Resurrecting Extinct Computers - The Connection Machine

Zoe-Rose Guy

MCompSci Computer Science - Part B

Trinity 2023

Word Count: 4989

Abstract

While the architectures of current commercial processors are well established, and relatively static [12, 16], the early days of computing saw extensive experimentation and exploration of alternative designs. These included the Connection Machine (CM-1) consisting of 65,536 individual one-bit processors connected as a 12-dimensional hypercube [9].

Through the development of a cycle accurate simulator of the Connection Machine, and several example programs, an evaluation of the machine has been conducted and its reasons for failure analysed. An RTL hardware description of the Machine's building block chip has also been created, which would allow a full replica to be constructed; both of these are important preservation steps for a piece of computing history at severe risk of being forgotten.

Quantitative evaluation of the Connection Machine provides mixed results. It performs remarkably well, even against hardware from almost 40 years later, on certain tasks: a breadth-first search algorithm runs at around 2 cycles per element, made even more astounding by the 1 bit word size and approximately 700 cycle latency of message passing. However, these factors become much more limiting in other tasks, stunting performance in some traditionally easily parallelisable applications, such as those in linear algebra.

Acknowledgements

First and foremost, I would like to thank my supervisor, who I credit with both sparking my interest in this field, and providing expert guidance on the project and its direction.

I would also like to thank Dr Paul Franzon of North Carolina State University, who's excellent course on Verilog and digital ASIC design is freely available on YouTube[1], and was an extremely valuable resource as I learned the language. So too was Stuart Sutherland's written reference for Verilog[2].

I'd also like to thank my friends and family, who have been in my corner continuously as I stressed about this project and the rest of my commitments. I am eternally grateful to them for putting up with me harping on and on about a "weird computer from the 80s" for the past 6 months.

Finally, to my college cat, who's company through my early morning library sessionsn has made them just that little bit more bearable!

[1]`https://youtube.com/playlist?list=PLfGJEQLQIDBN0VsXQ68_FEYyqcym8CTDN`
[2]`https://www.sutherland-hdl.com/pdfs/verilog_2001_ref_guide.pdf`

Contents

Chapter 1

Introduction

1.1 Motivations

The Connection Machine was a supercomputer designed in 1985 by W. Daniel Hillis [9]. It presented a fundamentally different computer, that can be thought of as "smart memory," [11] which can be issued SIMD style instructions to operate on its data massively in parallel. The CM-1 used 65536 parallel 1-bit processors each with their own memory, able to communicate via message passing to solve a wide range of complex problems. Without active preservation efforts, it is at severe risk of being lost to history.

The overarching motivation of this project is to create a suite of tools allowing anybody to understand the structure, benefits, and drawbacks of the machine, and allow them to write programs for it, in order to preserve this strange and innovative architecture. As few as seven Connection Machines were built [17], which are quickly ageing and may soon become unusable.

Historic preservation is often overlooked in computing, but it remains important for a number of reasons. Lots of systems and standards carry historical baggage; for example, modern Intel processors support 16-bit operations to allow compatibility with programs from almost 50 years ago [12], and North American television broadcasts at 29.97 frames per second as a hangover from the analogue NTSC standard [2]. Understanding historical context is important to understanding these systems and the logic behind their design. The Connection

Machine, niche as it is, likely still had an influence on modern supercomputing through its descendant, the CM-5, which was for a time the world's fastest supercomputer [18].

Modern CPU architecture in particular is very well established and hasn't experienced a true paradigm shift since the introduction of the ARM and other RISC processors in the 1980s [16], and arguably since the Manchester Baby implemented von Neumann architecture in 1948 [5]. In order to spur innovation into such stagnant fields, it is invaluable to be inspired by previous innovations, and learn from their successes and failures.

1.2 Contributions

This project contributes several pieces of code related to the Connection Machine.

The primary contribution is `libcm`, a full, cycle accurate, low-level programmable simulator of the machine. `libcm` takes the form of a C library, allowing a developer to write standard C code interspersed with library calls that issue instructions to the machine. This mirrors the physical machine's structure, whereby a sequential "host" computer would issue instructions to the cells as required. In fact, `libcm`'s structure has been carefully designed to reflect the machine's structure at the expense of speed, so that the source code may act as a clear reference for those trying to understand the machine's structure. `libcm` is as accurate as reasonably possible, however, due to the scarcity of sources and original hardware examples, there are bound to be some minor differences, which are enumerated in Section 3.2.

A Verilog RTL description of the Connection Machine's chip has also been provided. From this, with some simple changes to ensure compatibility with specific technologies, it would be possible to build a full replica of the machine, which is also important for preservation. This implementation has similar inaccuracies to `libcm`.

Several programs for the Connection Machine, including vector dot product and breadth first search, are included in Appendix A. Notably, an implementation of Feynman's logarithm algorithm [1] is also included, a historically significant algorithm which was run on the machine during its development [8]. The nature of the machine makes its programs seem alien and opaque; code must be verbosely commented to be understood.

A thorough evaluation of the architecture using `libcm` and the aforementioned programs has been conducted, in order to expose its strengths and weaknesses, and give evidence as to why no similar architecture exists today. The results are mixed. The machine excels in problems such as breadth first search, which can fully utilise the routing network and make use of the available parallelism, performing this task at a rate of just 2 cycles per element. However, in more structured communication networks with mathematical operations, performance is very poor, yielding a significantly worse CPE in vector dot product than the sequential implementation, despite the theoretical asymptotic advantage.

Finally, as Connection Machine code is very different to sequential code, a debugging tool, CMFrames, is also contributed. It allows full dumps of the machine's state created at runtime to be read and explored afterwards, to truly understand how the programs operate.

1.3 Structure of the Report

The next chapter briefly discusses the Connection Machine's history, and provides an overview of its instruction set and communication hardware. Chapters 3 & 4 then discuss in detail the implementations of `libcm` and the Verilog chip, in particular the implementation of the inter-processor communication router, the most complicated element of the machine. Their structures are discussed, which are designed to reflect the structure of the machine to allow them to act as reference implementations.

Chapter 5 then goes on to describe and demonstrate some Connection Machine programs, and provide timing results for them, contextualised with results of equivalent sequential programs on a modern CPU. This is followed by a discussion of those results, the machine's limitations more broadly, and how these may have been responsible for its failure. The demonstrated programs were chosen to cover a breadth of use cases for the machine, including using its routing network in pathological ways. It also provides a short discussion into the advantages to asymptotic run time that would be achievable on an infinite Connection Machine - whilst such a machine is obviously impossible to build, these advantages remain significant for even modest problems.

Finally Chapter 6 provides the author's reflection on the project, and outlines future work that could be undertaken to aid in the preservation of this piece of history.

Chapter 2

Background

2.1 History

The Connection Machine was designed in 1985 by Daniel Hillis [9] as part of his PhD thesis, and later built by the company he founded, Thinking Machines Corporation (TMC) [13]. Many notable individuals were involved in the company, including physicist Richard Feynman [8] and Internet Archive founder Brewster Kahle [4]. TMC produced several generations of the machine, namely the original CM-1, the CM-2 which found uses in scientific computing [7], and the ill-fated CM-5 [17].

2.2 Hardware Description

The CM-1 was built from 4096 chips, each of which contained 16 processing cells and a router. Processing cells were very simple, consisting of 4096 bits of memory, alongside 16 1-bit flags[1] [9]. The machine had only one “instruction,” parameterised by:

- Two 12-bit memory addresses, A and B
- Three 4-bit flag identifiers, R, W, and C
- One 1-bit condition variable

[1]8 general purpose flags, and 8 with special functions

- Two 8-bit truth tables, one for each of memory and flags
- One 2-bit direction variable

The cell takes the value of the bits referenced by addresses A and B, and the value of the flag R, looks up the corresponding value in the memory truth table, and stores it in address A. A similar process occurs for the flag truth table and the flag referenced in W. All of this occurs only if the value in flag C, the condition flag, is equal to the condition variable [9].

The machine provides a simple grid system for the cells on a chip to communicate with each other. Cells are able to send messages to their north, east, south, and west neighbours, selected based on the 2-bit direction variable [9].

The chips' routers are connected in a 12 dimensional hypercube topology. Cells can "inject" messages into their router via a special flag, which will then be sent by the routing network to the addressed cell. A cell can send a message to any other cell using their relative addresses on the hypercube, which will be delivered via a special flag in the cell [9].

A full description of the hardware can be found in Chapter 5 of Hillis's PhD thesis.

Chapter 3

The Simulator

The primary contribution of the project is a full simulator of the Connection Machine, provided as a C library named `libcm`. `libcm` seems to be the only existing low-level simulator of the Connection Machine[1], and has been designed to be cycle accurate to the largest extent made possible by the scarce available sources. Full source code for `libcm` is available on GitHub.

3.1 `libcm` Structure

The structure of `libcm` was deliberately designed to be similar to the structure of the Connection Machine to provide a transparent description of its behaviour. The library primarily provides a struct representing the machine, which contains an array of pointers to chips. Each chip is a struct containing a pointer to its router, and to its 16 processing cells. Each router contains its buffered messages and other data necessary for its function, as described in Section 3.3, and each cell is a struct containing its 4096 bits of memory and 16 flags. Executing instructions on the machine is achieved by calling the function `cm_exe()` on the top-level struct, which in turn calls corresponding functions on the chips and finally the cells. Wires between chips, used in inter-router communication, have been implemented using pointers,

[1]A higher level simulator ia available at `https://www.softwarepreservation.org/projects/LISP/starlisp/sim/`, which simulates the machines *Lisp interface

which are assigned when the initial `cm_build()` function is called.

3.2 Inaccuracies of `libcm`

There is little available literature regarding the Connection Machine. It is therefore inevitable that `libcm` will have some slight differences from the machine; the known differences are enumerated below:

- Referral, the process by which routers deal with more messages than their buffers can handle, is very poorly defined in Hillis's thesis. `libcm` implements this using a simple Hamiltonian cycle of routers, further explained in Section 3.3.
- Similarly, the communication protocol between routers and cells isn't defined; `libcm` uses a simple protocol whereby agents wishing to communicate first send a 1, for which the other must be listening.
- Routers contain buffers for seven messages as in Hillis's thesis [9], whereas the eventual routers had just five buffers [8].
- The "Global Pin", a 1-bit signal that may be asserted by any processor, is poorly defined in Hillis's thesis; this has been implemented by writing to a specific flag in the cell.
- `libcm` has a function to test if the router network is empty, which greatly increases the ability of the routers to deal with congestion; it is unclear whether such functionality existed in the actual machine.

3.3 Router Implementation

Whilst much of `libcm`'s implementation is self explanatory, the router is a fairly complicated piece of hardware that is worth describing in more detail. Similarly to other parts of the simulator, it is implemented as a struct, with its definition in `router.h` as follows:

```
    typedef struct rint
```

```
{
  Message *inports[DIMENSIONS];
  Message **outports[DIMENSIONS];
  Message *buffer[BUFSIZE];
  uint32_t listening[4];
  Message *partials[4];
  uint16_t *flags[1 << PROCESSORS];
  struct rint *referer;
  uint32_t id;
} Router;
```

As the router is connected to various structures, it understandably uses a lot of pointers to refer to them.

The `inports` array contains pointers to messages. When a message is received from another router, a pointer to it is placed in the array entry corresponding to the dimension of the hypercube along which it was sent. Similarly, `buffer` contains pointers to the messages currently stored in the router. The buffer itself is ordered, with messages at lower array indices having a higher priority over those with higher indices.

`listening` and `partials` are used in message injection from the processors. At the start of every petit cycle[2], all processors that wish to send a message write a 1 to their router data flag. A maximum of 4 of those are selected, and their identities placed into the `listening` array. A new message struct is also allocated on the heap, and its pointer is placed in the corresponding entry in `partials`. For the remainder of the injection cycle, bits are copied from cells' the router data flag into these partial messages. Once the message is complete, it is placed in the router's buffer. This data flag is accessible via the `flags` array, containing pointers to the values of flags inside the router's associated cells.

`**outports` contains pointers referring to locations in the `*inports` of other routers. By setting the dereferenced outport to a message pointer, routers are capable of sending messages

[2]the name given by Hillis to the routers' communication cycle, encompassing processors injecting messages, the transferal of messages across the network, and the delivery of messages to destination processors

to each other.

Finally, `*referer` points to an arbitrary different router in the machine. During `cm_exe()`, these are simply assigned in sequential order to create a Hamiltonian cycle. In the event of a buffer overflow, the incoming message can be offloaded to this other router to solve the problem. The router's absolute address, `id`, is also required to ensure that relative addresses are maintained - by XORing this with the address in the message, we obtain the absolute address of the message, which can then be XORed with the referrer's ID to set the relative address correctly.

3.4 Interface Implementation

The original Connection Machine's interface used a sequential "host" computer that could be programmed in *Lisp[3] [9], which would issue instructions to the machine. `libcm` operates at a lower level - programmers write a standard C program, interspersed with Connection Machine instructions, issued using calls to `cm_exe()`.

`libcm` provides several other functions for interfacing with the machine, including checking the routing network's status, speeding up routers for faster testing, and reading from the Global Pin.

`libcm` also provides the option to dump the state of the machine throughout the entire run of the program into a zip archive. Dumps can then by analysed using CMFrames to aid in debugging, though even modest programs generate large dumps.

[3]A Lisp variant with parallel functions and data structures, designed specifically for the Connection Machine

Chapter 4

Hardware Implementation

A “sketch Verilog” implementation of the Connection Machine’s chip has also been provided. With minimal changes, this could be synthesised into an actual piece of hardware; if several of these were built, it would be possible to build a full replica of the Machine. These changes mostly involve the mapping of pins to input lines. The chip requires at least 80 pins, including 55 for instruction parameters, bidirectional connections to 12 other routers, and several connections to the host. More would be required to communicate with memory if this is not stored on chip. The full source of this implementation is available on GitHub.

4.1 Router Implementation

As stated in Section 3.3, a lot of the building blocks of the Connection Machine are very simple. Processor flags are just 1-bit registers, and the “ALU” turns out to be nothing more than an 8x3 multiplexer. The router is the exception - it is in fact even more awkward to implement in hardware than software due to its priority system. In most cases, calculations involving priorities must be done in a single clock cycle, creating yet more complexity. A block diagram of the router is provided in Figure 4.1.

The injector portion, responsible for taking input from cells, and the heart, responsible for inter-router communication, are both fairly simple. Both simply read priority values, make calculations as to which buffer(s) to use, and send their results to multiplexers. Priority

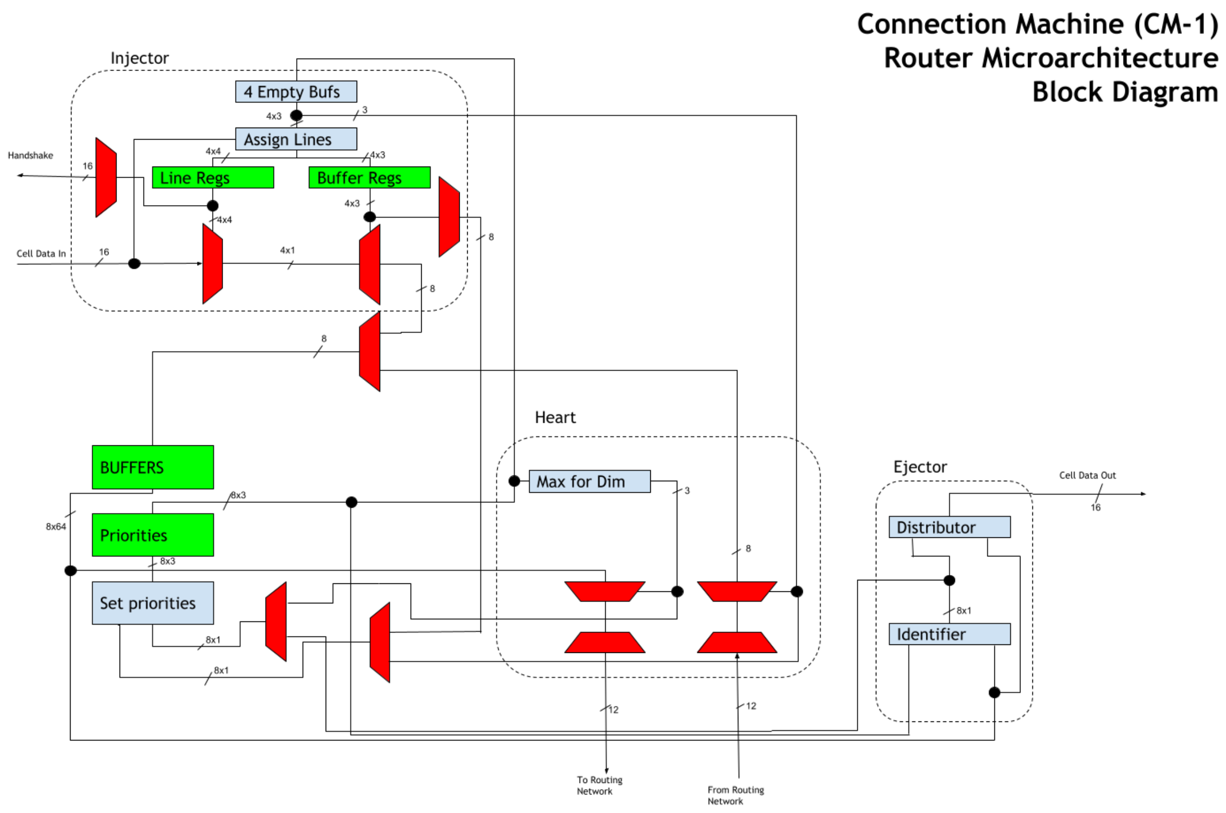

Figure 4.1: Block diagram of the router

calculations here are primarily done by searching for minimum/maximum values with a tree of comparators. Note that these portions do not set priorities, only signal a dedicated unit to instruct it to recalculate priorities.

The priority calculator *is* able to use multiple clock cycles. While a message is being sent or received, its priority cannot change, meaning the new priority value should be calculated over several cycles and updated following complete message transmission. The calculator therefore can iterate over all 8 possible priority values, incrementing the priority of any message with a lower priority than x if there is no message with priority x.

The most complicated part of the router is the ejector, responsible for sending messages from the network to the cells, which is necessarily quite large and complex. It can be broadly split into two components, named the identifier and the distributor. The identifier selects the messages to be sent to cells, either by selecting all messages destined for a cell on this chip, or using priority calculations depending on the delivery mode. This result is sent to the distributor and to the priority calculator to mark these messages for deletion at the end of the cycle. The distributor then takes the logical AND of each router's selection bit and the bit it's trying to communicate, then uses an array of barrel shifters to place this bit on the correct line to the cells, before taking the OR of all of these to produce the final output to the cells.

Chapter 5

Programs and Evaluation

The Connection Machine failed, but it is still possible that aspects of its architecture could be useful today; therefore it is useful to evaluate the machine. To this end, several programs have been developed that demonstrate various aspects of the machine's function, and their performance tested against sequential implementations.

Note that Connection Machine programs look very different to traditional sequential programs. Source code for the evaluation programs can be found in Appendix A.

5.1 Massive SIMD - Feynman's Logarithm Algorithm

Whilst working on the Manhattan Project, physicist Richard Feynman developed an algorithm for calculating logarithms [8, 1], which relies on the fact that any numbers between 1.0 and 2.0 can be expressed as a product of terms of the form $(1 + 2^{-k})$. This expression is very easy to calculate in binary - multiplication by $(1 + 2^{-k})$ is achieved with a shift and an addition - and the logarithm can be found by summing the logarithms of the component terms from a table [1]. This worked well on the Connection Machine as this table is small and could be shared by all processors [8]. This algorithm has been implemented for `libcm` during this project.

As this demonstrates simple SIMD operation, the program source is not particularly interesting. It is very similar to the sequential version, but with typical mathematical functions replaced with sequences of calls to `cm_exe()`.

5.2 Structured Communication - Vector Operations

A more interesting use of the Connection Machine is in vector operations. An example program sets up 2 vectors on cell 0 of various chips[1], with the 2 vectors separated by only a dimension. This allows messages to be sent between same-indexed entries in each vector with a relative address containing a single high bit. The program calculates a dot product, by means of sending each value from vector B into the corresponding cell in vector A, then performing an integer multiplication, before adding up all these values along vector A in $O(logn)$ using a tree-like structure as demonstrated in Figure 5.1, completing the addition in $O(logn)$.

5.3 Unstructured Communication - Breadth First Search

The previous programs are in a sense “well behaved” - communication is either not present or is very highly structured, and organised such that router congestion will never occur. This is, of course, unrealistic in practical programs.

Graphs can be implemented fairly naturally on the Connection Machine, with each cell functioning as a vertex, storing a list of addresses corresponding to vertices it shares an edge with. Algorithms requiring traversing edges, such as breadth first search, can be implemented by instructing vertices to send messages to all their neighbours. In this implementation, computation proceeds in rounds - there are sets of discovered and undiscovered vertices, and discovered vertices send messages to all their undiscovered neighbours. These messages include the relative address of the sender, so when the algorithm terminates, the back pointers can be followed to find the shortest path between a node and the initial vertex.

This algorithm is fundamentally different from the vector operation described in Section 5.2, as graphs have less structure than vectors - edges connect two arbitrary vertices. This

[1] Using more cells would cause router congestion, affecting performance.

gives rise to the issue of router congestion, as there are many messages in the router network going in many different directions. Consequently, messages may not arrive when expected, but several petit cycles later, and injecting messages may fail if the router is already full. It was therefore important to develop a system to deal with this.

The solution was to store a bitmap in each cell's memories, indicating whether or not it had yet sent a message along the corresponding edge. The vertices all have fixed out-degree, allowing this to be constructed. The program iterates over the edges, sending along that edge if its bit is high, and turning it low if the transmission succeeds. This repeats until all messages have been sent, which can be detected by assertions on the Global Pin.

This unstructured communication can saturate certain routers extremely quickly. During testing, even when increasing buffer size to a hundred messages, overflow occurred. Referral is an essential mechanism for this program.

5.4 Quantitative Results

For evaluation, the machine will be compared against a typical modern CPU running sequential implementations of the same algorithms on a single core. Results are given both in real time and cycles per element, taking a 4MHz clock for the Connection Machine and a 3.6GHz clock for the modern CPU. Note that the real time results for the sequential programs are adjusted for the overheads of generating their inputs and running the testing loop. Each sequential implementation operated for 10,000 cycles on a problem of size 65536, the size that fits onto the Connection Machine, except the vectors program, which multiplies 2048-vectors 1,000,000 times. Table 5.1 shows the timed results.

Results for the Connection Machine were taken using `libcm`, measured in cycles, taking an average of around 10 independent runs for BFS due to its variable run time depending on the graph structure. Cycles per element and estimated real time are provided, as well as an estimate of the time required to process 10,000 problems[2] for comparison to the sequential code. Table 5.2 shows these results.

[2] 1,000,000 for Vectors

Program		Time (s)	CPE
Feynman's Logarithm	Unoptimised	22.0	120.6
	`-O3`	7.4	40.5
Vectors	Unoptimised	14.2	25.0
	`-O3`	3.0	5.2
BFS	Unoptimised	17.8	97.9
	`-O3`	7.9	43.3

Table 5.1: Sequential program results

Program	Total Cycles	CPE	Real Time (ms)	Equivalent Time (s)
Feynman's Logarithm	7196	0.1	1.8	18.0
Vectors	15787	7.7	0.4	3946.8
BFS	138705	2.116	34.7	346.8

Table 5.2: Connection Machine Results

The evaluative testing paints a very interesting picture of the Connection Machine. Despite a time difference of nearly 40 years, it performs some 20% faster than the modern processor running unoptimised code in the calculation of fixed point logarithms. This is, however, to be taken with a pinch of salt - it is unsurprising that the connection machine would perform better simply because it can calculate 65536 logarithms at once, and a GPU logarithm algorithm with *some* parallelism would perhaps be a fairer comparison. The `-O3` optimised executable does outperform the Connection Machine implementation by a significant margin.

A more interesting result is that for breadth first search, which is difficult to parallelise on modern systems. Using the time for the unoptimised search, the Connection Machine runs around 20 times slower - very impressive, considering the slow speed of message passing and the 1-bit word length. Such tasks are those at which the Connection Machine excels, particularly due to the ability to send many messages in parallel to offset their latency. This was surprising - it was expected that this algorithm would perform poorly due to router congestion.

The only really disappointing result was the vector multiplication. The slowness is explained by the 700 cycle latency of message passing, creating a large constant on the theoretical $O(logn)$ algorithm. Dot product of vectors is also an operation that can be well parallelised on

Task	Algorithm	Average Case	Worst Case
Vector Addition	Textbook	$O^\infty(1)$	$O^\infty(1)$
Dot Product	Textbook	$O^\infty(logn)$	$O^\infty(logn)$
Matrix Multiplication	Textbook	$O^\infty(nlogn)$	$O^\infty(nlogn)$
Graph Search	BFS	$O^\infty(\|E\|log\|V\|)$	$O^\infty(\|E\|\|V\|)$
Set Membership	"Assertion Search"	$O^\infty(1)$	$O^\infty(1)$
Vector Search	"Treelike Search"	$O^\infty(logn)$	$O^\infty(logn)$
Sort	Bitonic Mergesort[3]	$O^\infty(log^2n)$	$O^\infty(log^2n)$

Table 5.3: Ideal asymptotic complexity values

modern processors using the Streaming SIMD Extension (SSE) and its successors for x86[10]. SSE is used for the `-O3` optimised vectors program, explaining its large speedup.

5.5 Asymptotic Analysis

Table 5.3 shows the asymptotic complexity of some algorithms, where $O^\infty(f(x))$ denotes the asymptotic run time on an idealised, infinitely large Connection Machine. "Assertion Search" is a very simple algorithm where cells containing the requested element simply assert over the Global Pin, and "Treelike Search" has cells communicate in a similar manner to the dot product algorithm to find the correct index.

Though obviously impossible to construct in real life, this provides useful results about the potential speedup available to modestly sized problems. With modern manufacturing, a Connection Machine with billions of cells is potentially feasible, providing significant speedup for matrix/tensor heavy applications in machine learning, for example.

5.6 Why the Connection Machine is No Good

The Connection Machine has been demonstrated to be an extremely powerful computer, able to perform remarkably well against modern machines, with the massive parallelism able to provide great speedup for modestly sized problems. What follows are opinions as to why it failed, and why no similar architectures are in use today.

The primary reason for the failure of the Connection Machine was likely the business case. The machine was expensive and its target towards AI research resulted in low sales [17]. Though it did have some uses in scientific computing, such as in quantum chromodynamics [8], it generally performed poorly in other fields [17].

Economic factors may explain the failure of the machine, but less so the architecture, which, as these results demonstrate, was also flawed. Hillis clearly intended to construct a very general purpose, blank-slate machine that could be used for a wide range of tasks; in doing so, he built a “jack of all trades, master of none.” One of the greatest limitations is the 1-bit word length, meaning that operations that would typically be thought of as atomic, such as addition and logical operations, take time relative to the word length of the values - it takes 32 cycles to add 32 bits numbers.

Message passing, a major mechanism responsible for the Machine’s power, has a very high latency. In the test programs that made use of it, most (upwards of 95% of) processor time was wasted waiting for messages to arrive, severely stunting the parallel advantage. It excels in simple SIMD applications, but this is neither surprising nor interesting due to the high thread count.

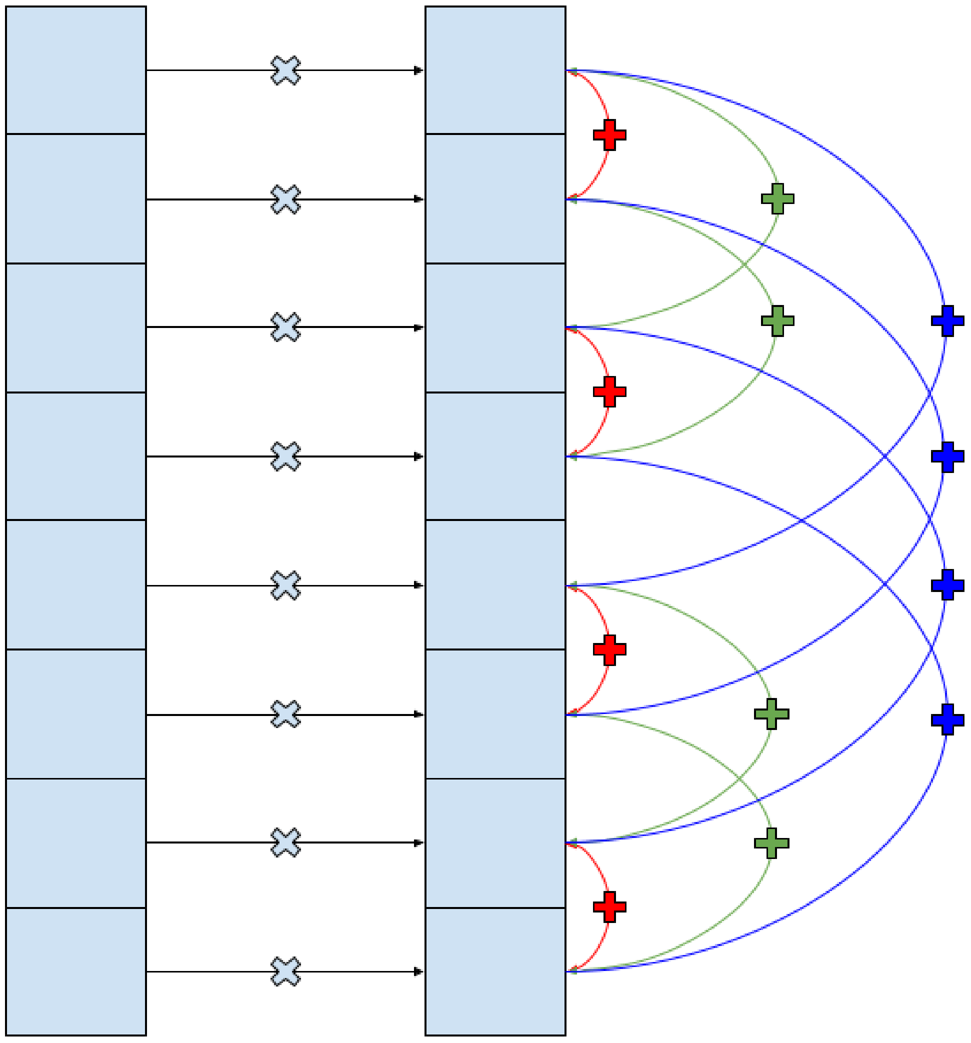

Figure 5.1: Communications made in vector multiplication.

Chapter 6

Conclusions

The primary contribution of this project is `libcm`, which is likely the only cycle accurate simulator of the Connection Machine. Its existence is critical for the preservation of this rare and aging architecture.

The evaluation of the machine also helps to explain why the innovative architecture failed, despite its incredible power in some applications, and has been superseded by more specialised parallel solutions. The hardware description of the machine is useful in preservation, as it would allow a full replica to be built, which will become important as original specimens degrade.

6.1 Reflection

Being a great fan of historic computing, I really enjoyed researching the Connection Machine and implementing `libcm` during this project. Though it is extremely difficult to find sources on the machine, I feel I did a good job of recreating it as accurately as possible without access to real hardware.

Having learnt about Verilog through Franzon's course over the summer, I was eager to make a start on the project, and able to demonstrate a mostly functional `libcm` by the Christmas Vacation. Hilary Term was spent mostly writing the RTL description, writing the example programs, and adding finishing touches to `libcm`. Due to my inexperience with

Verilog, I found that part of the project extremely challenging.

The implementation of CMFrames, used to analyse states of `libcm` to find bugs in programs, was very awkward due to the way dump data is gathered. Data is simply placed into a large binary file which is then placed into a zip archive on every cycle of the run of the program. This makes extraction of data in CMFrames very difficult and necessitated messy code.

6.2 Future Work

Programs for the Connection Machine were originally written in *Lisp [9]. `libcm` only adds Connection Machine machine instruction functionality to C, meaning original programs can't be run on `libcm`. It would be nice to see a front-end built for `libcm` that allows *Lisp programs to be run, both for historical preservation purposes and to make developing for `libcm` easier. This could be achieved by means of a transpiler, or perhaps with help from the existing *Lisp simulator[1].

Seeing as `libcm` is intended to act as a definitional simulator for the Connection Machine, it would be incredibly useful to experiment on original hardware, to allow its inaccuracies to be eliminated and better document the machine's function. However, owing to the limited number of machines produced, working examples are very scarce - at least one is located at the Computer History Museum, Mountain View, California [15], alongside an example of the empty casing [14]. I am unaware of any examples located outside of the United States, or even this one site[2].

Of course, more investigation could always be done into applications of the Connection Machine to better ascertain what tasks it provides advantages in. Perhaps a derivative architecture could be designed that fixes the problems outlined in Section 5.6. However, the machine's fundamental problems, as well as the fact that nobody has tried to build a derivative, make me sceptical that such architectures will ever be useful in solving real world problems.

[1]`https://www.softwarepreservation.org/projects/LISP/starlisp/sim/`

[2]CM-2 examples are more numerous, including at least one in Sweden [6]

Appendix A

Program Text

Connection Machine programs look very different from traditional sequential programs. Below is source code for the example programs used in Chapter 5, and the interface header for `libcm`.

A.1 Feynman's Logarithm Algorithm

```
#include "connection_machine.h"

#include <stdio.h>

#include <stdlib.h>

#include <time.h>


const uint32_t table[32] = /* Source: Wolfram Alpha */

{

  2147483648, // 0

  1256197405, // 1

   691335320, // 2

   364911162, // 3

   187825021, // 4

    95335645, // 5

    48034513, // 6
    24110347, // 7
    12078627, // 8
     6045200, // 9
     3024074, //10
     1512406, //11
      756295, //12
      378171, //13
      189091, //14
       94547, //15
       47274, //16
       23637, //17
       11819, //18
        5909, //19
        2955, //20
        1477, //21
         739, //22
         369, //23
         184, //24
          92, //25
          46, //26
          23, //27
          12, //28
           6, //29
           3, //30
           1  //31
};

/* Algorithm src: Marcus Ritt
```

```
 * (https://cstheory.stackexchange.com/users/2150/marcus-ritt)
 * Feynman and the logarithm, URL (version: 2010-12-03):
 * https://cstheory.stackexchange.com/q/3469
 */

int main()
{
  /* First things first is to load the numbers to be processed into the cells
   * of the connection machine. I'll use 16 32 bit numbers, each one greater
   * than 2^31 so it represents 1.something. These will be randomly
   * pregenerated (as well as the extreme cases). These will be placed in the
   * last 32 bits of memory of the processors on the first chip
   */

  cm *machine = cm_build();
  srand(time(NULL));

  Cell *proc;
  uint32_t i;
  for (i = 0; i < 65536; i++)
  {
    proc = machine->chips[i >> 4]->cells[i & 4];
    proc->memory[511] = random() & 255;
    proc->memory[510] = random() & 255;
    proc->memory[509] = random() & 255;
    proc->memory[508] = random() & 255;
  }
  printf("Numbers loaded in\n");
```

```
/* For the sake of demonstration, we can transfer the 1024 bit log table
 * into every cell one bit at a time, so the other processors can play
 * along in trying to log 0.
 */
for (i = 0; i < 1024; i++)
{
  uint32_t bit = (table[i >> 5] >> (31 - (i & 31))) & 1;
  cm_exe(machine, i, 0, 0, 0, 0, 0, (bit) ? SETO : SETZ, IDF, 0);
}
printf("Table loaded in\n");

/* Designate space for the other integers required. r will live at 1024-
 * 1055. k will be stored here in the host. tempr will be stored at 1056-
 * 1087, and finally logs at  1088-1119. Memory is initiallised to 0 in the
 * simulation, but for completeness on more unpredictable hardware, I will
 * write 0s in the appropriate places
 */
cm_exe(machine, 1024, 0, 0, 0, 0, 0, SETO, IDF, 0);
for (i=1025; i < 1120; i++)
  cm_exe(machine, i, 0, 0, 0, 0, 0, SETZ, IDF, 0);

/* Now for the actual main loop */
uint32_t k;
for (k = 1; k < 32; k++)
{
  /* The first step of this is to calculate tempr. This requires shifting r
   * into tempr, then adding r. Shifting is fairly easy, but we do need to
 * backill with 0s if necessary.
 */
for (i = 0; i < 32; i++) // tempr = r >> kShift
{
  if (i < k) cm_exe(machine, 1056+i, 0, 0, 0, 0, 0, SETZ, IDF, 0);
  else cm_exe(machine, 1056+i, 1024+i-k, 0, 0, 0, 0, CPM, IDF, 0);
}
/* Next comes the addition. By setting flag 8 to 0 and using it as a
 * carry flag addition can be done in 32 instructions in a for loop.
 */
cm_exe(machine, 0, 0, 0, 8, 0, 0, IDM, SETZ, 0);
for (i = 0; i < 32; i++) //tempr = r + (r >> k)
{
  cm_exe(machine, 1087-i, 1055-i, 8, 8, 0, 0, XOR, MAJ, 0);
  /* This effectively simulates a ripple carry adder */
}


/* Now, compare tempr to s. This can be done with 2 flags and iterating
 * over the number big end first. Both flags are initially high, and the
 * instructions are conditioned on  flag a being high. If there is a bit
 * in s bigger than the bit in tempr, then flag a goes down, s is bigger,
 * stop executing. If there's a smaller bit, flag b goes down, but we
 * keep on executing as everything is conditioned on flag a, so flag a
 * can eventually go down. But notably, whilst flag a can go down
 * arbitrarily, flag b goes down if and only if s is smaller than tempr,
 * so if tempr <= s, b will be high at the end.
 *
 * First, set flags 8 and 9 high to be a and b respectively.
```

```
   */

  cm_exe(machine, 0, 0, 0, 8, 0, 0, IDM, SETO, 0);
  cm_exe(machine, 0, 0, 0, 9, 0, 0, IDM, SETO, 0);

  /* Then do the iteration */
  for (i = 0; i < 32; i++) // if (tempr <= k)
  {
    /* 8 goes down if a bit in s is bigger */
    cm_exe(machine, 1056+i, 4064+i, 8, 8, 8, 1, IDM, 0b01000101, 0);
    /* 9 goes down if a bit in s is smaller */
    cm_exe(machine, 1056+i, 4064+i, 9, 9, 8, 1, IDM, 0b01010001, 0);
  }

  /* Now, if 9 is high, tempr <= s so we should add to logs and change r */
  cm_exe(machine, 0, 0, 0, 8, 0, 0, IDM, SETZ, 0);
  for (i = 0; i < 32; i++) //logs += table[k]
  {
    cm_exe(machine, 1119-i, (k << 5) + 31 - i, 8, 8, 9, 1, XOR, MAJ, 0);
  }

  for (i = 0; i < 32; i++) //r = tempr
  {
    cm_exe(machine, 1024+i, 1056+i, 0, 0, 9, 1, CPM, IDF, 0);
  }
}
cycles();
return 0;
```

```
}
```

A.2 Vector Multiplication

```
#include "connection_machine.h"

#include <stdio.h>

#include <time.h>

#include <stdlib.h>



int main()

{

  /* Lets set up 2 16 vectors. These will be set up to be on different

   * routers, on processor 0 of routers 0 through 15 for the left vector, 16

   * through 32 for the right vector. Note that we can really set these out

   * in any way so long as they're sufficiently structures to allow the

   * conversations to occur. Vector 0 will have a 0 in bit 0, whereas vector

   * 1 will have a 1 to differentiate them. To prevent spamming, both vectors

   * will have bit 1 set high

   */


  /* In the big version, let's do a similar thing, but spam give 2

   * 2048-vectors with rng 8 bit nos

   */


  cm *machine = cm_build();

  shouldntOr(machine);

  slowMode(machine);
```

```
shouldntDump(machine);

srand(time(NULL));

Cell *proc;
uint32_t i;
for (i = 0; i < 2048; i++)
{
  proc = machine->chips[i]->cells[0];
  proc->memory[0] = 1 << 6;
  proc->memory[511] = rand() % 256;

  proc = machine->chips[2048+i]->cells[0];
  proc->memory[0] = (1 << 7) | (1 << 6);
  proc->memory[511] = rand() % 256;
}

/* First, each item in vector 1 must send a message to the corresponding
 * entry in vector 0 with its value. CONVINIENTLY, the message length is
 * precisely the length of the 32 bit numbers we're storing! First, we need
 * to get whether we're going to send or not into a flag. This is gonna be
 * innefficient as it requires a petitSync, but in the context of a larger
 * system, it doesn't really matter.
 */

cm_exe(machine, 0, 1, 0, 15, 0, 0, IDM, 0b00000011, 0);
petit_sync(machine);
```

```
/* Now actually signal to send the message */
cm_exe(machine, 0, 0, 0, 5, 15, 1, IDM, SETO, 0); /*Sets a 1 in the router
                                                    data flag*/

/* The address of the message should be set so that only bit 5 of the
 * router address is high. There should be 7 low bits, a 1, then 8 0s
 */
cm_exe(machine, 0, 0, 0, 5, 15, 1, IDM, SETO, 0);
for (i = 0; i < 15; i++) cm_exe(machine, 0, 0, 0, 5, 15, 1, IDM, SETZ, 0);

/* The next bit is the always 1 formatting bit */
cm_exe(machine, 0, 0, 0, 5, 15, 1, IDM, SETO, 0);

/* Now we can start to pass in the actual data over the next 32 cycles.
 * We'll also comandeer bit 2 to act as a parity bit
 */
for (i = 0; i < 32; i++)
  cm_exe(machine, 2, 4064+i, 0, 5, 15, 1, XOR, CPM, 0);

/* Finally, add in the parity bit */
cm_exe(machine, 2, 2, 0, 5, 15, 1, IDM, IDM, 0);

/* Now, we can drop cycles until the message is delivered */
while (! machine->globalPin)
  cm_exe(machine, 0, 0, 5, 1, 0, 0, IDM, IDF, 0);

/* Listen for the incoming message, and copy the contents into memory */
// cm_exe(machine, 0, 0, 5, 15, 0, 0, IDM, IDF, 0);
```

```
for (i = 0; i < 32; i++)
  cm_exe(machine, 4032+i, 4032+i, 5, 0, 0, 0, IDF, SETO, 0);

/* Next is the actual multiplication operation. I'll use peasant
 * multiplication for this, overflowing into 64 bits, starting 3968.
 * However, only the lower 32 bits will be sent for addition due to test
 * numbers being sufficiently small.
 */

for (i = 0; i < 32; i++)
{
  /* Decide whether to actually run this cycle, based on the bit in the
   * original number
   */
  cm_exe(machine, 4095-i, 4095-i, 0, 15, 0, 0, IDM, IDM, 0);
  cm_exe(machine, 0, 0, 0, 14, 15, 1, IDM, SETZ, 0); /* Carry Flag low*/

  /* Then, perform the addition if required */
  uint32_t j;
  for (j = 0; j < 32; j++)
  {
    cm_exe(machine, 4031-i-j, 4063-j, 14, 14, 15, 1, XOR, MAJ, 0);
  }
}

/* Finally, we can start the collapsing addition. This can be done by
 * having each vec0 entry send on a dimension each turn its current running
 * value, add them up, then repeat for all 4 dimensions relevant. Each one
 * will share the running total.
 *
 * Due to the numbers being sufficiently small, the incoming value can be
 * stored at 3968. Then they can be added to the running 32 bit total at
 * 4000. Vec0s already have flag 15 high
 *
 * First, it will be necessary to get vec0 to have flag 15 set high. This
 * is basically just setting flag 15 to the and of bits 0 and 1
 */

cm_exe(machine, 0, 1, 0, 15, 0, 0, IDM, 0b00110000, 0);
  //maj is true if at least 2 are true. 0 is always low
for (i = 0; i < 10; i++)
{
  petit_sync(machine);
  cm_exe(machine, 2, 2, 0, 5, 15, 1, SETO, SETO, 0);

  uint32_t j;
  for (j = 0; j < 16; j++)
    cm_exe(machine, 0, 0, 0, 5, 15, 1, IDM, j == 11-i ? SETO : SETZ, 0);

  cm_exe(machine, 2, 2, 0, 5, 15, 1, SETZ, SETO, 0);

  for (j = 0; j < 32; j++)
    cm_exe(machine, 2, 4000+j, 0, 5, 15, 1, XOR, CPM, 0);

  cm_exe(machine, 2, 2, 0, 5, 15, 1, IDM, CPM, 0);
```

```
    /* Now, we can drop cycles until the message is delivered */
    //printf("%u\n", i); sleep(3);
    while (! machine->globalPin)
      cm_exe(machine, 0, 0, 5, 1, 0, 0, IDM, IDF, 0);

    for (j = 0; j < 32; j++)
      cm_exe(machine, 3968+j, 3968+j, 5, 14, 15, 1, IDF, SETZ, 0);

    for (j = 0; j < 32; j++)
      cm_exe(machine, 4031-j, 3999-j, 14, 14, 15, 1, XOR, MAJ, 0);
  }

  cycles();
  return 0;
}
```

A.3 Breadth First Search

```
/* Basic outline of the algorithm:
 * Each processor will function as the vertex of a graph. The first 1024 bit
 * will make up 64 16 bit pointers to other nodes in the graph (if we want to
 * make a less dense graph, we can just stop generating random pointers and
 * replace the later pointers with self loops). The next 64 bits will act as
 * a bitmap - initally all set to 1. The first pointer will be self loop for
 * easiness
 *
 * Conceptually, there will be 3 sets of processors, forming a partition of
 * all  processors. Name these undiscovered (N),  active (A), and done (D).
* These can be represented by bits in flags
* 8 and 9 (N.B as they're mutex, both off means its in D).
* Initially, A = {0}, N = everything\{0}, D = {}.
*
* Computation proceeds in rounds. Each round can be roughly described in
* pseudocode:
* for i from 0 to 63:
*   if bitmap(i) = 1 and proc in A:
*     attempt to send message 1 << 32 + self address to router at pointer i
*     if successful: bitmap(i) = 0
*
*   if proc in N and inbox(0) = 1:
*     copy inbox to longstore
*     inbox(0) = 0
*
*   wait until delivery stage
*
*   if proc in N:
*     copy message into inbox
*
* if bitmap is all 0 in all processors in A:
*   wait for outstanding messages to be delivered
*   D = D u A
*   A = all processors in N with longstore(0) = 1
*   N = N\{A}
*   next round
* else:
*   repeat the round
```

```
 *
 * The whole thing terminates when all processors are in D.
 */

#include "connection_machine.h"
#include <stdio.h>
#include <stdint.h>
#include <time.h>
#include <stdlib.h>

#define DEGREE 8

int main()
{
  /* First, lets set up the processors with random pointers */
  cm *machine = cm_build();
  shouldntOr(machine);
  slowMode(machine);
  shouldntDump(machine);

  srand(time(NULL));

  Cell *proc;
  uint32_t i, j;
  for (i = 0; i < 65536; i++)
  {
    proc = machine->chips[i >> 4]->cells[i & 15];
    proc->memory[0] = i >> 8;
  proc->memory[1] = i & 255;

  for (j = 2; j < ((2 * DEGREE) + 2); j++)
  {
    proc->memory[j] = rand() % 255; //Note these are relative addresses
  }

  //The next 64 bits need to be 1s. That's 8 bytes
  for (j=0; j<8; j++) proc->memory[128+j] = 255;

  //Lets also put the processors into the undiscovered set here too
  proc->flags = 1 << (15-8);
  // printf("%u\n", i);
}


//We put processor 0 in the active set.
machine->chips[0]->cells[0]->flags = 1 << (15-9);

//And with that, we're good for round 1!

uint32_t d = 0;
uint32_t a = 1;
uint32_t n = 65535;

while (a)
{
  for (i = 1; i < DEGREE + 1; i++)
```

```
{
  petit_sync(machine);

  //////////SENDING PHASE\\\\\\\\\\

  //Copy the bit from the bitmap into the message - indicate we want to
  //send a message if it’s unsent
  cm_exe(machine, 1024+i, 1024+i, 0, 5, 9, 1, IDM, IDM, 0);

  //Now copy the relative address
  for (j = 0; j < 16; j++)
  {
    cm_exe(machine, (16*i)+j, (16*i)+j, 0, 5, 9, 1, IDM, IDM, 0);
  }

  //Format bit
  cm_exe(machine, 0, 0, 0, 5, 9, 1, IDM, SETO, 0);

  //Comandeer bit 4095 as xor bit. Copy in the relative ROUTER address of
  //the pointerplus the absolute CELL adress. But first, 1 then 15 0s.
  cm_exe(machine, 4095, 4095, 0, 5, 9, 1, SETO, SETO, 0);
  for (j = 0; j < 15; j++)
    cm_exe(machine, 0, 0, 0, 5, 9, 1, IDM, SETZ, 0);

  for (j = 0; j < 12; j++)
  {
    cm_exe(machine, 4095, (16*i)+j, 0, 5, 9, 1, XOR, CPM, 0);
  }
```

```
for (j = 0; j < 4; j++)
  cm_exe(machine, 4095, 12+j, 0, 5, 9, 1, XOR, CPM, 0);

cm_exe(machine, 4095, 4095, 0, 5, 9, 1, IDM, IDM, 0);

//We now need to check the handshake and update the bitmap. The bitmap
//will be set to the and of the handshake and its current value.
cm_exe(machine, 1024+i, 0, 4, 0, 9, 1, 0b00001010, IDF, 0);

//////////WAITING PHASE\\\\\\\\\\

//The inbox will be the last 32 bits, 4064-4095. The longstore will be
//4032-4063. First work out if we should even copy. This is if the
//processor is in N and 4064 is 1. Copy this into scratch flag 15.
cm_exe(machine, 4064, 4064, 0, 15, 8, 1, IDM, IDM, 0);

//Then, conditioned on flag 15, we copy into the longstore
for (j = 0; j < 32; j++)
  cm_exe(machine, 4032+j, 4064+j, 0, 0, 15, 1, CPM, IDF, 0);

//Finally, scrub the bit on the inbox
cm_exe(machine, 4064, 0, 0, 15, 15, 1, SETZ, SETZ, 0);

//And now, we wait!
uint16_t timeout = 0; //Timeout required as occasionally due to repeats
                      //no message will be sent
if (! network_empty(machine)) printf("No messages sent!\n");
```

```
  else
  {
    while ((! machine->globalPin) && timeout < 10000)
    {
      cm_exe(machine, 0, 0, 5, 1, 0, 0, IDM, IDF, 0);
      timeout++;
    }
    if (timeout == 10000) printf("TIMED OUT!!\n");
  }

  //////////RECEIVING PHASE\\\\\\\\\\

  //Very simple - just copy the message directly into the inbox!
  for (j = 0; j < 32; j++)
    cm_exe(machine, 4064+j, 4064+j, 5, 0, 8, 1, IDF, SETO, 0);
}

//////////CONTROL STAGE\\\\\\\\\\

//WE NEED TO COPY TO LONGSTORE HERE orelse the last one won't send :(
cm_exe(machine, 4064, 4064, 0, 15, 8, 1, IDM, IDM, 0);

//Then, conditioned on flag 15, we copy into the longstore
for (j = 0; j < 32; j++)
  cm_exe(machine, 4032+j, 4064+j, 0, 0, 15, 1, CPM, IDM, 0);

//Finally, scrub the bit on the inbox
cm_exe(machine, 4064, 0, 0, 0, 15, 1, SETZ, IDM, 0);
```

```
//Now all messages have had a chance to send. We can evaluate to see if
//we should go into the next round or repeat this one.

//Firstly, we'll evaluate whether the round was completed successfully.
//This walks over the bitmaps in As and ors them all. If any are 1, we
//must go again, taking care to not drop a message

cm_exe(machine, 4095, 4095, 0, 0, 9, 1, SETZ, IDF, 0);
for (i = 0; i < DEGREE; i++)
  cm_exe(machine, 4095, 1025+i, 0, 0, 9, 1, OR, IDF, 0);
cm_exe(machine, 4095, 4095, 0, 1, 9, 1, IDM, IDM, 0);

//We can now check the global pin to see if we're done.
if (machine->globalPin) //NOT done
{
  //Repeat waiting and receiving phase

  //The inbox will be the last 32 bits, 4064-4095. The longstore will be
  //4032-4063.First work out if we should even copy. This is if the
  //processor is in N and 4064 is 1. Copy this into scratch flag 15.
  cm_exe(machine, 4064, 4064, 0, 15, 8, 1, IDM, IDM, 0);

  //Then, conditioned on flag 15, we copy into the longstore
  for (j = 0; j < 32; j++)
    cm_exe(machine, 4032+j, 4064+j, 0, 0, 15, 1, CPM, IDM, 0);

  //Finally, scrub the bit on the inbox
  cm_exe(machine, 4064, 0, 0, 0, 15, 1, SETZ, IDM, 0);

  //And now, we wait!
  uint16_t timeout = 0;
  if (! network_empty(machine)) printf("No messages sent!\n");
  else
  {
    while ((! machine->globalPin) && timeout < 10000)
    {
      cm_exe(machine, 0, 0, 5, 1, 0, 0, IDM, IDF, 0);
      timeout++;
    }
    if (timeout == 10000) printf("TIMED OUT!!\n");
  }

  //////////RECEIVING PHASE\\\\\\\\\\

  //Very simple - just copy the message directly into the inbox!
  for (j = 0; j < 32; j++)
    cm_exe(machine, 4064+j, 4064+j, 5, 0, 8, 1, IDF, SETO, 0);

  //Repeat the whole round
  continue;
}

//We now need to check for outstanding messages. I thought for a LONG
//time about the best way to do this, including stack and cascading
//addition based solutions, but the architecture of the connection
```

```
//machine just makes this VERY awkward. Instead, I'm going to cheat, and
//assume there's a line that is set low when all routers are empty. I'll
// implement this as a function in both the router and the connection
//machine libraries. This simply while there are still messages in the
//system, run the waiting and receiving phases

while (network_empty(machine))
{
  //The inbox will be the last 32 bits, 4064-4095. The longstore will be
  //4032-4063. First work out if we should even copy. This is if the
  //processor is in N and 4064 is 1. Copy this into scratch flag 15.
  cm_exe(machine, 4064, 4064, 0, 15, 8, 1, IDM, IDM, 0);

  //Then, conditioned on flag 15, we copy into the longstore
  for (j = 0; j < 32; j++)
    cm_exe(machine, 4032+j, 4064+j, 0, 0, 15, 1, CPM, IDM, 0);

  //Finally, scrub the bit on the inbox
  cm_exe(machine, 4064, 0, 0, 0, 15, 1, SETZ, IDM, 0);

  //And now, we wait!
  uint16_t timeout = 0;
  if (! network_empty(machine)) printf("No messages sent!\n");
  else
  {
    while ((! machine->globalPin) && timeout < 10000)
    {
      cm_exe(machine, 0, 0, 5, 1, 0, 0, IDM, IDF, 0);
      timeout++;
    }
    if (timeout == 10000) printf("TIMED OUT!!\n");
  }


  //////////RECEIVING PHASE\\\\\\\\\\

  //Very simple - just copy the message directly into the inbox!
  for (j = 0; j < 32; j++)
    cm_exe(machine, 4064+j, 4064+j, 5, 0, 8, 1, IDF, SETO, 0);
}

//Cool, we're here, which means the round is actually done! That means we
//just need to redo our sets, check if we're done, then move on to the
//next round/terminate as appropriate!

//First, put all processors from A into D
cm_exe(machine, 0, 0, 0, 9, 9, 1, IDM, SETZ, 0);

//Now, put correct processors from N with  into A and remove from N
cm_exe(machine, 4032, 4032, 0, 9, 8, 1, IDM, IDM, 0);
cm_exe(machine, 0, 0, 0, 8, 9, 1, IDM, SETZ, 0);

//Now, we can break if all processors are in D
cm_exe(machine, 4095, 4095, 8, 0, 0, 0, IDF, IDF, 0);
cm_exe(machine, 4095, 4095, 9, 1, 0, 0, IDM, OR, 0);
if (! machine->globalPin) break; //Done!!
```

```
    //else, some proc is not in D, so continue with the new round.
    //abort();
    d = 0;
    a = 0;
    n = 0;
    for (i = 0; i < 4096; i++)
    {
      for (j = 0; j < 16; j++)
      {
        uint16_t flags = machine->chips[i]->cells[j]->flags;
        if ( (flags >> 6) & 1) {
          a++; //Flag 9 set, in A
          // printf("%u\n", (i << 4) | j);
        }
        else if ( (flags >> 7) & 1) n++; //Flag 8 set, in N
        else d++;
      }
    }
  }
  cycles();
}
```

A.4 libcm.h

```
#ifndef CM_CM_H_
#define CM_CM_H_

#include "chip.h"
```

```
typedef struct
{
  Chip *chips[1 << DIMENSIONS];
  uint32_t petitCounter;
  uint8_t shouldOr;
  uint8_t slowMode;
  uint8_t globalPin;
  uint8_t dump;
} cm;

cm *cm_build();

void cm_del(cm *machine);

void cm_exe(cm *machine, uint16_t addrA, uint16_t addrB, uint8_t flagR,
            uint8_t flagW, uint8_t flagC, uint8_t sense, uint8_t memTruth,
            uint8_t flagTruth, uint8_t newsDir);

uint8_t shouldOr(cm *machine);

uint8_t shouldntOr(cm *machine);

uint8_t slowMode(cm *machine);

uint8_t fastMode(cm *machine);

void petit_sync(cm *machine);
```

```
uint8_t shouldDump(cm *machine);


uint8_t shouldntDump(cm *machine);


int network_empty(cm *machine);


void cycles();


/* Also define some useful truth tables for instructions */


#define AND 0b00000001

#define OR 0b01111111

#define XOR 0b01101001

#define IDM 0b00001111 //IDentity Memory, returns value in addrA

#define IDF 0b01010101 //IDentity Flag, returns value in flagR

#define CPM 0b00110011 //CoPy Memory, returns value in addrB

#define MAJ 0b00010111

#define SETO 0b11111111

#define SETZ 0b00000000


#endif
```

Bibliography

[1] Marcus Ritt (https://cstheory.stackexchange.com/users/2150/marcus-ritt). *Feynman and the logarithm.* Theoretical Computer Science Stack Exchange. URL:https://cstheory.stackexchange.com/q/3469 (version: 2010-12-03). Dec. 2010. eprint: `https://cstheory.stackexchange.com/q/3469`. URL: `https://cstheory.stackexchange.com/q/3469`.

[2] *ATSC Standard: Video - HEVC.* Standard. Accessed 11-5-23. Washington, D.C., US: Advanced Television Systems Committee, Mar. 2023. URL: `https://prdatsc.wpenginepowered.com/wp-content/uploads/2023/04/A341-2023-03-Video-HEVC.pdf`.

[3] K. E. Batcher. "Sorting Networks and Their Applications". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference.* AFIPS '68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 307–314. ISBN: 9781450378970. DOI: `10.1145/1468075.1468121`. URL: `https://doi.org/10.1145/1468075.1468121`.

[4] Joshua Benton. *After 25 years, Brewster Kahle and the Internet Archive are still working to democratize knowledge.* `https://www.niemanlab.org/2022/03/after-25-years-brewster-kahle-and-the-internet-archive-are-still-working-to-democratize-knowledge/`. Accessed 17-03-23. Mar. 2022.

[5] B. Copeland. "The Manchester Computer: A Revised History Part 2: The Baby Computer". In: *IEEE Annals of the History of Computing* 33.1 (2011), pp. 22–37. DOI: `10.1109/MAHC.2010.2`.

[6] DigitalMuseum. *Parallelldator*. Accessed 01-05-23. URL: `https://digitaltmuseum.se/021027765253/parallelldator`.

[7] T. Alan Egolf. “Scientific Application of the Connection Machine at the United Technologies Research Center”. In: *Proceedings of the Conference on Scientific Application of the Connection Machine*. NASA Ames Research Centre, Moffett Field, California: World Scientific Publishing Co. Pte. Ltd., Sept. 1988, pp. 38–63. ISBN: 9971509695.

[8] W. Daniel Hillis. “Richard Feynman and the Connection Machine”. In: *Physics Today* 42 (2 1989), p. 78. DOI: `https://doi.org/10.1063/1.881196`.

[9] W. Daniel Hillis. *The Connection Machine*. ACM Distinguished Theses. Cambridge, Massachusetts: The MIT Press, 1985. ISBN: 0262081571.

[10] Intel. “Intel® 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture”. In: Accessed 24-3-23. Dec. 2022, pp. 5–22. URL: `https://cdrdv2.intel.com/v1/dl/getContent/671200`.

[11] Brewster A. Kahle and W. Daniel Hillis. “The Connection Machine Model CM-1 Architecture”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 19.4 (1989), pp. 707–713. DOI: `http://dx.doi.org/10.1109/21.35335`.

[12] Bruno Lopes et al. *ISA aging: A X86 case study*. Accessed 11-5-23. 2013. URL: `https://www.researchgate.net/profile/Rafael-Auler/publication/260112900_ISA_Aging_An_X86_case_study/links/0f31752f9ceee27ead000000/ISA-Aging-An-X86-case-study.pdf`.

[13] John Markoff. *U.S. Awards Computer Contract: Thinking Machines Gets $12 Million to Develop Faster Supercomputer*. English. Copyright - Copyright New York Times Company Nov 29, 1989; Last updated - 2010-05-22. Nov. 1989. URL: `https://www.proquest.com/historical-newspapers/u-s-awards-computer-contract/docview/110347811/se-2`.

[14] Computing History Museum. *Artifact Details - Connection machine 1 supercomputer frame*. Accessed 01-05-23. URL: https://www.computerhistory.org/collections/catalog/102691297.

[15] Computing History Museum. *Artifact Details - Connection Machine CM-1*. Accessed 01-05-23. URL: https://www.computerhistory.org/collections/catalog/X1124.93.

[16] Leonid Ryzhyk. "The ARM Architecture". In: *Chicago University, Illinois, EUA* (2006).

[17] Gary A. Taubes. *The Rise and Fall of Thinking Machines*. https://www.inc.com/magazine/19950915/2622.html. Accessed 17-03-23. Sept. 1995.

[18] *TOP500 LIST - JUNE 1993*. Accessed 11-5-23. June 1993. URL: https://top500.org/lists/top500/list/1993/06/.