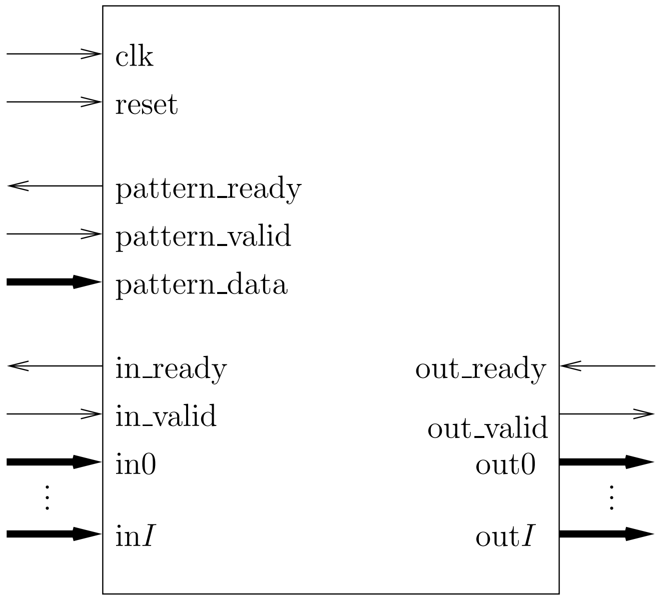

Figure 5.1: *A block diagram showing the inputs and outputs of each channel switch.*

To change the permutation, the pattern inputs are used. The pattern_ready and pattern_valid inputs work as usual. The data is send to pattern_data, which is eight bits wide regardless of the type of interleaver or its parameters. The data shall be the output of the generate_pattern_data() function.

Some versions of the interleaver require a pattern to be programmed into it before they will start interleaving. Those that do not will initially perform the null permutation, where the output is the same as the input.

When the user sends the hardware a pattern, the interleaver should deassert pattern_ready during the same clock cycle that the last pattern_valid is high. This is impossible to do purely with **always** @(**posedge** clk), as whether pattern_valid actually will be high cannot be known before that very clock cycle. An **assign** statement of some sort would be required for it to behave correctly. This could be a surprise to the user and affect the critical path.

In fact, the pattern_ready signal will not be deasserted until one clock cycle too late. It therefore does not quite follow specification. The data path solves the same problem by using a buffer at the output and does follow spec. The problem may be better solved, however, by instead signalling the user that an attempt to send data failed because the device was full. This has not been done.

5.3 Testbench

The generated code comes with a testbench. This testbench tests the hardware by sending it random inputs. At random times, but not before verifying that the current permutation is performed correctly, the testbench randomly generates a new permutation. The function generate_pattern_data() is called with that permutation as an argument, and the output of that function is sent to the interleaver. A few tens of patterns are usually tried before the testbench exits.

For the first half of the run, the testbench sets out_ready and in_valid randomly during each clock cycle, to test that handshaking works correctly. In the second half, a continuous run is tried. On average around 40 patterns will be tried.

6 Results

6.1 Testing

All interleavers were tested in the testbench with a few different values for inputs and period. There were no errors found except for small values for inputs to the Clos network. Not every combination of inputs and period that was synthesized later on was actually tested in the testbench.

A shell script was written to generate and then synthesize each type of interleaver with varying combinations of inputs and period. The output of each synthesizer run was saved in a log file but the resulting design was discarded. From the log, several parameters were extracted using another shell script: the maximum clock frequency, the amount of FPGA logic resources used measured in LUT-Flip Flop pairs, the amount of RAM used measured in bits and the time the synthesis took in seconds.

The interleavers were synthesized for a very large FPGA that was not actually used, so that they would fit for sure. The part number of the FPGA was XC5VLX330T-FF1738, which is a Virtex 5 chip with 300000 logic cells, 960 pins and 11 megabytes of built in RAM. It was not chosen for any reason other than that it was big and would fit the designs.

The FPGA has built-in memory modules. It does not report its resource use in area like an ASIC design tool might. Instead, it gives the RAM usage numbers and the logic slice usage separately. Design Compiler, a synthesis tool for ASIC, was able to synthesize the designs. A memory compiler was not available, however. Without a memory compiler, registers and combinatorial logic are used to implement memories, and this would not give an accurate representation of the resource use in a real implemetation, so Design Compiler was not used.

The parameters chosen were the following: The bit width of the samples was 4. The number of inputs varied. All powers of two from 2 to 32 were used.

Powers of two from 4 to 2048 were used for the period. All combinations of these parameters were tested giving 50 different version of each interleaver.

The first attempt was made on the initial versions of the interleavers which used shift registers for the pattern or, in the case of the bitonic sorter, comparators. It ran for a week on several different computers and had by that time only done some of the smaller configurations. The choice was made to use memories for storing the pattern instead. The shift register versions were not tested further, but they remain in the program.

When the interleavers had been reimplemented using memories, the synthesis was run again. This time it finished overnight on one computer. Parallel memories with 32 inputs still wouldn't synthesize, which is why they are absent from the data.

Clos networks with fewer than 4 inputs did not pass testing in the testbench and that is why they are absent. There was not enough time to fix this.

6.2 Results

The result of the synthesis runs are shown in this chapter. The clock frequency, logic complexity and memory sizes are plotted. Four cross-sections of each dataset are shown. One where $I = 8$ and P varies, one where $I = 4$ and P varies, one where $P = 512$ and I varies and finally one where the total number of channels is constant at $N = 4096$ but I and P vary from almost parallel to almost serial.

Figure 6.1 shows the clock freqency as inputs varies. The clock frequency for Clos network and Bitonic sorter are not strongly affected by the number of inputs, but the clock frequency of the Parallel memories decreases as inputs increases. This is probably because of the large multiplexers at the input, which are not pipelined.

Clock frequency as a function of period is shown in figure 6.2. Bitonic sorter is mostly constant while the other two decrease. The combinatorial part of the Clos network is almost completely unchanged as the period increases, so this result must be because larger memories have a greater latency. This would explain why Bitonic doesn't decrease as much. It uses a larger number of smaller memories. In both these graphs, however, the Clos network is the best regardless of parameters. This is a recurring pattern.

In figure 6.5 the complexity of the logic as a function of inputs is shown. It seems to be linear for Clos and Bitonic and quadratic for Parallel memories. This would make sense as the number of memories and multiplexers is the square of the number of inputs in that implementation. Again, Clos is the best everywhere.

Logic complexity of the Parallel memories and Clos network should not be dependent on period length. Figure 6.6 shows that, for the Clos network, it is not. The complexity of the Parallel memories does vary with period, but actually goes down for the largest period, so this result may not mean anything. Complexity of the Bitonic sorter goes up, since a longer period means more stages.

The memory use of the parallel memories is quadratic with respect to the number of inputs (shown in figure 6.9). The data memories should be linear, so

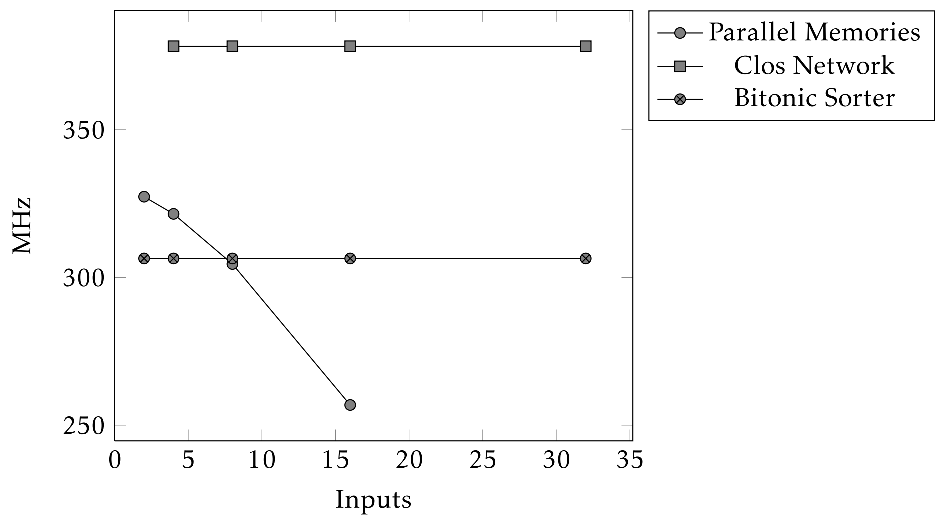

Figure 6.1: *The limit on clock frequency of the different designs depending on the number of parallel inputs when $P = 512$. Higher is better.*

this probably means that the pattern memories dominate. There is nothing more of interest here. The Clos network does the best as usual.

The logic complexity when $N = 4096$ shown in figure 6.8 decreases markedly as the switch becomes more serialized, as one would expect. The RAM use, shown in figure 6.12, behaves differently depending on the type of switch. For the Bitonic sorter, it increases somewhat. In the Clos network it decreases somewhat and in the parallel memories it decreases a lot. The reason for the big decrease in the parallel memories is probably that memory use is dominated by the pattern, which is $\mathcal{O}(I^2)$. As the period increases, the parallel memories come closer to the clos networks in their performance, which would be expected since a completely serialized parallel memories switch is the same as a completely serialized Clos network: a simple RAM memory.

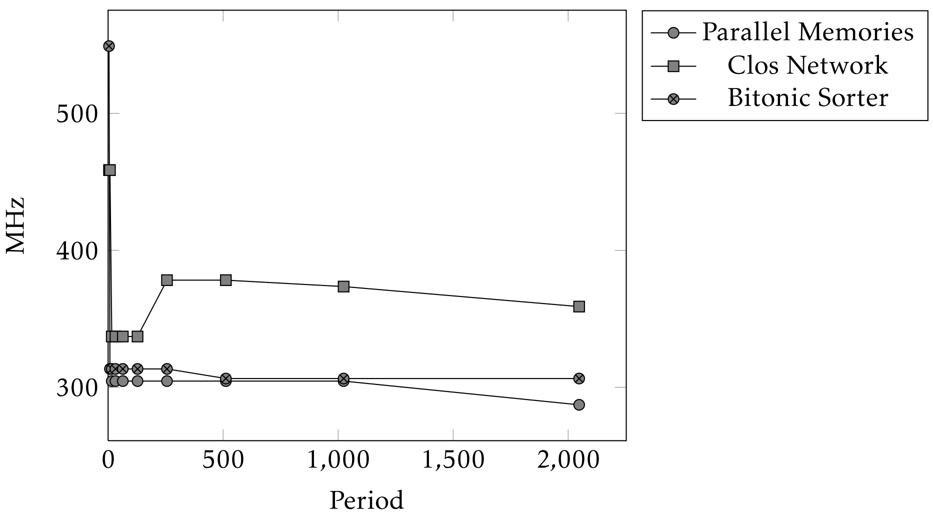

Figure 6.2: *Clock frequency depending on the period length when* $I = 8$.

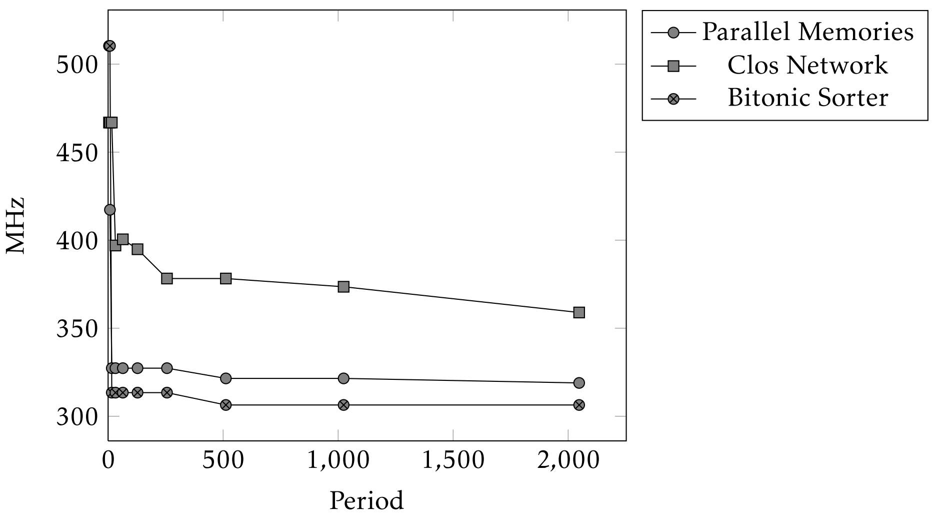

Figure 6.3: *Clock frequency depending on the period length when* $I = 4$.

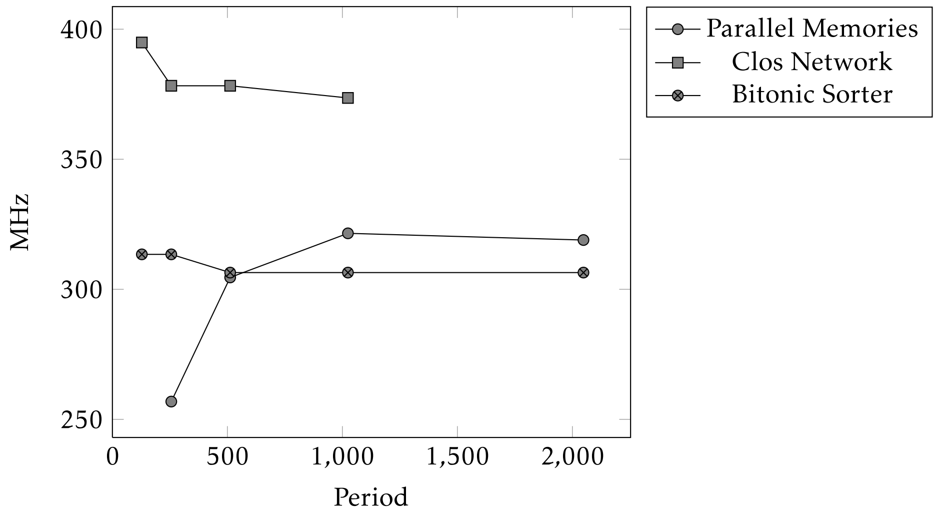

Figure 6.4: *Clock frequency depending on the period length when N is constant at 4096. The plot only shows P but $I = \frac{4096}{P}$.*

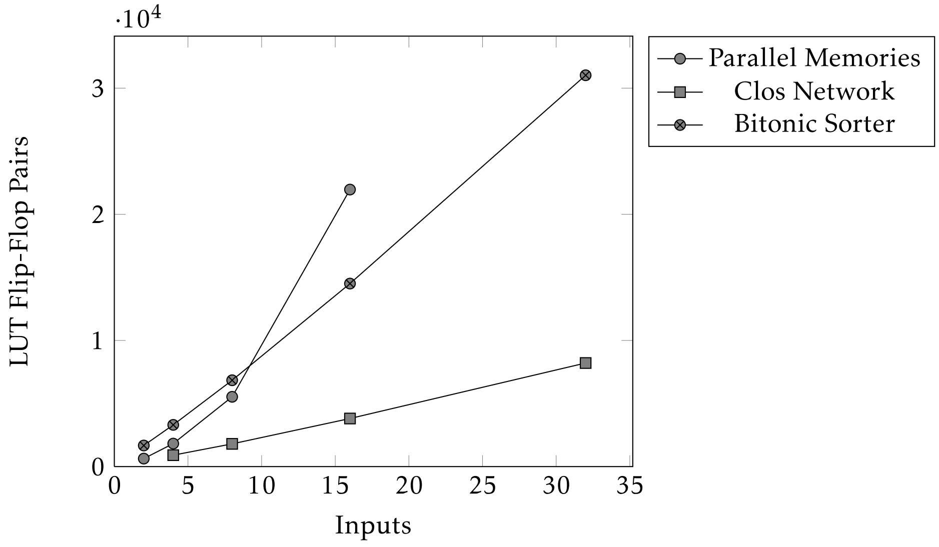

Figure 6.5: *Logic complexity by inputs when $P = 512$. Lower is better.*

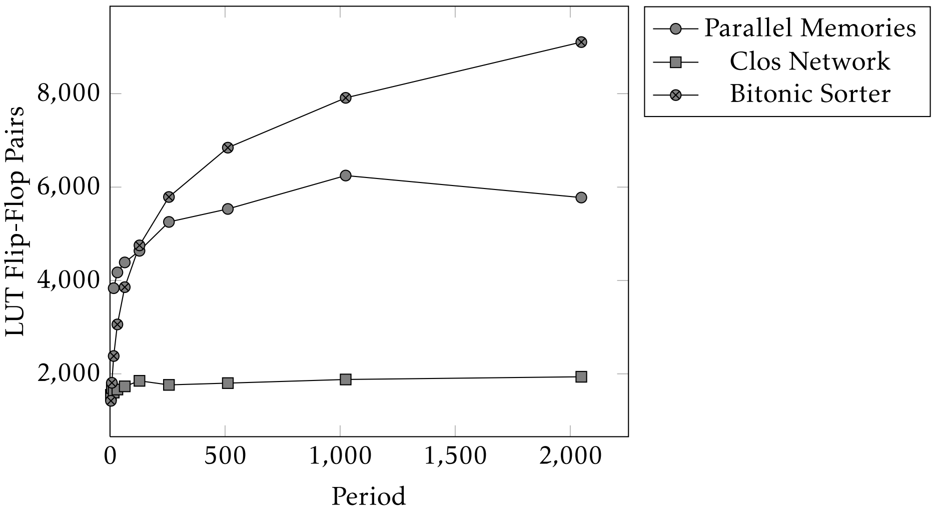

Figure 6.6: *Logic complexity by period when I = 8.*

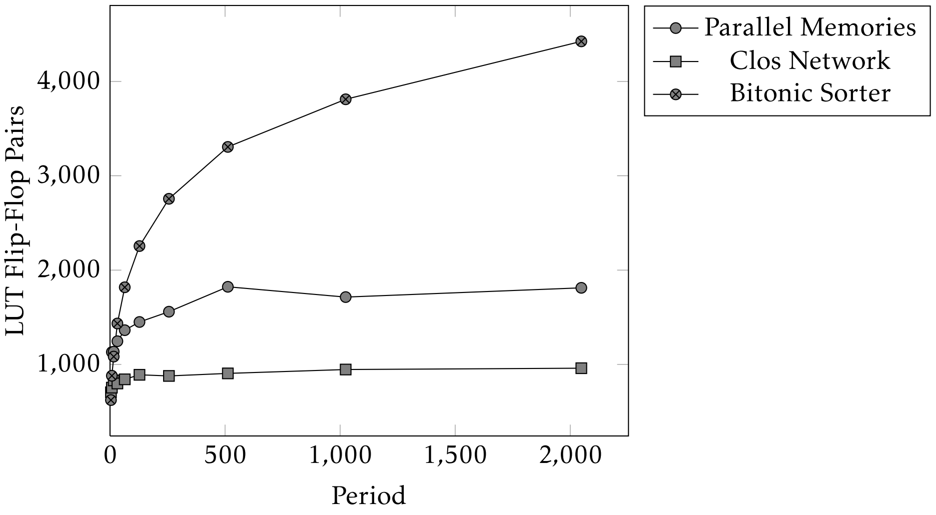

Figure 6.7: *Logic complexity by period when I = 4.*

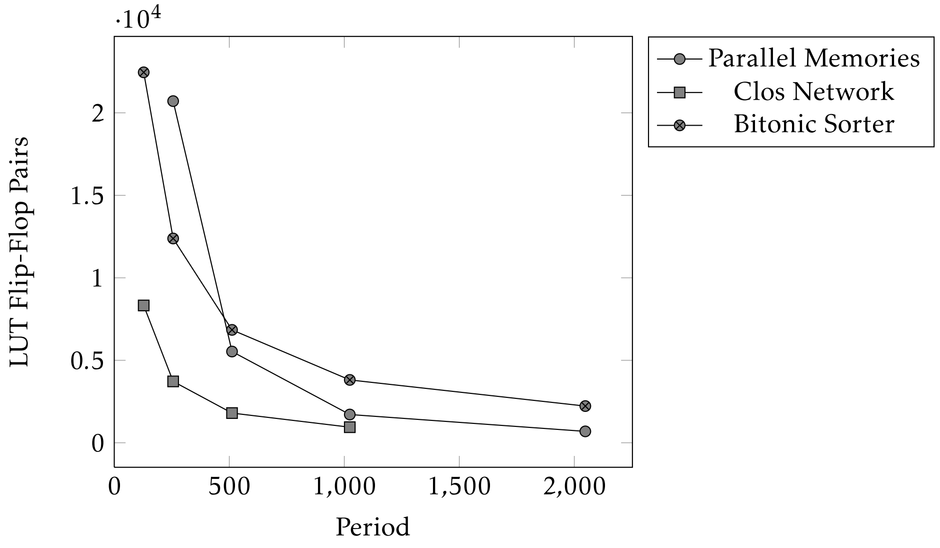

Figure 6.8: *Logic complexity by period when* $N = 4096$.

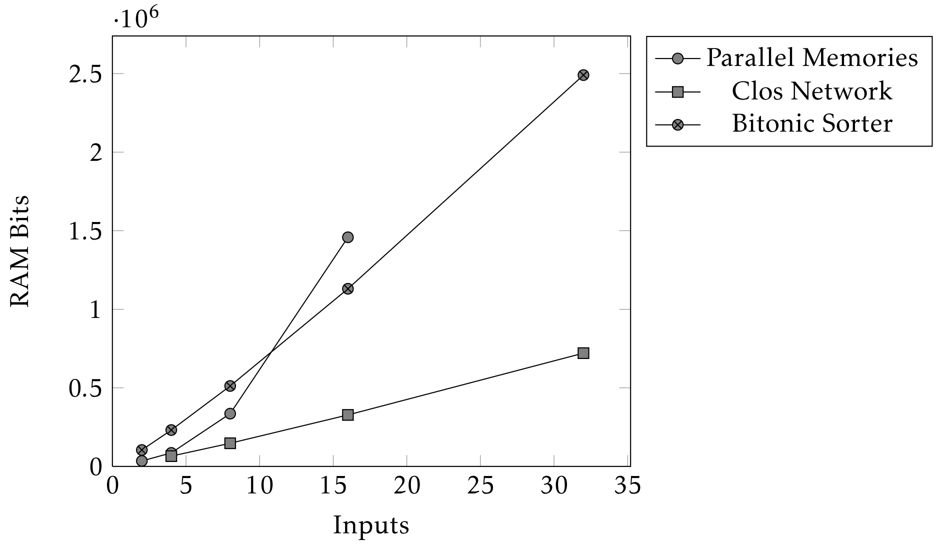

Figure 6.9: *Memory use by inputs when* $P = 512$.

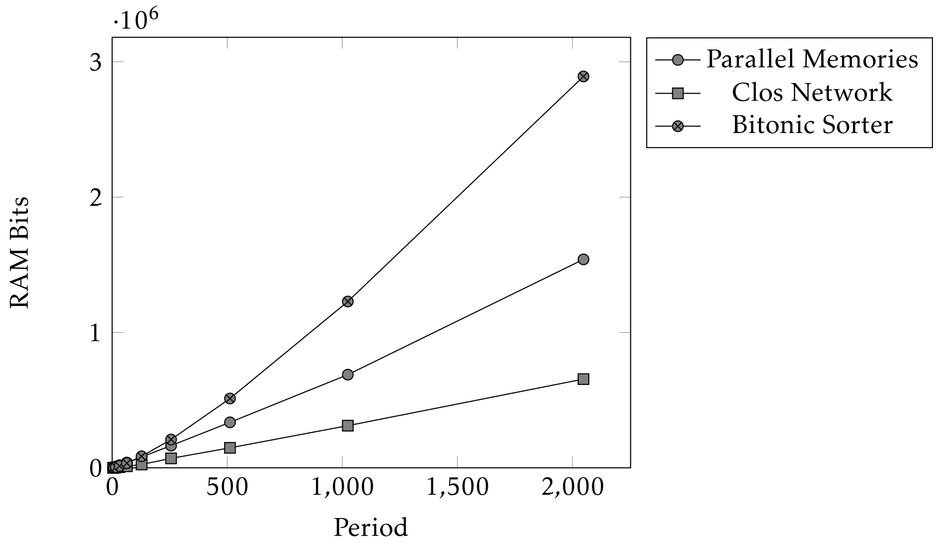

Figure 6.10: *Memory use by period when $I = 8$.*

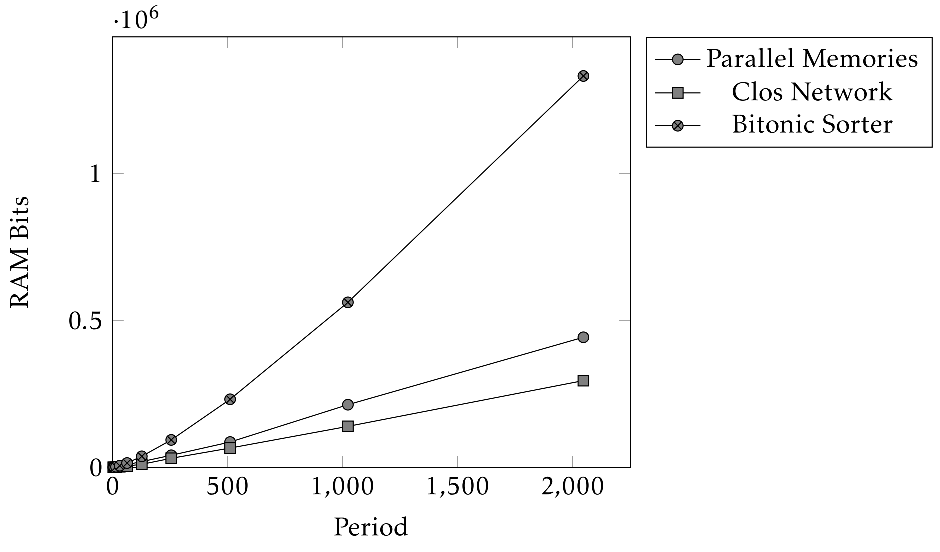

Figure 6.11: *Memory use by period when $I = 4$.*

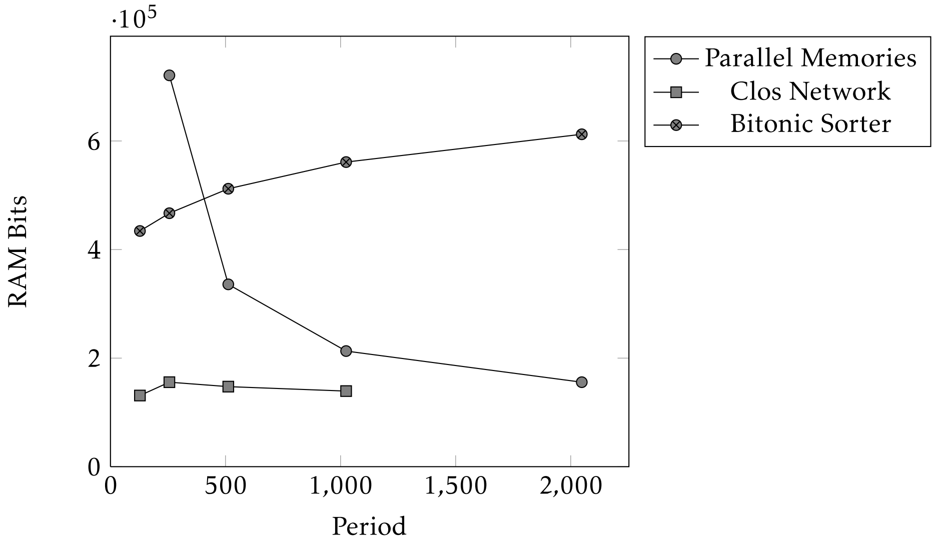

Figure 6.12: *Memory use by period when* $N = 4096$.

7 Discussion

7.1 Results

It's clear from the graphs that the Clos network is the best solution to the problem. The differences can be significant depending on what the parameters are. It's interesting to note how much better the clos network is than the Bitonic sorter despite the fact that they both are implemented using only single port and 2×2 swappers. A Benes network uses $\mathcal{O}(n \log_2 n)$ switches and a Bitonic sorter uses $\mathcal{O}(n \log_2^2 n)$, where n is the number of inputs, yet both can perform any permutation. The difference probably comes from the extra information available to the Benes network. Whether a swap needs to happen in a 2×2 switch in a sorter depends only on the two input values, whereas in a Benes network, the routing algorithm takes the entire state into account. The Bitonic sorter is not the most efficient sorter, however. The existence of simple $\mathcal{O}(n \log_2 n)$ sorting networks is an open question.

Another very clear result is that it was a terrible idea to use shift registers. Learn from my mistakes and use memories.

7.2 Future Work

The Bitonic sorter is currently limited to power-of-two parameters. It should be possible to construct a sorter for the next larger power of two and then simply ignore any swap that touches wires that are not used. Perhaps there are other problems with this approach, but it should be considered.

It is possible that the use of a "full" signal instead of a "ready" signal would simplify the hardware considerably, removing the need for a buffer on the output. The current handshaking scheme was chosen because I couldn't find any information on this on the internet and it seemed like a good choice at the time.

There are some variations on the Clos network that could be considered. A Benes network happens to be equivalent to two butterfly network opposite one another. Omega networks are equivalent to butterfly networks. Perhaps two Omega networks opposite one another would do the same thing and have some interesting properties. Another thing that could be tried is to create a big Benes network and then serialize it in a similar way to the Bitonic sorter, instead of using for the middle stage as currently. My humble prediction is that neither approach will give better results than the STS switch, though.

There is a sorting network with $\mathcal{O}(n \log_2 n)$ complexity and a reasonably simple implementation called the Zig-Zag sorter. Its latency is $n \log_2 n$ but this would not be a problem for our application. It may be worth investigating.

It's possible to extend the Parallel memories and STS interleavers so that they support variable patterns lengths (with a maximum of the one that was specified). This might be interesting in some applications. For it to be useful, more control over when a new pattern is selected would almost certainly be required.

An STS network using multiplexers instead of the Benes switches at the input and output may be better when the number of inputs is very small and the interleaver is implemented on an FPGA because most of them have very efficient multiplexers.

STS switches are not common in telephone systems. TST [3] switches are used instead, at least if the literature is to be believed. A TST switch has memories on the input and output and a crossbar switch (possibly Benes) in the middle. This is probably because the crossbar switch is more expensive than the memories, which would be the case when the number of inputs is larger (by some factor) than the period. The intended application has a much longer period than number of inputs, so the choice of STS over TST is the right one. Still, TST switches could be investigated.

In the STS switch, the Benes switches have one pipeline step for each 2x2 switch. This is good for clock frequency, but may be bad for area utilization and power use. An implementation without any pipelining or with registers only every nth step might be better, especially if clock frequency is limited elsewhere.

It may be worth looking at implementations where the number of outputs is different from the number of inputs.

Appendix

A

Synthesis Results

This appendix contains the results of the synthesis runs. Type, *I* and *P* were parameters. The datatype was 4 bits in all cases. The other columns are the results. LUT FF Pairs is a rough measure of the logical complexity not including memories.

Type	*I*	*P*	Clock (MHz)	LUT FF Pairs	Synth Time (s)	RAM Bits
Mem	2	4	419.62	411	170.13	0
Sort	2	4	549.10	288	162.50	0
Mem	2	8	416.54	481	160.16	0
Sort	2	8	534.90	404	154.21	0
Mem	2	16	418.13	497	356.17	0
Sort	2	16	510.35	566	170.61	0
Mem	2	32	415.95	497	184.13	0
Sort	2	32	313.43	705	232.35	2304
Mem	2	64	398.25	554	178.17	0
Sort	2	64	313.43	907	240.29	6272
Mem	2	128	327.33	627	200.15	5632
Mem	2	256	327.33	644	194.13	12288
Sort	2	256	313.43	1385	254.76	41472
Mem	2	512	327.33	633	198.20	34816
Sort	2	512	306.42	1671	272.39	104448
Mem	2	1024	327.33	646	231.37	73728
Sort	2	1024	306.42	1930	288.19	256000
Mem	2	2048	318.94	692	192.13	155648
Sort	2	2048	306.42	2226	348.24	612352
STS	4	4	466.82	705	242.12	0
Sort	4	4	510.35	622	168.13	0
Mem	4	8	417.27	1131	212.13	0

STS	4	8	466.82	754	194.16	0
Sort	4	8	510.35	880	214.13	0
Mem	4	16	327.33	1135	240.19	2048
STS	4	16	466.82	831	206.16	0
Sort	4	16	313.43	1083	232.43	1984
Mem	4	32	327.33	1246	230.19	4352
STS	4	32	396.99	798	208.19	2048
Sort	4	32	313.43	1434	246.13	5504
Mem	4	64	327.33	1363	240.13	9216
STS	4	64	400.50	842	264.13	4608
Sort	4	64	313.43	1818	256.13	14592
Mem	4	128	327.33	1450	250.13	19456
STS	4	128	394.87	890	238.16	10240
Sort	4	128	313.43	2255	304.17	37376
Mem	4	256	327.33	1559	260.13	40960
STS	4	256	378.24	878	234.14	30720
Sort	4	256	313.43	2756	330.19	93184
Mem	4	512	321.51	1823	266.12	86016
STS	4	512	378.24	905	218.14	65536
Sort	4	512	306.42	3306	382.14	231424
Mem	4	1024	321.51	1714	282.14	212992
STS	4	1024	373.56	946	258.14	139264
Sort	4	1024	306.42	3811	452.16	561152
Mem	4	2048	318.94	1812	262.14	442368
STS	4	2048	358.94	960	298.13	294912
Sort	4	2048	306.42	4425	640.18	1331200
STS	8	4	458.58	1551	266.11	0
Sort	8	4	549.10	1423	250.13	0
STS	8	8	458.58	1698	284.11	0
Sort	8	8	313.43	1809	276.14	1728
Mem	8	16	304.55	3835	334.15	9216
STS	8	16	337.10	1600	328.19	2304
Sort	8	16	313.43	2382	316.14	4864
Mem	8	32	304.55	4173	362.22	18944
STS	8	32	337.10	1665	266.10	5120
Sort	8	32	313.43	3059	312.14	13056
Mem	8	64	304.55	4387	372.14	38912
STS	8	64	337.10	1733	268.10	11264
Sort	8	64	313.43	3857	428.16	33792
Mem	8	128	304.55	4636	398.24	79872
STS	8	128	337.10	1851	318.27	24576
Sort	8	128	313.43	4752	504.15	84992
Mem	8	256	304.55	5254	440.15	163840
STS	8	256	378.24	1764	334.10	69632
Sort	8	256	313.43	5788	502.15	208896
Mem	8	512	304.55	5532	434.22	335872

STS	8	512	378.24	1802	288.10	147456
Sort	8	512	306.42	6844	996.18	512000
Mem	8	1024	304.55	6247	454.15	688128
STS	8	1024	373.56	1880	312.55	311296
Sort	8	1024	306.42	7911	832.19	1228800
Mem	8	2048	287.24	5776	626.36	1540096
STS	8	2048	358.94	1938	358.47	655360
Sort	8	2048	306.42	9105	980.18	2891776
STS	16	4	457.52	3615	386.10	0
Sort	16	4	435.58	3432	584.18	0
STS	16	8	390.51	3329	380.11	2560
Sort	16	8	313.43	4160	488.57	4352
STS	16	16	390.51	3442	752.18	5632
Sort	16	16	313.43	5385	578.15	11776
Mem	16	32	256.84	15996	1354.36	87040
STS	16	32	390.64	3536	820.14	12288
Sort	16	32	313.43	6867	666.15	30720
Mem	16	64	256.84	19035	1398.66	176128
STS	16	64	400.50	3625	522.28	26624
Sort	16	64	313.43	8464	852.68	77824
Mem	16	128	256.84	18317	1412.25	356352
STS	16	128	394.87	3871	456.18	57344
Sort	16	128	313.43	10252	980.18	192512
Mem	16	256	256.84	20711	1496.34	720896
STS	16	256	378.24	3721	496.18	155648
Sort	16	256	313.43	12384	1474.22	466944
Mem	16	512	256.84	21960	1594.52	1458176
STS	16	512	378.24	3811	494.11	327680
Sort	16	512	306.42	14508	1428.74	1130496
Mem	16	1024	256.84	22648	1552.27	2949120
STS	16	1024	374.27	3893	548.11	688128
Sort	16	1024	306.42	16669	1300.20	2686976
Mem	16	2048	256.84	25388	1812.27	5963776
STS	16	2048	359.49	3989	638.12	1441792
Sort	16	2048	306.42	19179	2422.27	6275072
STS	32	4	455.42	7996	904.14	0
Sort	32	4	435.58	8244	1104.36	0
STS	32	8	390.51	7351	1922.19	6144
Sort	32	8	313.43	9530	900.17	10752
STS	32	16	390.64	7582	2176.21	13312
Sort	32	16	313.43	12263	1308.89	28160
STS	32	32	390.64	7691	1944.18	28672
Sort	32	32	313.43	15107	1848.53	71680
STS	32	64	400.50	7902	1146.15	61440
Sort	32	64	313.43	18554	2828.29	178176
STS	32	128	394.87	8324	1190.17	131072

Sort	32	128	313.43	22456	3045.18	434176
STS	32	256	378.24	7996	1122.14	344064
Sort	32	256	313.43	26733	4619.79	1040384
STS	32	512	378.24	8210	1324.15	720896
Sort	32	512	306.42	31022	3306.86	2490368

Bibliography

[1] Chihming Chang and Rami Melhem. Arbitrary size Benes networks. *Parallel Processing Letters*, 7:279–284, 1997. Cited on page 17.

[2] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 1952. Cited on page 3.

[3] A.A. Collins, R.D. Pedersen, and C.B.I.I. John. Time folded tst (time space time) switch, January 25 1977. URL `https://www.google.com/patents/US4005272`. US Patent 4,005,272. Cited on page 36.

[4] Amir Eghbali, Håkan Johansson, Per Löwenborg, and Heinz G. Göckler. Dynamic frequency-band reallocation and allocation: From satellite-based communication systems to cognitive radios. *J. Signal Process. Syst.*, 62(2):187–203, February 2011. ISSN 1939-8018. doi: 10.1007/s11265-009-0348-1. URL `http://dx.doi.org/10.1007/s11265-009-0348-1`. Cited on page 2.

[5] H. Johansson and O. Gustafsson. Filter-bank based all-digital channelizers and aggregators for multi-standard video distribution. In *2015 IEEE International Conference on Digital Signal Processing (DSP)*, pages 1117–1120, July 2015. doi: 10.1109/ICDSP.2015.7252052. Cited on page 2.

[6] Håkan Johansson and Per Löwenborg. Flexible frequency-band reallocation networks using variable oversampled complex-modulated filter banks. *EURASIP J. Appl. Signal Process.*, 2007(1):143–143, January 2007. ISSN 1110-8657. doi: 10.1155/2007/63714. URL `http://dx.doi.org/10.1155/2007/63714`. Cited on page 3.

[7] Gordian Prescher, Tobias Gemmeke, and Tobias G. Noll. A parametrizable low-power high-throughput turbo-decoder. In *2005 IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP '05, Philadelphia, Pennsylvania, USA, March 18-23, 2005*, pages 25–28. IEEE, 2005. ISBN 0-7803-8874-7. doi: 10.1109/ICASSP.2005.1416231. URL `http://dx.doi.org/10.1109/ICASSP.2005.1416231`. Cited on page 17.