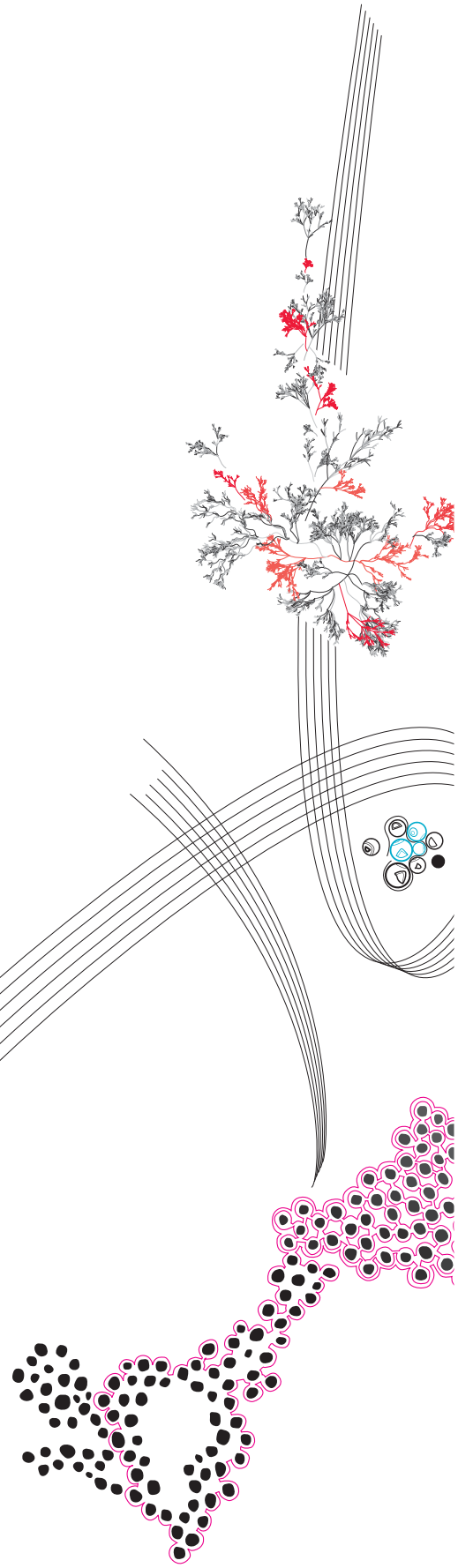


MSc Computer Science
Final Project

Mapping Hardware Descriptions to Bittide Synchronized Multiprocessors for Instruction Level Parallelism

Daan Middelkoop

Supervisors:
Dr. ir. S.H. Gerez
Prof. dr. ir. A.B.J. Kokkeler
ir. H.H. Folmer

September, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Efforts to enhance computational speed have been ongoing since the inception of processors. This research explores a way to program multiprocessors using HDL (Hardware Description Languages). A mapping has been created between HDL and synchronized multiprocessors, specifically bittide synchronized multiprocessors. The bittide network is a new approach to synchronize multiprocessors to allow for instruction-level parallelism. The central research question is *Can the bittide network be programmed using hardware description languages (HDL) in a way that exploits the available parallelism?* HDLs provide inherent fine-grained parallelism as the components in circuits lay physically next to each other and operate simultaneously. The bittide network provides the ability for instruction-level parallelism needed to exploit this fine-grained parallelism. In this research, an approach is successfully demonstrated which maps hardware to a DAG (Directed Acyclic Graph) of RISC-V instructions, which is then scheduled over a bittide network of RISC-V cores. A simulator has been implemented capable of simulating this network which was used to verify the correctness of the mapping. The mapping was able to find a high degree of parallelism in simple programs. 16x16 matrix multiplication requiring 8705 cycles on a single core could be executed in 95 cycles on a (slightly unrealistic) fully connected network of 100 RISC-V cores, meaning a 91x speedup.

1 Introduction

Researchers have been looking to increase computation speed since the first processor was developed. Throughout history many ideas have been thought of that increase the computation speed of processors. These generally fall into one of two categories, namely spatial and temporal solutions, in other words, either do more things at the same time (you need more physical hardware that is active at the same time) or use the same hardware more efficiently over time to increase computation speed. These systems must remain synchronized in order to achieve correct computation. Modern processors often have multiple cores that run completely independently and synchronization happens through various constructs, like locks and atomic operations, with significant overhead. Another possibility is to have all cores running on the same clock so the system behaves in a deterministic manner and these synchronisation constructs are not necessary.

As applications continue to demand greater performance, finding efficient and effective ways to maintain synchronisation is crucial. The bittide network [12] is a novel system designed to address this problem by offering a decentralized approach to synchronisation. Traditional hardware has a single central clock generator that provides a clock signal to the hardware. Bittide eliminates the need for physical clock distribution and provides applications with a perfectly synchronized logical clock. Instead of the physical clocks being perfectly synchronized, the system behaves as if this is the case on a logical level. How this is achieved is explained in Chapter 5.

Despite the promising features of the bittide network, there is still a lack of research on how to use this synchronized multiprocessor architecture efficiently. At the time of writing no solution exists which is capable of mapping algorithms to run on the bittide network. This research aims to bridge this gap by exploring the possibility of programming the bittide network using HDL (Hardware Description Languages) to exploit the available parallelism. This is possible due to the inherent nature of hardware, where different components are physically spread out in space and might thus expose more parallelism than traditional programming languages. The main research question guiding this study is: *Can the bittide network be programmed using hardware description languages (HDL) in a way that exploits the available parallelism?*

This research identifies a possible mapping between HDLs and the bittide network, determines appropriate benchmarks for measuring the exploited parallelism, and assesses whether this approach scales well as the size of the bittide network increases. The findings from this research could potentially open up new avenues for using HDLs in distributed systems, offering a new way to tackle the complex problem of multiprocessing.

Concretely this thesis explores the following:

- A mapping of HDLs to a graph of instructions (Chapter 6 and 8.1)
- A scheduler to run this graph on a bittide network of RISC-V cores (Chapter 8.2)
- A simulator capable of simulating a bittide network of RISC-V cores to verify the correctness of the mapping and scheduling (Chapter 8.3)

This thesis is structured as follows: In Chapter 2, background will be provided on parallelism. In Chapter 3 the research question and sub-questions of this thesis are formulated. In Chapter 4, we look into similar research and build a theoretical basis for our approach. In Chapter 5, the bittide network is explained in detail. Chapter 6 discusses the mapping from hardware to the bittide network. In Chapters 7 and 8, we explain our design choices and implementation, and in Chapter 9, we show and discuss our results.

2 Background

2.1 Parallelism

One of the inventions to increase parallelism within processors was the superscalar processor. Within superscalar CPUs, there are multiple functional units capable of executing an instruction, which are all connected to the same register bank. This allows the CPU to execute multiple instructions simultaneously if there is no dependency conflict. Within this CPU there is extra hardware that looks ahead in the instruction stream to find instructions whose dependencies are already resolved and can thus already be executed. These instructions are then already executed by the extra functional units. The CPU also performs optimizations to increase the number of instructions that can be executed in parallel, such as dynamically reordering instructions. All of the synchronization between the functional units happens in hardware and is mostly taken care of by the shared clock. As a result, the programmer/compiler does not have to be aware of this process at all, although the compiler might try to generate instructions that can be executed in parallel as much as possible. These superscalar CPUs are an example of ILP (instruction-level parallelism) which is a form of fine-grained parallelism.

Another improvement was the multi-core processor. By introducing multiple independent cores (with their own register banks and functional units) entire execution paths can be executed in parallel. Since these execution paths are independent and cores might arbitrarily stall for many reasons, these cores will not run in sync, and thus, necessary synchronization has to be applied which can be quite costly time-wise. Due to this synchronization and extra communication overhead, it is infeasible to use instruction-level parallelism on this system. Instead, Thread level parallelism is used which is a form of coarse-grain parallelism. The design of these threads and which code runs in which thread often ends up falling on the programmer, which is a complicated task.

To execute a program faster, as much of the parallelism within this program should be exploited as possible. If this is done at a finer level, more of the parallelism might be exposed which can then be exploited. Therefore it is advantageous to look for ways to increase the exploitation of fine-grained parallelism.

A recent competitor for fine-grained parallelism is Google's bittide synchronisation [12]. This is an abstraction over hardware to achieve a network in which there is a fixed cycle delay from node to node within the network. This could be a network between two processors on a die, or between processors in different machines and can be seen as more of a generalization of systems like the RAW machine [13]. These nodes do not even strictly have to be processors, as long as they operate on a cycle basis. These bittide nodes have only one way of communicating information and that is via the bittide network which keeps them synchronized. Thus each bittide node has its own registers, memory, and other resources required to operate. However, due to the synchronous nature of this communication network, very fine-grained parallelism can be achieved because there is no overhead for synchronization. This bittide network requires a special compiler as the behavior of each of the nodes within the network has to be synchronized with the others ahead of time. This requires an extra form of spatial scheduling next to temporal scheduling. The details of the bittide network are further explained in Chapter 5.

2.2 FPGA's and Yosys

These days there exist open-source FPGAs (Field Programmable Gate Arrays) and open-source tooling to program those FPGAs. FPGAs are integrated circuits that can be con-

figured after manufacturing. This is usually done using hardware description languages. Yosys [20] is one of the open-source tools that can synthesize HDLs to configure FPGAs. On top of being able to synthesize for FPGAs, Yosys is also capable of transforming HDLs into graph representations and optimizing/altering these graphs. For example, changing Verilog process definitions into multiplexers and registers. Furthermore, Yosys can change the HDL into LUT (Lookup Table) implementations. All of these representations can be extracted from Yosys in the form of an easy-to-use Json format. These points are convenient when taking HDL as a starting point for any project since it saves the trouble of having to write your own VHDL/Verilog interpreter. In this thesis, the outputs Yosys provides are used as a starting point for both implementations found in Chapter 8.

3 Research Question

The goal of this thesis is to investigate a new approach to programming synchronous multiprocessors, particularly the bittide network. This problem is two-fold. While numerous attempts have been made to program multiprocessors with conventional imperative languages, this thesis seeks to determine if a mapping between hardware descriptions and multiprocessors is possible. Furthermore, the specific multiprocessor will be a bittide network of RISC-V cores which is so new that at the time of writing no solution to program a bittide network exists. Given the synchronous nature of the bittide network, it is possible that the inherently parallel structure and well-defined dependencies of hardware descriptions can be mapped onto it.

The main research question and several sub-questions are:

- Can the bittide network be programmed using hardware description languages (HDL) in a way that exploits the available parallelism?
  - Does a mapping between HDLs and the bittide network exist?
  - What are appropriate benchmarks to measure the exploited parallelism?
  - Is the approach able to exploit the available parallelism found in hardware descriptions?
  - Does the approach scale well when the size of the bittide network increases?

By addressing these sub-questions, insight should be gained into the feasibility of the approach to use HDLs to program the bittide network or even similar multiprocessing models.

4 Literature Review

4.1 Similar Architectures

4.1.1 Raw machine

Although the method that the bittide network uses to achieve synchronized processing to allow more parallelism is new, the resulting platform is not. Multiple architectures exist with similar behavior. The RAW machine [13] is a machine with a grid of processing units, each with its own register bank and some memory, connected using a synchronous network. This network can carry a word of data in one clock cycle to a neighbor processing unit. The entire architecture is driven by the same clock and thus runs synchronously. This architecture allows for ILP over multiple processors because data can be transferred between processors quickly and synchronously (most times) thus not requiring expensive synchronization mechanisms. However, in contrast to superscalar processors, dedicated compilers must be built for such architectures. This is because each processor and the network nodes require their own instruction streams that are built together instead of having a single instruction stream in which the hardware itself finds parallelism opportunities. In the case of the RAW machine, a dedicated compiler called RawCC has been developed that can exploit the fine-grained parallelism that this platform provides. RawCC can compile C code to run on the RAW machine. There are many obstacles that the compiler has to overcome to be able to do this. Because each part of a C program can alter arbitrary memory, the compiler tries to statically infer memory locations and map this memory to the processor that is operating on it. When multiple processors operate on it, the data is exchanged via the network via statically scheduled instructions, thus not requiring routing information. Sometimes the data can not be statically inferred and a dynamic asynchronous strategy is used to route the data to the processors requiring it. This requires a lot more overhead so it is important that the compiler can statically interfere as much as possible. The compiler also has to deal with branches in the program, making sure that every processor is aware of which branch the program is in. A lot of these problems are due to the nature of the C language and might be fixed by using a more appropriate language.

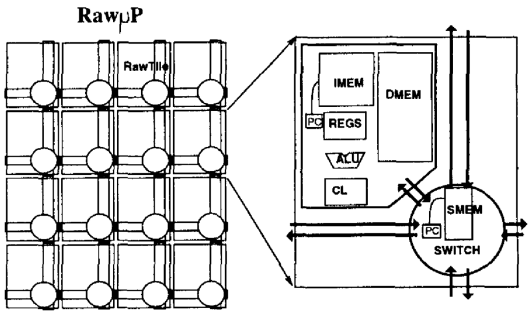


FIGURE 1: A RAW microprocessor is a mesh of tiles, each with a processor and a switch. The processor contains instruction memory, data memory, registers, ALU, and configurable logic (CL). The switch contains its own instruction memory. [13]

The RAW machine hardware side exposes a platform that is similar to a bittide network of CPUs. Both systems allow sharing data with neighbors in a fixed amount of clock cycles and can route this data on demand to different nodes (the paths that data follow can differ per cycle). Both systems can scale indefinitely without suffering clock speed penalties. There are a few key differences between the two platforms. The RAW architecture is

limited to a grid of nodes, limiting the connectivity of each node. The Bittide network is less limited as it can handle arbitrary layouts and distances between the nodes in exchange for a latency penalty. A last key difference is that the RAW machine is still managed by a single clock signal generated somewhere on the chip. Because of the decentralized clock behavior of the bittide network, it has no trouble with getting its clock signal to each node. The bittide network can even operate over multiple chips.

4.1.2 VLIW architectures

Another approach that was made was a VLIW (Very large instruction word) architecture [10]. This is an earlier attempt at instruction-level parallelism. The proposed architecture consists of clusters which each have their own registers, memory, and a few functional units for integer / floating point operations. The clusters are connected by busses which can transfer scalar values. There is a single instruction stream that controls the behavior of all of the clusters. these are very large instructions that instruct each of the functional units of each of the clusters at the same time. The behavior of this architecture is very similar to the bittide network and the RAW machine. This architecture seems less scalable due to the single instruction stream which has to reach all of the clusters. However, compiling to this architecture seems very similar to the bittide network. There are nodes that do processing and that can communicate scalar values in constant time with their neighbors, all while having a synchronized clock. There does seem to be an interesting difference with this architecture in comparison to both bittide and the RAW architecture, and that is branching. Due to the nature of a single instruction stream, every processor on this architecture always branches at the same time. There is a single program counter. It does seem to be possible to simulate individual branching by generating extra instructions, thus it is unclear whether this makes a difference in the complexity of compiling for this architecture. The paper also proposes a compiler for this architecture. A lot of the compiler again focuses on the problems associated with references that were seen with the RAW compiler as well, indicating that allowing this behavior is a bad match for such architectures.

There are many more VLIW architectures out there and different approaches for generating instruction streams. The concept they all share is that since it is only a single instruction stream, the executions on the different cores can not diverge like the Raw machine and thus the need for synchronization is not there. Another approach to a VLIW architecture is ReVAMP [6] in which they construct an in-memory computation architecture for general purpose computing. This machine is based on the 3-input majority function with one of the inputs inverted which can be used to compute any Boolean function. The paper describes a method to generate instructions from a Majority Inverted Graph. This is a directed acyclic graph that can be used to represent logic not unlike combinatorial logic in hardware descriptions. In the proposed architecture the program counter can not be manipulated, thus this machine is not capable of looping or branching.

4.2 Compilers

The instruction level parallelism available in traditional instruction streams is very limited [19] and the available parallelism can often be fully exploited by superscalar processors. Superscalar processors work by taking the instruction stream and looking ahead for instructions that can already be executed. This means that instructions very far into the stream are often not considered, for example after branches. There is another project called TRIPS [16] that tries to solve this problem by better understanding the instruction stream sent to it and thus looking much further ahead in the stream. This is done using an

approach called EDGE (Explicit Data Graph Execution) which tries to identify complete blocks of instructions with their own data and moves all of these instructions together with their data to separate cores for execution. Both the RAW and VLIW architecture come with their own compilers. Not only because the kind of instructions generated need to be very different but also because they employ techniques to extract more parallelism out of the program to justify the amount of parallelism the architectures can handle. Instead of identifying the parallelism at run-time, this is the responsibility of the compiler. The exact parallelism exploited is already determined after compiling the program. This does mean that the compiler must be fully aware of the underlying architecture and programs will only work on the exact architectures they are compiled for. It seems that to allow for more parallelism this must be considered a lot earlier already than in the final instruction stream. The aforementioned strategies focus on the formed instruction blocks and streams as a last step of compilation to improve parallelism. Another avenue is the compilation process before the translation to instructions has been made so that loops and other high-level concepts remain intact. One of these approaches is described in [9] where instead of instructions, the program is first translated into their new programming model called Asynchronous Graph Programming. Here the entire program is described as a directed graph of single assignment semantics, tracking dependencies and thus exposing parallelism more elegantly.

4.3 Languages

Parallelism can also be expressed at the source level. It is common for programs to make calls to threading libraries such as pthreads [2] as a form of explicit source-level parallelism. Of course, this assumes thread-level parallelism implemented by the programmer, while this research focuses on instruction-level parallelism. It is important to select a language that is appropriate for the job. As we have seen with the RAW machine which is quite similar to bittide, the C language has quite a few problems to deal with due to the ability to arbitrarily change any memory location from any place in the program. This even required the RAW machine to implement a hardware solution (the dynamic routing nodes) and the entire program can no longer be run synchronously since this causes arbitrary delays. Within this research, we are exploring the mapping between hardware descriptions and the bittide network. Hardware descriptions or languages similar to that (Clash [5] for example) are pure and do not suffer from all the problems that pointers cause within these systems. In [8] a new functional language called NOVA is proposed, which through the use of higher-order functions can express parallelism. All the calculations are functions from fixed inputs and thus you only have to route the inputs to the right nodes and this can always be done statically. This looks similar to the approach of the TRIPS architecture in which they try to collect blocks of instructions that rely on some input data but nothing else to execute. Hardware descriptions follow this same pattern and could therefore be a good fit for such a parallel system.

4.4 Scheduling

Imperative languages are quite easily transformed into a single instruction stream. But we want to capture the available parallelism at compile time. To generate programs for the bittide network, a schedule must be generated for each of the processors within this network. One way to do this is to start by capturing all of the dependencies within a program into a dependency graph. Generating the perfect instruction schedule for completely synchronized systems falls within the category of NP-Hard problems [11]. In asynchronous

situations, you have to cut the problem up into pieces and generate an efficient schedule for each piece. Then assume that the synchronization overhead between these pieces does not affect performance too much. However, with a completely synchronous system, you have the option to not cut the problem up into pieces to find more efficient schedules. This means you now have to schedule the entire program all at once. Luckily scheduling research has advanced a lot and there have been many solutions proposed for this scheduling case. The scheduling problem at hand is that we have a set of tasks (instructions) and a set of cores on which these tasks can be executed. Then there is a communication delay between these cores which must be taken into account. The execution time for communication is not homogeneous and the execution time for tasks does not have to be. What we know is that the system is synchronous and therefore deterministic thus the execution times for tasks are constant. One of the popular algorithms for this problem is *Heterogeneous earliest finish time*, which belongs to the class of list-based scheduling algorithms.

4.4.1 Heterogeneous earliest finish time

The HEFT algorithm [18] is a heuristic that minimizes the total execution time of a task set. It consists of two phases. First the task prioritization phase. In this phase, each task within the task set is assigned a priority. This priority is determined by the distance of the task in the graph to its furthest leaf node. Thus the task that has the most generations of children coming after it gets the highest priority. Then it sorts the tasks by priority, from highest priority to lowest. Next is the processor selection phase, in which the tasks will be taken in order of their priority and tried on each processor. The processor on which the task finishes earliest (this includes communication time for the data to be transferred to this processor) is then selected and the task will be permanently scheduled on this processor, continuing with the next task until all tasks are scheduled. This algorithm is very fast and generally performs reasonably well. This algorithm does not consider its children when selecting a processing core, thus sometimes resulting in very obviously bad schedules. Since the algorithm is already so fast, alterations have been made that consider the children of a task as well during processor selection, or even altering the order in which tasks will be scheduled, resulting in often significantly better schedules. This improvement is proposed in [7]. Another sibling algorithm of HEFT is Predicted Earliest Finish Time (PEFT) as described in [4]. This algorithm improves HEFT by implementing an optimistic cost table to enable look-ahead scheduling while maintaining the same time complexity.

4.4.2 Critical path on a processor

CPOP (Critical path on a processor) is another algorithm originally proposed in [18]. This algorithm finds the critical path of the directed acyclic graph, that is the longest path from an entry node (node without dependencies) to an exit node (node that other nodes do not depend on) in the graph. The minimum length of the produced schedule is bound by the length of the critical path. The algorithm then proceeds to schedule each of the nodes in this critical path to the same processor, eliminating communication delays in this path and thus minimizing its length. Nodes that are not in this path will be selected from a priority queue that prioritizes the node with the longest distance from an entry node plus the longest distance from an exit node. The selected node will be placed on a processor that minimizes its completion time.

4.4.3 Task duplication

Another scheduling approach is based on task duplication. The algorithm proposed in [3] considers the possibility of duplicating nodes in case the communication overhead is larger than the computation cost. This algorithm again considers the critical path but also tries to schedule other nodes on which the critical path depends efficiently, possibly duplicating them.

4.4.4 Other algorithms

There exist many other scheduling algorithms. Each of them works differently. In [17] various scheduling algorithms are compared. These algorithms employ different heuristics and techniques including genetic algorithms, Monte Carlo simulations, and various heuristics similar to HEFT.

4.5 Evaluation

Evaluation of the approaches in this thesis is not straightforward. Real-world applications have years of micro-optimisations behind them making it difficult to outperform them without applying every possible optimization as well, which due to a lack of time is infeasible for this thesis. Thus evaluation has to be done on a more theoretical basis. Many papers concerning parallelism focus on the speed-up of certain applications when parallelism is applied. In the RAW paper, a collection of benchmarks has been run on the MIPS processor (a sequential processor without any parallelism) versus the RAW platform. Then the speedup in cycles is measured and compared to arrive at a performance increase/decrease. Although a benchmark against the state of the art would be more beneficial, this still provides valuable data on the exploited parallelism and could indicate similar performance gains when applying similar strategies to the state of the art. As there are many different task loads, each exposing different amounts of parallelism, it is also important to consider a broad task set in these benchmarks. Both the RAW paper and the VLIW paper considered matrix multiplication and other matrix-based algorithms as some of their benchmarks as there is a lot of parallelism to be exploited. In [15] and [14], again multiple of the benchmarks are matrix based. Other algorithms like sorting or prime number generation could also provide interesting results. In [7] the proposed graph scheduling algorithms are evaluated using various shapes of directed acyclic graphs. These are hypothetical graphs of what calculations could look like and could be applicable bench-marking, because for evaluation it does not matter what the computed results mean, as long as the computations are representative of real-world applications, and each different graph in that paper represents a different kind of calculation.

In [13] results are provided for matrix multiplication and Conway's Game of Life when run on the RAW machine in comparison when compiled for the MIPS processor (a processor without any parallelism). The results for this can be found in Table 1. In Chapter 9 these results will be compared against the implementation created in this thesis.

TABLE 1: Speed up on the RAW platform with the number of cores = N

	MIPS cycles	N=4	N=8	N=16	N=32
Matrix multiplication	2.01M	3.6	6.64	12.20	23.19
Conways Game of life	2.44M	3.0	6.64	12.66	23.86

Unfortunately for the VLIW architecture in [10] they present results for an ideal machine, thus they provide no information on the number of cores. Furthermore, the size of the matrices for multiplication is not specified, and can thus not be reproduced.

5 Bittide

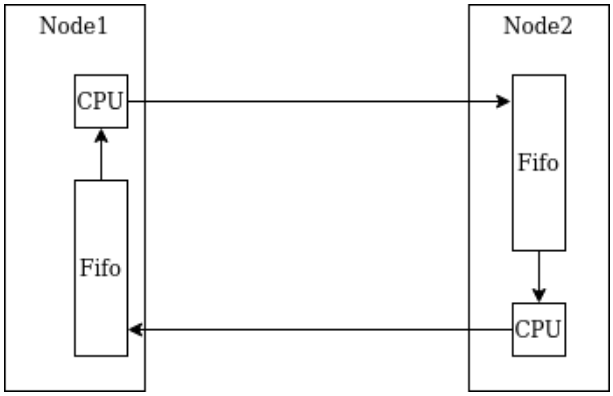


FIGURE 2: A bittide connection between two nodes

5.1 Network

The bittide network [12] describes a communication protocol that can connect different nodes in the network in such a way that they can perform synchronous computations together. The network forms a basis for a synchronized logical clock, meaning that all the nodes in the network can run in lockstep with one another. Thus if each node was a processor, all processors could step together to the next state, without one running faster than the other. This is achieved by the way the nodes communicate within the network. The core of this mechanism is formed by the elastic buffers in the network. The elastic buffer is a FIFO (first in first out) buffer that shrinks or extends depending on the speed at which items are added/removed. If there is a connection between two nodes in the network then there will be an elastic buffer on both nodes unique to this connection, an example of this can be seen in Figure 2. Each clock cycle the nodes are responsible for sending a data packet over their links towards the FIFO buffer on the other side and they must also consume a data packet from the FIFO buffer on their side. The result is that in each clock cycle the nodes exchange information. This can be information relevant to the currently running programs or not, the only important thing for the bittide network is that data is exchanged every cycle. The packets are for data communication as well as synchronization. This data can be anything but for the rest of the thesis, it is assumed that this is a scalar number of fixed bit length, which can be relevant for the program but is mostly zeros. because each node always consumes a packet from a FIFO buffer and a packet is added to the buffer by the other node, the FIFO size should remain constant. However, this is only true if the nodes run at the same clock frequency. If one node has a faster frequency and thus consumes more packets from its buffer and produces more packets for the other buffer, the buffer occupancies diverge. And the variation in the occupancy of the elastic buffers is exactly what is used to synchronize the system. If a buffer slowly drains empty, the node is running faster than it should. And if a buffer slowly fills, the node does not run fast enough and must process more packets. Thus the execution speed of a node can be adjusted when this happens. The total amount of packets in the system remains constant, since each time a processor consumes a packet it also must produce one and visa versa. Because of this property, we can use the buffer occupancy of a link to directly control the

frequency of the corresponding node. If a buffer empties this must mean that the other side fills up, thus one must slow down and the other must speed up.

5.2 Multiple nodes

Previously the case was discussed for two nodes with a single connection between them, however, this concept can be scaled up to multiple nodes. For example, a node might have two neighbors. This means that for both neighbors it has an elastic buffer and each cycle it must consume a packet from both elastic buffers, but it must also produce a packet over both links each cycle. for the control of the speed, both buffer occupancies must be taken into account. When both are empty the node must slow down, if both are full it must speed up and if one is full and the other empty, it must do nothing. In the last case, it is up to the other two nodes to slow down/speed up. Here you can see that via a shared neighbor node, the other two nodes are getting synchronized as well.

5.3 Fixed delay in communication

Even though the clock speeds of nodes might converge, an important observation is that these clocks do not have to run perfectly synchronously with each other. As long as the buffers do not empty or overflow, a logical level of synchronization has been reached already. By strictly communicating with other nodes using the network, we have access to a synchronized layer on top of this network. If the system started with 5 packets in all of the elastic buffers, there is a known communication delay of exactly 5 cycles between each node and this delay remains constant for the entire duration of the program. Even if an elastic buffer has a lower occupancy than 5, this means that this node is relatively "ahead" of the rest, which makes sense because it processed more packets already, but the current packet it is processing still was produced when the cycle counter of the other node was 5 less than this nodes counter now.

5.4 Network initialization

As the total amount of packets in the network stays constant during operation, the elastic buffers must contain some initial packets when starting. Since the buffer occupancy is directly proportional to the communication delay between nodes it makes sense to want to start with a minimum buffer occupancy, however, data is not exchanged over the link instantly, and there is a small propagation delay. Nodes physically further apart need more packets in their buffer to compensate for more propagation delay. Furthermore, nodes might have an irregular frequency (Certainly when the system starts and has not converged yet), and a larger buffer occupancy provides some leeway in these situations so that the entire system does not stall.

5.5 Gearboxing

Since all of the nodes in the bittide network have to be synchronized the entire network can only go as fast as the slowest node. To combat this issue a few exceptions can be made to the rules explained above. If one node is capable of running twice as fast as the other, it is possible to create a proportional frequency link between them. If the faster node only produces and consumes a packet every two of its cycles, then it can run twice as fast as the bittide network operates. Its frequency would synchronize to twice the frequency that the rest of the system is running on since it has to run twice as fast to consume and produce a packet. Using this principle the bittide network could operate under various interesting

conditions. For example, multiple processors on the same chip could operate using a high-frequency bittide network, while at the same time being connected over a lower-frequency bittide network to a different chip, so that the entire system is still synchronized. This concept could be extended to synchronize an entire data center together, as long as all the connections are managed by bittide links. However, this linking of different frequencies is not used in this thesis and it is assumed that all nodes operate at the same frequency.

6 Hardware on bittide

Hardware design is a complicated task. There are many considerations to be made to create "good" hardware, depending on the final platform. Whether the hardware runs on an FPGA or in another form differs. Introducing the bittide network into the mix complicates it even further. In this chapter, the created mapping from hardware to the bittide network is discussed. a few of the important considerations will be discussed when creating a mapping of hardware to a bittide network with processor-like nodes, meaning nodes that do instruction-based computation on a cycle basis. Unlike in hardware, within the bittide network wires are more of a concept and less of a physical thing. Values are stored in registers in between every computation and combinatorial paths have to be scheduled in the temporal domain as well instead of just the spatial domain. The bittide nodes used have a limited instruction set that is optimized for certain tasks. Below a few of the important considerations are discussed that arise when creating a mapping between hardware and this bittide network.

6.1 Mapping

The core idea behind the mapping relies on two interesting properties of circuits. Circuits have parallelism, all wires and components operate independently of each other in parallel. Circuits are also static, their layout is predetermined and does not change. The first property indicates that when mapping a circuit to a multiprocessor, there is parallelism to be exploited for the multiprocessor, and the second property indicates that this can be achieved by a static schedule predetermined ahead of time. In the mapping proposed in this thesis, this has been the thought process. The base components of the circuits (think of adders, multiplexers, multipliers) are mapped to CPU instructions so that the multiprocessor can execute them, and then much like FPGA place and routing, instructions have to be placed on processors and data between them routed. The core of this mapping is very simple and based on existing ideas, like scheduling and place and routing, however, due to the differences between CPUs and FPGAs, some circuits might map better to multiprocessors than others. In the rest of this chapter, different properties of different circuits will be discussed, and how they map to multiprocessors.

6.2 Pipelining

When developing hardware for an FPGA, registers are often used not only to keep state but also to pipeline the design, so that specific clock timings can be achieved. However, the constraints on these clock timings are more relaxed when targeting the bittide network. The implementation proposed in this thesis already takes care of all of the temporal scheduling of the hardware designs. It has to do this anyway to fix the communication timings within the bittide network. This does mean that the developer no longer has to consider pipelining. However, the state is still important. This leads to the observation that there are different types of registers, state registers, and others. While the proposed scheduler does not care what type of register it is, it is important to note that pipelining registers put unnecessary restrictions on the schedule and thus might harm the performance of the final schedule. This is just one example of how even though someone has experience with hardware design, they still have to learn how to design for bittide. This does raise the question of how good this mapping between hardware and the bittide network is to be used in practice. Pipelining is not the only difference here. Normally hardware developers have to focus on the combinatorial path with the longest execution time to speed up the

design, but this is also taken care of in the proposed scheduler. The scheduler will simply assign more processing power to the longest path, such that all processors are busy as much as possible and thus shortening any path will improve the design speed. In Figure 3 two versions of a very simple example hardware description are given, incrementing the input three times, and below example programs are given which could be the result of mapping this to a processor. The pipelined circuit has shorter combinatorial paths and should be able to run at a higher frequency. The produced programs, however, require equivalent processor cycles, and thus no throughput benefit can be gained with the pipelined version. Notice that in the pipelined program, the program still has to run multiple times to produce results, just like the pipelined circuit version produces results after multiple cycles.

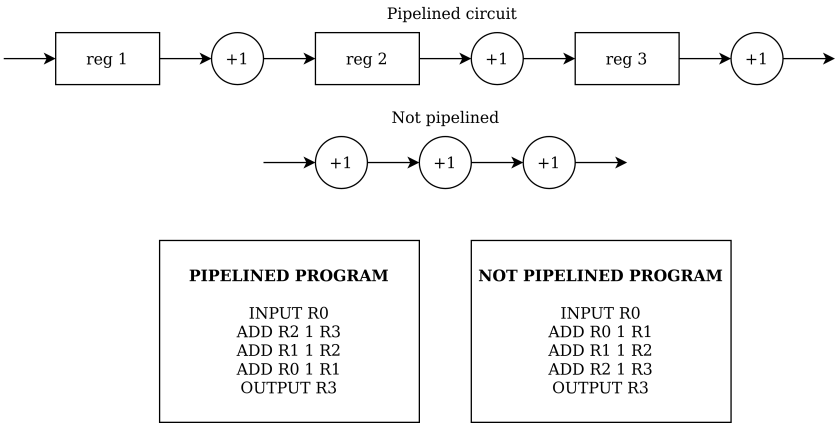


FIGURE 3: A very simple circuit, both with and without pipelining, and example programs that these circuits might be mapped to

6.3 Instruction mapping

Yet another difference between running hardware on FPGAs vs. the bittide network is the consideration of what kind of combinatorial paths should be implemented. Because an FPGA has wires and LUTs, it is trivial to arbitrarily switch and operate on a bit level. However, processor-based architectures are constrained by their instruction set. Thus implementations that focus heavily on bit "magic" might be emulated using many instructions and thus observe a significant performance decrease. It is better to focus on operations that can run natively on the provided instruction sets. This could be very counter-intuitive for FPGA engineers as using these operations (for example 32/64 bit sized operations) is often a big waste on FPGA platforms. For example, shuffling the bits of a 32-bit number to arbitrary locations would be a trivial task on an FPGA, but on a CPU it would require many instructions to isolate the bits and recombine them into the correct result.

6.4 Branching

When encountering selection logic in hardware, any of the branches could be selected, and thus each of the branches must be physically placed on the chip and thus will always be evaluated. It is easy to see how this is necessary for hardware but when running on a cycle-based processor this is different. Efficient processor code might first evaluate the selection line and based on the result only compute a single branch. This seems straightforward, but given that this is evaluated on a bittide network, execution times of the different branches

are important. One solution could be to pad branches with empty instructions to make each path the same length. Another solution could be to inform other processors which branch has been taken and to make sure that the other processors also branch into paths that comply with the new timings. Another option is to simply calculate both branches always. Calculating both branches always might seem like a performance waste, however, this allows for parallel computation of both branches and the selection line. The result is that the branches are not dependent on the selection line during scheduling. If there are idle processors, these branches could be evaluated in parallel to the selection line. This behavior is similar to branch prediction on modern processors and could increase performance. Of course, this is completely situational and the optimal strategy is probably somewhere in between these options.

7 Design choices

To produce a working prototype for a mapping of hardware to the bittide network various choices have to be made. The bittide network itself is relatively abstract and has many possible implementations. Hardware is also a broad category that has to be narrowed down. This chapter names and elaborates several choices made to constrain the implementation in order to build a real-world prototype.

7.1 Hardware description

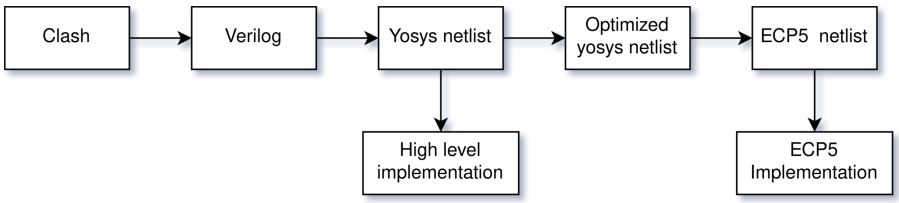


FIGURE 4: A hardware synthesis pipeline and the two places where the implementations tap in

The original idea behind this research is to apply synchronous hardware descriptions to the bittide network and see if there is a mapping. Due to this, we restrict ourselves to hardware or hardware-like descriptions as a start of our process of describing the network. Clash was considered as a starting point for the compiler but it is much more complex than the Json net-lists that Yosys can produce, thus in this research it was opted to go for that instead. However, pursuing Clash in the future could be a promising lead. Yosys was chosen due to it being open-source and a very versatile tool in converting Verilog to appropriate other models. The choice was made to restrict ourselves to Verilog code that Yosys will transform to either a high-level representation or an ECP5 synthesized version. This is visualized in Figure 4. For the high-level variant, the choice was made to optimize the Verilog description as little as possible and only convert it to a graph representation from a Json netlist. However, processes are being reduced to register transfers. The choice to alter the graph as little as possible is based on the notion that these programs optimize toward FPGA targets or other hardware architectures. These optimizations do not make much sense when targeting the bittide network.

7.2 RISC-V

The bittide network does not specify anything about how the nodes perform computation. It just describes the synchronized communication behavior between them. Nodes could be traditional processors, FPGAs, ASICs, or anything else, as long as they can somehow comply with the notion of cycles and consume and produce data at those cycles. For this research, we have opted to go for a RISC-V CPU as a node because of its simplistic nature and the existing research into scheduling for processors. There also exists a setup of a functioning bittide network with RISC-V nodes which also influenced this decision [1].

7.3 Network layout

To reduce the scope of the problem the layout of the bittide networks has been reduced to fully connected graphs of a variable number of nodes. The decision to make the network

fully connected is because the scheduling problem is already complex enough, but without a fully connected network, it would require path-finding for the data dependencies as well (To connect nodes to each other via other nodes). Furthermore, a cycle of data transmission on the bittide network does not always have to be at the same time as a CPU cycle on a node (see 5.5). The number of CPU cycles could be a multiple of the number of bittide cycles, however, in this research we have assumed them to be the same. We have opted to target networks with different buffer lengths for their bittide links so that the scheduling algorithm can be observed under low and high communication delays.

7.4 Compiler (Mapping and Scheduling)

To implement the mapping from hardware descriptions to the bittide network, it was decided to create two programs. A mapper whose purpose is to convert the hardware descriptions to RISC-V instructions with identical behavior, and a scheduler responsible for assigning each instruction to a core and a time slot, or specific CPU cycle at which the instruction executes. The scheduler works based on the HEFT algorithm, elaborated in Chapter 4.4.1. HEFT was first chosen as an initial algorithm because of its simplicity, to be later replaced by a more sophisticated algorithm, however after implementation, it became clear that this algorithm was good enough and the decision was made to focus on other parts than the scheduling. Details of both the mapper and the scheduler can be found in Chapter 8.1 and 8.2 respectively.

7.5 Simulator

To test the implementations of the compilers and schedulers, a simulator was developed that can simulate a network of RISC-V nodes. This simulator simulates the behavior of the network to the level of the elastic buffers and their contents. The simulator requires all communication between cores to go over the bittide network. It does not simulate the individual frequencies of the nodes and the slow convergence because that part is unnecessary in testing the correctness and performance of the compiled schedules. The simulator requires a program for each node, containing instructions for both the RISC-V core and to drive communication over the bittide connections. Since programs like matrix multiplication have inputs and outputs, the RISC-V instruction set is extended with two instructions, namely an input and an output instruction which the simulator will use to inject information into the simulation and produce the outputs back to the user. For the moment these input and output instructions are the only form of IO that is implemented and act like regular instructions, and can also be scheduled at any node. Input could be done in multiple ways. It might be more realistic to limit input and output nodes to certain machines in the network so that they fill the role of "data loaders" for the network. Nodes that can bridge data between the bittide world and the outside. Another option could be a form of interoperability between programs running on the bittide network so that they can pass data to each other, which would be interesting for future work.

7.6 Benchmarks

To arrive at tangible results about the performance of both the mapping and the scheduler, a few algorithms have been implemented in Verilog to benchmark everything. Four classes of problems have been selected, namely matrix multiplication, Conway's Game of Life, Prime number generation, and random instruction graphs (Here instruction graphs are directly generated and thus the mapping is bypassed). Matrix multiplication and Conway's

Game of Life have been selected because of the known parallelism that can be exploited in these operations, and because in the literature they are being compared with often, thus providing a reference point for comparison. Prime number generation has been selected as a highly sequential algorithm (The implementation used in this thesis checks primes one by one). And lastly, random graphs have been selected as a way to give more general insights into the performance for less specific problems. Implementations for all the benchmarked algorithms can be found in the appendix. All of these problems will be mapped to the bittide network and scheduled, after which the amount of cycles needed to run all instructions sequentially is measured, and the number of cycles it takes on various sizes of networks (in terms of nodes). This can then be compared against results found in the literature. The decision was made to not choose an existing benchmarking set because of three issues, First of all, since these hardware descriptions run on a bittide network of CPUs, they can not be compared against other projects using these benchmarks. As the underlying architecture would be too different, the comparison would be meaningless. Secondly, due to time constraints, not the full set of features of Verilog is usable or implemented in this thesis, thus these benchmarks would not work without adding more functionality. Lastly but related to the previous point, these benchmarks have been optimized for hardware synthesis and not to be run on a CPU, and thus the results would paint a skewed picture.

8 Implementation

In this chapter, the implementations for mapping and scheduling of the hardware descriptions onto the bittide network are explained. The implementation consists of two parts, first the mapping and then the scheduling. the mapper generates a DAG (Directed Acyclic Graph) of RISC-V instructions from the hardware descriptions. This graph then has to be scheduled onto the specific nodes of the bittide network. An initial implementation has been made that differs in many aspects from the final implementation and details and initial results of this implementation can be found in Appendix B.

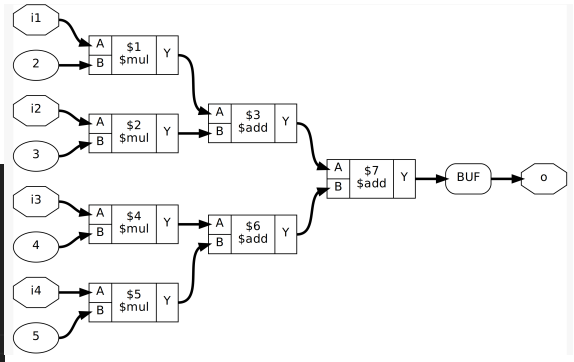
In this chapter, the terms circuit register and CPU register will be used and should not be confused with one another. A circuit register is a register specifying behavior in the hardware description that will be mapped to the bittide network, while a CPU register is a register in a RISC-V CPU within the bittide network. This CPU register stores values inside the CPU between instructions issued. In this chapter, the terms circuit cycle and CPU cycle will also be used, which also have little to do with one another. A circuit cycle refers to a single cycle in the circuit described in the hardware description and refers to a cycle in which all combinatorial logic is processed and stored inside the circuit registers. A CPU cycle on the other hand refers to a single cycle of a RISC-V CPU in which a single RISC-V instruction is executed and stored into a CPU register.

8.1 Generating the DAG

```
module adder(i1, i2, i3, i4, o);
    input [0:32] i1;
    input [0:32] i2;
    input [0:32] i3;
    input [0:32] i4;
    output [0:32] o;

    assign o = ((i1 * 2) + (i2 * 3)) + ((i3 * 4) + (i4 * 5));
endmodule
```

(A) Verilog code describing some program



(B) The same program converted to a graph by Yosys

FIGURE 5

The mapper from hardware descriptions to RISC-V instructions does not parse Verilog directly. Instead, Yosys (Chapter 2.2) is used to transform the Verilog into a netlist representation of the hardware circuits. An example of such a netlist is given in Figure 5b. This netlist consists of registers and combinatorial logic. The program in 5b does not contain registers so only combinatorial logic is shown. Except for circuit registers, these netlists are already directed acyclic graphs. Thus the compiler has to convert each of the nodes in the netlists to RISC-V instructions. For most nodes this is trivial, an $add or $mult node has to be converted to a RISC-V addition or multiplication instruction respectively. Some nodes require more complicated conversions, such as an $eq node, which should output a zero or one, based on whether the two inputs are equal or not. There is no direct instruction in the basic RISC-V instruction set that has this behavior, and thus a combination of instructions needs to be issued. In the case of $eq equal behavior

can be achieved by issuing a subtract instruction followed by a Set-Less-Then-Immediate-Unsigned instruction. By subtracting two numbers, and checking if the result is (unsigned) smaller than 1 the behavior of $eq is achieved. Another example is $mux or a multiplexer. This behavior can be implemented in multiple ways but the chosen implementation was $(A * S) + (B * \bar{S})$ Where S is the selector and A and B are inputs. Now that it is explained how the nodes in the graph are mapped to instructions, the last part that remains are circuit registers. To explain how circuit registers are implemented a few things need to be recalled. Circuit registers carry data over cycles within a circuit. All of the combinatorial logic is converted into a program that runs on the bittide network. Thus registers need to carry data between full executions of the program or an entire cycle of the original circuit. To achieve this, a circuit register is implemented by inserting extra nodes into the graph without instruction, sort of like a wire, just passing data through. However, instead of passing data through, the dependency direction is flipped. In the case of a normal wire, where the output of an instruction would be written to a CPU register, and then the next instruction would read it from the CPU register, the dependencies are flipped. So the instruction Writing to this node has to execute after the instructions reading from this node. The result is that when running a program the CPU register corresponding to this node is first read, and later in the program this CPU register is written to, only to be read again at the start of the next program and thus carrying the data over to the next program execution. This change in how the circuit registers are mapped also maintains the acyclic nature of the produced DAG. At this moment there is a DAG of RISC-V instructions and the first step is nearly completed. The CPU register locations for each instruction still have to be set. Now that there is a DAG of instructions, this can be done by simply traversing the DAG by starting at a node that has no dependencies or whose dependencies already have an assigned CPU register and assigning the next free CPU register location to that node. This step can be repeated until each node in the graph has an output CPU register location assigned. Then by looking at the input nodes for each node, the input CPU registers can be set based on the output CPU registers of these input nodes. This is not an efficient assignment as many CPU registers could probably be reused, and the resulting programs thus use way more CPU registers than necessary, but it helps the debug process. Furthermore, this could be optimized later in the process. An example instruction DAG is given in Figure 6

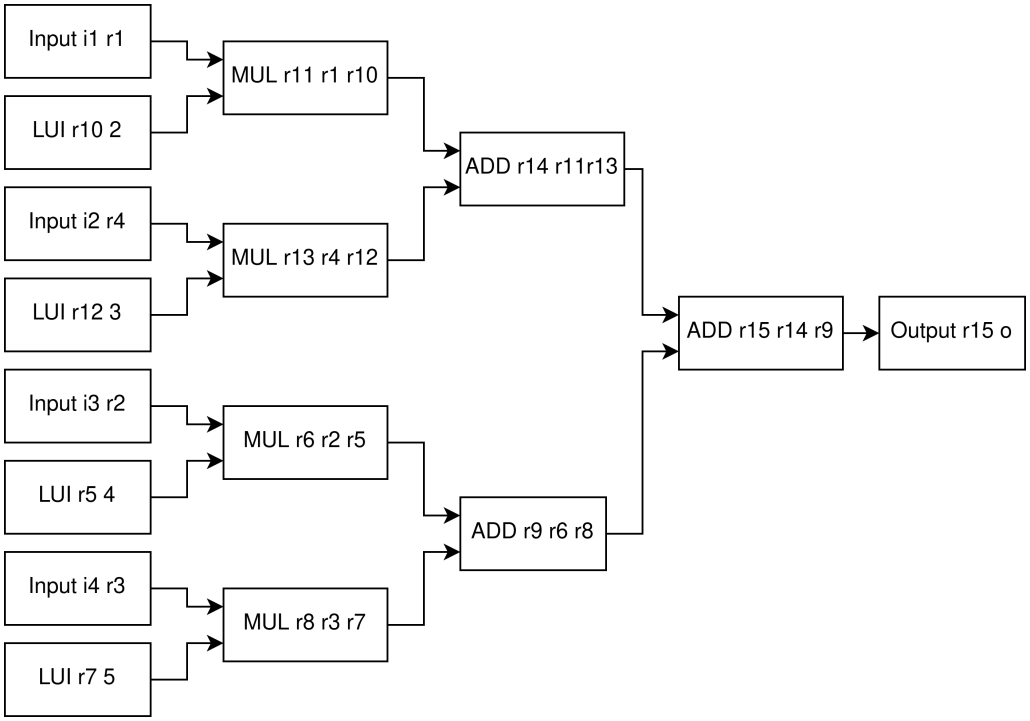


FIGURE 6: Instruction DAG generated from Figure 5b

8.2 Scheduling the DAG

Now that an instruction DAG as shown in Figure 6 is given, this has to be scheduled over the bittide network. A decision has to be made on which instruction runs in which CPU cycle, and on which core. In case data has to be moved between cores, communication instructions have to be inserted. All of this is done using the HEFT (Heterogeneous Earliest Finish Time) Algorithm [18] described in Chapter 4.4.1. The algorithm works in two steps. First, a priority queue is constructed based on the node's upward rank. The upward rank for a node is determined by taking the maximum upward rank of its immediate successor nodes and adding the average computation and communication delay of the node to that score. The average computation delay is 1 CPU cycle as every instruction takes one CPU cycle. The average communication delay is the average delay of the specific bittide network that is being targeted. This upward rank represents the critical path to the furthest exit node or the time that the program needs after this instruction to finish. After each node is assigned an upward rank, the nodes are put into a priority queue from highest to lowest rank. After this queue is constructed, the algorithm proceeds to the second step, the actual scheduling. A node is taken from the priority queue and selected for scheduling until the queue is depleted. Since the queue is constructed in a way where successors always have a higher priority, it is unnecessary to check whether its dependencies have been scheduled already. The first step to scheduling the node is to retrieve where and when its dependencies have been scheduled. Then the algorithm will try to schedule the node on each core and select the core on which the instruction finished earliest. When trying to schedule a node on a specific core, first it is checked whether the dependencies were scheduled on the same core. If this is not the case, a send, and receive will be scheduled first to get the data to the correct core. The send and receive will be scheduled at the first empty slot after the dependency is scheduled. What constitutes an empty slot can differ and in the Results (Chapter 9) multiple options are evaluated, for example defining a slot to be empty when

there is no communication already going over that specific link, or when no communication at all was happening for the cores in this cycle. After the communication is scheduled and the data is brought to the selected core, the instruction can be scheduled on this core. This happens similarly to the send, and receive instructions where the first possible slot is selected. Again there are multiple ways to define an empty slot, for example when there is no computation instruction, or when neither computation nor communication is happening in that CPU cycle. After this process has happened for each core, the core for which the instruction finished earliest is selected. The instruction (and optional send and receives) will be permanently added to the schedule and the next node is selected from the priority queue. After the queue is empty and all instructions have been scheduled, each core is left with a schedule of which instructions it will run at which moment. Holes in this schedule have to be filled with NOP instructions, and the schedule for each node has to be padded with NOPs so that each schedule has the same length. If the program is meant to repeat (in case the original hardware description contains registers and is meant to calculate results over multiple cycles), a jump instruction has to be inserted on each core to let the program start over when reaching the end. An example of a produced schedule can be found in Figure 7. In order to arrive at this schedule, the upward rank is determined, which is 10 for the leftmost column of nodes in Figure 6, 8 for the second until 2 for the last column containing the output instruction. This results in the leftmost column being scheduled first and the rightmost column last. Each instruction is scheduled on a core/cycle where it finishes first.

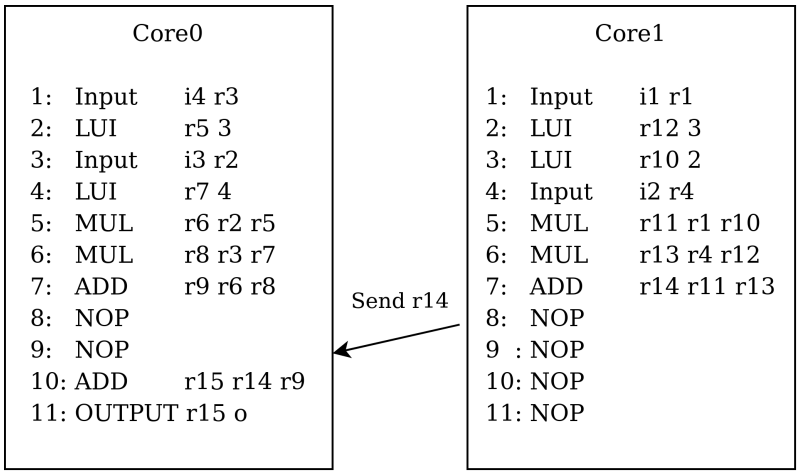


FIGURE 7: Schedule for 2 nodes generated from Figure 6. In this case, the send instruction happens in parallel to RISC-V instructions and is not the responsibility of the cores themselves but the network

8.3 Simulator

To test the correctness of the provided mappings and schedules, a simulator has been developed that accepts inputs provided by the scheduler. The simulator simulates RISC-V cores on an instruction level with the assumption that each instruction takes a single cycle (The scheduler makes the same assumptions). Each core has an amount of registers and memory that is configurable. As the scheduler does not optimize register usage it is often necessary to have unrealistically large register banks. The zero register always reads zero, and can thus be used to discard outputs and read zeros where necessary. Furthermore, the basic RISC-V instruction set is implemented with an extension for multiplication and

division. The simulator starts by initializing the network, this means instantiating all cores and the links between them. The links will be filled up with zeros as initial data, corresponding to the delay on these links. As explained in Chapter 5 the link delay corresponds with the amount of packets on the network and as the total amount of packets in the network does not change we need to initialize the network with a predetermined amount of packets on specific links. Next, each core will be assigned a program provided by the scheduler, and the program counter is set to zero. After this, the simulator goes into a loop where in each generation it lets each CPU step forward. This means that in the simulator the frequencies of the cores are all synchronized and can not diverge. However, as communication still happens exclusively via the elastic buffers that does not influence testing correctness. Within each CPU step first, the incoming links are processed, meaning data is read from each link and stored in the indicated registers (zero if no register is provided). Then the instruction is executed, possibly updating the program counter and register values. Lastly, the outgoing links are populated with values from registers specified by the provided schedule. The keen observer might note that this order allows a core to receive a value, use that value, and send the result over a link, all within the same cycle. This is done purposely to keep the simulator flexible for possible future implementations. However, it was decided that this might be unrealistic to do in a single cycle, therefore it is the responsibility of the scheduler to generate schedules that do not abuse this possibility. The simulator provides two interesting instructions outside the RISC-V instruction set, the input and output instructions. These instructions are meant for the program to be able to communicate with the outside world, for example, to read in a matrix to multiply, or to provide the result so correctness can be verified. The simulator is provided not only with a schedule and a network layout but also with a file containing inputs for the running program. When an input instruction is encountered, the input file will be opened and the next input value will be read which has the tag corresponding to the input instruction. For example, there might be an Input i1 r2 instruction, which opens the input file, reads the next input with the tag i1, and stores the value in register r2. The output instruction works similarly, except that it will pipe the tag together with the value to stdout of the simulator program.

8.4 Benchmarks

Exact Verilog implementations for some of the matrix multiplications, Conway's Game of Life, and prime number generations can be found in Appendix A. For matrix multiplication and Conway's Game of Life, the sizes of the inputs differ but the general implementation remains the same. The generation of random graphs however requires a little bit more explanation. The graphs are generated as uniformly as possible. When generating a graph of N nodes and E edges, a matrix of size N*N is created and filled with zeros. In this matrix, one represents an edge from the node on the left to the node on the bottom. Thus in a 2x2 matrix, a one in the left bottom represents an edge from the second node to the first. For a graph with E edges, a random place containing a zero in the bottom left triangle of the matrix is selected and replaced with a one. This is done E times, creating a graph with E edges. Since each node represents an instruction and instructions within the RISC-V instruction set have at most two dependencies, another constraint needs to be applied. Whenever a column already has two ones, and thus the bottom node already has two incoming dependencies, no ones can be added to this column. The result is a DAG of N nodes and E edges, in which each node has at most 2 dependencies. The nodes are then replaced with instructions, where each node without dependencies is turned into an INPUT instruction, nodes with a single dependency are turned into an ADDI (Add

Immediate) instruction, and nodes with two inputs are turned into an ADD instruction. For the execution time or generated schedule, it does not matter much which instructions are chosen (as long as the number of dependencies matches) but it is required that each node has an instruction and these instructions have been arbitrarily selected. Furthermore, all of these benchmarks operate on data. This data is provided into the bittide network using the INPUT instructions described above in Chapter 8.3. These INPUT instructions are capable of loading a single scalar value into a register, and thus operating on a 32x32 matrix requires 32*32=1024 INPUT instructions to load all of the data into the CPUs. Similarly, the results of the benchmarks will be provided using OUTPUT instructions, which again can read a single register and produce a single scalar value.

9 Result & Evaluation

In this chapter findings are represented, elaborated, and evaluated. The implementation will be subjected to multiple problems, among which are matrix multiplication, Conway's Game of Life, Generating primes, and random graphs. Then the parallelism achieved will be measured when running these programs on a 12-node RISC-V bittide network. Later, a few problems will be measured when scheduled on a 1- to 50-node network, and Lastly, a comparison will be made against the RAW machine shown in Chapter 4.1.1.

9.1 Higher level Implementation

First, the results are presented when running each problem on a 12-core RISC-V bittide network. Each problem is scheduled 500 times because the HEFT scheduling algorithm is non-deterministic and thus produces different results each time. The number of cycles that the produced program gives is measured and in the table, the minimum, maximum, and average number of cycles is shown. As the number of nodes in a problem represents the number of instructions in the dependency graph, the minimum cycle length on a single core machine would be the number of nodes in the network, as there is no extra communication delay. This is also directly the maximum number of cycles as each cycle can always be filled with an instruction. Based on the number of nodes a speed-up factor can be calculated. This is the amount of speed-up achieved in comparison to a single-core machine. The Min Factor, Max Factor, and Avg Factor represent the speed-up in comparison to a single-core machine for the minimum schedule length, maximum schedule length, and average schedule length over the 500 generated schedules.

To get realistic results, a realistic implementation of the bittide network must be assumed. For the benchmarks, a fully connected bittide network is assumed. However, it might be unrealistic that a single node can at the same time receive data over all its connections and send data back over the connections. This requires the register bank of the processor to have many ports, in the case of a 12-node network, each register bank requires 25 ports. Thus results are presented for three different node designs. A node that has unlimited ports on its register bank, and therefore unlimited communication parallel to computational instructions running on the RISC-V core. A node is represented with a single register bank, thus the node has to either communicate over a single link or make a computation. Lastly, a hybrid node is represented with a few ports on its register bank, allowing limited communication in parallel to computational instructions.

9.1.1 Benchmarks

To have any meaningful evaluation of the implementation an array of different benchmarks has been created. To be able to compare results with existing literature, problems have been picked that existing implementations have been evaluated for as well. In [13] their implementation is tested against 32x64 * 64x8 matrix multiplication and a 32x32 board of Conway's Game of Life. Thus these are included in this thesis as well. More about the implementations of these benchmarks can be found in 8.4. The comparison between the RAW machine and this thesis is made in and below Table 7 and 8.

9.1.2 Unlimited communication

TABLE 2: Unlimited communications on a 12-node network

Instance	Nodes	Edges	Min Cycles	Max Cycles	Avg Cycles	Min Factor	Max Factor	Avg Factor
mat4x4	160.0	240.0	16.0	18.0	17.08	8.889	10.0	9.368
mat5x5	300.0	475.0	26.0	27.0	26.62	11.111	11.538	11.27
mat16x16	8704.0	16128.0	726.0	726.0	726.0	11.989	11.989	11.989
conway3x3	333.0	549.0	33.0	37.0	35.046	9.0	10.091	9.502
conway4x4	592.0	976.0	52.0	54.0	53.532	10.963	11.385	11.059
conway5x5	925.0	1525.0	79.0	80.0	79.698	11.562	11.709	11.606
conway16x16	9472.0	15616.0	791.0	791.0	791.0	11.975	11.975	11.975
random	72.0	99.0	15.0	32.0	21.822	2.25	4.8	3.299
random	160.0	240.0	20.0	38.0	27.32	4.211	8.0	5.857
random	300.0	475.0	26.0	44.0	34.112	6.818	11.538	8.795
random	8704.0	16128.0	726.0	726.0	726.0	11.989	11.989	11.989
primes	43.0	60.0	17.0	17.0	17.0	2.529	2.529	2.529

In Table 2 the results are presented for the amount of parallelization found on a 12-core network. As can be seen, the amount of parallelism found seems to correlate a lot with the problem size. This is logical because firstly, matrix multiplication and Conway's Game of Life are arbitrarily parallelizable when the size of the problem grows, and in general, the chance to have two instructions that can be executed in parallel likely grows when the number of instructions grows. It is interesting to note that in a few cases, the theoretical maximum performance is achieved. As there are 12 cores, the program can be sped up at most 12 times. In the case of 16x16 matrix multiplication, the generated schedules consist of 726 cycles. The theoretical maximum is 8704 / 12 = 725.33, which rounded up is exactly 726 cycles. This also holds for 16x16 Conway's Game of Life and the largest instance of the random graphs. Note that in cases where the theoretical limit can not be achieved, there can be a significant difference between the best and worst-case scenarios. This suggests that performance gain might be possible using better scheduling algorithms as HEFT is often not able to exploit all available parallelism.

9.1.3 Limit to either communicate or calculate

TABLE 3: Communication and computation managed by the RISC-V Core on a 12-node network

Instance	Nodes	Edges	Min Cycles	Max Cycles	Avg Cycles	Min Factor	Max Factor	Avg Factor
mat3x3	72.0	99.0	23.0	55.0	36.33	1.309	3.13	1.982
mat4x4	160.0	240.0	43.0	113.0	70.326	1.416	3.721	2.275
mat5x5	300.0	475.0	85.0	201.0	130.776	1.493	3.529	2.294
mat16x16	8704.0	16128.0	2624.0	7000.0	4399.416	1.243	3.317	1.978
conway3x3	333.0	549.0	204.0	351.0	313.102	0.949	1.632	1.064
conway4x4	592.0	976.0	402.0	628.0	547.868	0.943	1.473	1.081
conway5x5	925.0	1525.0	556.0	909.0	776.312	1.018	1.664	1.192
conway16x16	9472.0	15616.0	4972.0	6642.0	5896.344	1.426	1.905	1.606
random	72.0	99.0	40.0	79.0	67.914	0.911	1.8	1.06
random	160.0	240.0	114.0	177.0	160.416	0.904	1.404	0.997
random	300.0	475.0	263.0	334.0	309.422	0.898	1.141	0.97
random	8704.0	16128.0	8794.0	9196.0	8907.3	0.946	0.99	0.977
primes	43.0	60.0	35.0	35.0	35.0	1.229	1.229	1.229

In table 3 the results are displayed for the situation in which the RISC-V core can perform either a communication or computation instruction. The maximum average speed up observed is 2.293 times faster with the 5 by 5 matrix multiplication. In the case of the 5x5 matrix multiplication, the smallest generated schedule is 85 instructions long and the largest 201. This is a significant difference and would mean a speed-up of between 1.49 and 3.52 times. Overall each schedule is far away from the theoretical limit of a 12 times speed up.

9.1.4 Limited to 2 receive, 2 send, and one instruction in parallel per cycle

TABLE 4: Communication is limited to 2 receives, 2 sends, and one instruction on a 12-node network

Instance	Nodes	Edges	Min Cycles	Max Cycles	Avg Cycles	Min Factor	Max Factor	Avg Factor
mat3x3	72.0	99.0	13.0	15.0	13.096	4.8	5.538	5.498
mat4x4	160.0	240.0	17.0	19.0	18.076	8.421	9.412	8.852
mat5x5	300.0	475.0	27.0	29.0	27.664	10.345	11.111	10.844
mat16x16	8704.0	16128.0	727.0	727.0	727.0	11.972	11.972	11.972
conway3x3	333.0	549.0	33.0	39.0	35.32	8.538	10.091	9.428
conway4x4	592.0	976.0	52.0	55.0	53.516	10.764	11.385	11.062
conway5x5	925.0	1525.0	79.0	81.0	79.664	11.42	11.709	11.611
conway16x16	9472.0	15616.0	791.0	791.0	791.0	11.975	11.975	11.975
random	72.0	99.0	16.0	32.0	21.784	2.25	4.5	3.305
random	160.0	240.0	21.0	43.0	28.648	3.721	7.619	5.585
random	300.0	475.0	28.0	48.0	35.804	6.25	10.714	8.379
random	8704.0	16128.0	727.0	730.0	727.014	11.923	11.972	11.972
primes	43.0	60.0	17.0	17.0	17.0	2.529	2.529	2.529

In the previous section the results were shown for a network with unlimited communication, however, due to physical limitations with the register banks and other components this does not seem likely to be possible in reality. Thus in Table 4, the results are shown if the communication can still happen in parallel but is more limited. In this case, a bittide node is limited to two sending over two and receiving over two links in a single cycle. This requires a more realistic number of extra ports on the register banks. The performance for most of the problems has decreased somewhat which is to be expected. The performance is still much better than in table 3 and when increasing the problem sizes the theoretical limit can still be achieved.

9.1.5 Scaling bittide network

The next results show how well the approach scales with the bittide network.

TABLE 5: The achieved parallelism when scheduling 5x5 matrix multiplication over different network sizes

Num Cores	Min Cycles	Max Cycles	Avg Cycles	Avg Factor	Effiency
1	301.0	301.0	301.0	0.997	0.997
5	61.0	61.0	61.0	4.918	0.984
10	32.0	32.0	32.0	9.375	0.938
15	23.0	25.0	23.78	12.616	0.841
20	19.0	22.0	20.64	14.535	0.727
30	17.0	21.0	17.76	16.892	0.563
40	17.0	20.0	17.08	17.564	0.439
50	17.0	20.0	17.18	17.462	0.349

In Table 5 the results are presented for mapping the 5x5 matrix multiplication problem on networks with 1 to 50 cores. An extra column is included displaying the efficiency of the achieved parallelism. This is the average factor divided by the number of cores. As can be seen in the table, at a relatively low core count, there is enough parallelism in the problem for the amount of cores and the parallelism efficiency is very high. However, as the core count increases the efficiency decreases and it seems that around 40 cores, adding extra cores does not increase the amount of parallelism achieved, stagnating at around a 17 times speed-up in comparison to a single core network.

TABLE 6: The achieved parallelism when scheduling 16x16 matrix multiplication over different network sizes

Num Cores	Min Cycles	Max Cycles	Avg Cycles	Avg Factor	Effiency
1	8705.0	8705.0	8705.0	1.0	1.0
5	1742.0	1742.0	1742.0	4.997	0.999
10	872.0	872.0	872.0	9.982	0.998
15	582.0	582.0	582.0	14.955	0.997
20	437.0	437.0	437.0	19.918	0.996
30	292.0	292.0	292.0	29.808	0.994
40	219.0	219.0	219.0	39.744	0.994
50	176.0	180.0	176.1	49.426	0.989
75	118.0	126.0	119.62	72.764	0.97
100	91.0	100.0	94.68	91.931	0.919
150	64.0	69.0	66.16	131.56	0.877
200	52.0	55.0	53.5	162.692	0.813
250	52.0	55.0	53.34	163.18	0.653
300	52.0	54.0	53.08	163.979	0.547

TABLE 7: The achieved parallelism when scheduling 32x64 by 64x8 matrix multiplication over different network sizes

Num Cores	Min Cycles	Max Cycles	Avg Cycles	Avg Factor	Effiency
1	35329.0	35329.0	35329.0	1.0	1.0
2	17665.0	17667.0	17665.6	2.0	1.0
4	8833.0	8833.0	8833.0	4.0	1.0
8	4417.0	4417.0	4417.0	7.998	1.0
16	2209.0	2209.0	2209.0	15.993	1.0
32	1105.0	1105.0	1105.0	31.971	0.999

TABLE 8: The achieved parallelism when scheduling 32x32 Conway's Game of Life over different network sizes

Num Cores	Min Cycles	Max Cycles	Avg Cycles	Avg Factor	Effiency
1	37889.0	37889.0	37889.0	1.0	1.0
2	18945.0	18945.0	18945.0	2.0	1.0
4	9473.0	9473.0	9473.0	4.0	1.0
8	4737.0	4737.0	4737.0	7.998	1.0
16	2369.0	2369.0	2369.0	15.993	1.0
32	1185.0	1185.0	1185.0	31.973	0.999

In [13] the problems from Tables 7 and 8 above are also benchmarked. Directly two things become clear. Firstly, for both problems on a single core machine, their implementation takes 2M+ cycles, while the mapping proposed in this thesis can solve both problems in less than 40k instructions. This seems very extreme but can be largely attributed to the following factors. An assumption was made that the RISC-V bittide network would take 1 cycle per instruction always, which is slightly unrealistic and the floating point math in their implementation likely takes up more cycles per instruction. Furthermore, their im-

plementation has to deal with loops, which require extra instructions. Lastly the RISC-V implementation assumes a very large register bank, while the RAW machine might need to use loads and stores in between. Due to these factors, it is impossible to say whether the presented mapping performs better or worse than their compiler. The second observation has to do with the amount of parallelization found in both platforms. While the efficiency of the RAW machine seems to degrade quickly with matrix multiplication and Conway's Game of Life, it seems that the proposed scheduling for the bittide network can find a lot more parallelism. At 32 cores, the efficiency is still nearly perfect for both problems. This can likely be attributed to the way data is stored in both systems. In the RAW machine, data is pinned to a certain machine, and then the scheduler has to insert extra instructions to move that around, to feed all the instructions needing that data. The RISC-V implementation however keeps this data in their registers with their instructions, thus scheduling instructions already tries to minimize the data transfers and therefore solves this problem more efficiently. This however can only be done because in this implementation all of the data is statically allocated, while the RAW machine can deal with the dynamic allocations possible in C++.

10 Conclusion

In search of new parallel ways to increase parallelization, this thesis looked into the possibility of using hardware descriptions to exploit the parallelism provided by bittide synchronization [12]. To do this an implementation was made that compiles hardware descriptions into RISC-V instructions and schedules them onto different nodes of the bittide network. The resulting schedules were evaluated based on the parallelism they exploited and how well this scaled with the size of the bittide network. The research question of this thesis was *Can the bittide network be programmed using hardware description languages (HDL) in a way that exploits the available parallelism?* The research question is answered as a result of the subquestions.

Does a mapping between HDLs and the bittide network exist?

The implementations presented in this thesis can transform the hardware descriptions into programs that run on the bittide network producing equivalent results. This is demonstrated by the mapping implemented in this thesis. The implementation uses various heuristics to overcome the complications in finding this mapping, thus the implementation takes mere seconds to schedule the largest presented problems on the bittide network. The precise algorithmic complexity for finding these mappings has not been analyzed.

What are appropriate benchmarks to measure the exploited parallelism?

According to the literature review, many of the benchmarks used to test parallel architectures rely on matrix-based algorithms. algorithms like matrix multiplication have a lot of parallelism that could be exploited by such platforms. In this thesis, sequential benchmarks and random graphs have also been implemented which demonstrated how the implementation held up under conditions where there was less parallelism than with matrix multiplication.

Is the approach able to exploit the available parallelism found in hardware descriptions?

The results found in Chapter 9 confirm that there is parallelism to be found in these hardware descriptions that remains intact when mapping to the bittide network. All of the problems presented observed performance increases when mapped over multiple cores. When comparing the implementation to the RAW machine, much more of the parallelism seems to be exploited. Even the prime number generation problem observed more than a 2.5 times increase while only consisting of 43 instructions, while the larger problems observed much more parallelism.

Does the approach scale well when the size of the bittide network increases?

The results in Chapter 9 show that at some point no more performance is gained when adding more cores. But until that moment, adding more cores is very effective in increasing the performance. Table 6 shows that with 16 by 16 matrix multiplication, over a hundred cores can be used quite efficiently.

Can the bittide network be programmed using hardware description languages (HDL) in a way that exploits the available parallelism?

To answer the main research question. Using hardware description languages and mapping these onto a bittide network seems an effective way to use the parallel nature of the bittide network. A high degree of parallelism was observed in the mapping produced for fully connected bittide networks using nodes based on RISC-V cores.

11 Future work

In this research, a way to compose programs to run on the bittide network was proposed. This is a first step because both the bittide network and the approach of using HDL to produce exact timed schedules for instruction-level parallelism are new. There are many directions that this research can go into. In this chapter, a few possible avenues will be discussed

11.1 Functional languages

In this research, the focus was on HDL because of its functional nature. However multiple limitations have been discussed in Chapter 6 like branching and the mapping of instructions. Furthermore, designing hardware is seen as a complicated task and there might be too few developers in this area. Thus an interesting avenue to explore would be to look at different languages that share good properties with HDL and have fewer of the downsides. The Haskell core language would be particularly interesting as it is also a pure language but can express constructs like recursivity. HDLs feel very restricted in what they can do since they can only schedule spatially over the available area while the bittide network provides the extra temporal dimension and is therefore less restrictive. This means that many of the constructs of Haskell that map poorly to hardware like recursion might run very elegantly on the bittide network. All while possibly keeping a high degree of parallelism.

11.2 Bittide

For this thesis, the bittide network operated in a very simple mode. For example, the assumption was made that for each CPU cycle, the bittide network would also do a transfer instead of applying gear ratios (See Chapter 5.5). One can imagine the bittide network operating in a data center to connect servers where internally within the CPUs in a single server, the bittide network would transfer with high frequency, while the communication between servers would happen at a significantly lower rate. This has many implications for scheduling towards this network. For example, a program DAG containing highly connected clusters, while having little connection between clusters might map very well to such a system, while other programs might map poorly. These consequences of different bittide parameters have not been explored in this Thesis.

11.3 Inter-operability

Another interesting avenue to explore could be interoperability on the bittide network. Modern computers almost always have a layer of operating system between the hardware and programs being executed, which the current methods in this thesis do not allow. This can be implemented in many ways. Maybe a subset of bittide nodes can be reserved for the operating system, and other nodes can communicate with this subset of nodes to communicate with the operating system. This would be interesting because the operating system and other programs have to be completely synchronized. There would also be a lot of complications because the program has to be scheduled over a subset of the bittide network, in a specific way to conform to the timing that the operating system nodes require. This would complicate the scheduler immensely. Also, the program has to be rescheduled every time the availability of nodes changes.

11.4 RISC-V

Currently, the project has many limitations that make it less appropriate for real-world applications. For example, a simple RISC-V implementation was taken that does not implement concepts like pipelining and assumes all instructions take the same length. Modern processors have many details that an instruction-level scheduler should be aware of. Furthermore, details like register spilling (where there are not enough registers so some of them should be emptied into memory) are not taken into account. Also, modern-day compilers perform many optimization tricks while the implementation described in this research does no form of optimization. The result is that it is difficult for this research to compare itself with state-of-the-art in performance. Thus a project that tries to optimize this implementation and get every bit of performance out of it could be interesting to see where this research stands and if it could be a possible solution to increase processor performance. The bittide network is a very flexible system and there is not even a requirement for all of the nodes to be the same. In a case where multiple different servers have to be connected using bittide, it is possible that the servers do not even share the same instruction set. Or even more extreme, some of the nodes in the network do not necessarily need to be processors but could be FPGAs or other accelerators as well.

11.5 Analysis

Due to the deterministic nature of the bittide network, many of the normally uncomputable problems in the analysis of multiprocessor environments could be reviewed. One of the reasons the bittide network was designed was to have guarantees over things like buffer overflows which should not be able to happen on a synchronous platform like the bittide network. The product of this research can produce exact schedules and can thus make many guarantees over and provide exact values for throughput and latencies. It would be interesting to have follow-up research to see how far this analysis can go and whether it can help make more guarantees about parallel programs.

References

[1] bittide-hardware. URL: https://github.com/bittide/bittide-hardware.

[2] pthreads(7) - linux manual page. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html.

[3] I. Ahmad and Yu Kwong Kwok. A new approach to scheduling parallel programs using task duplication. *Proceedings of the International Conference on Parallel Processing*, 2, 1994. doi:10.1109/ICPP.1994.37.

[4] Hamid Arabnejad and Jorge G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25:682–694, 3 2014. doi:10.1109/TPDS.2013.57.

[5] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Cash: Structural descriptions of synchronous hardware using haskell. *Proceedings - 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2010*, pages 714–721, 2010. doi:10.1109/DSD.2010.21.

[6] Debjyoti Bhattacharjee, Rajeswari Devadoss, and Anupam Chattopadhyay. Revamp: Reram based vliw architecture for in-memory computing. *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, pages 782–787, 5 2017. doi:10.23919/DATE.2017.7927095.

[7] Luiz F. Bittencourt, Rizos Sakellariou, and Edmundo R.M. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010*, pages 27–34, 2010. doi:10.1109/PDP.2010.56.

[8] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. Nova: A functional language for data parallelism. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 8–13, 6 2014. URL: https://dl.acm.org/doi/10.1145/2627373.2627375, doi:10.1145/2627373.2627375.

[9] Sebastien Cook and Paulo Garcia. Arbitrarily parallelizable code: A model of computation evaluated on a message-passing many-core system. *Computers 2022, Vol. 11, Page 164*, 11:164, 11 2022. URL: https://www.mdpi.com/2073-431X/11/11/164/htmhttps://www.mdpi.com/2073-431X/11/11/164, doi:10.3390/COMPUTERS11110164.

[10] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandra Nicolau. Parallel processing: A smart compiler and a dumb machine. *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1984*, 9:37–47, 6 1984. URL: https://dl.acm.org/doi/10.1145/502874.502878, doi:10.1145/502874.502878.

[11] Y. K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, IPPS/SPDP 1998*, 1998-March:531–537, 1998. doi:10.1109/IPPS.1998.669967.

[12] Sanjay Lall, Calin Cascaval, Martin Izzard, and Tammo Spalink. Modeling and control of bittide synchronization, 2021. URL: https://research.google/pubs/pub50734/.

[13] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. *ACM SIGOPS Operating Systems Review*, 32:46–57, 10 1998. URL: https://dl.acm.org/doi/10.1145/384265.291018, doi:10.1145/384265.291018.

[14] Jack L Lo, Susan J Eggers, Joel S Emer, Henry M Levy, Rebecca L Stamm, Dean M Tullsen, J L Lo, S J Eggers, H M Levy, ; J S Emer, R L Stamm, and ; D M Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems (TOCS)*, 15:322–354, 8 1997. URL: https://dl.acm.org/doi/10.1145/263326.263382, doi:10.1145/263326.263382.

[15] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. Impact of instruction-level parallelism on multiprocessor performance and simulation methodology. *IEEE High-Performance Computer Architecture Symposium Proceedings*, pages 72–83, 1997. doi:10.1109/HPCA.1997.569611.

[16] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. *Conference Proceedings - Annual International Symposium on Computer Architecture, ISCA*, pages 422–433, 2003. URL: https://dl.acm.org/doi/10.1145/859618.859667, doi:10.1145/859618.859667.

[17] Khushboo Singh, Mahfooz Alam, Sushil Kumar, and Sharma M Tech. A survey of static scheduling algorithm for distributed computing system. *International Journal of Computer Applications*, 129:975–8887, 2015.

[18] Haluk Topcuoglu, Salim Hariri, and Min You Wu. Task scheduling algorithms for heterogeneous processors. *Proceedings of the Heterogeneous Computing Workshop, HCW*, pages 3–14, 1999. doi:10.1109/HCW.1999.765092.

[19] David W Wall. Limits of instruction-level parallelism. URL: https://dl.acm.org/doi/pdf/10.1145/106972.106991, doi:10.1145/106972.106991.

[20] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free verilog synthesis suite. 2013. URL: https://yosyshq.net/yosys/.

A Benchmarks

A.1 Conway's Game of Life

```
    module GameOfLife_3x3(input wire current_state_0_0,
input wire current_state_0_1,
input wire current_state_0_2,
input wire current_state_1_0,
input wire current_state_1_1,
input wire current_state_1_2,
input wire current_state_2_0,
input wire current_state_2_1,
input wire current_state_2_2,
output reg next_state_0_0,
output reg next_state_0_1,
output reg next_state_0_2,
output reg next_state_1_0,
output reg next_state_1_1,
output reg next_state_1_2,
output reg next_state_2_0,
output reg next_state_2_1,
output reg next_state_2_2);
assign next_state_0_0 = (current_state_0_0 && (current_state_2_2 + current_state_2_0
+ current_state_2_1 + current_state_0_2 + current_state_0_1 + current_state_1_2 +
current_state_1_0 + current_state_1_1 == 2 || current_state_2_2 + current_state_2_0 +
current_state_2_1 + current_state_0_2 + current_state_0_1 + current_state_1_2 +
current_state_1_0 + current_state_1_1 == 3)) || (!current_state_0_0 &&
current_state_2_2 + current_state_2_0 + current_state_2_1 + current_state_0_2 +
current_state_0_1 + current_state_1_2 + current_state_1_0 + current_state_1_1 == 3);

assign next_state_0_1 = (current_state_0_1 && (current_state_2_0 + current_state_2_1
+ current_state_2_2 + current_state_0_0 + current_state_0_2 + current_state_1_0 +
current_state_1_1 + current_state_1_2 == 2 || current_state_2_0 + current_state_2_1 +
current_state_2_2 + current_state_0_0 + current_state_0_2 + current_state_1_0 +
current_state_1_1 + current_state_1_2 == 3)) || (!current_state_0_1 &&
current_state_2_0 + current_state_2_1 + current_state_2_2 + current_state_0_0 +
current_state_0_2 + current_state_1_0 + current_state_1_1 + current_state_1_2 == 3);

assign next_state_0_2 = (current_state_0_2 && (current_state_2_1 + current_state_2_2
+ current_state_2_0 + current_state_0_1 + current_state_0_0 + current_state_1_1 +
current_state_1_2 + current_state_1_0 == 2 || current_state_2_1 + current_state_2_2 +
current_state_2_0 + current_state_0_1 + current_state_0_0 + current_state_1_1 +
current_state_1_2 + current_state_1_0 == 3)) || (!current_state_0_2 &&
current_state_2_1 + current_state_2_2 + current_state_2_0 + current_state_0_1 +
current_state_0_0 + current_state_1_1 + current_state_1_2 + current_state_1_0 == 3);

assign next_state_1_0 = (current_state_1_0 && (current_state_0_2 + current_state_0_0
+ current_state_0_1 + current_state_1_2 + current_state_1_1 + current_state_2_2 +
current_state_2_0 + current_state_2_1 == 2 || current_state_0_2 + current_state_0_0 +
current_state_0_1 + current_state_1_2 + current_state_1_1 + current_state_2_2 +
```

```
current_state_2_0 + current_state_2_1 == 3)) || (!current_state_1_0 &&
current_state_0_2 + current_state_0_0 + current_state_0_1 + current_state_1_2 +
current_state_1_1 + current_state_2_2 + current_state_2_0 + current_state_2_1 == 3);

assign next_state_1_1 = (current_state_1_1 && (current_state_0_0 + current_state_0_1
+ current_state_0_2 + current_state_1_0 + current_state_1_2 + current_state_2_0 +
current_state_2_1 + current_state_2_2 == 2 || current_state_0_0 + current_state_0_1 +
current_state_0_2 + current_state_1_0 + current_state_1_2 + current_state_2_0 +
current_state_2_1 + current_state_2_2 == 3)) || (!current_state_1_1 &&
current_state_0_0 + current_state_0_1 + current_state_0_2 + current_state_1_0 +
current_state_1_2 + current_state_2_0 + current_state_2_1 + current_state_2_2 == 3);

assign next_state_1_2 = (current_state_1_2 && (current_state_0_1 + current_state_0_2
+ current_state_0_0 + current_state_1_1 + current_state_1_0 + current_state_2_1 +
current_state_2_2 + current_state_2_0 == 2 || current_state_0_1 + current_state_0_2 +
current_state_0_0 + current_state_1_1 + current_state_1_0 + current_state_2_1 +
current_state_2_2 + current_state_2_0 == 3)) || (!current_state_1_2 &&
current_state_0_1 + current_state_0_2 + current_state_0_0 + current_state_1_1 +
current_state_1_0 + current_state_2_1 + current_state_2_2 + current_state_2_0 == 3);

assign next_state_2_0 = (current_state_2_0 && (current_state_1_2 + current_state_1_0
+ current_state_1_1 + current_state_2_2 + current_state_2_1 + current_state_0_2 +
current_state_0_0 + current_state_0_1 == 2 || current_state_1_2 + current_state_1_0 +
current_state_1_1 + current_state_2_2 + current_state_2_1 + current_state_0_2 +
current_state_0_0 + current_state_0_1 == 3)) || (!current_state_2_0 &&
current_state_1_2 + current_state_1_0 + current_state_1_1 + current_state_2_2 +
current_state_2_1 + current_state_0_2 + current_state_0_0 + current_state_0_1 == 3);

assign next_state_2_1 = (current_state_2_1 && (current_state_1_0 + current_state_1_1
+ current_state_1_2 + current_state_2_0 + current_state_2_2 + current_state_0_0 +
current_state_0_1 + current_state_0_2 == 2 || current_state_1_0 + current_state_1_1 +
current_state_1_2 + current_state_2_0 + current_state_2_2 + current_state_0_0 +
current_state_0_1 + current_state_0_2 == 3)) || (!current_state_2_1 &&
current_state_1_0 + current_state_1_1 + current_state_1_2 + current_state_2_0 +
current_state_2_2 + current_state_0_0 + current_state_0_1 + current_state_0_2 == 3);

assign next_state_2_2 = (current_state_2_2 && (current_state_1_1 + current_state_1_2
+ current_state_1_0 + current_state_2_1 + current_state_2_0 + current_state_0_1 +
current_state_0_2 + current_state_0_0 == 2 || current_state_1_1 + current_state_1_2 +
current_state_1_0 + current_state_2_1 + current_state_2_0 + current_state_0_1 +
current_state_0_2 + current_state_0_0 == 3)) || (!current_state_2_2 &&
current_state_1_1 + current_state_1_2 + current_state_1_0 + current_state_2_1 +
current_state_2_0 + current_state_0_1 + current_state_0_2 + current_state_0_0 == 3);

endmodule
```

A.2 Matrix Multiplication

```
module matrix_mult (
```

```
    input signed [31:0] m1_0_0,
    input signed [31:0] m1_0_1,
    input signed [31:0] m1_0_2,
    input signed [31:0] m1_1_0,
    input signed [31:0] m1_1_1,
    input signed [31:0] m1_1_2,
    input signed [31:0] m1_2_0,
    input signed [31:0] m1_2_1,
    input signed [31:0] m1_2_2,
    input signed [31:0] m2_0_0,
    input signed [31:0] m2_0_1,
    input signed [31:0] m2_0_2,
    input signed [31:0] m2_1_0,
    input signed [31:0] m2_1_1,
    input signed [31:0] m2_1_2,
    input signed [31:0] m2_2_0,
    input signed [31:0] m2_2_1,
    input signed [31:0] m2_2_2,
    output signed [31:0] result_0_0,
    output signed [31:0] result_0_1,
    output signed [31:0] result_0_2,
    output signed [31:0] result_1_0,
    output signed [31:0] result_1_1,
    output signed [31:0] result_1_2,
    output signed [31:0] result_2_0,
    output signed [31:0] result_2_1,
    output signed [31:0] result_2_2
);

    assign result_0_0 = (m1_0_0 * m2_0_0) + (m1_0_1 * m2_1_0) + (m1_0_2 * m2_2_0);
    assign result_0_1 = (m1_0_0 * m2_0_1) + (m1_0_1 * m2_1_1) + (m1_0_2 * m2_2_1);
    assign result_0_2 = (m1_0_0 * m2_0_2) + (m1_0_1 * m2_1_2) + (m1_0_2 * m2_2_2);
    assign result_1_0 = (m1_1_0 * m2_0_0) + (m1_1_1 * m2_1_0) + (m1_1_2 * m2_2_0);
    assign result_1_1 = (m1_1_0 * m2_0_1) + (m1_1_1 * m2_1_1) + (m1_1_2 * m2_2_1);
    assign result_1_2 = (m1_1_0 * m2_0_2) + (m1_1_1 * m2_1_2) + (m1_1_2 * m2_2_2);
    assign result_2_0 = (m1_2_0 * m2_0_0) + (m1_2_1 * m2_1_0) + (m1_2_2 * m2_2_0);
    assign result_2_1 = (m1_2_0 * m2_0_1) + (m1_2_1 * m2_1_1) + (m1_2_2 * m2_2_1);
    assign result_2_2 = (m1_2_0 * m2_0_2) + (m1_2_1 * m2_1_2) + (m1_2_2 * m2_2_2);

endmodule
```

A.3 Prime number generation

```
    module prime_number_count(
    output reg [31:0] last_prime,
    input clk
);

  reg [31:0] sieve_index = 0;
```

```
  reg [31:0] test_prime = 0;
  reg current_is_prime;


always @(posedge clk) begin

    if (sieve_index < test_prime) begin
      current_is_prime <= current_is_prime &&
      ((test_prime / sieve_index) * sieve_index) != test_prime;
        sieve_index <= sieve_index + 1;
    end else begin
        if (current_is_prime) begin
            last_prime <= test_prime;
        end
        sieve_index <= 2;
        test_prime <= test_prime + 1;
        current_is_prime = 1;
    end
end


endmodule
```

B Initial Implementation

The initial implementation is based on synthesis for the ECP5. This consists of a compiler and a scheduler.

B.1 Instruction generation

The compiler is responsible for converting different platforms to RISC-V instruction programs to be scheduled for execution. The ECP5 compiler compiles ECP5 Json files produced by Yosys to our internal representation of RISC-V instructions ready for the next step. This compiler converts each ECP5 cell to a different program. The top-level ports are treated as their own cells and thus get their own programs. The ECP5 implementation has cells with inputs and outputs that are connected. Each bit has a unique id and if an input and output share an id, this means that these components are connected via this specific bit. A specific output bit could be connected to multiple inputs of different cells. For each of these connections, a specific send instruction has to be generated to send this bit to that specific program. To do this the first step is to collect for each unique bit, all the locations where this bit is used as input so that we can later generate a write instruction for each of them. After identifying all connections we can compile each different cell. Input ports are compiled by first generating an Input instruction which is a special instruction that tells the environment that this node expects input from the outside environment now. In a simulation, this could mean user input. In a real setting, this could be resolved by the layer between the bittide network and the outside world. After this input instruction, send instructions are generated. A send instruction is generated for each cell that requires this input as mentioned in the previous chapter. Output ports work similarly except that they first generate a Recv instruction to receive a bit within the network and then generate

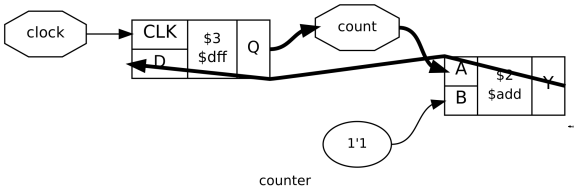
an Output instruction to output this data to the environment. When not using dedicated hardware (like multipliers etc.), ECP5 synthesizes the rest into 2 different components, LUTs and Flip-Flops. For the LUT, Recv instructions are first generated for the relevant inputs. Then RISCV instructions are generated to find the resulting value based on the lookup table. and then Send instructions are generated to forward this result to the next node. An important detail here is that not all instructions have to be executed each time. Loading the lookup table into a register only has to happen once for example. And there is a high chance that not all inputs are connected to the lookup table thus these instructions also only have to happen once at the beginning. Instructions that only have to happen once at the beginning are identified and added to the setup part of the program instead. Flip-flops are implemented slightly differently as they act like registers. This means that their behavior mostly comes down to first sending the previously stored value to the network and then receiving the next value from the network. apart from this detail, the rest is implemented in a similar way as the other components, including identifying instructions that only have to happen once and adding them to the setup part instead. Currently, the compiler does not support dedicated hardware but in a future iteration these components could be identified and in a similar way instructions for them could be generated. All the collected programs are stored together in a model, converted to Json, and stored in a file for the scheduler to take over.

B.2 Scheduler

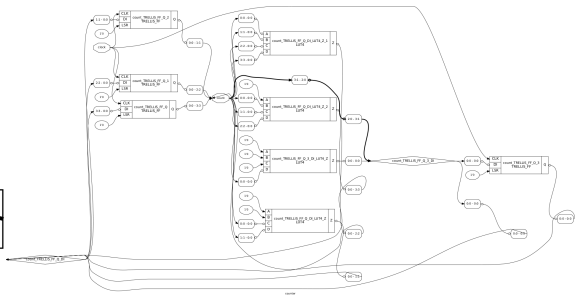
The scheduler performs three important steps. Map each program to a different node, create a dependency graph of all instructions, and create a schedule by resolving the timings in the dependency graph. Each program has Send and Recv instructions. These instructions correspond to their counterparts in different programs. When mapping two programs with a Send Recv pair between them to hardware nodes, it is required that these hardware nodes have a bittide connection between them to be able to exchange this data. This problem is a sub-graph problem (identifying a sub-graph of connected programs in the graph of connected hardware nodes). and this problem is NP-complete. The implementation for this is a simple brute-force approach. Each instruction reads or writes to certain registers. A dependency graph can be created by identifying which instructions are dependent on which other instructions by identifying the registers they are writing to or reading from and finding other instructions that also use those registers. For example, an instruction that reads from r0 should depend on the last instruction that has been written to r0 because if it was scheduled earlier, this register would have a different value. Similarly, a write instruction depends on all instructions that are read from it until the previous write, as when writing too early, the read instructions would again have a different value. These dependencies have the property that all instructions should have some lower bound on their delay respectively to their dependencies. They can not be scheduled before their dependencies, but they can be scheduled arbitrarily later. Because this is a prototype, in the implementation each instruction is dependent on all instructions before it in the program, disallowing changing the order in which instructions are executed. There are two special case instructions, namely the Recv and the Send instructions. The Recv instructions are dependent on their corresponding Send instructions, however, there should be an exact delay between the Send and Recv instructions. Thus the Send instruction depends on the Recv instruction, however, the Recv instruction also depends on all instructions that the Send instruction depends on. This ensures that when we try to schedule the Send/Recv pair, both their dependencies are already scheduled and we can safely schedule these without having to move them. A directed graph is constructed

where each instruction corresponds to a node, and each node has connections to all of its dependencies. After the dependency graph is constructed, the timings can be resolved. The timings are resolved using a first-fit approach: First, a node is selected for which all of its dependencies are already scheduled. Then this node is scheduled at the first possible time. If this node happens to be a Send instruction, its Recv instruction is immediately scheduled after. This results in a working schedule that could be very inefficient. An example approach to improving upon this is to figure out afterward which instructions could have been scheduled later. The total throughput of the schedule is dependent on the timing of the first instruction of a node and the last. the throughput is dependent on the node for which this difference is the largest. Thus there are two ways of improving this, first by reducing the time at which the last instruction is scheduled, however, due to our first fit algorithm, each instruction is already scheduled at the earliest possible time. However, often instructions can be scheduled later, by trying to schedule instructions later, the total time between the first instruction and the last of each program could be reduced. This creates a sort of pipe-lining effect. After each instruction has a time at which it needs to be executed an exact schedule can be created. The period of the programs is identified by finding the program with the longest execution time. The other programs are padded with NOP (no operation) instructions to make sure each program has the same period. Empty spots in the schedules are also filled with NOP instructions. Afterward, a JAL (Jump and Link) instruction is added to let all programs jump to their beginning again. This results in a program for each hardware node (including nodes for which no actual instruction was scheduled, just NOPs and JALs) all of equal length.

B.3 Results



(A) The graph for a counter



(B) The same counter synthesized to ECP5 LUTs

The initial approach that makes use of code synthesized for the ECP5 FPGA was able to create schedules. However, a simple 3-bit counter would require more than 10 cores to schedule and would take many cycles per increment. All this could be implemented on a single core in 3 cycles per increment (an add instruction, output instruction, and jump instruction). This is because each of the ECP5 nodes would be a separate program that would be scheduled on its own core. This means that most of the nodes, each of which is very simple, would have to first fetch their data from other nodes. On top of this, synthesizing hardware for an ECP5 reduces the implementation to a bit level. So a 3-bit counter is not simply one add instruction but the behavior is implemented using many 4-bit LUTs. The one exception is that the ECP5 has dedicated multipliers (and other hardware) which it will try to target when synthesizing. However, our counter would be implemented using a LUT, which must first fetch its 3 bits from 3 different cores, then execute the LUT

operation in a few cycles (building the index, then checking the bit in that index) then send its bit along. And a 3-bit counter requires a few LUTs. The result is that trivial operations for which the RISC-V core has dedicated instructions will be split over many cores requiring enormous execution and communication overhead. From these results, it is clear that the ECP5 approach will produce efficient results. Due to the synthesis to a bit level and the use of instructions operating on many bits at the same time within the RISC-V processor, this is a bad match and will never reach acceptable performance levels, however, it shows that there exists a mapping between the bittide network and hardware. On top of that, the usage of ECP5 synthesized code allows us to run everything that is able to synthesize to the ECP5 on the network. Thus existing code and anything produced anywhere between Clash and VHDL/Verilog will also run on the network. This implementation could still find usage in simulating designs to test for correctness or other related purposes.