POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Master degree course in Computer Engineering

Master Degree Thesis

Design of a High Performance Memory Packet Decoder for Streaming Architecture on FPGA

Supervisors:
prof. Claudio Passerone
prof. Carlos Carreras Vaquer
dipl. Roberto Sierra Cabrera

Candidates
Filippo Mangani
matricola: 241916

Academic Year 2018-2019

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The need for solving problems that are way too complicated to be solved by hand or mind is not a recent one. All computers have as primary purpose to solve problems efficiently and fast. However, there is a set of problems that occurs mostly in the scientific and engineering fields that requires an extreme use of computing capabilities to be solved. These problems aren't new and neither their algorithmic solution. However, the technology used to solve these problemd using machines is instead fairly new as the Moore law has allowed enhancing the computing technology to a point that a set of problems that were thought to be unsolvable due to the complexity of their algorithms are today solvable with the use of machines. One example of this kind of problems is the Schrödinger equation, which is capable of describing completely the atomic structures of molecules. As we can imagine, the more complex is the molecule, the more complex will be solution of the Schrödinger equation.

The Schrödinger equation was proposed in the beginning of the 20th century, but its solution using computer simulation wasn't possible until recently. The same concept is applicable to the Navier-Stokes equations, that describe the behaviour of the interaction between fluids and solids, where the more complex is the solid, the more complex is the solution of the equation. Also these equations have been proposed in the 19th century, but they couldn't be applied directly to problems as their solution was too hard in not ideal cases. As we can understand, these sets of equations are more and more essential to run precise simulations for the scientific and engineering communities. However, the main obstacle in all these problems is the time required to solve them, and therefore any solution that increases the

performance will be competitive in the market. Accelerating this set of algorithms in any possible way is a necessity of the whole scientific and engineering community.

The field of High performance Computing (HPC) is where algorithms requiring very long computational times are optimized to achieve ever increasing performance. While parallel processing has always been the key aspect of HPC, it also poses serious problems related to communication and synchronization that must be solved in order to support cooperative execution of programs among many processing elements. HPC has traditionally relied on general purpose processors as main devices, frequently organized in clusters or connected by means of specialized networks in supercomputers. Their model of computation (i.e. Von Neumann) is robust and reliable, so the approach to connect many processors using such model was the first one to provide good results. However, such computing model is sequential in nature and even though it has been adapted to exploit instruction level and thread level parallelism, it is not necessarily the best approach to exploit data level parallelism. So in recent years there is a tendency in complementing the processing capabilities of general purpose processors with hardware accelerators that act like co-processors and specialize in exploiting data level parallelism. These accelerators are mainly based on one of two types of devices : GPUs (Graphic Processing Units) and FPGAs (Field Programmable Gate Arrays). GPUs, that are originally designed to perform graphics applications have shown to be more performing than CPUs for a wide set of applications. This is due to the fact that GPUs can exploit data parallelism while CPUs cannot. This means that for all those algorithms where we have to perform the same operations on a wide set of data, GPUs can perform better than CPUs. On the other hand, there is also another device that is competitive when we have to accelerate algorithms: the FPGA.

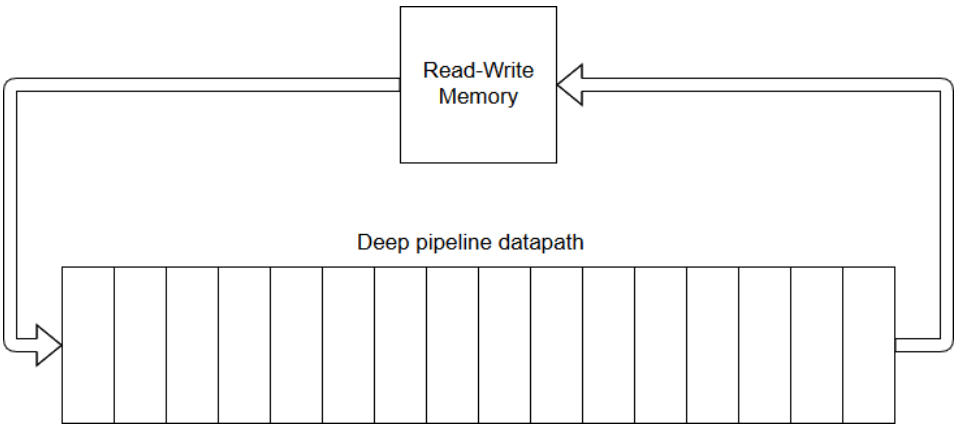
While up to recently the FPGA was merely used as a substitute of ASICs in low-volume productions, these devices can exploit both spatial and temporal data parallelism in a completely flexible way. This means that we can adapt the circuit to the application in a way that it can perform faster than GPUs and CPUs. This fact wasn't exploited up to recently, because using FPGAs involves an additional (and very complex) step than GPUs and CPUs, and that's the hardware design of the circuit that will run the application. However, new developments are allowing some degree of automation in the process. The goal is to achieve high-performance hardware implementations in reduced times. While current design tools from high-level languages have clearly reduced design times, the resulting implementations can hardly qualify as high-performance, as required by HPC applications. Therefore, there is an ongoing research effort in the direction of producing high-performance implementations for HPC accelerators. This thesis aims at contributing to the use the FPGAs to optimize the execution of a set of algorithms that today aren't yet implemented efficiently in FPGAs.

In the field of high performance computing, our research team is specifically interested in the field of Computational Fluid Dynamics (CFD). This field is a branch of HPC whose purpose is to solve with electronic hardware the equations needed to perform Fluid Dynamics simulations, that's to simulate the behaviour of the dynamic of the fluids around solids, or between two fluids. An example of a branch of engineering that extensively uses CFD is aerospace engineering: whenever we have to simulate the behaviour of a wing, an airfoil, or the whole plane when it's flying, we have to apply the CFD of equations to models that describe both the object and the fluid. The basic equations used to solve this problem are the finite volume equations of Navier-Stokes which, in turn, are based on the equation of Euler. These equations are capable of describing the behaviour of a fluid in terms of its density, energy, speed and their derivatives. The aim of the research group is to use the properties of the FPGAs to accelerate the algorithm related to the two and three dimensional solutions of the Navier-Stokes and Euler equations. While the work presented in this thesis has general application in implementations based on streaming architectures and is not directly related to any CFD application, a proper knowledge of finite volume applications and of the methods needed to solve this kind of problems are necessary to understand some aspects of the issues this thesis is about to solve.

The FPGA core advantage with respect to GPU and CPU is the capability of being fast, programmable and flexible, allowing custom circuit design adapted to the application. This means that in order to achieve maximum performance we have to exploit the spatial and temporal parallelism as much as possible as the device we are using is capable of granting this kind of optimization. Streaming architectures, where the dataset flows through a datapath using deep pipelining and many parallel functions, fully exploits the potential advantages of the FPGAs. Therefore, they are the preferred solution when considering the design of an HPC application in an FPGA. This kind of architecture differs mostly from the standard architecture (i.e. the Von Neumann architecture) of processors in the access to memory. While in a general purpose processor the access to memory is provided randomly, as we have to provide maximum flexibility to the processor, in streaming architectures the accesses are performed sequentially and data are streamed through a deep pipeline. Since the FPGA can implement full data flow graphs in parallel in addition to the deep pipelining, we can achieve a throughput that is way higher than in a standard Von Neumann architecture. Figure 1.1 shows an hypothetical schematic of the streaming architecture. The datapath consists of a series of pipeline stages of variable width, and the access to the memory is sequential [1]. This means that the memory itself doesn't behave like a RAM but like a FIFO. It should be noted that in a pure streaming architecture a continuous flow of data is processed in the device as in a digital signal processing (DSP) application. This

is due to the iterative nature of the HPC application being considered. Since the datasets correspond to large meshes processed using finite volume methods it takes much longer to transfer the dataset than to traverse the processing device and the streaming architecture can be safely applied.

Figure 1.1. Basic model of Streaming Architecture



1.2 Objectives of this thesis

The purpose of this thesis is to create a FPGA module that will manage the interface between a on-board DRAM memory and the memories inside the FPGA itself. The design will be based on the streaming architecture concept. The module will have to decode the data that are encoded in an optimized way inside the external memory. Inside the external memory data are organized as packets. This module has the primary purpose of feeding the datapath with one packet in each clock cycle. Since packets have different lengths, this is an ideal and impossible requirement, and some control will be added to manage some cases. The decoding of packets is performed to achieve maximum performance. Therefore, the module will receive sequential data in the external memory port, it will extract the packets of different lengths, and will store them in individual RAM memories so the packets can be processed by the rest of the system.

This module will have to satisfy the rules imposed by the streaming architecture approach and the paradigm of high performance computing that is to keep the performance as high as possible. It will also be taken in consideration the area taken by the module, as this module has to co-exist with the remaining part of the

system that is programmed as well inside the FPGA. We will see later that this module can easily become the bottleneck of the whole system, and therefore it is crucial to optimize it as much as possible, as the performance of the whole system depends on the performance of this module;

The design has been thought and subsequently made to be as generic as possible. The data coming from the memory will not be treated as specific to the system itself, but will be treated as abstracted multi-purpose data, which is stored in external DRAM and transferred sequentially to the FPGA as a frame of packets with different lengths. In fact, the frame design (i.e. the most appropriate way to optimize the sequential data) is also an important part of the work developed here. This fact is the main point of separation from the rest of the system: this module has been intended not as a part of the algorithm of acceleration but as an interface between the memory and a generic algorithm inside of the FPGA that uses the streaming architecture concept. However, the CFD application will provide the specifications and limitations in the working conditions that need to be considered in order to avoid too many design choices. Still the module accounts for a very small part of the design itself, it basically just defines the input configuration of the design. In order to implement the design of the module, we'll have to go through a process of decision making and a process of determination of the specification before arriving to the design itself.

This module implements is a new concept in FPGA design, since as far as we know, the streaming architecture hasn't been used for CFD applications before. This module therefore has been designed from scratch; the only previous work has been theoretical and performed by the research team. It is important therefore to explore in detail the challenges posed caused by the streaming architecture concept but also coming from the performance optimization goal, these topics will be developed and discussed in this thesis as well.

In summary, the core points of this module and therefore the main objectives of this thesis are:

- Design of a data decoder of packets received from memory that will provide the data for a pipelined design in the FPGA implementing the stream processing model of computation. The main challenge of the design is to handle packets of different sizes at maximum performance. This is a novel approach that fills the requirements to stream process unstructured data, as other previous approaches to data decoding have always relied on packets of fixed width.
- Data frame design: structure of the sequential data from external memory to favour fast decoding in the interface module.

- Performance optimization: The performance of this module will affect the performance of the whole system and therefore it's crucial that the module is optimized in terms of performance.
- Area limitation: As performance can be usually improved by using more computational area, we cannot abuse it as this module has to co-exist inside the FPGA with the remaining part of the system. However, this is a secondary goal.
- Portability and abstraction: We will abstract the module from the CFD application, and we will make the module independent from the specific algorithm implemented inside the FPGA.
- Streaming architecture: This module is intended to be used for a streaming architecture based system and it will follow the principles of this kind of architecture, including deep pipelining and control flow.

The software used for the design is Quartus Prime 16.2 [9], an IDE made for the synthesis of the VHDL designs for Intel's Altera FPGAs; The whole project was initially intended to be implemented on a GiDEL PROCStar III board [8] with Stratix 3 FPGAs and later on a recently acquired Nallatech 520C with s Stratix 10 FPGA [7], but as this design is intended to be generic, the board shouldn't be relevant. For the simulation of the system ModelSim 2 VHDL edition of 2008 [10] has been used. Also Python 3.6.5 [11] has been used to do the configuration part of the system. This software has been run in a PC with a Intel i7 processor running a Windows 10 operating system.

1.3 Thesis structure

This thesis is composed by six chapters and two appendixes. The first chapter contains the introduction, where we have discussed the background and objectives of the thesis. In the next chapter, the core topics of this thesis will be addressed : the problems arising from unstructured meshes, how from the definition of unstructured mesh we arrive to the definition of the problem of this thesis, which is to decode packets of variable lengths. It is also discussed the overall view of the system, the modules that aren't included in this work but that are necessary to the processing of the algorithms. An analysis of the streaming architecture and the definition of its main rules will follow. The chapter will finish with the definition of the optimized data frame and the specification of the Packet Decoder.

Chapter 3 will define the specifications of each component of the Packet Decoder starting from a detailed specification of the Packet Decoder itself. It will

also present the Packet Decoder from an external point of view, defining the relationship between its components and the interfaces with the external modules. In the end, it will also present the design techniques and the coding style adopted during the development of the design.

Chapter 4 consists of the description of the implementation of the four components of the Packet Decoder. It will have a back to front set up as it will start from the component interfacing with the outputs going back to the component interfacing with the inputs. Each component will be analyzed and described in detail.

Chapter 5 shows which kind of tests have been made to verify the Packet Decoder and the results obtained from these tests. The first part consists on a functionality analysis, where we prove that our design works correctly and that complies with the specification made in Chapter 2 and Chapter 3. In the next section, the resource and performance tests are presented.The main parameters analyzed in that section are are clock frequency and Resource utilization.

Chapter 6 contains in the conclusions, where there is a summary of the achievements of this thesis and a section dedicated to further improvements to the design. Finally there is the bibliography, where all the sources used to write this document are cited.

Appendix A is a corollary to the maths used in Section 4.3. Appendix B shows the results of the functionality test described in Chapter 6. The input frame and the relative output frame of the testbenches made to verify the functionality of the Packet Decoder are included.

Chapter 2

Data encoding for stream processing

2.1 Computational Fluid Dynamics

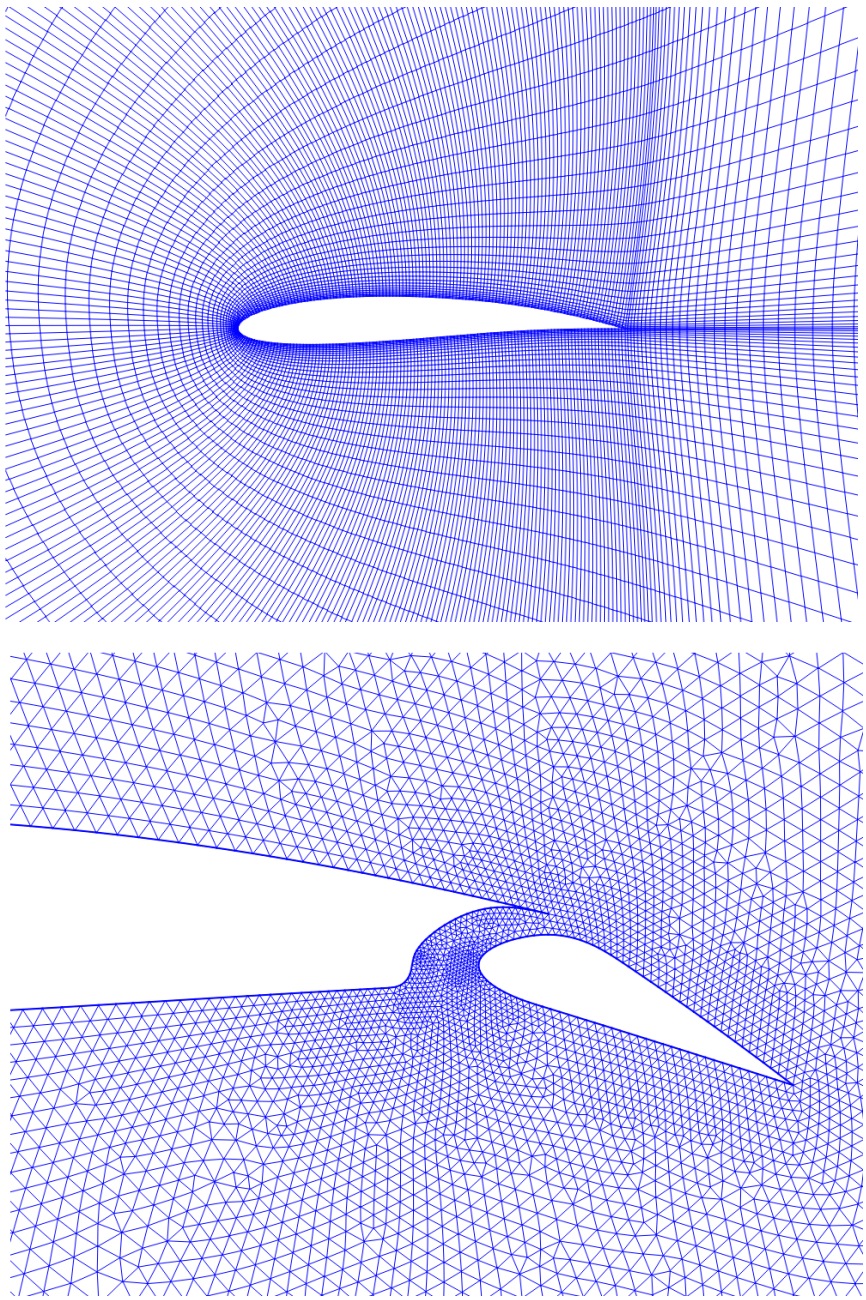
CFD algorithms are very important for aerospace engineering, as they obtain the solution of a set of differential equations that describe the interaction between different fluids or between fluids and solids. As with many other HPC algorithms, the equations that model the fluid dynamics are way older than their actual implementation as algorithms. One of the milestones of fluid dynamics are the Navier-Strokes equations, that define the dynamic of non compressible fluids by means of momentum conservation, mass conservation and energy conservation laws. Starting from these equations we can determine the various properties of the flow of fluid around surfaces. This is very important for stability studies of aeronautical industries. The most frequently used numerical methods used to solve these equations around a surface are the finite volume methods. They are based on the representation of a surface and its surroundings as a graph of vertices and edges. This system representation is widely used by NACA, the association that determines the aeronautical standards in the USA. The base concept of this set of algorithms is to compute the needed differential equations within a grid, that will represent the space around the object that is submerged into the fluid. For each point of the grid, finite volume methods will compute the requested variables by solving the differential equations around the point. Therefore, the grid is defined as a set of differential volumes around the points and the flow equations are solved in each volume. This process is iterative until a stable solution is found. The main problem that arises from this algorithm is that since we have an increasingly complex grid (i.e. more and more complex objects and scenarios are simulated), the algorithm

to compute these variables is computationally expensive because the equations are computed in millions of volumes (cells) in the grid over and over again until the result converges. So to make the simulation efficient and complete, there is a need to run such algorithms in a way that is the most performing possible. This is why Computational Fluid Dynamics is a branch within the HPC family of applications.

The grid describing the volume space around the object constitutes the dataset to be processed by the algorithm. It's usually represented as a mesh of vertices connected by edges describing the control volumes where the differential equations are solved in every iteration of the algorithm. Typically a mesh is described as a connected graph where each vertex is connected to at least other two vertices. The algorithm can be computed by traversing the vertices or the edges of the mesh, but in any case the algorithm has to be performed on each vertex or each edge. Whether we choose to compute the algorithm based on edges or on the vertices is a design choice. Meshes can be structured or unstructured as shown in Figure 2.1. The first image shows a structured mesh: each point in the space has a fixed number of neighbouring vertices, and the space is homogeneously described with polygons each one having the same order. In the case of the chosen figure, the polygons are all rectangles and the order of connectivity of each vertex is four. On the other hand unstructured meshes do not have a fixed connectivity: the vertices have different degrees of connectivity.

We can compare the two kinds of representations in terms of their connectivity and number of vertices. While both represent correctly the volume, the structured mesh has a fixed connectivity degree for each vertex. Therefore, if great detail is required in some parts of the volume, also great detail must be used in the rest of the volume. On the other hand, unstructured meshes describe more easily different parts with different densities of vertices. So the problem with structured meshes is that to represent the same volume with the same precision we need a higher amount of vertices. This fact becomes a mayor problem as we reach the critical parts of the representation. While for most regular parts - the external parts - this isn't a problem as the required precision is not high, in the part where the computation has to be more accurate the structured mesh requires more vertices than the unstructured mesh. In other words, unstructured meshes adapt better to the precision required in the different parts of the volume. This is a major problem in terms of performance, as for structured meshes we would have to compute the equations on each vertex/edge and the more of them there are, the greater the workload to reach a solution. This the key reason why the trend in HPC is to use unstructured meshes instead of structured ones.

Figure 2.1. Structured mesh vs unstructured mesh



A key aspect of unstructured meshes is that they are irregular: we do not know a priori what is the connectivity of each vertex. It may vary from a number that is very high in the most critical part of the mesh up to a small number in the border of the mesh. Therefore, unstructured meshes generate irregular memory access patterns when considering a Von-Neumann model of computation in CPUs. Even though GPU's support parallelism, they also rely on random memory accesses, so the irregular patterns caused by unstructured meshes are not well suited for GPUs either, particularly considering their fixed regular architecture. FPGAs using a streaming architecture seem less affected by these irregularities, so this is one of the reasons why improved performance is expected when unstructured meshes are processed in FPGAs. However, one key point is to appropriately store in memory the mesh data so it adapts well to streaming computation model in the FPGA. This typically involves sorting the mesh vertices.

The processing of the mesh is performed by solving the differential equations that describe the behaviour of the fluid around each vertex of the mesh, that is, in each control volume of the mesh. Such computation depends on the data associated to the vertex and its neighbours. When considering stream processing of the mesh in FPGA, the mesh is read from memory and flows through the datapath in the FPGA where the processing takes place. Therefore, it is crucial that when the equations are computed in the control volume of a vertex, its data and the data of its vertex neighbours are available at the time inside the FPGA so computation can proceed. In other words, the mesh should be coded in memory so that each vertex and its neighbours are located as close as possible in the flow of data arriving from the memory and entering in the FPGA. The goal is that the mesh data is only sent once to the FPGA per iteration, so all uses of a vertex data (to compute its new state or the new state of its neighbours) must occur while the vertex data is in the FPGA as part of the flow of mesh data. The term ' vertex window ' is used to describe the distance in terms of mesh data between the first use of the vertex data and the last use of the vertex data during the computation. Therefore, in order to achieve that at each moment in time the FPGA is storing a vertex for the duration of its window, the mesh data has to be organized so that the maximum window size is minimized. It has been proposed [3] that the best way to achieve this is by encoding the mesh by sorting its vertices according to a breadth-first traversal of the mesh. The root vertex where the traversal starts can have a significant impact on the maximum window obtained, so there are different proposals to detect such root vertex (an NP-complete problem).

As the breadth-first traversal proceeds, vertices and edges are added to the mesh representation. Note that typically a mesh (a graph) is described by a list of vertices including the vertex data, and a list of edges including the identifiers of the edge's endpoints and the edge data. However, separate lists are not the

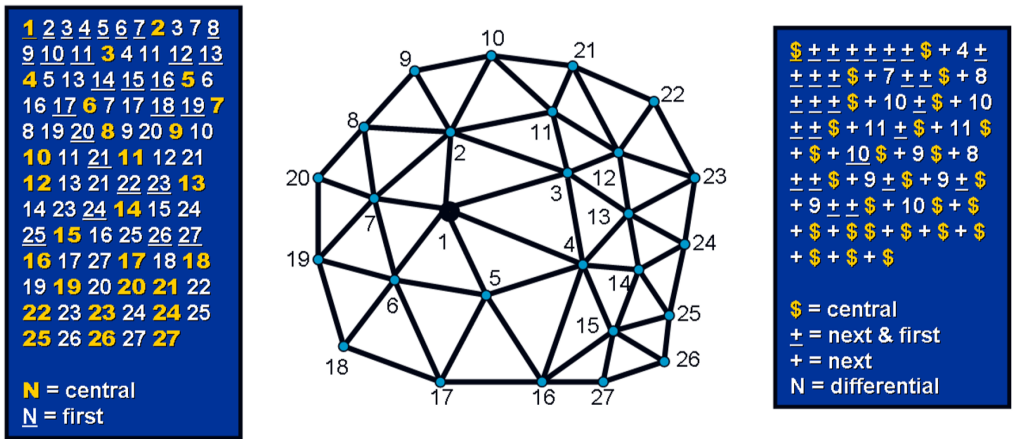
most efficient way to describe the mesh for stream processing in an FPGA because the FPGA does not have the storage resources to store the lists, so it is much better to have a unique stream of data that includes both, vertices and edges, and that flows from memory to FPGA. Mesh data can be classified as geometry data, including connectivity (i.e. edge endpoints), coordinates, volumes, normals etc. and algorithmic data (i.e. the actual conservative variables associated to vertices that must to be stored between iterations, like densities, velocities, energies etc...). Geometry data are assumed to remain unchanged so it is read-only data, while algorithmic data changes in every iteration so it is read-write data. Since FPGAs in accelerator boards usually include more than one external DRAM per FPGA, it makes sense to store read-only data in one DRAM and read-write data in a different DRAM. Since algorithmic data is associated to vertices, once vertices are sorted as a result of the breadth-first traversal, algorithmic data can be simply read or written following such ordering. Since all vertices have the same amount of algorithmic data, memory accesses in this case are performed sequentially and only require the appropriate buffering in the FPGA [2]. Geometry data, however, includes the description of the geometry which as previously mentioned, is highly irregular and requires more careful treatment. The memory interface module developed in this work solves the problem posed by such irregular data. The results of the breadth-first traversal of the mesh are represented on the left of Figure 2.2 Starting from vertex 1, its neighbouring vertices are included thus describing the edges connected to vertices 1. Then each such neighbours is visited (its identifier is noted in hold) and the corresponding neighbours not previously included are added to the mesh description. This continues until all vertices in the mesh have been visited. Only the first time a vertex identifier is included in the description, its geometry data are also included. Such identifiers are underlined in the description of Figure 2.2 . Edge data is included when listing each neighbour of a vertex as it defines the other endpoint of the edge.

However, as the mesh is represented in this way, the first vertices hold much more information than the last ones. This means that we cannot encode each vertex in a memory line of fixed length, as it would be extremely inefficient. Not only some vertex identifiers have associated geometry data, but the number of neighbours per vertex is not fixed. So considering the units that compose the mesh description and that we call packets, it is clear that a mesh will require packets of different sizes to be described. Examples of packet descriptions can be "vertex with data", "vertex without data","edge with data","two edges without data" and so on.

Once the mesh is represented it goes through a phase of compression. Since vertices are numbered as the mesh is breadth-first traversed, it is observed that visited vertices have increasing identifiers, so the identifiers can be substituted by

a flag ($). The neighbours are also included following increasing identifiers which in some cases are consecutive. Again, in these cases a flag (+) can substitute the identifiers, and only when they are not consecutive the identifier is represented by the offset with respect to the consecutive identifier. This compression method reduces the size of the connectivity description of the mesh up to 90% as the nodes can be listed in binary form and not with unsigned numbers representing identifiers. Once the grid is encoded, it is going to be stored inside an external DRAM memory and the sent to the FPGA, where subsequently will be decoded and then used by the algorithm.

Figure 2.2. Encoding and compression of an unstructured mesh [3]



2.2 System description

This thesis deals with a part of a larger project that is about the acceleration of CFD algorithms processing unstructured meshes using FPGAs. The procedure to elaborate this system starts from the encoding and compression of the mesh described in the previous section. This procedure is done in software on the host computer. The whole system exploits the advantage given by the fast programming of the FPGA. The whole design of an algorithm is made generic and configurable to support different kinds of meshes. A type of mesh is what defines the different types of packets used to describe it. This is achieved by dividing the project in two different parts. The first part, done on the host computer performs these operations:

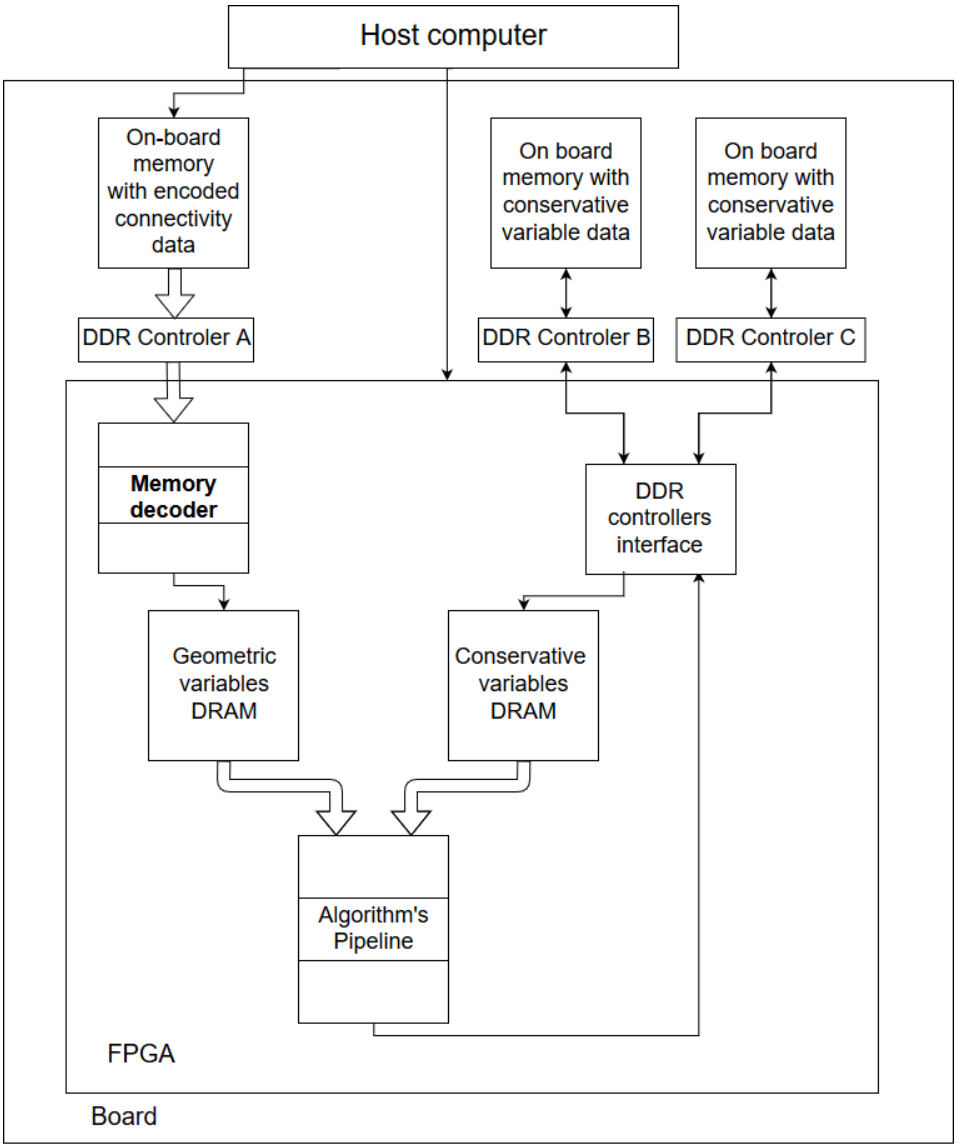
- Mesh encoding and compression
- Configuration of the design
- FPGA programming

The mesh encoding and compression has been described in the previous section. The configuration of the design is done by software. It has the purpose to adapt the system to the particular mesh type to be processed. In this case it will be performed by a Python software before the programming itself. It basically changes the VHDL code in those cases where generics weren't enough to make the VHDL code generic. The FPGA programming consists of the configuration of the FPGA for the particular algorithm and interface, that at this point is adapted to the needs of the mesh type. In this phase, the CFD algorithm is translated to VHDL directly from a C specification with an in-house tool developed by the research team. It then goes through a synthesis phase, that depends on the board, on the FPGA and the tools provided by the FPGA vendor, through an implementation and assembler phase where the synthesized design is translated into the board's devices, and finally to the bitstream phase where we program the FPGA itself with the VHDL code we have generated. Of course performance, area and power consumption depend on the way the system is synthesized and implemented on the FPGA.

The second part of the project loads the mesh to the board DRAMs and runs the algorithm that was programmed on the FPGA. As we can see in Figure 2.3 the design in the FPGA can be subdivided into three parts:

- Algorithm's Pipeline
- Geometric variables decoding
- Double data rate (DDR) interface

Figure 2.3. Schematic of the whole project



The algorithm's pipeline is the implementation of the CFD algorithm itself. As we have seen, it is designed as the rest of the system in a streaming architecture fashion. The design is based on deep-pipelining and also includes flow control to minimize the impact of stalls in the stream caused by the memory interface or inherent local feedback loops when performing reduction operations (accumulation, max, min). The interesting point of this part of the system is that instead of computing the algorithm by traversing the mesh by vertices, as in the mesh connectivity coding, the mesh is now traversed by edges. This happens for a simple reason: while vertices need a changing number of neighbours to compute them, edges only depend on two vertices, the endpoints, to be computed. This makes the design more straightforward and simple.

This pipeline performs all the operations of the CFD algorithm itself. This algorithm is iterative and requires two kinds of data [1]: data about the geometry of the mesh and data about the conservative values of the algorithm: energy, velocity and density at each vertex of the mesh. All these data except for the edge data are stored in block RAMs inside the FPGA when they arrive from the external DRAMs as the stream of data flows through the FPGA. The edge data are stored in a FIFO so it is the ordering of edges what controls the computation. On every cycle the computation of a new edge starts. The edge endpoints act like addresses for the internal block RAMs that are accessed randomly and provide the data of the two vertices required to compute the edge. This data, along with the edge data (i.e. its normal) enters the algorithmic pipeline where it proceeds for hundreds of stages to compute one iteration of the algorithm. The resulting conservative variables are produced at the end of the pipeline and stored in the external DRAM that keeps the read-write data. This process repeats for every new iteration while some internal control variable data detects that the conservative variables have converged to stable values (or for a predefined number of iterations).

The data representing the connectivity are static data that are written to a DRAM memory during the configuration phase. Once they are sent from the memory to the FPGA they have to be decoded in order to be used in the pipeline. This is the part this thesis deals with. We have to feed the algorithm's pipeline with the semantic and connectivity data of the mesh, and since we need to perform this operation at maximum speed, we have to decode the data frame arriving from the memory in a way that supports maximum performance. The data stored here are static, they don't change during the execution of the algorithm, and they only need to be read at the beginning of each iteration. This will be explained in detail in the next sections.

As previously mentioned, the handling of the conservative values (i.e. read-write data) is performed directly by the DDR memory controller since they are

simply read and written sequentially. Since these values are both requested at the beginning of the iteration and are also rewritten at the end of the iteration they require an interface to manage these variables. These read-write data are stored on an on-board DRAM memory and need to be transferred to and from the memory of the FPGA. It has been shown in [2] that the most efficient way to manage these memory accesses is to use two memories at the same time: at each iteration one memory stores the new results of the algorithm while the other provides the algorithm with the data of the previous iteration. When an iteration is completed, the two memory switch their roles: the memory that has been read is now written and the memory that was written is now read. The DDR interface has to make sure that this operation goes without problems and at maximum performance. Differently from the memory decoder there isn't any encoding and decoding of the values provided by the algorithm. They are simply stored in sequence and read in the same sequence in the next iteration. So this interface does not have to deal with problems posed by the interface that handles the connectivity data which is the interface designed in this work.

2.3 Memory bottleneck

The key issue of this project is to achieve a very high performance implementation. The streaming architecture grants us the possibility to have an extremely performing design when it comes to the execution of the algorithm, as we can add as many stages in the pipeline as needed. Furthermore, this system has also the priority of being as simple as possible in order to avoid overheads in the design. As previously mentioned, the computation of the algorithm is performed traversing the mesh by edges, because this avoids the complexity implied by a vertex-based traversal: an edge requires two vertices to be computed, while a vertex may need a variable number of edges not known at compile-time.

However, the same approach is not applied for the storage of the grid connectivity data. As we have seen in Section 2.1, the key point of the design is to optimize the communication and thus we need to compress the information about the connectivity as much as possible. This translates to the fact that we cannot use edges to represent the grid when storing the information about the connectivity inside the FPGA, because each edge would be represented with two vertices that surely would be listed more than once, thus causing an overhead in the representation. Consequently, the grid geometry data is represented traversing the mesh by vertices and coding the data as described in a previous section. The vertices are therefore decoded and translated into vertices inside the algorithm's pipeline by the module that is described in this thesis.

The key point is that in order to grant the functioning of the system we do need both the geometry data and the conservative data. In order to have the algorithm to perform its operations we need to provide the geometry and conservative data at a rate that matches the execution of the the algorithm pipeline. The throughput of both the memory decoder and the DDR interface have thus to be optimized at the maximum in order to grant the success of the system. When considering the throughput of the internal pipeline which processes one edge per cycle and comparing it with the amount of data that must be transferred from memory and the maximum frequency of the DDR interface in the board being considered, it happens that the throughput of the algorithm pipeline is greater than the bandwidth of communication between the on-board memory and the FPGA: there is a higher throughput of edges from the algorithm's pipeline than of vertices from the memory. This makes the design of this decoder a key part of the system as it deals with the bottleneck of the execution of the whole system. Therefore, it is vital to optimize this module as much as it is possible.

This problem translates into three types of performance optimizations:

- Optimization of the encoding and compression of the information
- Optimization of the data frames with mesh information
- Optimization of the implementation of the hardware decoder that converts data frames into geometry data

While the optimization of the encoding and compression of the grid has already been addressed by the team in a previous work [3] as explained in the previous section, the two remaining optimizations are approached in this thesis and they will be discussed later on this work. The optimization of the encoding of the grid's information simply means that in order to grant the maximum performance we have to store inside the memory as much as information as possible using as few bytes possible, so that each time we will read from the memory it will have the maximum possible vertex's data density. In other words, we want to transfer the mesh in the minimum possible time. This is why the representation has been chosen to be made in terms of vertices, as we cannot afford any kind of memory overhead: the edge representation would lead to a memory overhead that is intolerable. The optimization of the data frame with the mesh information deals with the problem of transferring the data from memory to the FPGA. The grid data encoded according to the previous optimization is a series of packets of different sizes. These packets have to be assembled in a continuous sequence to be stored in memory and issued to the FPGA where it will be decoded. This sequence has to include the packets and some additional information like the packet type (i.e. size) or markers to identify the end of the sequence. We call this sequence the data frame. The structure of the

data frame plays a crucial role when the decoding of the mesh has to be performed at maximum speed. Such structure is analyzed and defined in the last section of this chapter. The optimization of the implementation means that starting from the encoded information received in the FPGA we have to restore the meaning of the mesh information, that has been fragmented for the sake of performance, in the most efficient way possible within the paradigms of a streaming architecture.

2.4 Streaming architecture implementation techniques

Pipelining

The streaming architecture implies that the data are transferred sequentially and processed in a way that data packets are processed independently from each other. This allows us to use one very powerful technique to improve the performance of the design: pipelining. The concept of pipelining is based on the fact that the maximum frequency we can obtain from a design depends on the critical path of the combinational logic between two sequential circuits [4]. The longer is the combinational logic path, the lower will be the frequency of the design. This wouldn't be a problem if we have to perform an atomic operation, but since we are working with a long sequence of data packets, the whole computational process has to be repeated for each packet. This leads to the metric that influences the most: throughput, that in our case is evaluated as $\frac{data\ packets}{seconds}$. Without the use of pipelining the computation would be very slow as the time needed to process each data packet would be very high. Since we cannot eliminate the combinational logic what we can do is to introduce stages (i.e. registers) so that at each stage the combinational logic is freed and ready to compute a new packet as soon as it has performed the operations on the previous data packet. Therefore the less complex it is the combinational logic, the faster will be the system. Hence pipelining. This technique is made to reduce the length of the critical path in a digital circuit. Pipelining asserts that in a DAG (direct acyclic graph) we can add a stage of delays in the circuit if this stage cuts the circuits in two separated sub-graphs without altering its functionality.

Flow control

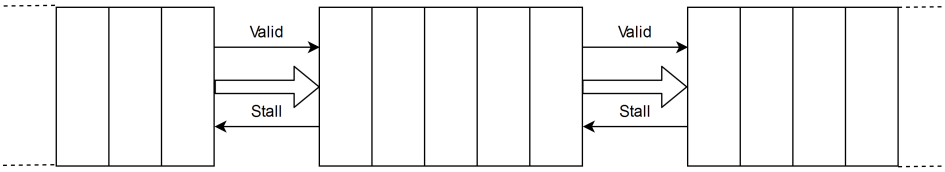
The whole system is based on the concept of a streaming architecture. To grant maximum performance we have to exploit all the strong points of the FPGA. In our case is the temporal parallelism: in all the system each operation is divided in as many stages as possible in order to grant the maximum operating frequency. Even though pipelining is implemented in all parts of the design the fact that processing speeds are different in the memory interface and decoder (conditioned by the memory bandwidth) and in the internal pipeline leads to a problem regarding the availability of the data, particularly since we are trying to exploit the algorithm for irregular data structures. This problem is approached by implementing some method of flow control among the complete pipeline structure that allows to stall some part of the pipeline if data cannot advance while allowing that other parts of the pipeline still continue with the processing. Necessarily there will be clock cycles where data aren't valid, it can be a partial form of a valid data that is going to be ready in a few clock cycles. This type of data has to be discarded in a way or another and the most effective way is not to let the next section to take them as input. The whole structure can also be busy doing an operation on the same data at a certain moment. This makes it impossible to take new data in the module.

Since there are many operations to perform and they differ from each other, to simplify the design it is necessary to divide the design in multiple independent parts or modules that perform different groups of operations. Since all these operations are done in a pipelining fashion, all the modules behave similarly from an external point of view: the input stages have to be ready and not stalling to allow the insertion of new data in the module. On the other hand, all the outputs have to communicate the validity of the value they are sending to the next module in order to prevent errors. It may happen that some operations take more than one clock cycle to be performed. During this time, the output of a module is not yet the correct one, and thus it isn't valid. This means that each output has also a line which communicates the validity of the data, that each input has a line which communicate its availability to the previous stage, that is whether the stage can or not take any new input. This means that in order to proceed with the stream of data of a module we need to have this two conditions: The output of a module has to be 'valid' data and the next module has to be 'ready' to take new data. If these two conditions occur the data can be transferred from the source module to the next module. However, it remains to be defined how the source module behaves when the next module is not ready to take new data. Depending on how we implement the module we can still proceed with the execution of some valid data inside the module if the previous condition is not met at the output of the module. A module can therefore be implemented in two ways :

- Rigid
- Elastic

The rigid way is a simplified way where a module can work if and only if there isn't a stall. Both data, valid and invalid are going to be frozen inside the pipeline if a stall appears at the module's output. The elastic way is an optimization that allows the processing in a module if a stall appears at its output by discarding the invalid sections stored inside the pipeline. Once the pipeline is filled with valid data and if the next module is still stalling, the module will freeze. This concept is valid through the whole design and it applies to every module, including the memory. It is clear that in order to achieve the maximum possible throughput, it's necessary to reduce to the minimum the number of stalls and the number of invalid data as they take away precious clock cycles. The concept of flow control is implemented in each component of the system in order to grant independence and modularity of each operation, so that each module can be implemented independently from the others as a black box.

Figure 2.4. Interfaces of modules with flow control

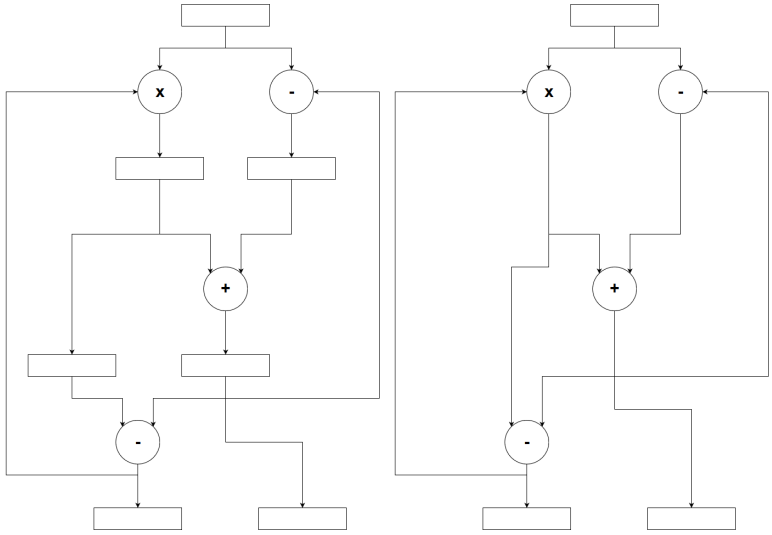


Loop reduction

The whole concept of the streaming architecture is based on pipelining. Unfortunately, pipelining goes with a big problem: whenever there is a loop inside a pipeline, a set of hazards may occur. This means that if there is a loop, there should also be a whole part of the design dedicated to manage the hazards the loop would generate. As our design has to be as optimized as possible, hazards may lead to a whole set of delays and area waste that are inconvenient for our purposes. Unfortunately, in multiple parts of the design there are local loops that cannot be removed while maintaining the functionality. The way we deal with them is simple: we don't deal with them. Since all the problems of the hazard come to effect if there are pipeline stages inside the loop, we avoid to put any kind of stage inside the loop, and all the combinational part of a loop is kept between two pipeline stages. This may become a problem regarding performance if the combinational part of the loop is not small; it will lead to a slower module. However, the loops

that occur can be reduced up to a point where they do not cause such problem, as all the unnecessary parts can be computed outside the loop stage. As we will see later on, the loop part in the memory decoder module is the longest path and it will be the critical path of the whole design. Figure 2.5 shows two examples of graphs with loops. The one on the left is not valid, as it adds delays to the graph that are inside a loop and this translates into hazards. The graph on the right is valid instead, as all the computations that have to be done are inside a single pipeline stage

Figure 2.5. Invalid and valid graphs



2.5 Summary of the key issues in the design of the Packet Decoder

This thesis is about the design and the optimization of the part of the system dedicated to the encoding and decoding of the data regarding the connectivity of the mesh grid. The problem by itself isn't complex, as we just have to process a series of encoded data. The key issue is that we have to perform this operation optimizing the performance as much as possible. The challenge of the whole design relates to the data representation: we are going to transmit data packets that are variable in length, the vertices of the grid can have any number of neighbouring vertices and the vertex data is only transferred the first time its identifier appears in the

sequence. Thus this translates into packets of data that aren't fixed in length. The second aspect of the problem is that we do have a bus to communicate between the FPGA and the memory, and this bus has fixed width. We do have to find a way to fit this variable information inside the memory in a way that can be transmitted in an efficient manner.

Another aspect is portability. The design has to be portable for different boards and thus a generic design had to be made, since different boards have different memory bandwidths and different widths in memory ports. Furthermore, the design of the decoder has to be valid for whichever kind of mesh. Different types of meshes include different types of geometry data and lead to different packet sizes, thus leading to different designs. This is a key point in the specification as the design has to be made not only portable for different boards, but adaptable to the needs of each mesh.

The performance optimization also declines into the hardware optimization. It is crucial to obtain a maximum performance when we are designing the hardware module, as we have to grant that the whole system works at the maximum frequency possible. This problem will be addressed with the use of the streaming architecture and therefore applying its specific design techniques. Regarding flow control we select the elastic approach that grants that if there is a stall in the next sections of the pipeline, the previous parts will stop elaborating only if all the stages of the pipeline are filled. This means that once the stall is over, the previous section won't have any invalid data.

The last key issue is area. This design has to be placed inside a FPGA, and while area isn't a key problem of the design, we cannot accept a solution that causes an extreme overhead in area or power, as the FPGA resources are shared with the remaining parts of the system.

2.6 Data frame optimization

One critical aspect of the design of the data decoder is that unstructured meshes translate into irregular data. This means that they cannot be represented in a way that matches perfectly the fixed memory port width of the design. Since the communication between memory and FPGA is the bottleneck of the whole design, an efficient encoding of the meshes is required in order to grant the best performance. Unstructured meshes are encoded as explained in Section 2.1. This encoding has to fit inside the on-board memory in the most efficient way possible. The most effective way to fit the compressed mesh into the memory is to divide each set of

data representing the connectivity and geometry of a vertex in a packet of variable length. The packets describing the mesh geometry will be referred to as the payload. Each vertex will have a corresponding packet with all the information necessary to represent it and its connections to its neighbouring vertices. The problem is that since there are vertices that have higher connectivity than others, they hold much more information than other vertices that have lower connectivity.

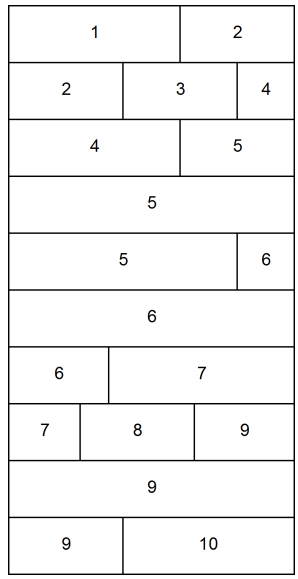
Therefore, since all the vertices are ordered, the first data of the stream will have a larger packet, while the last data will have a smaller ones since edges (connections) are only included in the data stream. In conclusion, all packets have a variable length that may or may not be larger than the memory port width and can also be zero if all related data has already been sent. This method also abstracts the module from the application itself: by defining the packets as generic memory information we do not care anymore whether the data inside the payload is a vertex or whatever else, we just care that this data has some lengths from a possible set of length that goes from a maximum number down to zero.

It is important to realize that the memory itself behaves like a streaming architecture module. It has valid and stall lines, and at each clock cycle it transmits a line of memory of the width of the bus. It basically behaves like a FIFO. We have to grant that each line transmitted holds as much useful information as possible. To achieve maximum performance, we store the beginning of a new packet after the end of the previous one, filling therefore the whole memory. Figure 2.6 shows the packet encoding in terms of memory lines of the size of the memory port. Each number corresponds to a packet, and each packet is sliced into multiple parts to fill each line of memory. Each time there is a transmission from the memory to the FPGA, the maximum data possible (i.e. a memory line) is transmitted per cycle. It is however important to consider that each packet doesn't have a clear distinction as seen on the image. Each packet is composed as a series of bits and there isn't any kind of way to distinguish between the end of a packet and the beginning of the next one from the point of view of the decoder, as they look as a uniform series of bits.

In the end, these data will have to be recomposed to the original form, as they hold no meaning for the purpose of the algorithm encoded in this way. The goal of the memory decoder is to start from an encoded frame like in Figure 2.7a and to end with a set of separate packets as shown in Figure 2.7b. The whole hardware design has to translate the encoded form in Figure 2.7a into a dataset that looks like the one proposed on the right of the Figure of 2.7. The specification of the design data frame to be used by the memory decoder will revolve around this problem.

Each mesh has a unique correspondence with its encoding. This means that

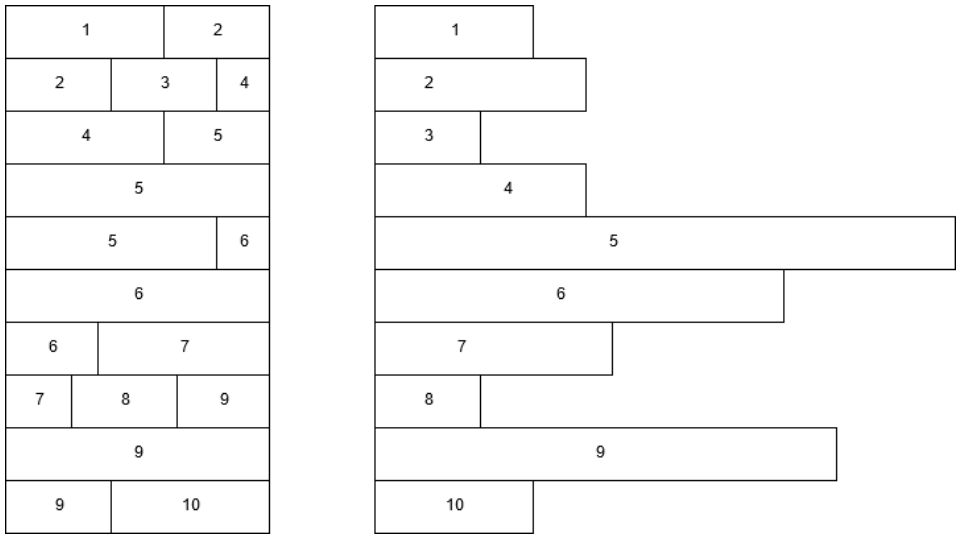
Figure 2.6. Efficient packet configuration example



we can determine the possible sizes of the packets before the execution of the algorithm. The set of possible packet lengths is determined when the mesh is pre-processed at the host. The size of each packet is however necessary to the design, as each packet may have whichever length among all the possible lengths declared at configuration time. It is certainly an information needed at execution time, as we do have distinguish between the end of one packet and the beginning of the next one in order to recompose them correctly.

This information (i.e. the length of each individual packet) is therefore added into the memory and transmitted to the FPGA along with the payload information. This new information is called header. It can be a meaningful number or just a unique series of bits so that each series maps to a packet length. A main problem that arises from this kind of encoding is how to determine whether some data is header data or whether it is payload. There are multiple possibilities to deal with the headers. Since this problem is strictly related to the decoder it can be dealt with multiple approaches. All come with some advantages and some drawbacks. They are analyzed in the following subsections. The design of the data frame has been performed as part of the design of the memory decoder in this thesis.

Figure 2.7. Encoded and decoded packets



2.6.1 Header before the payload

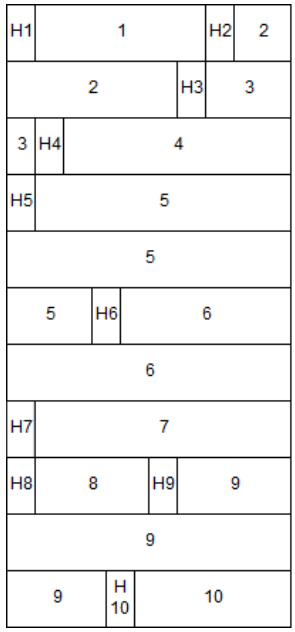
In order to separate each packet from the next one, a first simple and intuitive approach is to separate each packet with a unique series of bits. However a series of hard problems emerge when selecting this approach :

- This series of bits has to be reserved and cannot be found inside the payload.
- The size of the payload remains unknown and therefore it is complex to make the reconstruction at fast speed.
- A long header causes an unnecessary overhead in the mesh representation in the memory

This approach reveals to be too much of a mess and therefore a slightly more complex solution is taken into consideration: to provide the information about the length of the payloads in the headers. While this approach is memory effective, as we always know which part is payload and which part is not, it is highly problematic due to the fact that the decoder doesn't know in advance the dimension of each payload, so the information about it becomes just an offset to the payload size. Not knowing the size of the payload in advance would also apply on the header, as it just would become and extension of the payload. Those facts would lead to performance inefficiency: the lack of instantaneous knowledge of the width of the

packet would mean that each time a new packet arrives, we would have to wait until it's effective length is computed. This translates into a waste of clock cycles and therefore performance inefficiency. However the advantage of this design is that we can perform an efficient memory encoding. Figure 2.8 shows this frame encoding: each small rectangle before each payload encodes the size of the payload that follows it.

Figure 2.8. Data frame using the "header before payload" approach



2.6.2 Lines of headers and lines of payloads

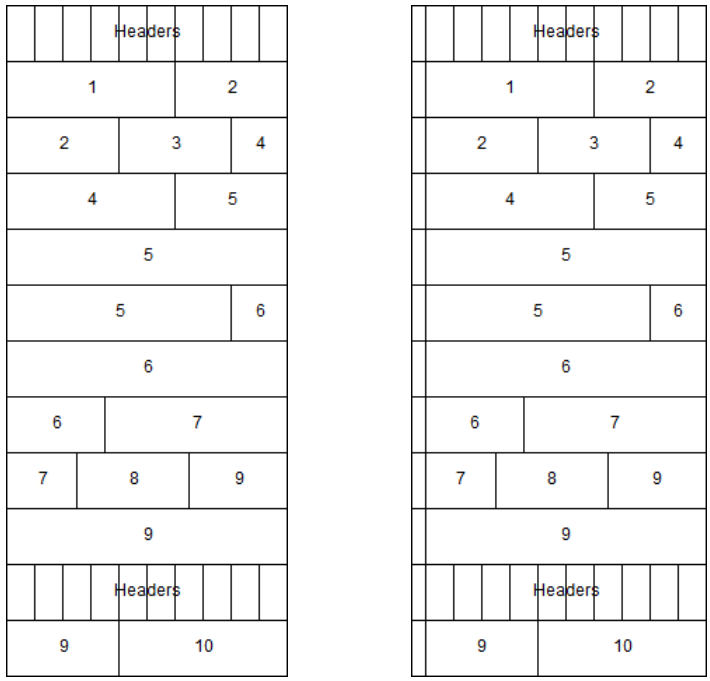
An alternative to the previous approach is to send several headers before the corresponding payloads, so by the time the payloads are received in the FPGA their lengths have been properly computed from the information in the headers. With this approach we make a distinction between lines of headers and lines of payload. A memory line can be filled with payloads -that have variable lengths- or headers,

that would have fixed lengths. While this approach is good in terms of the feasibility, it leads to further problems that may reduce the memory efficiency and performance. The main problem is that the decoder in this case cannot distinguish between the two different lines, and additional information about the kind of line has to be added to the data frame. Some inefficiencies with this approach are :

- Header lines may not perfectly fit in a memory line and some bits will not be used.
- Difficulty with computing when the decoder needs a new header line. Since it's not perfectly clear when the decoder will need a header line, as it depends on the whole system, there may be some delay due to lack of available headers at a some point in the decoding.
- The encoding of the headers may not be perfectly efficient as there may be some packets that can be represented with a smaller number of bits than the selected fixed width.
- The lines of headers break the flow of payload, so they might add stalls in the system

However, since it is the most feasible approach for high-performance and despite the previous set of inefficiencies it is considered that the streaming of data can be continuous in most cases. The lines of headers, that in theory would add a useless clock cycle to the decoding, are usually not a problem since there will be other stalls in the rest of the system (for example, due to local reduction operations) that would stop nevertheless the flow. Those stalls can be exploited to cover the inefficiency caused by the header lines. While using headers of fixed width may seem memory inefficient, as using the binary encoding there may be cases where we need an extra bit just because we have a number of cases (i.e. possible packet lengths) that is slightly more than a power of two, this approach is necessary as the complexity of having headers of different lengths would lead to a much more complex design that may lead to performance and area inefficiency. Therefore, despite it is not perfectly efficient in terms of memory usage it is the approach chosen for the implementation due to its feasibility and the small area it requires inside the FPGA to decode the memory lines. Figure 2.9 shows the encoding of the data frame using this method. In the left image we have a clear distinction between the lines of headers and the lines of payload. However due to the fact that both are an indistinguishable series of bits from the decoder point of view is not obvious to determine whether a line corresponds to payload or to headers. This problem is solved with the approach shown on the right, where each line of data has attached a bit at the beginning that determines whether the line contains payloads or headers.

Figure 2.9. Data frame using the "lines of header and lines of payload" approach



2.6.3 Others versions

The two approaches presented above are just some of the possible encodings of the data frame. There are more examples of encoding that were discarded for other reasons. These versions are briefly introduced here:

- Abolition of the header information: this version would imply that the header information would be completely stored inside the FPGA during its configuration. This is highly inefficient due to the fact that the storage resources needed to store the header information would be too large. This approach leads to an area inefficiency so high that it would be intolerable.
- Payloads of all possible sizes: with this version the payload could have a continuous range between zero and the maximum packet length. This version, while being extremely generic would imply a variable header dimension, that would lead to a second level of the same problem of the headers: lines of headers of headers. Each header would not be encoded but would directly provide a number. The advantage here would be that the design would be

completely generic and would hold a minimum amount of memory space inside the FPGA. However, this approach was discarded as there isn't a need for all this flexibility, as the encoding of the mesh will always lead to a limited set of possible sizes a packet can possibly have. Furthermore it would lead to a memory inefficiency as we would need extra lines for the header of headers, as the headers wouldn't be encoded.

- Transmission of all the headers at the beginning of the frame. This has the same problem of the first example of this list. We would need many storage resources in the FPGA dedicated to hold the values of the headers. This problem makes this kind of approach unfeasible, as it would take out significant amount of resources needed by other modules of the system.

These alternative versions were discarded. However, it is important to keep in mind that this design hasn't some other similar implementations. Also, these versions aren't appropriate for this particular application, but in other circumstances these kind of approaches may be useful and recommended.

2.7 Unavoidable performance overheads and relative solutions

In the end, the chosen encoding was the one described in Subsection 2.6.2 . This design has however its own performance inefficiencies, both in terms of throughput and in terms of frequency. Some of them are inherently unavoidable due to the variability of the lengths of the packets. The unavoidable performance loss is that, since we have packets that may be bigger than the memory line, it necessary takes more than one clock cycle to decode those packets. This means that there will be a series of clock cycles where there aren't available output packets and thus there is a decrease in the throughput. This problem is however manageable by making the design elastic: whenever there is a stall from the algorithm, whenever the edge's pipeline doesn't need a new vertex, we can use this clock cycle to decode more packets. This fact can be exploited up to the length of the decoder pipeline, as once it is filled it cannot decode any more lines from the memory.

Another unavoidable performance loss is the fact that there may be packets smaller than the memory line. In this case we cannot set multiple lines from the memory as the system can process at maximum one packet per clock cycle. This however is not a real problem, as it only means that we can afford to make the processing of the edges more efficient. Since the pipeline processing the edges is already more performing than the decoder part, this is actually not an issue.

Other cause of performance loss are the headers as they carry information that is not strictly necessary for the data processing, but they are instead necessary for the data decoding. They cause a loss of a clock cycle each time a header line arrives. This problem can be managed by exploiting the fact that there are some packets smaller than the memory line. Whenever we encounter those kind of packets we virtually have another clock cycle to gather header's information. During the processing of packets larger than the memory port width there is a constant need for payload lines and if we don't have available headers we have to stop the flow of payloads to take a header line. This problem can be addressed in the same way we addressed the problem of having packets larger than the memory width: if a stall from the edge's pipeline occurs, we can use this clock cycle to take another header line from the memory.

Another performance loss is the information about the type of memory line. The decoder needs to know whether a line of memory is a payload line or a header line. This fact may or may not lead to a performance loss depending on the method to decode lines used by the decoder. In the selected approach, the performance loss is in terms of memory throughput. Using a bit to communicate the type of the packet we lose bandwidth, that translates to a loss of one bit per clock cycle.

2.8 Specification summary

In summary, the core problems of this design are :

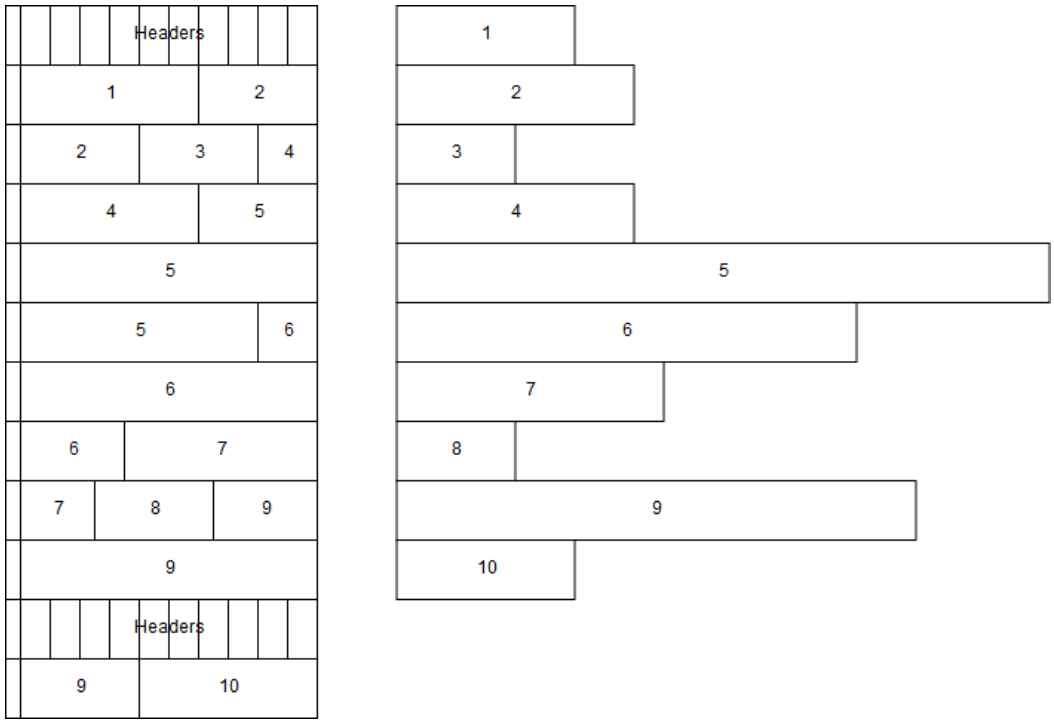
- The port width of the on-bard memory is fixed, while the length of the packets is not. This is the core problem of the design.
- The packets are fragmented in the memory and have to be recomposed correctly.
- The size of each packet is a property of the packet. This means that we have to send this information during the transmission of the packets, and process it at execution time.
- The design has to be portable and adaptable to each kind of grid (i.e. to different sets of possible packet sizes) and different boards (i.e. memory port widths).
- It has to operate using the streaming architecture paradigms: pipelining , flow control, loop reduction.

The optimization paradigms are:

- Data encoding optimization: each memory line coming from the memory has to be filled as much as possible with useful payload data.
- Decoder module optimization: The module has to be optimized in terms of operating frequency. We have to maximize it using pipelining as much as possible. It also has to be efficient in terms of throughput, so every stall coming from the edge processing part has to exploited to optimize the performance.

In the end we have chosen to go on with the data encoding design explained in 2.6.2. This means that the memory will provide to the FPGA a continuous series of packets that are fragmented and, furthermore, there are some lines of headers that will need to be managed. On the other hand, the next part of the FPGA design will need each of the packets that was previously encoded correctly decoded : each packet has to be recomposed and sent to the next stages of the system. Figure 2.10 shows the inputs and the outputs of the memory decoder.

Figure 2.10. Inputs and outputs of the memory decoder



Chapter 3

Architecture of the Packet Decoder

3.1 Specification of the module components

In the previous chapter we have seen how from the top-level view of the application of the whole project, passing through the implementation of the whole system, we have arrived to the specification of the module to be implemented. However, the specification of the module only shows the problem from an external point of view, and nothing is specified about the details of the inner components. To arrive to the solution of this problem we cannot solve it directly in a monolithic way, as it would lead to complex and messy solutions. The flow control technique allows us to divide each module using the streaming architecture into various sub-modules without altering the functionality and keeping the constraints of the streaming architecture valid. In order to solve the problem, the way to go is to divide the Packet Decoder into multiple components, following the *divide et impera* paradigm. In this section, the specification of each inner component of the Packet Decoder will be defined from an external point of view. The detailed description of each component will be addressed in the next chapter.

The first thing to set clear is the terminology we are going to use. There are multiple concepts that will be essential to understand the way the system is going to be implemented. Those terms are :

- Lines of memory
- Packets
- Payload

- Headers
- Frames

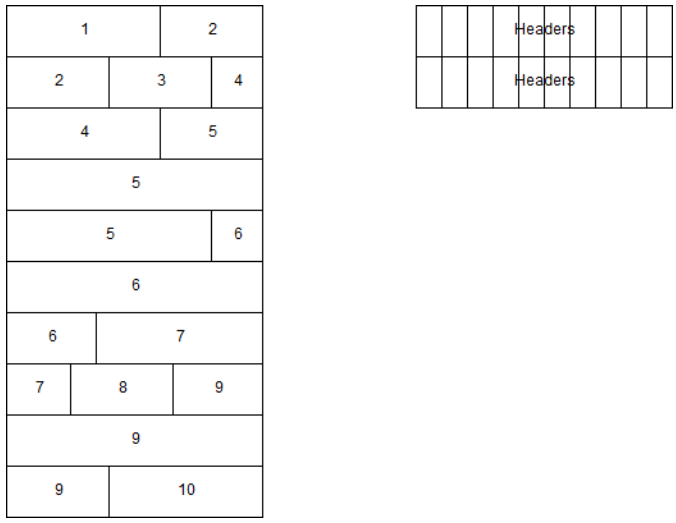
With memory lines we define the data incoming from the on-board memory. These data are organized as lines of data, and at each clock cycle one line of memory can be sent from the memory to the decoder. Packets refer to the data units that describe the dataset. This data is encoded and fragmented inside the memory and it has to be composed by our memory decoder. Payload refers to all the geometry/connectivity data contained in the packets. It doesn't refer to one single packet but it is about data belonging to packets. This is useful as there is another kind of data independent from the packet's data coming from the same memory that are the headers. A header contains the information that has a one-to-one relationship with a single packet. This set of data is independent from the payload data and it will be treated in a different way than payload data. The frame is the complete sequence of data coming from the memory, holding both payload and header information.

The first operation to be done on the frame as it arrives to the FPGA is to divide the frame into two different sub-frames: one for the header lines and one for the payload lines. This is done at the beginning of the decoder and it will also discard the extra bit describing the type of memory line. This can be done as the data regarding headers and payloads are independent one from each other, and thus we can treat them independently. Figure 3.1 shows the two different frames on which the original frame shown in 2.10 is subdivided. This makes even more clear why we have to distinguish between header lines and payload lines. They will be treated differently as they belong to two different frames. Figure 3.1 shows the sub-frame division: on the left we see the payload frame and on the right we see the headers frame. The extra bit shown in 2.10 is now discarded.

By focusing on the payload frame shown in 3.1 and the output packet sequence shown in 2.10 we see that packets are fragmented in the input, while at the output are composed instead . By analyzing the payload frame we can see that the way we have to rebuild the packets depends on how they are distributed in terms of memory lines. In the previous chapter we have already seen the problems arising from having variable packet lengths. However this property of the packets has to be analyzed in further detail to get the solution of the decoding process.

The core point is that when packets are encoded they can belong to multiple lines of memory depending on how long the packets are and on how large the width of the bus is. There are therefore three ways a packet can be spread into the lines of the memory, and this depends on the width of the packet:

Figure 3.1. Payload and header sub-frames



- The packet is larger than the memory port width
- The packet is smaller or equal than the memory port width
- The packet has zero length

The first case happens mostly at the beginning of the transmission. As we have seen the most connected vertices are the first ones to be listed and they will need large packets to hold all their values. As we go through the frame, the packets switch from being larger than the port width to packets smaller than the port width, as the packets not located at the center of the mesh will be encoded with less and less data because some of the vertex neighbours have already been sent. In the end there is going to be a whole sequence of packets of zero length that will represent the vertices where data and neighbours have already been sent. This case is peculiar and has to be addressed as well.

It is important to notice that the payload frame however is not ordered in terms of packet lengths, it may happen that some packet is larger than the previous and than the next one, and this means that we cannot divide the payload frame furthermore into sub-frames (i.e. the part of the frame with large packets, the part of the frame with small packets, etc...). We have to grant that our system is capable of decoding each kind of packet at whichever time during the frame transmission.

If we look at the payload frame, the three cases previously listed are valid. In the first case, the case a packet is larger than the port width, the packet can be spread on a maximum of N lines of memory, where N is equal to

$$N \leq \lceil \frac{packet\ length}{width\ of\ the\ bus} \rceil + 1 \tag{3.1}$$

For example, in Figure 3.1, if we consider the packet 5 we see that its length is approximately 2.2 times the width of the bus (i.e. the width of the rectangle), and it is spread exactly on three lines: the third, the fourth and the fifth. The plus one in the formula above is necessary as each packet may occupy an extra line if it starts at the end of the memory line: this is the case of packet 9, whose length is more or less 1.8 times the memory port width but it still occupies three lines of memory as it starts close to the end of the memory line.

The second case is when there is a packet smaller than the port width. In this case the packet will be spread on one or two lines of memory depending on where it starts. In Figure 3.1 we see some packets smaller than the port with, such as packet 1 and packet 2. The first one occupies only one line of memory, the first line, while the second one occupies two memory lines, the first and the second lines. The difference between the number of lines occupied stands, as mentioned in the previous paragraph, on where the packet begins. In the case of packet one, the packet starts at the beginning of the line and thus it occupies only one line. In the case of packet two, it begins at the end of the first line and thus it is spread on two lines. The main problem with these packets is that the first line of memory holds the information about two distinct packets, and thus it will be necessary to hold this data for more than one clock cycle. This doesn't happens for example in the fourth memory line, where all the memory line belongs to one single packet and once it is used it can be discarded.

The last case is when a packet has zero length. In this case the packet will necessarily take only one line as it will be virtually located between two consecutive packets. The only way we can know that a zero length packet exist is through its header: it will determine whether there is or isn't a zero length packet. Equation 3.1 is valid also for packets smaller than the port width or packets of zero length, and thus it is valid for all packets.

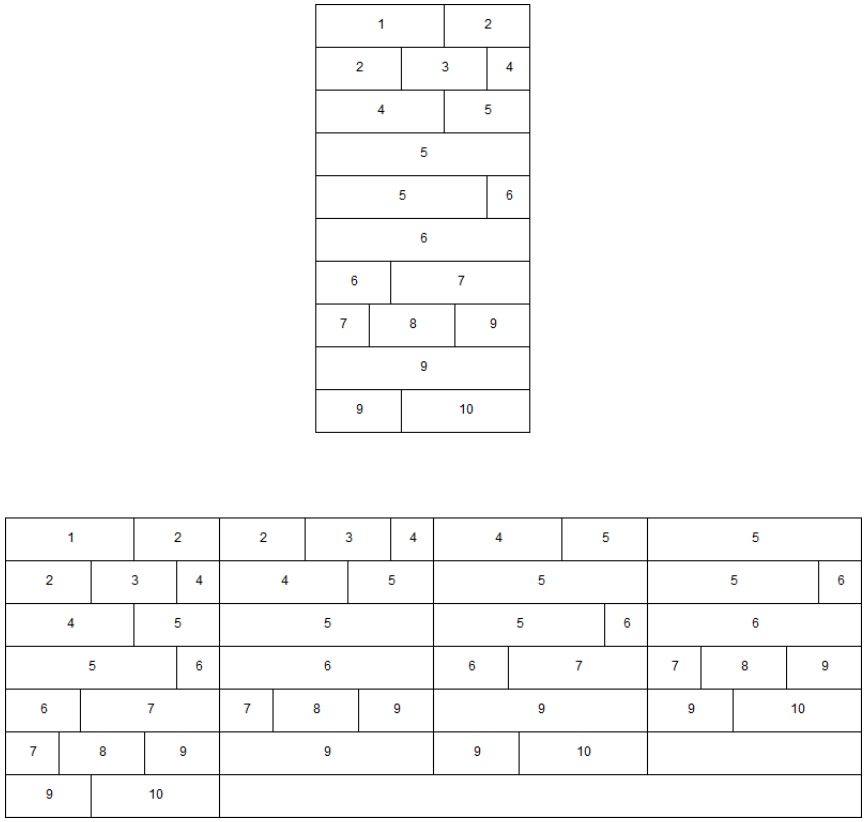
The first step to be done in order to solve the problem is to store all the N memory lines needed by a single packet to be reconstructed. In order to have a complete packet we need at a certain clock cycle all the lines of memory that contain the data of the packet. Those lines of memory could be ordered in a chaotic way, but in order to have our packet completely built we have to align them in a single register that is going to be wider than the memory port width. This

extended register has to be exactly $Max(N)$ times the memory port width, so it can hold all possible packets. This representation is going to be redundant and incorrect, as the packet can be smaller than this extended register and can start in a position that isn't the beginning of this extended register. However, it is an intermediate passage that is necessary to decode our packets, as it is the way we gather payload information from the memory. This is shown in Figure 3.2. Associated with each packet we have a sequence of lines of memory where the packet is completely rebuilt. We can see that the format of the input data is not the one proposed in Section 2.8, as it misses both the header information and the bit that determines the difference between a payload line and a header line. This is intended to be in this way because this part of the system exclusively addresses the problem of gathering payload lines, while there is another part of the system dedicated to divide the frame into two sub-frames, one for the header lines and one for the payload lines. We can also see that the width of this component output port is equal exactly to $Memory\ bus\ width * max(N)$ where $max(N)$ which is four as the packet 5 could be spread on a maximum of four memory lines. It has to be noticed that Figure 3.2 shows only the valid lines of the payload frame and of the built packet frame. There are necessary invalid lines inside the built frame that are omitted for the sake of simplicity.

Each reconstructed line can hold more than one packet. In this case the register value has to be held for more than one clock cycle in order to extract all the packets it holds. This problem has been addressed in Section 2.7: when a packet is smaller than the memory line, it is necessary that we hold this intermediate form for more than one clock cycle. The memory, on the other hand, according to the streaming architecture, can provide at maximum one line of data at each clock cycle. This means that in order to build packets like the packet 5 in Figure 3.2 we will need more than one clock cycle. In the bottom part of Figure 3.2 we see that some output lines have to be used more than once. The first, second and sixth line hold information about at least two packets. In the case of the first line the first and the second packet, and thus it has to be held for at least two clock cycles. The same applies to second line for the packets 3 and 4 and to sixth line for packets 8 and 9. We can notice that in all these cases there are packets that are smaller than the memory line. Inside each line there may be more than two packets. For example in the first and second lines (referring to the bottom part of Figure 3.2) packet four is completely built. This may lead to some redundancy as we in theory could use both lines to gather packet 4. This redundancy is resolved by granting that each built packet has to begin inside the first memory line part of the built line. In this way the design is more simple and avoids overhead in performance as the time to generate a new built line or to keep the previous line is the same: one clock cycle. What we ultimately see is that not only this component builds the packet from the payload frame, but also performs a coarse grained shift (in this

case to the left) about a multiple of the size of a memory line on the built lines. In the construction of the fourth line we see that we shift the previous line about two memory lines. This consequently means that we will have to ask two new payload lines from the memory, and thus it will take two clock cycles to perform this operation. The intermediate form of this line will be an invalid line. These lines are omitted from the bottom part of figure 3.2. The specification of being capable at the same time to keep a packet line for more than one clock cycle, to build a new extended line and to interface with the payload frame generates extra corner cases than the one listed above. These cases will be further analyzed in the next chapter as they depend on the implementation of this component. This procedure, that is the one of taking lines from the payload frame and to build a partial data format which holds the built packet is done by a component that is called *Packet Builder*.

Figure 3.2. Inputs and outputs of the Packet Builder



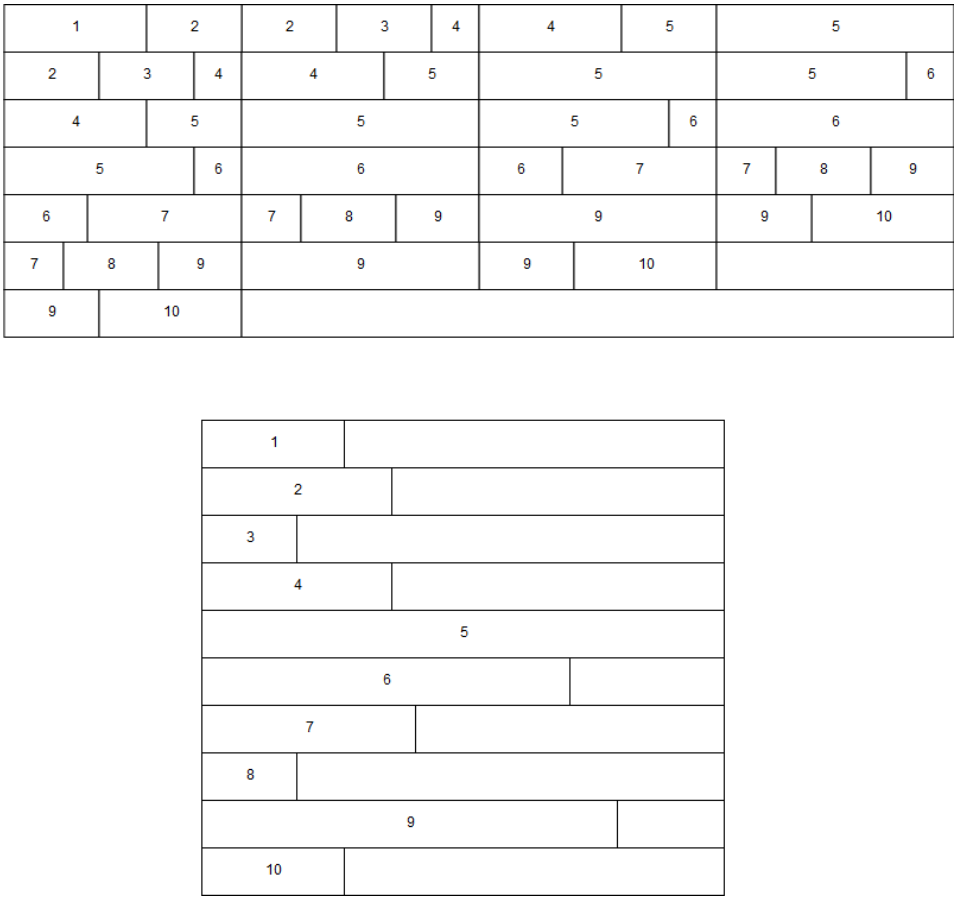
The data lines shown in the bottom part of Figure 3.2 already hold the complete packet. The problem we have now is that there is too much useless information inside these lines. For example, the packet 3 is held inside both the first line and the second line of the bottom part of Figure 3.2 . In order to have the final form of the data, we need to eliminate both the first part of the new extended line and the last part of the extended line in order to have the packet completely decoded. The simplest way to go is to shift each built line to the right or to the left about the size of the part that is between the beginning of the line and the packet we have to extract. In this way we have at the output a line that exactly contains a packet that begins or ends with the respective beginning or ending of the packet, that is the format we have to send to the FPGA memory. The problem is that by shifting in one direction we are holding the values left on the other direction, and since the width of the output line cannot change after it is programmed on the FPGA this is an unavoidable problem. This problem will be solved by giving extra information to the FPGA memory about the length of each packet exiting from the decoder. In this way the edge's algorithm knows exactly when the packet ends and therefore it can gather the information correctly. The width of the external bus must be capable to transmit each possible packet, therefore the width of the external bus will be exactly the maximum length among all the possible lengths the packets in a certain frame can have. The component that has to perform this operation is the *Packet Shifter*. It takes the output of the Packet Builder, a value holding the number of bits it will have to shift and another data about the dimension of the packet. It therefore shifts to the left or right the line given by the Packet Builder, and it also forwards the value about the length of the packet. Figure 3.3 shows the inputs and the outputs of the Packet Shifter. We can see that the output width is exactly the same width of the biggest packet inside the frame, that in this case is the width of packet 5, the white rectangles on the right of each packet are useless data that will be filtered by the algorithm's pipeline using the information about the length of each packet.

A problem of both the Packet Shifter and the Packet Builder is that they need some additional information about the packets in order to work. The Packet Builder has to know exactly how many lines a packet spreads over, the N value for each packet, while the Packet Shifter has to know exactly how much the built line has to be shifted, and also the FPGA memory requires the length of each packet.

To address all these problems we have to make another module that feeds both the Packet Builder and the Packet Shifter with this values. This component is called *Instruction Computer*, and what it basically does is to take the header information and to compute for each packet both the shift value and the N value. It has to be noticed that the header information doesn't hold the value about the length of the corresponding packet; but its value has a one-to-one correspondence with

one specific packet and encodes the length information. In this way we can decode the header to gather all the information necessary by this module to compute the two values needed by the Packet Builder and Packet Shifter.

Figure 3.3. Inputs and outputs of the Packet Shifter



The Instruction Computer doesn't know anything at execution time about the packets that are incoming from the memory. It has to gain the information about the sequence of packets from the data frame. This information is provided by the headers, that are given to the module at a certain frequency. The values of these headers hold information about the packet lengths, but it has to be encoded in

order not to waste bits in the memory line. These headers have to be stored inside the FPGA and have to feed the Instruction Computer in order to make it work. This module that processes the headers is called *Header Decoder*, as it is basically a FIFO memory designed to hold the headers. What it does is not only to store headers, but also to interact with the memory as it has to be fed by the header's frame. Since there is only one memory port but there are two modules interacting with the memory independently, the Packet Builder and the Header Decoder, there has to be some more control dedicated to joining the two interfaces. It has also to be noticed that this component is strictly related to the way we build the frame: depending on the way we decide to store the header lines in the frame and on how we discriminate between header and payload lines the design of this module may change.

3.2 Module schematic

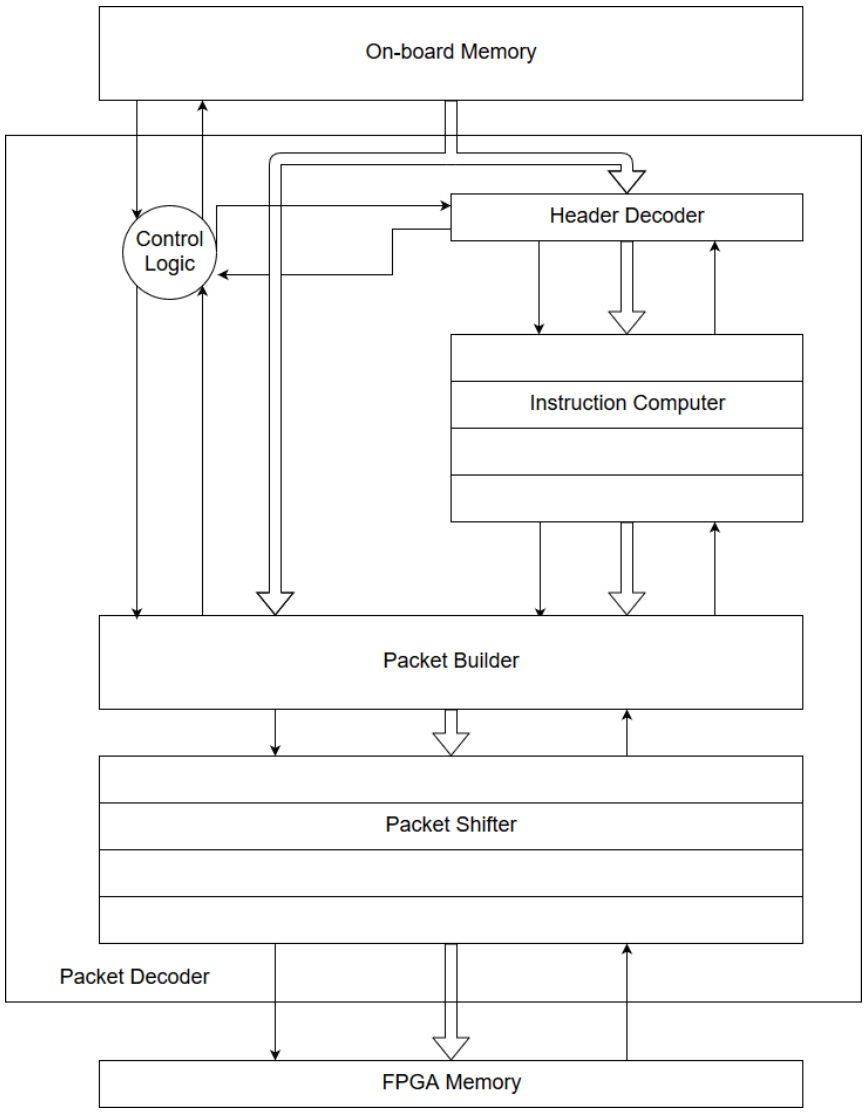
The previous section showed how from the specification of the module we can achieve a more complex system made of four different sub-modules or components:

- Packet Builder
- Packet Shifter
- Instruction Computer
- Header Decoder

The complete module will be composed by the sub-modules and by their joint control. Figure 3.4 shows the module schematic within the system. We see that all the interfaces of the components follow the flow control paradigm (thin arrows). All of them have three signals exiting each component and three signals entering each component (except for the Packet Builder which has two input interfaces). The signal represented with a thick arrow represents the data line. As each component in the flow control paradigm behaves like an FIFO, the data passing through the components will behave like a FIFO as well. This data will pass in the form of lines of data. The other two thin arrows represent a stall request and a valid bit. The valid bit signals to the next component whether a line of data is valid or not. This is useful in order to accept or discard the incoming lines. The stall line instead tells to the previous component that the component is already busy and it is incapable of taking any more lines of data. The previous component has therefore to stall. This same concept is applied to the external interface. However, since all the sub-modules have to perform different tasks and are independent, and

since there is more than one component interfacing with the external memory some control logic has to be added in order to manage the control signals coming from the sub-modules.

Figure 3.4. Block schematic of the design



An issue that may show up from this approach is that we are actually breaking some pipelining rules, as we aren't having the same number of pipeline stage if we take into the account the data lines going from the external memory to the Packet Builder and the header datapath coming from the Header Decoder up to the Packet Builder. This apparent rule break doesn't apply as we can model the system as if the two sub-modules are actually taking data from two different memories. We can think that there are two different FIFO memories, one holding the payload frame and the other the header frame. The fact that there is only one memory is a constraint that does not affect the functionality of the system because the flow control at the input of the Packet Builder synchronizes the payload lines coming from memory with the data provided by the Instruction Computer.

3.3 External interfaces

The module has two different interfaces with the external system. The first one is between the module and the memory which holds the geometry values. The second one is between the memory decoder and the memory inside the FPGA.

We can think about the interface between the Header Decoder and the external memory and the interface between the Packet Builder and the external memory as two independent interfaces that are joined as a final step in the design. The interface between the Packet Builder and the memory is indeed simple: the Packet Builder knows from the Instruction Computer how many lines of data it has to get from the memory. Since the memory behaves like a FIFO, it has a line communicating whether the incoming lines are valid -for example if the memory is empty it will tell that the incoming data are invalid-, the Packet Builder will instead ask through the stall line a line of data. This line in an ideal environment would never be active (set), however it may be active when there isn't an instruction available or if the Packet Shifter is stalling.

The interface between the Header Decoder and the memory works basically in the same way. It has a line coming from the memory telling whether the data are valid or not, and a line coming from the header register telling whether the Header Decoder needs a new line of headers or not. This line is not always active as the Header Decoder has a width that grants that a certain number of headers can be stored inside this FIFO. It will ask another line only when the Header Decoder is empty.

On the other interface we have to send the data decoded to the memory of the FPGA. The data format is shown in the bottom of Figure 3.3. We are going to

send a packet completely built. However since the packets have different widths, we have to grant that each packet can be transmitted to the FPGA memory. This translates into a bus whose width is equal to the maximum number of bits a packet can possibly have. This means that packets that are smaller than the maximum width will necessarily have an additional number of bits attached to the end of the line. This problem will be solved by adding to the output bus another line that tells the width of each packet starting from the left. In this way, the next sections of the system will be capable of determining the exact bits belonging to the packets. This interface is exclusively between the Packet Shifter and the memory inside the FPGA. It will also have a validity line coming from the shifter that tells whether a packet is valid or not. The data outgoing may not be valid in the case when we have to decode a packet that is larger than the width of the bus. In this case, there will be some partial lines of data where the packet isn't still completely built and therefore in the end this data will not be valid. There is also a line coming from the FPGA memory that tells whether the memory is full or not. If this happens the module will not send any new data and it will stop operating. The memory on the FPGA also works as a FIFO.

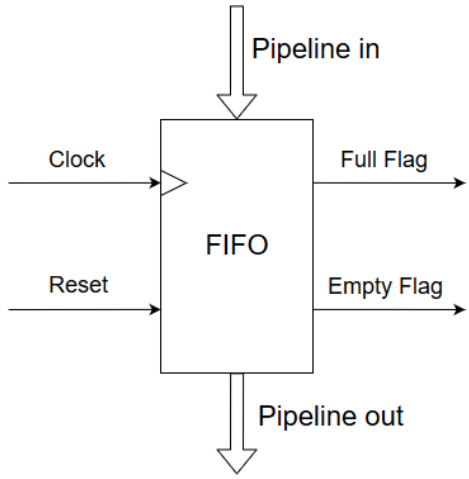
Figure 3.5 shows an example of standard FIFO memory module. In our case we adapt the interface of this memory to the flow control module. We use the full flag as a stall line, and it communicates to the previous stages of the pipeline to compute the data. When it's set it means that the memory is full and thus that we have to stop the computation. On the other hand, the empty flag can be used as a valid line. When the memory is empty the data sent to a module will not be valid and therefore these data won't be used.

3.4 Design techniques

This section is dedicated to the description of the techniques that have been used when designing the memory interface. These techniques are not completely standard, and in order to understand the implemented components that will be described in the following chapter we have to describe them. As explained in Section 1.2 we have used VHDL 2008 version as hardware description language, and Python 3.6.5 as programming language.

One of the problems addressed in the previous section is that we have to abstract the design and make it valid for whatever kind of packet length. In order to do so, first we have to declare in a file all the possible lengths the packets of the frame may have, and also the width of the bus of the memory, that is a core variable that may change from board to board. The file where there will be written this core

Figure 3.5. Schematic of a FIFO module



variables is a simple text file with a structure like this

- bus width
- packet length #1
- packet length #2
- ...
- packet length #N

All these variables have to be expressed as hexadecimal numbers. From this primitive file we extract all the useful information needed by our design.

3.5 Generics

The key point to abstract the system from the physical implementation on any board is to make all the design parameters as generic as possible. The way we decided to go to abstract the design was to use the generics description tool provided by the VHDL language. This tool still isn't complete when it comes to defining the variables of the design, as sometimes we have to use different kind of approaches to make the design more accurate. We have chosen to use the following three approaches when it comes to the implementation of the generics variables.

- Direct use of generics
- Use of a package of global variables
- Use Python software to generate VHDL code

The first case is the direct use of generics. Each component has its port's width not determined by a fixed number, but determined by a generic variable that is the generic defined in the top part of each design. This grants us that each port of our design is portable and flexible to represent every possible number. This is useful when defining the interfaces, that depending from the configuration file may vary a lot in the various designs. Furthermore, some designs will directly depend on the generic values.

The generic tool is however a bit limited in some cases. We will encounter cases where we have to grant that some variables used by more than one component have to be the same, and also to generate some generic variable to declare the components of a module. Finally, we have to grant that after the configuration process the design is ready to be synthesized and implemented in the fastest possible way, without adding variables that the final user may insert incorrectly. In order to speed up the process and to avoid confusion with the use of generics we decided to make the use of variables defined in VHDL packet files, and to make these files readable by every VHDL file. In this way we can grant that once we have written this values inside the packet file, each VHDL file will read the variable correctly, and that this value will be the same for all the components of the system.

All the previous values to be written on the package file have to be extrapolated from the configuration file. In order to do so we decided that the best way to go is to generate the generic variables file with a software in Python that takes as input the configuration file, and rewrites some basic variables inside a VHDL file. All the other variables derived from these basic ones will be instead computed in another VHDL packet file. Furthermore, there are some components in the design that have to be pre-programmed at configuration time in order to work correctly. This happens mostly when we need the result of a complex operation that would reduce the performance of the design inside the FPGA. All these operations are performed before the compilation and synthesis of the VHDL design. In this way we grant that we have a simple user interface that is reduced only to the configuration file, and no generic variables have to be declared inside the VHDL code.

3.6 VHDL style

When we had to implement the components using an hardware description language we use some design choices regarding which kind of style we had to choose to implement the system. We first choose VHDL over Verilog because my formation from the university has privileged the teaching of VHDL with respect to Verilog.

The VHDL language is very versatile as it is thought to be a generic hardware description language. However, there is more than one possible approach when using VHDL. The two main applications of this hardware description language are

- ASIC design
- FPGA programming

Since we have to program an FPGA, we had to go for a style that would be different from the ASIC design style. When describing an ASIC component we can go up from the lowest level of abstraction to describe it (bottom-up approach). Since everything is going to be implemented up to the physical layer, we can define how the architecture of each component it is going to be implemented on the chip. For example, we can choose to implement an adder in many different ways: from a ripple carry adder to a carry select adder and so on. This is different when programming an FPGA. If we are going to implement our components in a custom way, we are going probably to incur in some overheads, as there is already inside the FPGA some hardware dedicated for the purpose we have. There are in fact adders and multipliers already implemented inside the FPGA, and they are already optimized for their purpose. Creating a custom device would be an overhead in most of the cases as we would need to make it by programming the LUTs and the interconnections between them, and this would lead to an inefficient way of using our FPGA. Most of our work is therefore going to avoid this kind of structures, as the latest form of VHDL grants a simple and fast way to do operations between any data types, given that there is an on-chip block to perform the operation. This basically means that we are going to privilege the behavioural style of coding with respect to the structural style, as the structural way of coding is going to generate overheads when specifying the component down to a too low level of abstraction.

VHDL is also a strongly typed language. In this language, we have to clearly define the type of each operand when performing an assignment as both of the types in the operation have to be the same. Since sometimes the data we are dealing with are too complex to be represented by a VHDL standard data type, there are some cases where we are creating a more complex data type in order to keep the structure simple and close to the flow control design implementation. Most of the times, when we have to communicate between two different components a set

of data that is not unique, but is composed by two or more types, we are going to include this information in a single data type, so that we keep clear which signal is or isn't inside the communication bus. This happens especially inside the Packet Builder, as we will need to deal with a big amount of different signals that in the interfaces is better to keep united, as they are transmitted all together through a bus between two different components.

Chapter 4

Implementation of the Packet Decoder components

In the previous chapter we have described how to address the problems posed in Chapter 2 by decomposing the structure of the Packet Decoder into four sub-modules or components.

- Packet Shifter
- Packet Builder
- Instruction Computer
- Header Decoder

The block which has to divide the frame into two different sub-frames will be addressed inside the Header Decoder, as it is the block which interfaces with the input the most and, therefore, has a more intimate dependency on the format of the input lines. These sub-modules however present multiple possible implementations, and the solution to each one of the presented sub-modules has to be chosen appropriately in order to optimize the performance and to make the module portable for multiple kinds of FPGAs. The sub-modules are here presented in a back to front order: starting from the output we will go back to the inputs as each time we describe a component we also add specifications to the previous one, by defining its output format. The description of the components will start therefore with the module that interfaces directly with the output: the Packet Shifter.

4.1 Packet Shifter

The Packet Shifter is the component that has to perform the operation of deleting the exceeding information from the lines provided by the Packet Builder. The specification of this component has already been discussed in Chapter 3. However, here is a summary of its specification:

- It has to remove the first part of the lines provided by the Packet Builder.
- It has to do it within the streaming architecture; it, therefore, has to behave like a FIFO module
- It has to perform this operation at the maximum possible throughput while not exceeding the area usage
- It has to be portable to multiple FPGAs and, therefore, it has to be a generic design

4.1.1 Background

The core point of this component is held inside the name itself: the simplest way to perform the operation described above is to do a shift to the right or left of the lines coming from the Packet Builder. There are some different ways to perform this operation. Firstly, it is a key point to understand how does a shifter work.

The shifter is a block that is commonly found inside digital components as it can multiply or divide by powers of two efficiently. First of all, we have to set clear that there is more than one possible operation that can be performed by the shift block. Namely :

- Arithmetic shift
- Logic shift

The arithmetic shift performs the operation on signed numbers. This means that the operation will keep the sign of the operand as it is. The logic shift is instead more direct and simply adds zeros to the left or right of the word we have to shift. Since we aren't dealing with signed numbers, the arithmetic shift isn't useful to our purpose. It is instead useful to see which kind of logic shifters have been already implemented, and which one fits better in our specification.

An example of a shifter that can perform any kind of logic shift is the Barrel Shifter. The core ability of the barrel shifter is that it can perform any kind of shift without using sequential logic (i.e. using a single clock cycle). While this can be useful in architectures that have fixed pipeline stages, in our case we don't have

any limitation in the temporal parallelism and thus this design was discarded as it requires many resources to be implemented. Furthermore, this design is mostly used inside full or semi-custom designs, which isn't our case

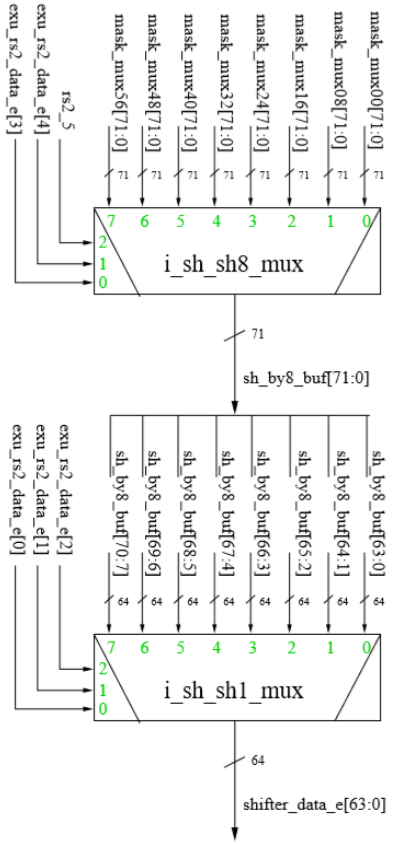
An interesting approach is the one provided by the Ultrasparc T2 Niagara [5] shifter module (Figure 4.1). This module is important to us as the Ultrasparc Niagara T2 is a processor that is optimized in terms of performance, as it was intended to be used in server machines. The T2 shifter, which resides inside the ALU which is inside the execution unit, can perform both the arithmetic and the logic shift, but only the logic shift is the one which is interesting for our purposes. The T2 shifter can perform shifts in a multi-level fashion. In particular, the T2 shifter is based on three levels of shifting: the first level consists in the selection of a mask which will provide the inputs of the next levels of the shifter. The masks provide eight words, each of these has already performed a shift by a multiple of eight and, depending on the type of shift it has to perform, this words will be logically or arithmetically shifted to the right or to the left, or rotated to the right or to the left. Inside Figure 4.1 this is shown in the input lines of the multiplexer at the top. These inputs are determined by the selected mask. The second level performs the coarse level shifts. The initial word is shifted by multiples of 8 bits up to 64 bits. This is achieved by the use of a eight to one multiplexer, which has as input the already shifted values provided by the mask as shown in Figure 4.1. This stage has to select the correct coarse grained shift provided by the mask. After this phase there is the fine grained shift that shifts the value by 0,1,2,3 up to 7 bits. This is also provided by a multiplexer which has as input the value generated by the previous multiplexer and this value is divided into eight different words. Each one of this words performs the shift by selecting the appropriate bits of the previous word. In Figure 4.1 the second multiplexer performs this shift. The inputs are always the same previous word but inside a specific range. Each on these ranges are shifted incrementally by one starting from the first input up to the eighth input. As this block is defined by three levels that together make a DAG, this block can be easily pipelined by subdividing the component into three stages: a stage for the mask, a stage for the coarse level shift, a stage for the fine grained shift. Since this module can support a high throughput and performs the shift in a simple and modular way, it has been chosen as the basis for the shifter we had to implement.

4.1.2 Critical issues

Compared with the T2 shifter, our block has relevant different specifications:

- Our shifter has to be implemented on an FPGA, not on a full custom chip
- More pipelining has to be included to optimize performance

Figure 4.1. Level two and three of the Ultrasparc T2 Niagara Shifter



- We do not need to shift the whole word, just the lengths of a memory line almost
- The amount of shift to be shifted is not fixed, it has to be generic

The first point implies that the T2 shifter is optimized inside the environment it is working. It is indeed probable that there are different modules inside the processor that work at a slower rate than the T2 shifter. This may be the reason why it isn't pipelined and so it isn't highly optimized for performance. This is not our case, as we have to grant maximum performance without considering the other parts of the system. Further optimizations of the T2 shifter will therefore be considered. The basic optimization with respect to the T2 shifter is to divide the coarse grained

shift and the fine grained shift into multiple sub stages, creating a pipeline stage whenever possible.

Secondly, we have to consider that the incoming lines have the biggest width among all the possible size of the packets. This means that there isn't a case where we have to completely shift an incoming line. The maximum possible shift we have to make is instead equal to the width of a line of the memory bus, as the edge case is that the packet begins exactly at the last bit before the end of the line. This means also that the width of the data coming from the Packet Builder has to be exactly

$$packet\ length\ +\ memory\ linewidth\ -1\ Bits \tag{4.1}$$

These are the two parameters that will influence the design the most.

This design not only has to be optimized, but it has also to be generic. This means that we have to grant that this module works for whichever packet length and whichever memory line. This influences the design a bit, as the T2 shifter is intended for a fixed data length, and therefore we have to modify the design in order to make it portable.

4.1.3 Implementation and block diagram

Figure 4.2 shows the schematic of the design. As we can see, there are three ports coming from the Packet Builder. The first port has the data to be shifted, in the format shown in Figure 3.2. In addition there are two more ports, one for the line of the header, which describes the length of the packet. This information is necessary for the module that is going to use this data and it is just forwarded to the end. The third port has the shift value line, a line of data which holds the value about how much the packet inside the data line will be shifted. The lengths of these ports are respectively :

$$Maximum\ packet\ length\ +\ memory\ line\ width\ -1\ (Dataport)$$

$$\lceil \log_2 Maximum\ packet\ length \rceil\ (Headerline)$$

$$\lceil \log_2 Memory\ line\ width \rceil\ (Shiftline)$$

On the other side, there are the ports that will communicate the data to the memory inside the FPGA. This ports are the shifted data port and the header port; the first port will have a width equal to $Maximum\ packet\ length$ while the

second port width is the same as the input port for the headers. As we see in Figure 4.2, the design is regular and is similar in a way to the T2 shifter. Considering the stages after the top one, at each stage we have a two to one multiplexer that can perform a partial shift of the input data. After the first stage, the number of bits eventually shifted at each stage is equal to

$$2^{\lfloor \log_2 Memory\ line\ width \rfloor - (\#stage - 1)}$$

If the amount of bits to shift is greater than this value, the shift will be performed and the same amount of bits shifted will be attached to the end of the register in form of zero bits, otherwise the lines will simply be forwarded. This procedure is repeated until we will reach the shift value of one, where we will have to make a shift of just one bit. In this way we can directly use the number in the form of an unsigned integer provided by the shift value line to control the multiplexers, as any number represented as an unsigned integer is equal to

$$N = \sum_{i=0}^{n} x_i 2^i$$

where N is the number in the decimal form, x_i is the i-th bit of the number represented in binary. This means that each bit of a binary number correspond to 2^i in its decimal version. Therefore if we have to make a shift by five bits, since the number 5 is represented as 101 in binary, we can decompose the shift into

$$1 * shift\ by\ 4 + 0 * shift\ by\ 2 + 1 * shift\ by\ 1.$$

There is however a problem with this implementation.The problem is that we are considering that the memory width is exactly a power of 2. This means that our system works perfectly if the bus width is a power of 2, which rarely is the case for FPGAs, but it doesn't work for systems that have a memory port width that may differ from a power of two. In these cases, the approximation of the bus width will leave some cases unsolved: whenever we have to shift a value that is inside this range

$$2^{\lfloor \log_2 Memory\ line\ width \rfloor} \leq x < Memory\ line\ width \tag{4.2}$$

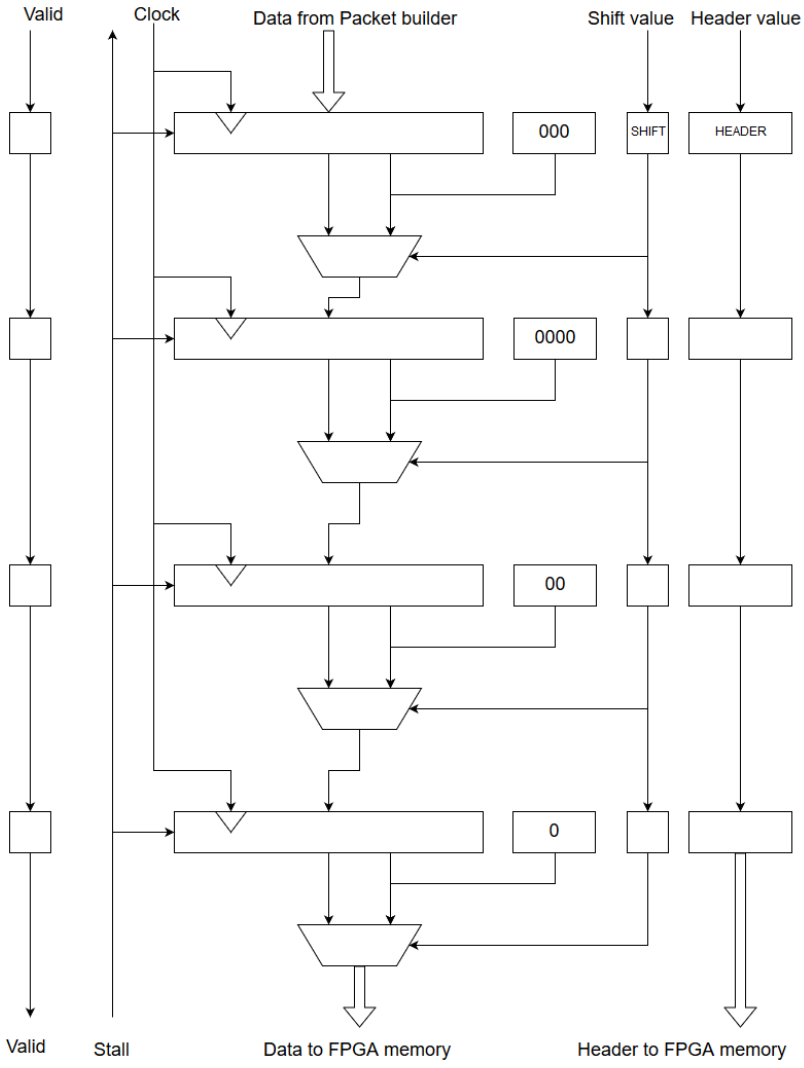
the value cannot be possibly shifted, as it will necessarily remain an unshifted part equal to

$$x - 2^{\lfloor \log_2 Memory\ line\ width \rfloor} \tag{4.3}$$

The way to solve this problem is to add an extra pipeline stage at the beginning of the module that will perform exactly this shift. If the value to shift goes inside the range shown in 4.2 it will be shifted exactly by the value in 4.3. The remaining part (4.4) will be shifted by the regular part of the shifter as it is necessary inside the range 4.5.

$$Memory\ line\ width - 2^{\lfloor \log_2 Memory\ line\ width \rfloor} \tag{4.4}$$

Figure 4.2. Schematic of a rigid Packet Shifter with an input line width equal to 11 bits



If the value to shift isn't inside the 4.3 range, we simply use the regular part of the shifter as it can perform any shift inside the range in 4.5

$$0 \leq x < 2^{\lfloor \log_2 Memory\ line\ width \rfloor} \tag{4.5}$$

In this way the block is both high performing and generic as the operation between any two pipeline stage is selected by a two to one multiplexer, that can be implemented directly on LUTs in parallel, as a LUT in the FPGA has at least four inputs and one output. The portability requirement is also achieved as the module can perform a shift of any value between zero and the width of the bus. Therefore, the parameters needed to configure this block are

- Memory bus width
- Maximum packet length
- # of bits of the header value
- # of bits of the shift value

4.1.4 Elasticity

As mentioned in Section 2.3 the flow control technique is based on the concept of validity and stall. This same concept is applied to the Packet Shifter: lines incoming from the Packet Builder have attached a validity bit, and the lines processed in the Packet Shifter can go to the FPGA memory only if the FPGA memory isn't full, which is signalled through the stall line. This creates an optimization problem: each line incoming will be elaborated unless the memory is full, disregarding the validity of the data. This means that some pipeline stage will be filled with data that isn't valid at some point in the execution. This in fact is unavoidable as long as the FPGA memory can store data lines, as the FPGA memory is assumed to discard the invalid lines. When a stall from the FPGA memory occurs, there are however two ways to proceed in the Packer Shifter:

- Rigid way
- Elastic way

The rigid way is the simplest one: whenever a stall from the FPGA memory occurs, all the component will pause and the stall signal will propagate a to the Packet Builder, as well through the outgoing stall line. In this way the Packet Shifter will be freezed and valid data both with invalid data inside its pipeline. This however is an overhead, as the invalid data can be discarded and replaced with valid data already inside the Packet Shifter even if the output is stalled.

The elastic way of managing flow control solves the problem of having invalid data inside the pipeline. When a stall from the memory occurs, instead of freezing

the component, it allows new data to shift as long as there are pipeline stages filled with invalid data. This way grants that when the stall from the FPGA ends there will be a sequence of packets ready to be sent to the FPGA memory, locally increasing the throughput of lines. As the paradigm of performance optimization applies to this component as well as to all the others, this is the implementation that has been applied. The rigid way of stalling the pipeline was the original version of the shifter, as it was simpler and it was firstly implemented to avoid problems regarding its simulation. It works in this way: each pipeline stage is made as a register which has four inputs: the data incoming from the previous combinational circuit, the clock signal, the reset signal and the enable signal. In the rigid way of implementing the stall, we simply connect each enable signal to the stall line coming from the FPGA memory. Whenever the stall signal from the FPGA memory occurs, all the registers will be disabled and will keep the value of the previous clock cycle. In the elastic way, the stall signal changes from register to register: it will stall only if the FPGA stall signal is active and all the pipeline stages between the stage being considered and the FPGA memory port are filled with valid data.

This leads to the chain shown in Figure 4.3. There is a chain of AND gates, starting from the last stage and going through the pipeline up to the stall line going to the Packet Builder. The stage of pipeline next to the interface with the memory checks if the valid value related to its stage is 1 -valid- and if the stall line coming from the FPGA memory is 1 -stall- if those two conditions appear the stage will be stalling and it will signal this fact to the previous stage. The inverted signal will determine the eneble signal of the registers inside that pipeline stage. All the other stages will also check two values: the related valid value and the stall signal coming from the the next stage of the Packet Shifter. If all those two lines are set, then the register will be disabled and it will keep the value it already holds. The same concept is propagated up to the interface with the Packet Builder and it applies also to the stall signal going to the Packet Builder. In this way we can manage all the possible stall signals coming from the FPGA. This translates into the following cases:

- A stall that lasts more clock cycles than the number of clock cycle needed to fill the Packet Shifter with valid data
- A stall that lasts less clock cycles than the number of clock cycle needed to fill the Packet Shifter with valid data

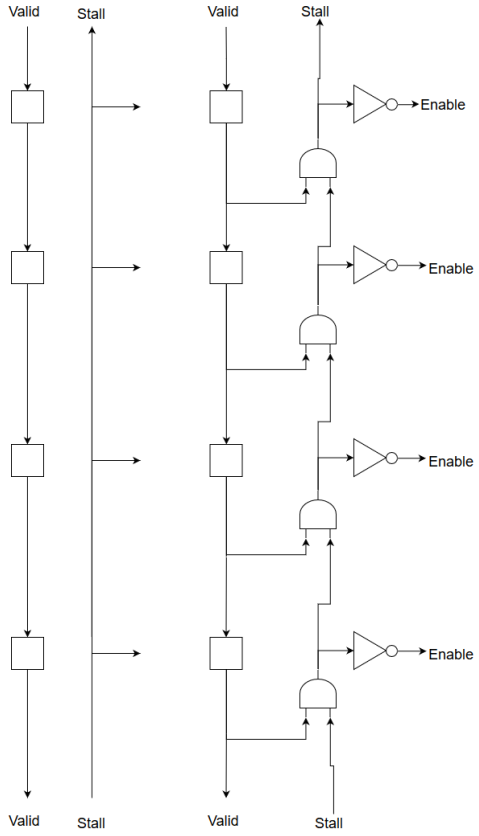
In the first case, the pipeline stage will continue to be filled up to the point that all the pipeline is full. If the pipeline is filled with valid values, then all the component will stall and it will communicate to the Packet Builder to stall as well. In the second case the shifter will continue to work as long as it isn't

filled, but since all the enable lines of the registers monitor the stall line of the FPGA memory, if it goes down it will simply continue to work as if nothing had happened. In the end there will be a sequence of valid packets reaching the output.

The problem with this design is that it introduces a chain of AND gates that is as long as the pipeline length. This chain is unfortunately impossible to pipeline, as the data flow of the stall signals is going in the opposite direction of the data to be shifted. The length of this chain will affect the maximum operating frequency of the module. Figure 4.3 shows on the left the rigid way to manage the stalls, and on the right the enhanced elastic way to manage the stalls. We can see that the critical path is the chain of AND gates. It is linear with respect to the number of pipeline stage and it is equal to one logic gates multiplied by the number of stages minus one. Therefore in terms of the on-board memory bus width the critical path length is logarithmic and it is equal to

$$\lceil \log_2 Bus\ width \rceil\ LUTs$$

Figure 4.3. Rigid way and elastic way to manage stalls.



4.2 Packet Builder

In the previous section we have shown how the Packet Shifter module has been implemented. Here we are going to describe the implementation of the Packet Builder. The behaviour of this component has been described in Section 3.1. Nevertheless, here is a summary of its specification:

- It has to transform the payload encoded in the memory to a format acceptable by the Packet Shifter.
- It has to perform this operation following the streaming architecture concept which implies a flow control technique
- It has to be optimized in terms of performance

Since this component has as main purpose to gather payload correctly from the payload frame, it will interface with the payload FIFO. As explained in Chapter 3, this frame can be thought to be independent from the header frame, and therefore, this component can be modeled as an interface between the payload frame and the Packet Shifter. This is a core component of the system, as it is the direct interface with the payload frame. To solve the problem addressed in the specification, it is important not to have a monolithic approach, but it is better to divide the component into several sub-components, each of these with a single specific purpose.

As this component is an intermediate module between the external memory and the Packet Shifter, it is immediate to set the specifications of two sub-components: one as an interface with the payload frame and the other as an interface with the Packet Shifter. The interface with the payload frame is called Payload Reader, and has to deal with the control signal coming from the payload frame, such as the valid/invalid signal, and the stall signal coming from the Packet Shifter.

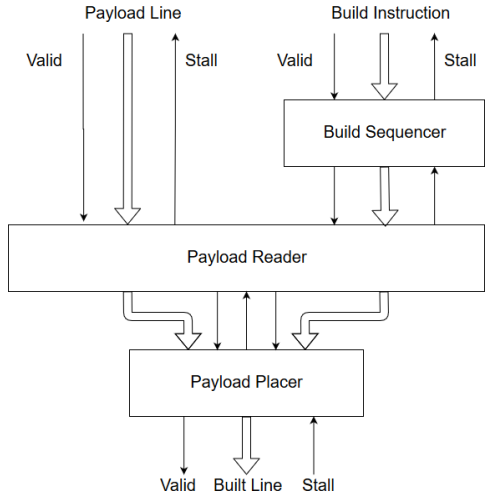
The second sub-component is the interface with the Packet Shifter, and has the main purpose of providing valid extended payload lines. This component is called Payload Placer, as it has to put each payload line in the correct register, and to generate a valid-invalid signal for the Packet Shifter, and also to manage the stall signal coming from the Packet Shifter itself.

The Packet Builder also includes another interface, the one between the Instruction Computer and the Packet Builder itself. This interface is called Build Sequencer. Its main purpose is to control the other two sub-components. It has to manage correctly all the possible cases that the two other sub-components can encounter, and avoid the generation of an incorrect extended line. As explained in Chapter 3, the Packet Builder requires additional information to correctly build the

packet and this information, which is generated by the Instruction Computer, is sent to and managed by this sub-component. This sub-component can be thought as a finite state machine and it will be modeled in this way.

From a top-level point of view, this component is modeled as shown in Figure 4.4; The Build Sequencer is placed before the Payload Reader. This happens because the Build Sequencer has to control both the Payload Reader and the Payload Placer. Some control signals generated by the Build Sequencer will be used by the Payload Reader, and the remaining signals will be used by the Payload Placer, and therefore they are forwarded by the Payload Reader. Each component has only one stage of registers, so this whole component can be seen as a component with three pipeline stages . What could look like a pipeline rule break, as the data coming from the Instruction Computer have one more layer of registers compared to the path of the payload lines path, shouldn't be seen as a rule break as the payload frame can be modeled a an FIFO component with undefined depth. The last component is the Payload Placer, which gathers payloads from the Payload Reader and control signals from the Build Sequencer. We see that all the sub-components support the flow control technique.

Figure 4.4. Block diagram of the Packet Builder's sub-components.

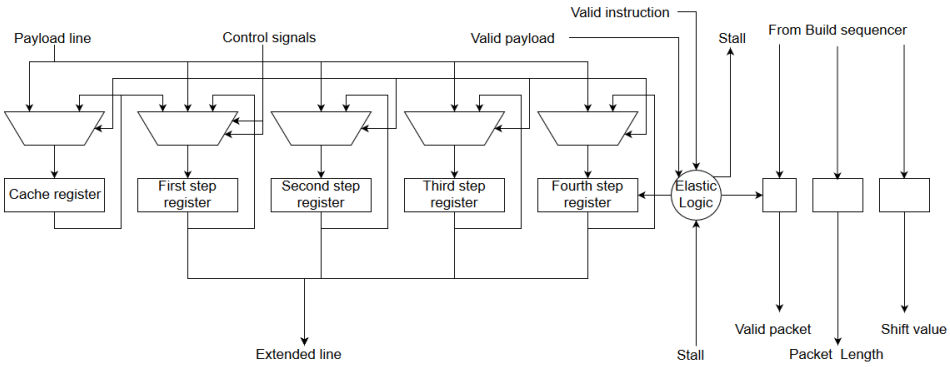


4.2.1 Payload Placer

This module has three interfaces, two input interfaces and one output interface. The two input interfaces have to gather data from the Build Sequencer and from the Payload Reader. The output interface has to issue data to the Packet Shifter. The data required by the Packet Shifter includes the lines to be shifted, the packet length and the shift value. While the length of the packet and its shift value aren't computed by this module, the payload has to be formatted by this module. The format of the data is shown at the bottom of Figure 3.2 and the output port width is $Extended\ width = Maximum\ packet\ length\ +\ memory\ linewidth\ - 1$.

While the width of the Packet Shifter input port is shown above, the correct way to build an extended line is to do exactly as specified in Chapter 3, that is, to align a series of registers, each one of the width of the external memory bus as shown in Figure 4.5. In this way, we ensure that our extended line can hold all the lines required by the Packet Shifter. The number of aligned registers is equal to the $max(N)$ value in Equation 3.1. The way the payload lines have to be stored is explained in Section 3.1. Each valid built packet has to begin exactly inside the first register of the set of aligned registers. There can be more than one packet that begin inside the first register, and in this case the extended line has to be held for more than one clock cycle, and at the output the valid signal will be set for two or more consecutive cycles.

Figure 4.5. Payload Placer schematic



The core point of this sub-component is to put correctly the lines gathered from the Payload Reader into the extended register of the width shown above. The problem that arises from the encoded memory is that there are three possible cases in which a packet can be distributed inside the on-board memory:

- The packet is larger than the on-board port width
- The packet is smaller or equal than the on-board memory port width
- The packet has zero length

These cases have to be treated properly. The Packet Shifter can take invalid data as input, but in any of the previous cases there has to be exactly one clock cycle where the packet is built correctly, so that the valid signal to the Packet Shifter can be set, and the extended line forwarded.

In the first of the above cases, or when a packet is larger than the width of the external memory bus, the packet has to be stored in more than one payload line inside the extended register. Each of the corresponding payload lines has to be placed in the correct register. Since the control signals which determine how many lines have to be fetched from the payload frame come from the Build Sequencer, the only thing to do here is to put them in the right place. The way to go is to put a two to one multiplexer at the input of each step register, with one input with payload line and the other input with the output of the step register itself. The lines that control these multiplexers are managed by the Build Sequencer. In the case that there is a packet larger than the linewidth, one multiplexer chooses the payload line and all the other multiplexers choose to keep the register value. In this way, we ensure that each packet is put in the correct place. As explained in Section 2.6, there will be some clock cycles when the extended line is not complete, and the built packet is not valid.

There is, however, a complication to this simple concept. There are some cases where there are more packets stored inside one single payload line. When this happens, as explained in Section 2.6 and in Section 3.1 as well, we have to keep for more than one clock cycle the same payload line. The way to go to solve this problem is to add another step register of the linewidth of the external memory bus, with a multiplexer at the input controlled by the Build Sequencer like all the other step registers. This register is called the Cache Register, and it doesn't interface with the Packet Shifter, but with the first step register of the extended line as shown in Figure 4.5. Whenever a payload line is requested, it is automatically stored in this cache line. The line that pilots the multiplexer above this register is the same that asks to the memory to ask for a new payload line.

Figure 4.5 shows the schematic of this component. There are N step registers (in this case N=4 as we are referring to the case shown in Figure 3.2), each of them having a multiplexer controlled by the Build Sequencer. The first step register has a three to one input multiplexer since it has to be capable of selecting the Cache Register as input. This means that there are two lines controlling its multiplexer.

The valid line is simply forwarded from the Build Sequencer to the Packet Shifter, just like the shift value and the packet's length value. The stall line together with the control lines have the role of blocking the register from allowing additional information, and the stall line is then forwarded to the Payload Placer and Build Sequencer.

4.2.2 Payload Reader

The main purpose of the Payload Reader has is to read correctly the payload frame. It has to manage appropriately the stalls coming from the Packet Shifter and the valid signal coming from the payload frame. It also has to manage the stall going to the payload frame, and to forward the control signals coming from the Build Sequencer together with the packet length and the shift value.

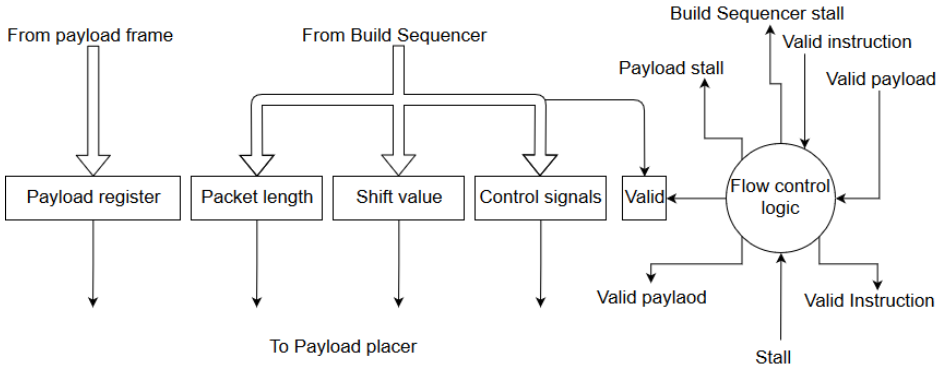
The stall signal coming from the Packet Shifter simply blocks the reading of the payload frame. It is also forwarded to the Build Sequencer. The valid signal coming from the payload frame will block the reading of the payload frame. The valid signal coming from the Build Sequencer will stop the reading of a new payload, as there may be instructions not valid coming from the Build Sequencer. The Build Sequencer also has to tell the Payload Reader whether to ask or not for a new payload. This happens, for example, in the third clock cycle of the scenario in Figure 4.6, where the Payload Placer doesn't need to ask for a new payload line. We can see that in this case the stall line behaves like a request line for a FIFO, as whenever there isn't a stall the payload placer is requesting a new line from the payload FIFO. The packet length and the shift value are also forwarded if they are valid. Otherwise they will be discarded. The valid line for the Packet Shifter is also forwarded.

Figure 4.7 shows the diagram of the Payload Reader. It is simply a register for the payload line, a register for the instruction signals for the Payload Placer and some control circuit for the flow control management.

4.2.3 Build Sequencer

As we have previously seen, both the Payload Reader and the Payload Placer have to be controlled. The Payload Placer has N+1 two to one multiplexers to be controlled. The output validity bit also has to be determined. On the other hand, the Payload Reader has to know whether to read or not to read the payload frame. The task of generating these control signals is performed by the sub-component called Build Sequencer. This block gathers data from the Instruction Computer and processes or forwards such information to the Payload Reader and the Payload Placer. It is the control core of the Packet Builder as it generates most of the

Figure 4.6. Payload Reader diagram



signals needed in most cases.

There are several ways to design this component. Since it is a control unit, we have to choose between the most common practices that deal with control units. In the architectures of pipelined processors, there are two main ways to create a control unit. One is through the use of a Finite State Machine (FSM): states are defined and have outputs depending on the state and on the instructions arriving from the memory. The other way is to create micro-instructions which will control directly each stage of the pipeline. Our case is slightly different from the one of a standard pipelined processor, as in our case there aren't any feedbacks between different pipeline stages. This avoids a lot of problems regarding hazards, as explained in Section 2.4. From a system perspective, the control machines have been distributed, and each component has its control logic and its flow control logic. This is a partial view about the control machines, as there is in fact only one control machine, this one. Most of the control in the other components is dedicated instead to flow control.

When choosing which type of control unit to develop, we have chosen to use a micro-architecture approach, as we have two components that have distinct signals to manage and many cases to handle. However, this is a partial view of this component. If we look at micro-instruction oriented control units inside standard pipelined processors, they have in a way a finite state machine to handle the sequential circuit. The point is that a micro-instruction oriented control unit has a very simple and small finite state machine, which basically only increments the memory address and thus is not classified as a finite state machine control unit, but as a micro-architecture control unit. In our case, we have a more sophisticated

temporal logic, and therefore the control unit is modeled also as a finite state machine. Our approach is to treat the output signals as micro-instructions and the overall component as a finite state machine.

At each clock cycle the Build Sequencer will send a packet of instruction signals (without considering flow control) to the Payload Placer and Payload Reader. These instructions behave effectively as outputs of a Mealy finite state machine, as in fact they are composted from a register which holds the current state, a register that holds the next state and a combinational circuit that will compute the output signals of each instruction depending also on the inputs of the Packet Builder.

There are two types of instructions we have to distinguish:

- First type: in this clock cycle we are going to build a new packet
- Second type: in this clock cycle we are going to compute a packet that needs to be built in more than one clock cycle, and this clock cycle isn't the first one for this packet

These instruction, which are the outputs of our FSM will be discussed in the next subsection.

As we have said this sub-component behaves like a Mealy finite state machine; Figure 4.8 shows the machine for $max(N) = 4$. The $max(N)$ value which has been defined in Section 3.1 is the maximum number of payload lines a packet can be distributed over. Each packet has therefore its relative N value, which is described in Equation 3.1, and from now on will be referred as the Used Lines variable. The maximum number of Used Lines ($max(N)$) is the core parameter for this machine, and the Used Lines variable (N) is an input of this machine. The Instruction Computer has to continuously provide this value to the Packet Builder.

Each machine has a number of states equal to the maximum number of Used Lines. It has a state whose output are first type instructions and $max(N)-1$ states whose outputs are second type instructions. Depending on the Used Lines value, the machine goes from the first state to one of the remaining states, and at the end of each second type instruction state the machine goes back to its initial state, which is the first type of instruction state. In the diagram, not all the outputs have been shown. It is only shown which register has to be selected. By this we mean that the chosen register selects its input from the payload line, and all the other registers don't. This means that the first step register in the case it's not selected will choose the Cache Register as input. The signals regarding the payload read for the Payload Reader, the validity of the packet at the output of the Payload

Placer, and the signal controlling the Cache Register multiplexer are shown in each transition. The inputs of this machine are basically three:

- Valid Build-instruction / not valid Build-instruction.
- Used Lines equal to 1,2,3 or 4.
- Zero shift equal to 1 or equal to 0.

Zero shift means that the incoming shift value is equal to Zero. Used Lines equal to 1,2,3 or 4 means that the Packet is distributed on 1,2,3 or 4 lines of payload. It has to be noted that the Step Registers are controlled by a word which uses a One-Hot encoding. We could have encoded this information with a binary number on two bits in this case, but since the Payload Placer requires a One-Hot encoding to pilot the multiplexers, any other encoding is considered redundant. Valid /not valid Build-instruction means that the input signals are not valid. This is a flow control signal, but it is handled in this machine as well. The remaining flow control is handled separately and will be discussed in the end of this section.

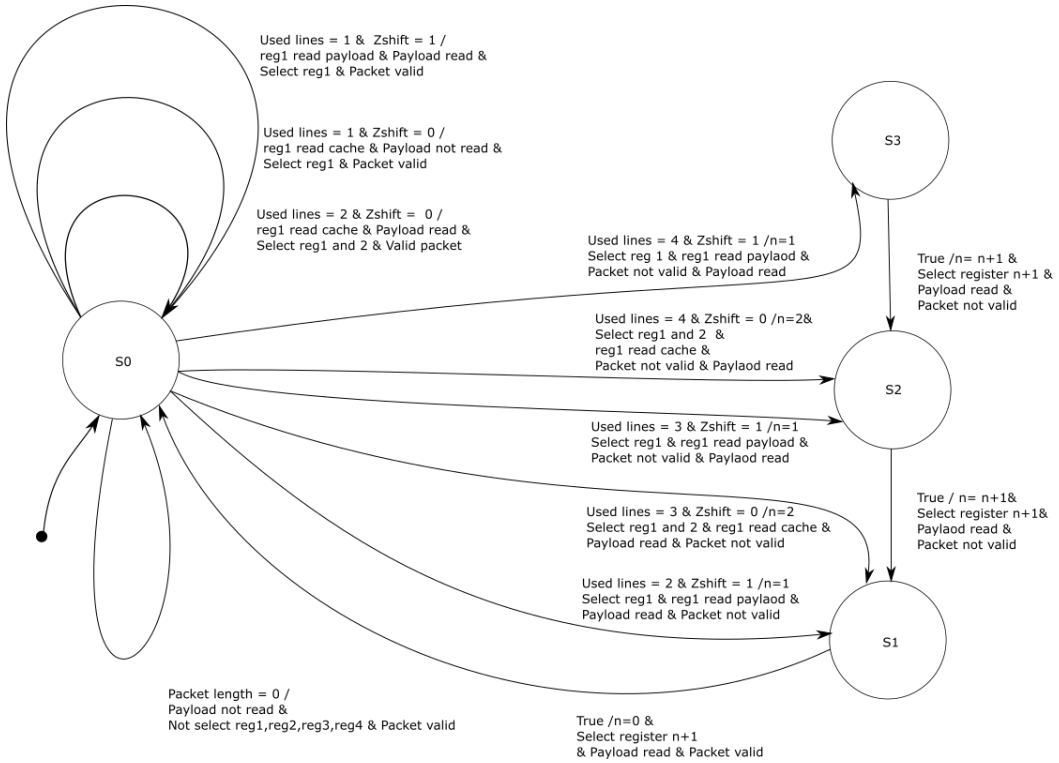
4.2.4 First type instructions

The first type of instructions, which are the outputs of the state zero (S0) of the FSM, have the complex task to manage all the cases when the used lines are equal to one or two, thus controlling the first step register, the second step register and the Cache Register. It has also to determine whether to read or not the payload frame, the validity of the output line and if there is a need to go to a second type instruction state. This type of instructions behave like the outputs of a Mealy state machine. Consequently the outputs have to be determined considering the inputs of the machine.

We have in total four output signals and the register word to manage. The register word is encoded with the One-Hot encoding, and whenever a step register has to be enabled, the bit controlling its multiplexer will be set to one, thus selecting the payload line as input of the step register. The remaining signal will determine :

- If we have to read or not a payload line.
- Whether the first step register has to read from the cache or not.
- If the built packet is valid or not.
- If we need a second type of instruction or not.

Figure 4.7. State diagram when the maximum number of Used Lines is 4



The signal that determines whether to read from the payload line is the same that enables the Cache Register. This means that whenever the payload frame is read and the incoming payload line is valid the Cache Register will automatically store the new payload line. The line that determines whether to read the cache or to read the payload for the first step register is apparently in conflict with the register word, as they both control the input multiplexer of the first step register. However, the register word has higher priority, and when it isn't set, the register won't read any input. When it is set, depending on the other line of the multiplexer, the first step register will select the Cache Register or the payload line. here is a table resuming the cases of the multiplexer of the first step register.

Registers word[0]\Cache select	0	1
0	Keep previous value	Keep previous value
1	Read payload line	Read Cache Register

There is a total of six cases that have to be determined in this type of instruction, and to determine these cases we need the following input signals:

- Used lines equal to One
- Used Lines equal to Two
- Shift value equal to Zero

Used Lines equal to one or two means that a target packet is completely stored inside only one or two lines of payload. Zero shift means the the shift value is equal to zero, and therefore that the packet exactly begins at the beginning of the first payload line it is stored in. Therefore, the first type instructions control six different cases:

- All the needed payload bits are inside a new payload line, not in the Cache Register
- All the needed payload bits are inside the Cache Register
- Part of the needed bits are inside the Cache Register, part come from a single new line
- We need two new payload lines without the use of a Cache Register
- We need more than one new line with the use of the Cache Register
- We need more than two new payload lines without the use of a Cache Register

The first case happens when the packet is smaller than the memory line and it is completely held inside the memory line; plus it means that it starts exactly at the beginning of the corresponding line of payload. Therefore, the line inside the Cache Register doesn't hold the packet and the first step register has thus to read the payload line. The Payload Reader has therefore to ask for a new line from the payload frame and to store it both in the first step register and in the Cache Register of the Payload Placer. The remaining step registers don't have to store any new line, and will hold their respective previous values. Since the packet ends in this payload line, there aren't going to be transitions to second type instruction states, and the next state is going to be a first instruction state. The signals values that determine this case are: Zero shift active, Used Lines equal one active and Used Lines equal to two not active. The corresponding output signals are: Read payload line, first step register read from payload, and all the other signals hold the previous value. Table 4.1 contains the signal values for all the cases explained in this subsection

The second case happens when the packet is completely held inside a payload line and this payload line is already stored inside the Cache Register. This happens each time the packet it is completely held inside exactly one memory line but it doesn't start at the beginning of the line of payload. In this case the cache already contains the packet we are going to need and therefore we will not need a new line of payload. The difference between this case and the previous one is that we do not need a new payload line but instead we have to use the value inside the Cache Register. Furthermore, the first step register will read from the Cache Register, not from the payload line. As in the previous case, there is no transition to second type of instructions states.

The third case happens when we have a packet that belongs to two lines, and one of them is already inside the cache. This means that we cannot discard the cache and that we need to ask for a payload line. The line asked is going into the second step register only, the first step register is going to read the cache value. The Cache Register is going to read the payload line, as every time the payload is read the cache will automatically store the payload line read. Both the first and the second step registers are enabled. As a packet is decoded in one clock cycle, we do not need extra instructions of the second type.

The fourth case is the same as the first case, but with two lines of payload. The packet starts exactly at the beginning of the payload line and it ends in the second line of payload. This means that we are going to ask for a new payload, but the first step register won't read from the cache but from the payload line. Also we are going to generate a second type instruction in the next clock cycle. It has to be noticed that this case is equal to a case where a packet is placed inside more than two lines, which is the sixth case listed, as the difference is that the next state is going to be corresponding second type of instruction state different from N=2.

The fifth case happens when we have a packet that is partially held inside the cache and the rest is distributed among more lines of payload. The cache is selected as input for the first step register, and the payload is requested and stored inside second step register. This means, as Case 3, that both the first and the second step registers are enabled. The next state is going to be a second type instructions state.

We have to notice that in this state we handle also the Used Lines equal two case. This may seem redundant as the second type of instruction handles all the cases when Used Lines is greater than one. There is also a state dedicated to the Used Line equal two state. However the difference between the Used Lines equal to two and the remaining number of Used Lines stands in the third case of the ones listed above. In that specific case the Used Lines are equal to two, but the payload line to ask is only one. This means that the packet is built in just a clock

Table 4.1. First type of instruction inputs and outputs

Case	Inputs		
	Zshift	Ule1	Ule2
C1	1	1	0
C2	0	1	0
C3	0	0	1
C4	1	0	1
C5	0	0	0
C6	1	0	0

Case	Outputs				
	Payload read	Cache select	Registers word	Valid packet	Second type instruction
C1	1	0	1000	1	0
C2	0	1	1000	1	0
C3	1	1	1100	1	0
C4	1	0	1000	0	1
C5	1	1	1100	0	1
C6	1	0	1000	0	1

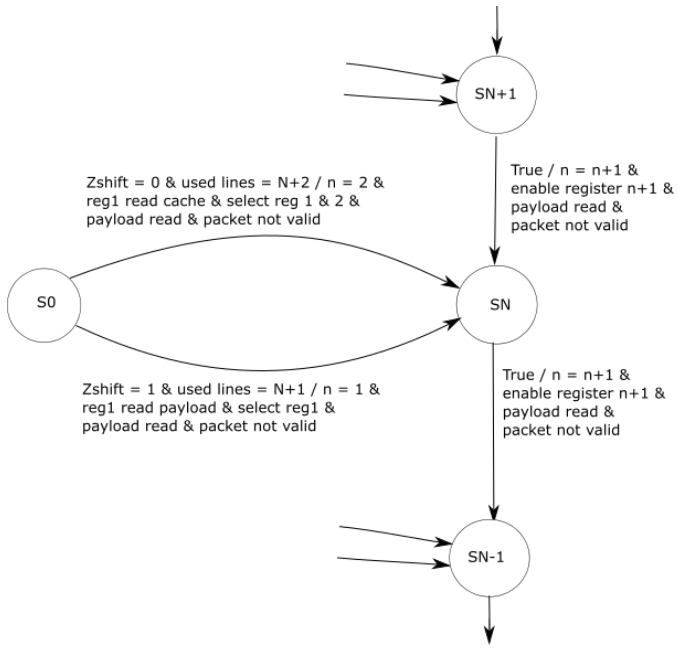
cycle, and thus it is a first type instruction. In case four instead, we have to go to another state even if the Used Lines are two because we need another payload line, just like with every other number of Used Lines.

4.2.5 Second type instructions

The second type instructions are needed after cases four, five and six of the first type instruction. When this situation happens we have to deal with the states that follow the first type instruction state until the packet is completely built.

The second type of instructions is represented in the state machine by all the states except for the first, which is the one described above. All these cases behave similarly: The state to go is determined by the number of lines the packet is distributed in the frame (Used Lines), which is an input, and by the Zshift value. As we have seen in the previous section, the Zshift value determines whether we use two or one step registers in the first type instruction state. Therefore, having the Zshift line set means that the state number has to be Used lines minus 2, otherwise the state number will be Used Lines minus 1. The register word is going to be incremented at each transaction. In the transaction from S0 to a generic SN state, as explained in the previous subsection, the active register will be only the first in the case Zshift is set, and the register to be activated in the following transition has to be the second. Otherwise, both the first and the second registers will be activated, and thus the register to be activated in the following transition

Figure 4.8. N state diagram



has to be the third, and so on for the next stages. This number is encoded with the One-Hot encoding (4 is going to be 0001, 3 is going to be 0010 etc...). The valid line is inactive, as the packet is completely built only if all the registers contain their relative lines, and this happens only during the first type instruction state, which is state S0. The payload frame has to be read and the next state is going to be the N-1 state, where N is the number of the state. N = 0 means that the next state is S0 which corresponds to a first type instruction. Figure 4.8 shows the details of a generic N state. This diagram is not valid for the $Smax(N)$ state and the S1 state, as the $Smax(N)$ state has only one input transition (only if Zshift is set), and the S1 state has a different output transition as the packet is valid in the final transition to the S0 state.

4.2.6 Packet Builder scenario

In this subsection is presented how the payload frame proposed in Chapter 2 is processed by this component. Figure 4.9 shows the behaviour of the Payload Placer and some output signal of the Build Sequencer. Each horizontal line represents one

clock cycle, the first clock cycle is at the top of the figure and the last clock cycle is at the bottom of the figure. In this scenario the frame is supposed to start at the beginning of the reported payload frame and to finish at the end of the reported paylaod frame.

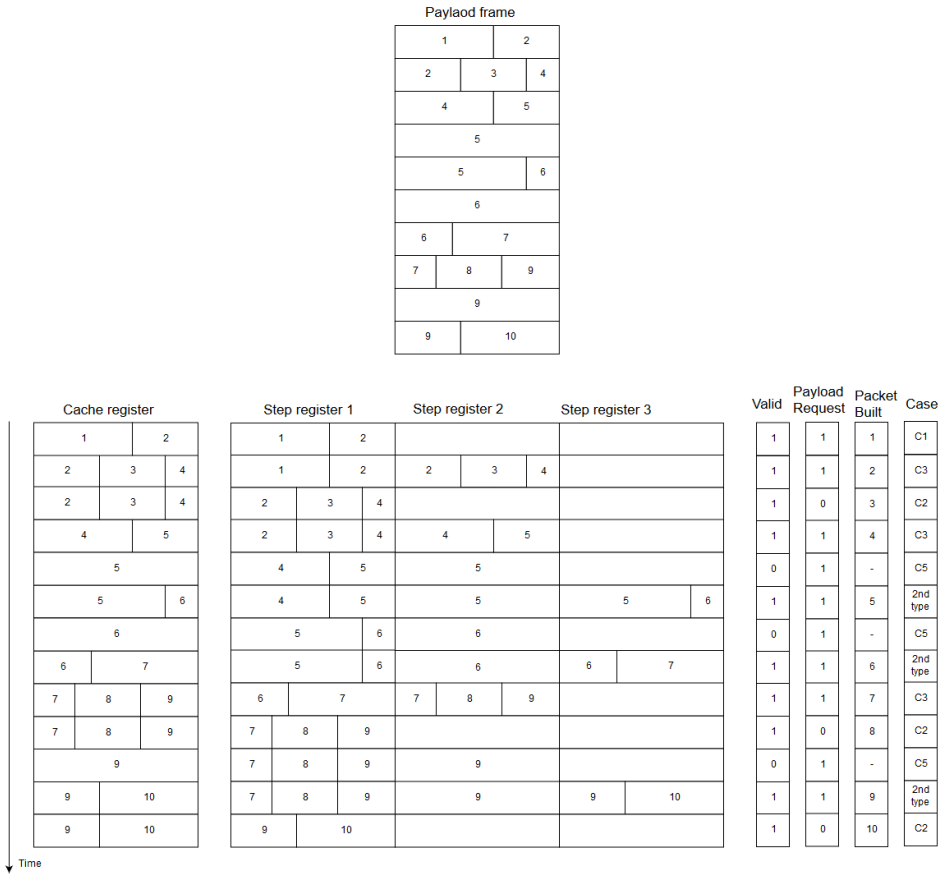
On top of Figure 4.9 is shown the payload frame. This frame, however, is not temporally related to the built frame as there are clock cycles when the Packet Builder doesn't ask the payload frame any new line. On the figure above is reported the temporal visualization of the content of the Payload Placer. On the left of Figure 4.9 there is the Cache Register. We see that whenever the payload request line is active, it automatically stores the input payload line. On the center of the figure it is reported the content of the step registers. Only three of the four registers are reported in Figure 4.9 (packet 5 can possibly be stored on four payload lines) as in this frame the fourth step register is never used and thus it is redundant to show it on Figure 4.9. On the right there are reported some signals generated by the Build Sequencer, and on the foremost right are reported the cases of the Build Sequencer. It has to be noticed that the payload request value is correct for every case, but since in the pipeline of the circuit the Payload Reader is one stage before the Payload Placer, the payload request happens one cycle before it is reported on the relative line. Instead, since the payload request line also controls the input multiplexer of the Cache Register, the Cache Register is enabled in the same clock cycle the payload request line is reported. It is also reported which packet is built whenever a built line is valid. If a step register is disabled, the rectangle is reported as white. It means that it holds the value held in the previous clock cycle.

In this scenario proposed in Figure 4.9 there are reported four out of six cases of the first type instruction and one second type instruction. The first line is a case 1 of the first type instruction. The packet to be built is packet 1 and it begins exactly at the beginning of the payload line. This means that Ule1 = 1 and Zshift = 1. The payload is requested, and the line is therefore automatically put in the Cache Register and also in the first step register. All the other registers are disabled. The packet is valid and thus sent to the Packet Shifter.

The second line is instead a case 3 of the first type instruction. In this case the packet is partially hold in the Cache Register and partially on the next payload. This means that a payload line is requested, and then put both in the Cache Register and in the second step register. The first step register instead reads from the Cache Register. The packet is valid.

The third line is a case 2 of the first type instruction. Since packet 3 is completely held inside the Cache Register, there is no need to ask for a new payload line. The first step register reads from the Cache Register and the built line is valid.

Figure 4.9. Packet Builder scenario



The fifth line is a case 5 of the first type instruction. The packet 5 is held in three different lines, and part of the packet is inside the Cache Register. What happens is that the first step register reads from the Cache Register, the second step register and the Cache Register read the payload line, and there is a transition to a second type instruction state, which referring to Figure 4.7 would be S1. The output is therefore not valid.

The sixth line is a second type instruction, and in particular is the transition from S1 to S0. This means that the third step register stores the payload line, the

payload is read and therefore also the Cache Register stores the payload line. The packet is valid and it is the packet number 5. All the other lines are cases of the one previously listed, so they are not reported.

If we compare Figure 4.9 to to Figure 3.2, there are some differences. The first difference is that not all the registers are always full of valid data. For instance in the first line of the step registers of Figure 4.9 only the first step register holds useful data. Figure 4.9 also is more accurate than Figure 3.2 as there are reported also clock cycles with invalid data and clock cycles with repeated lines referring to different packets. It also may seem that there isn't any coarse grained shift, but in reality it happens each time the first step register reads from the Cache Register (except after case 2 and case 1). In this scenario it happens six times. The main difference here is that the packets are built in a reverse order from the coarse grained shift. Instead of shifting payload lines from the right to the left, the packets are built from the left to the right and the lines are eventually shifted only at the end of the building of a packet.

4.2.7 Instruction Computer interface

While in the Payload Reader we have addressed the interface with the payload frame, The Build sequencer also interfaces with the Instruction Computer. The Build Sequencer component requires a series of data related with each packet being decoded. These data are computed outside the Build Sequencer in order not to slow down the execution, as it would cost too much in terms of performance if they were obtained inside the Build Sequencer itself. The information required includes

- Used Lines
- Used Lines equal to one.
- Used Lines equal to two.
- Packet starts at the beginning of the memory line
- Packet length is equal to zero

Used Lines describes the number of memory lines the packet is spread over. If it ends or begins at the exact end of a line, the final or beginning line counts as one. This information is needed in order to determine to which state the machine has to transition. It has also to be computed when the used lines are equal to one or two as this influences the way we use the cache and also the way we switch between first and second type of instructions. We also have to know whether the packet starts at the beginning of the memory line, as this influences the way we use the Cache Register. The case when the packet length is equal to zero is needed in

order to produce a useless instruction. It could be computed also inside the Build Sequencer, but it was preferred not to do it in order to keep the design simple. All this information is provided by the Instruction Computer. As this component will behave as all the others using the the flow control technique, the data arriving can be valid or invalid and the Build Sequencer can stop the Instruction Computer with a stall line if it cannot take any new instruction.

4.2.8 Flow control management

This component has by three pipeline stages: one for the Build Sequencer, one for the Payload Reader, and one for the Payload Placer. As with every other component, the flow control technique is implemented by means of stall lines and valid lines. In this case, the component has an incoming stall line from the Packet Shifter, a valid line coming from the payload frame and a valid line coming from the Instruction Computer. It also has to signal the stall to the payload frame and to the Instruction Computer.

By implementing the streaming architecture, this component complies with its specification. It's optimized through pipelining, every feedback is kept inside a pipeline stage (in this case it is the Build Sequencer) and the flow control is also implemented. In this case, it uses an elastic approach. If a stall occurs, each pipeline stage checks whether the incoming line is valid or not. If it is valid it processes and stores the value. Otherwise it discards it and it will not signal a stall until a valid line occurs. The concept is the same already explained inside the Packet Shifter section: the registers are enabled if the stall line is not active and all the previous stages don't signal any stall and if the line being processed is not valid. This means that each component has to provide to the next the validity of its output. In the case of the Payload Reader, which has two input interfaces the output stall signals are treated independently. The one going to the payload frame signals a stall if the Payload Reader's register has a valid payload line and if the Payload Placer is signalling a stall. The stall line going to the Build Sequencer set active if the stall line coming from the Payload Placer is active and if the instruction coming from the Build Sequencer is valid (control signals, shift value and packet's length value). This means that the Build Sequencer has to signal the validity of each instruction computed as well.

In the Build Sequencer the flow control is treated independently from the finite state machine described above, as mixing both would complicate the design. It is instead made as simple as in the other two stages: if the stall line coming from the Payload Reader is active, the Build Sequencer generates the same instruction of the previous state and signals the stall to the Instruction Computer. If the

incoming line from the Instruction Computer is not valid, if the Build Sequencer is in S0, it performs a transition to S0 where it doesn't ask for payload lines but it asks only for Instruction Computer lines and disables all the step registers. If it is not in state S0 and an invalid line from the Instruction Computer occurs, it will continue the transitions down to S0 as it doesn't need any line from the Instruction Computer. In all the other cases, when the Build Sequencer is in S0, it keeps asking for instructions to the Instruction Computer.

4.3 Instruction Computer

In the previous sections we have seen how the packets coming from the memory have to be decoded. However, both the Packet Builder and the Packet Shifter need some additional information to work. This information has to be provided at execution time otherwise the whole system cannot work. The information needed by the remaining parts of the system includes

- Build instruction information for the Packet Builder: Used Lines, Used Lines equal to one and two, when a packet begins or ends at the beginning of the memory line, when a packet length is equal to zero
- Effective shift values: the number of bits each packet built by the Packet Builder has to be shifted by the Packet Shifter
- Header: the FPGA internal memory requires information about the length of each packet

This information has to be given to both the Packet Builder and the Packet Shifter in one way or another. As this information is not directly related to the lines of packets, it has to be extracted from the lines of headers. In Section 3.1 we have explained that payload lines and header lines can be treated independently. This means that the header lines will be sent to the system in a way that is exactly equal to the way we send lines of payloads to the system. This means that we have to treat these data as if they where coming from a FIFO memory, and thus that we have to keep the streaming architecture concept of flow control, pipelining, and loop reduction. Also performance optimization and portability are required.

4.3.1 Information analysis

There is more than one way to achieve this result. In Section 2.6, we arrived to a configuration of the encoded data where we have both lines of payload and lines of headers. The information to be provided to the components described above is directly obtained from the headers or by elaborating the information inside the

headers. There are, therefore, two ways to go to define the contents of the headers. One is to directly decode the information encoded inside the header into the required format. Another way is to store inside the headers partial data about the packets and then to transform them into the required information.

The first approach may seem the simplest one, as the only thing to do is to encode the headers to keep them compact and to decode each one of them in the relative build instruction information and effective shift and header. This, however, is an impossible design. While some kind of encoding-decoding of the header is required to optimize the size the headers are going to occupy inside the memory, both the effective shift and the build instruction information haven't a one-to-one relationship to the corresponding packet.

Let's start with the effective shift which is the value the Packet Shifter needs in order to shift correctly the built packet. In this case, we assume a bus width equal to a power of two. In this way, the effective shift value is exactly equal to the corresponding unsigned value, as explained in Section 4.1.3. If we take a series of built packets like the one shown in the bottom part of Figure 3.2, we notice that each packet needs to be shifted a value that doesn't depend on the packet itself, but on the previous sequence of packets. The shift value to be computed is equal to

$$Shift\ value\ =\ (Length\ of\ the\ previous\ packet\ +\ Previous\ shift\ value) \mod Bus\ width$$

Symbols are assigned to each value in order to make a more compact formula: X_i = shifted value referring to packet number i; L_i = length of the i packet ; B = bus width. Therefore the equation is going to be the recursive formula shown in 4.6

$$X_i = (L_{i-1} + X_{i-1}) \mod B \tag{4.6}$$

Since the first packet starts at the beginning of the line the initial condition is

$$X_0 = 0 \tag{4.7}$$

As it can be seen the shift value does not depend directly to the related packet, but it depends on the payload frame. We cannot decode directly this value from the relative header because as we have explained in Section 2.6 we cannot hold all the information about the sequence of the packets inside the FPGA as it would require too much area. Instead, what we get from the header frame is a series of headers, that we assume that report the length of each packet. Equation 4.6 could be optimized if we would have an ordered sequence of packets, but this is not the case. We have to grant that our system works for any random sequence of packets that have lengths belonging to the configuration file shown in Section 3.4.

This fact has been explained in Section 3.1: while the packets on average decrease in size in the frame due to the structure of the unstructured meshes, we cannot assume that all the frame is ordered in terms of packet lengths.

The same concept is applied to the Used Lines value. We cannot directly compute how many lines a packet will occupy, since we have to assume that a packet can start in any position in a memory line. By looking at the payload frame we can see that each packet will be stored into different lines depending on the length of the single packet, on the number of bits between the beginning of the packet and the beginning of the line, and on the width of the memory bus. If we look at the bottom part of Figure 3.2, we see that the number of lines the packet 2 is stored in is more than one even if its length is smaller than the width of the memory bus. It is stored inside two lines of payload. Packet 1 is instead stored inside only one line of payload as it exactly begins at the beginning of the frame. Overall the formula describing the behaviour of this value is:

$$U_i = \lceil \frac{X_i + L_i}{B} \rceil \tag{4.8}$$

U_i refers to the number of used lines. The approximation is by excess as each packet, even a zero length packet, is stored inside at least one memory line. This component is therefore going to compute these values starting from data that have a one-to-one correspondence to each packet and that will be decoded from the header values. These values will be determined at the end of the next section.

4.3.2 Equation optimization

The core point of this design is that it has to be optimized in terms of performance. This concept applies as well for this component, and while, in theory, we could just use the formulas declared in the previous subsection, this would necessary lead to reductions in performance as the direct implementation of the equations shown above doesn't lead to the use of simple multiplexers or logic gates, but instead it leads to the use of mathematical operators, which are more complex and time consuming. The previous equations have therefore to be transformed in a way that they don't include complex operations, so the optimization by means of pipelining can be achieved in a simple way.

Equation 4.6 shows how the shift value has to be computed. We can see that there are some problems related to this formula. First of all, it is recursive. This implies that a loop is needed in order to compute this value, as each recursive algorithm can have an iterative implementation. This is contrary to the paradigms of the streaming architecture. However since this algorithm is necessary to our design, we have to be sure that a complete iteration is performed within two stages

of the pipeline. The second problem is that there is division in Equation 4.6. The division is a very expensive operation, as it would have to be implemented directly on the FPGA using LUTs and FFs, leading to a large area and long delays. This two problems come from the data we are using as starting point to compute Equation 4.6. While the loop is unavoidable, the use of the division can be avoided.

The core change from the Equation 4.6 is to eliminate the division from the algorithm. This means instead of getting the length of each packet from the headers, we decode directly from the headers the result of a division. In this way the division is not performed on the FPGA but on the host computer at configuration time. Instead of providing only the value about the length of each packet we are going to give to the FPGA the value regarding L_i *mod* B. With this value available we can compute again the X_i value. Equation 4.6 thus becomes Equation 4.9. Their equivalence is demonstrated in Appendix A.1

$$X_i = (L_{i-1} \mod B + X_{i-1} \mod B) \mod B \tag{4.9}$$

We still have two module operations, that would look like a worsening of the problem. However, this two module operations will result to be redundant or easily to compute. The $X_i \mod B$ operation is in fact redundant, as the X_i value is always less than B. This fact is granted by the behaviour of the Packet Builder. It grants that each packet built begins in the first line of the extended register, as it performs a coarse grained shift on each packet larger than B. This is explained in Section 3.1, and the reason it performs it is also to simplify this algorithm. So since X_i is always less than B

$$0 \leq X \mod B < B \quad \forall X \in \mathbb{N}, B \in \mathbb{N} - \{0\} \tag{4.10}$$

Th validity of the above inequality is demonstrated in Appendix A.2 Equation 4.9 becomes

$$X_i = (L_{i-1} \mod B + X_{i-1}) \mod B \tag{4.11}$$

The second module operation has to be performed, as the sum of $L_i \mod B + X_i$ can be more than the B itself. However, it can be computed avoiding the use of a divisor and it will be transformed into an algorithm that will use only adders/subtractors and multiplexers. Equation 4.12 is composed by the two terms: $L_{i-1} \mod B$ and X_{i-1}. This two terms are by the property of the module operation shown in Inequality 4.10 always less than B. This means that

$$L_{i-1} \mod B + X_{i-1} < [\max (L_{i-1} \mod B)] + [\max X_{i-1}] = 2B \tag{4.12}$$

Since the module operation doesn't depend on the quotient of the division but on the product of the dividend and the approximation by defect of the quotient (see A.1) this means that the module operation can be written as a simple subtraction

between the dividend and a multiple of the divisor if we know in which range the dividend is

$$\begin{aligned} A \mod B &= A - kB \\ if\ kB < A &< (k+1)B \\ \forall A \in \mathbb{N}, B \in \mathbb{N} - \{0\}, k &\in \mathbb{N} \end{aligned}$$

This equivalence is demonstrated in Appendix A.3. Since we know that the dividend is inside a range that is exactly $[0; 2B[$ we can perform the operation of the module by using multiplexers and subtractors as Equation 4.11 can only result into two different cases: If

$$L_{i-1} \mod B + X_{i-1} < B$$

the result is equal to

$$X_i = L_{i-1} \mod B + X_{i-1}$$

Otherwise it is going to be

$$X_i = L_{i-1} \mod B + X_{i-1} - B$$

Therefore we can have the solution in terms of $L_i \mod B$ without using any multiplication or division. The solution to this problem is therefore

$$X_i = \begin{cases} L_{i-1} \mod B + X_{i-1} & \text{if } L_{i-1} \mod B + X_{i-1} < B \\ L_{i-1} \mod B + X_{i-1} - B & \text{if } L_{i-1} \mod B + X_{i-1} \geq B \\ 0 & \text{if } i = 0 \end{cases} \tag{4.13}$$

In this way we can solve the problem only using adders, subtractors and multiplexers since the values $L_i \mod B$ are given by the host computer at configuration time as they have a one-to-one correspondence with the length of the packets. A problem that arises from this version of the equation is that we have to perform a comparison between $L_{i-1} \mod B + X_{i-1}$ and B. This may seem an expensive operation, but in reality there is a simple way to do this comparison, as it is explained next.

The value X_i is an integer unsigned number as it is always greater or equal to 0. This value can be used directly by the Packet Shifter only if the bus has a width equal to a power of two. In all the other cases, we have to transform this value in order to make it compatible with the design of the shifter, that has an additional stage that can shift the value shown in Equation 4.3. The way to go to transform the unsigned value of X_i to a value that can be used by the Packet Shifter starts form the Subtraction 4.3. We have a stage of the shifter that can only perform a single shift equal to

$$B - 2^{\lfloor \log_2 B \rfloor} \tag{4.14}$$

while the remaining part of the shifter can perform any shift in the range

$$0 \leq x < 2^{\lfloor \log_2 B \rfloor} \tag{4.15}$$

For simplicity we are going to call k to the value $2^{\lfloor \log_2 B \rfloor}$. So there are two possibilities. The first oneis that $x < k$, in this case we have to use the part of the shifter that can perform the shift inside the range shown in Equation 4.16; otherwise the word has to be split in two different parts : k and $x - k$. This means that the k value will be transmitted through a single line to the first stage of the shifter, and the remaining part will communicate the additional shift to the bottom layers of the shifter. The word that is composed by this two terms doesn't represent the X_i value in the form of an unsigned integer. Therefore the X_i value has to be transformed into a new format.

The first layer of the shifter has been made to perform only a k shift, while the remaining layers of the shifter have been made to perform the $x - k$ shift. Therefore, the X_i value computed in Equation 4.13 is transformed into a new word that behaves like this

$$Xreal_i = \begin{cases} X_i & \text{if } X_i - k < 0 \\ X_i - k & \text{if } X_i - k \geq 0 \end{cases} \tag{4.16}$$

$$Kline_i = \begin{cases} 0 & \text{if } X_i - k < 0 \\ 1 & \text{if } X_i - k \geq 0 \end{cases} \tag{4.17}$$

The width of the $Xreal_i$ word is equal to $\lfloor \log_2 B \rfloor$ bits while the $Kline_i$ value only has one bit. The k value will be computed once again at configuration time at the host so we aren't going to perform any base 2 logarithm operation in our system.

With these equations we have solved the problem of feeding correctly the Packet Shifter. However the build instruction information still has to be sent to the Packet Builder and therefore it has to be computed. We have to compute how many payload lines does a packet occupy. This value has been already seen in the previous subsection in Equation 4.8. Once again, we fall into the problem of performing a division inside the FPGA. This is a problem that can also be avoided in a similar way we did for the values needed by the shifter. In this case, we are adding the additional information of

$$\lfloor \frac{L_i}{B} \rfloor \tag{4.18}$$

to the information to be decoded from the headers. This information will also we computed at configuration time by a host computer, and therefore it isn't a

problem anymore. Equation 4.8 becomes Equation 4.21 since we are using an approximation by defect. The module value of $L_i \mod B$ is needed as

$$\frac{L_i}{B} = \lfloor \frac{L_i}{B} \rfloor + \frac{L_i \mod B}{B} \tag{4.19}$$

$$U_i = \lceil \frac{X_i}{B} + \lfloor \frac{L_i}{B} \rfloor + \frac{L_i \mod B}{B} \rceil \tag{4.20}$$

$$U_i = \lceil \frac{X_i + L_i \mod B}{B} \rceil + \lfloor \frac{L_i}{B} \rfloor \tag{4.21}$$

This equation transforms into the next one since we are going to use an approximation by defect of the division in Equation 4.20 The problem now is how do we compute the value of

$$J_i = \lceil \frac{X_i + L_i \mod B}{B} \rceil$$

This expression can take only three values: 1 in case $X_i + L_i \mod B$ is less or equal than B and 2 if $X_i + L_i \mod B$ is greater than B, 0 if $X_i + L_i \mod B$ is equal to zero. Therefore, the value U_i translates into this equation:

$$U_i = \begin{cases} \lfloor \frac{L_i}{B} \rfloor & \text{if } X_i + L_i \mod B = 0 \\ \lfloor \frac{L_i}{B} \rfloor + 1 & \text{if } 0 < X_i + L_i \mod B \leq B \\ \lfloor \frac{L_i}{B} \rfloor + 2 & \text{if } B < X_i + L_i \mod B < 2B \end{cases} \tag{4.22}$$

This can be easily achieved with the use of adders and multiplexers as the division is already done by the computer at configuration time. The last information we have to decode is the one required by the FPGA memory: the length of the packet. Since we already have the quotient and the module of the length of the packet, with the bus width we could obtain this value by computing

$$L_i = \lfloor \frac{L_i}{B} \rfloor * B + L_i \mod B$$

However, this would imply the use of a multiplier that is considerably slower than any other operator we are using inside this component. Since the length of a packet has a one-to-one correspondence with the packet, each packet's length is going to be decoded directly from the header and then it will be forwarded to the next components.

4.3.3 Implementation and schematic

The data needed by the Packet Shifter and by the Packet Builder are obtained from with three different data :

- L_i, or the length of each packet
- $L_i \mod B$, or the reminder of the division between the length of each packet and the width of the memory bus
- $\lfloor \frac{L_i}{B} \rfloor$, or the quotient of the division between the length of each packet and the width of the memory bus

These data are going to be processed in the way explained in the previous subsection in order to obtain the data required by the components. For the Packet Builder :

- U_i, the number of memory lines packet i occupies
- $U_i = 1$, we need a line that tells when the used lines are equal to one
- $U_i = 2$, we need a line that tells when the used lines are equal to two
- $X_i = 0$, we also need to communicate when the shift value is equal to 0
- $L_i = 0$, we finally need to communicate when the length of the packet is equal to 0

For the Packet Shifter

- $Xreal_i$ and $Kline_i$, the Packet Shifter needs to know what kind of shift it has to perform

For the FPGA memory

- L_i, the FPGA memory needs to know the length of each packet as each packet is presented in a line of bits greater or equal to the packet i length.

All these data have to be computed starting from the data listed at the beginning of this section.

Let's start with the most complex values that are $Xreal_i$ and $Kline_i$. In order to compute these values, as explained in the previous section we need to compute the X_i value, that is the actual shift, represented by an unsigned integer number, to be performed by the Packet Shifter.

X_i is obtained using Equation 4.13. This equation is indeed easy to implement without using complex elements that would slow down the computation. In fact, we basically have to decide between two options; the $i = 0$ case is solved by the reset signal. Since the reset signal sets all the registers to zero, the first X_i value is going to be zero. We have to signal this by activating the valid line one clock

cycle before our system is going to be fed with values. This approach also helps us for the remaining part of this computation, since we do need to compute the X_i value based on the previous values ($L_{i-1} \mod B$ and X_{i-1}) by simply setting the valid bit one clock cycle before we grant that the sequence is correct from the beginning, and we do not have to insert stalls in this part.

Figure 4.10. Non-optimized and optimized circuit to obtain X_i

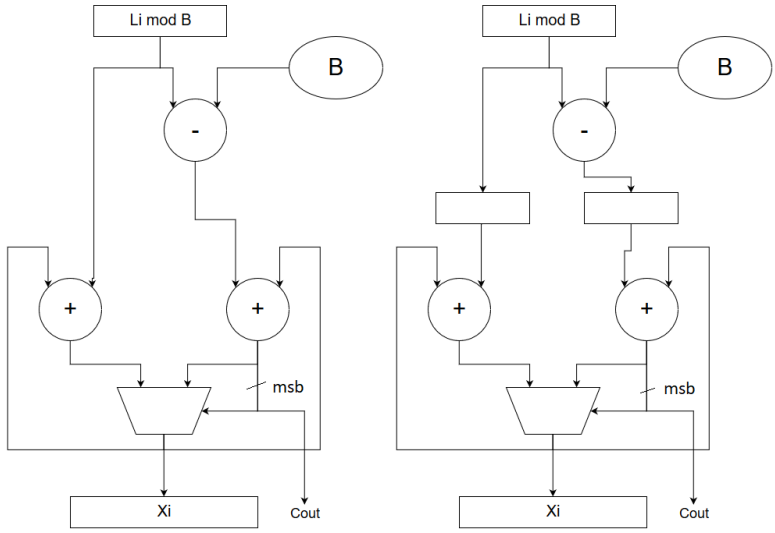
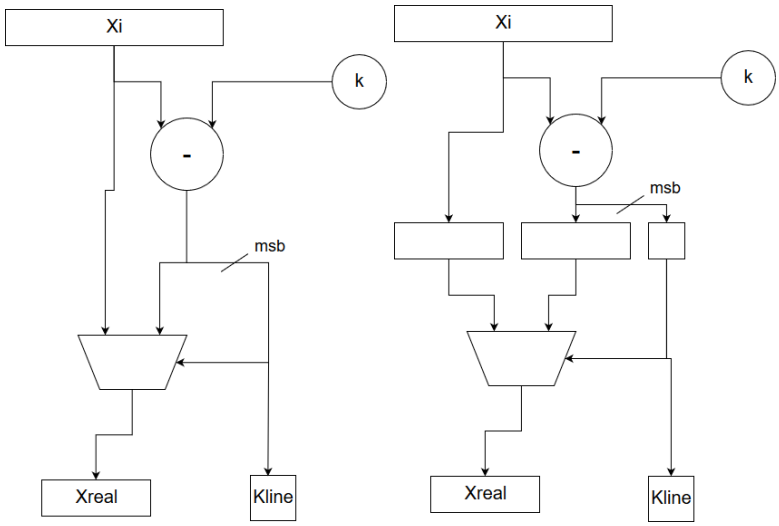


Figure 4.10 shows the DFG related to this part. Firstly we compute $L_{i-1} - B$. Then we make two additions in parallel, one is $L_{i-1} - B + X_{i-1}$ and the other is $L_{i-1} + X_{i-1}$. In order to decide which one is the correct one for the actual value we have to see if $L_{i-1} + X_{i-1}$ is less than B or not. The simplest way to decide this is to look at the $L_{i-1} - B + X_{i-1}$ addition. If this sum is less than zero the most significant bit of the result is going to be 1, as the result will be negative since signed numbers are represented on a Two's Complement representation. This bit line is going to be inserted inside a multiplexer that consequently will choose which result is the correct one. This means that we need to perform this operation on a word that is going to be one bit larger than the actual unsigned result.

As we can clearly see there is a loop in this operation. This is an unavoidable loop, as this operation is recursive. However this isn't a problem as long as we do not insert any pipeline stage inside the loop, as explained in Section 2.4 Without any optimization the critical path of this operation, is the one that goes through

the subtractor, adder and multiplexer. This can be optimized as the subtraction operation can be done before the loop itself. In this way we reduce the loop to just two adders in parallel and a multiplexer. Figure 4.10 shows on the right the optimized version of this part of the component that computes X_i

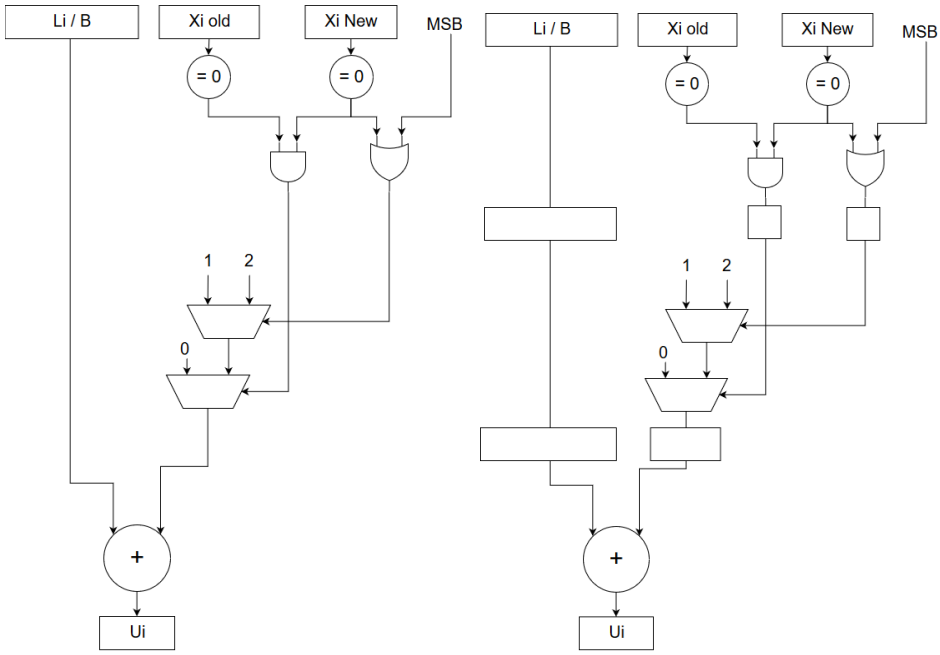
Figure 4.11. Non-optimized and optimized circuit to obtain $Xreal_i$ and $Kline_i$



The next circuit is the one that computes $Xreal_i$. The Packet Shifter needs a particular format for the shift value. as explained in Equations 4.16 and 4.17. We have a line that tells the shifter to shift by k and the remaining lines will tell to the shifter to perform a shift in the range described in Equation 4.16; This is simply achieved by subtracting to the X_i vaule the k value. If the result is greater than zero, this data will be put in the $Xreal_i$ line, and the $Kline_i$ is set active. Otherwise, the $Kline_i$ is going to be 0 and the $Xreal_i$ line is going to be X_i. Figure 4.11 shows on the left the DAG corresponding to this operation. As we see we still represent the result of the subtraction with one additional bit, as we need the most significant bit (msb) of the result to control the multiplexer, as it represents whether the result is greater than zero or not. This circuit doesn't have any loop and therefore is described by a DAG. We can optimize this design more easily than the previous one by just adding pipeline stages between each operator. This is implemented on the right part of Figure 4.11

The last computations to be performed are the ones related to the U_i value.

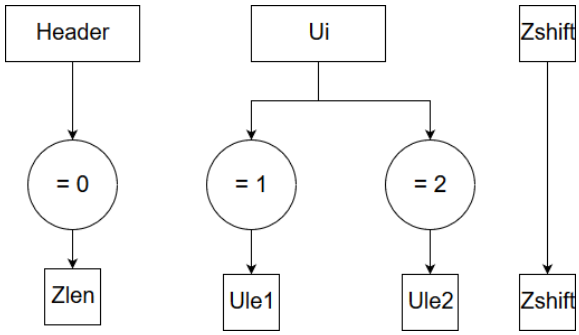
Figure 4.12. Non-optimized and optimized circuit to obtain U_i



The way to compute this value is shown by Equation 4.22. Since the $\lfloor \frac{L_i}{B} \rfloor$ value is provided by the Header Decoder, the computation of the U_i value is simple. Figure 4.12 shows on the left the circuit that computes this value, and on the right the optimized pipelined version of the same circuit. The way this circuit is made stems from the analysis of Equation 4.22. In Equation 4.22 we have three distinct cases which lead to three distinct results. The way to go to implement this three conditions is to use a three to one multiplexer. A three to one multiplexer has to be controlled by two control lines. In Figure 4.12 we can see that we have two cascaded multiplexers, one with as input a 0 and the output of the previous multiplexer, and the other with as inputs the numbers 2 and 1 represented as unsigned integers. Those values will be then added through an adder to the $\lfloor \frac{L_i}{B} \rfloor$ value. The way we have to compute the values of the lines controlling the two multiplexers is explained in the conditions of Equation 4.22. The first condition to be met is that when the value $X_i + L_i \mod B$ is equal 0, then the output of the multiplexer has to be zero. This value is composed by two terms, X_i and $L_i \mod B$. If both those two values are equal to zero, then the result has to be zero

as well. We can check if the first term is zero by doing an equality check to the X_i value. We can see instead if the $L_i \mod B$ is equal to zero by doing an equality check to the X_{i+1} value. If this value, which is the X_i value for the next packet, is equal to zero means that the packet ends exactly at the end of a memory line, and thus that also the $L_i \mod B$ value is also equal to 0. By joining this two values with an AND gate we can control the second multiplexer shown in Figure 4.12 and thus accomplish the first term of Equation 4.22. In the remaining cases we have to check whether $X_i + L_i \mod B$ is less or equal than B. The line that tells us if $X_i + L_i \mod B$ is less than B is the MSB signal of the adder in the loop that performs the $X_i + L_i \mod B - B$ sum. This has already been used inside the loop multiplexer. It is set to 1 when the result is less than zero, and it is set to 0 if the result is greater or equal than zero. It has to be noticed that the adder inside the loop actually computes the $X_{i-1} + L_{i-1} \mod B - B$, so the MSB line has to skip a pipeline stage to represent the MSB of the $X_i + L_i \mod B - B$ addition. Since the Equation 4.22 tells that the input of the adder has to be one in the case that $X_i + L_i \mod B$ is lesser or equal than B, we have to add to this signal the case when the $X_i + L_i \mod B$ is equal to B. This happen in the following case: The packet ends exactly at the end of a memory line, but it doesn't begin at the beginning of a memory line. Thus the correct way to pilot the multiplexer is to use an OR gate between the equality check of the X_{i+1} value and the MSB value. If the result is 1 the multiplexer will select 1, otherwise it will select 2. This is shown in Figure 4.12 which shows on the left the DAG and on the right the optimized version of the circuit.

Figure 4.13. Ule1, Ule2, Zlen and Zshift computing



We further have to tell to the Packet Shifter when U_i is equal to 1 and when U_i is equal to 2, and also when the packet's length is equal to 0. This is performed in a different pipeline stage as we need the result of U_i to determine these values. There

are specific operators to perform the equality check, that is as computationally expensive as a subtraction. All these operations are performed in parallel in the same pipeline stage as shown in Figure 4.13

4.3.4 Complete component and critical path

The previous section has shown all the parts of this component. By joining them appropriately we obtain the component shown in Figure 4.14. We see that it has a fixed number of seven pipeline stages. In order to know which is the critical path of this component we have to analyze all the combinational circuits between two pipeline stages.

The first stage is the subtractor for the X_i computation. This value will have a delay that depends on the number of bits of the subtractor. This is valid for ripple carry adders, that may or may not be the way this operators is implemented in the FPGA. The first subtractor has

$$W_1 = \lceil \log_2 B \rceil + 1$$

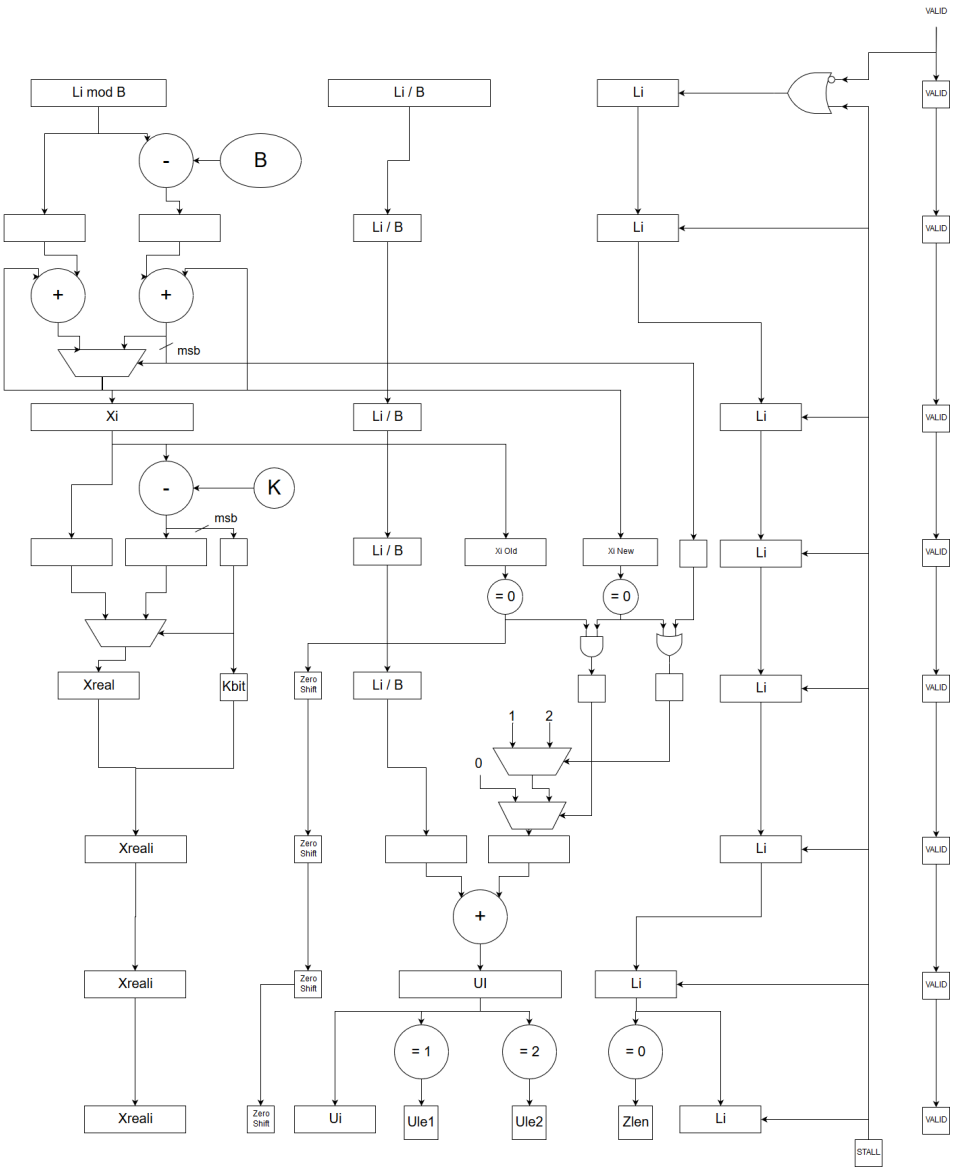
bits as it depends on the value B. The next adders are the ones inside the loop. Both have the same width of the previous subtractor, so they will perform at the same speed. The adder on the left is going to have one bit less as it doesn't need to know the sign of the result. The multiplexer delay has to be added, as the last bit of the adder on the right is the select line of the multiplexer. The multiplexer is implemented with W_1 LUTs in parallel, so the critical path of this stage is the delay of one W_1 adder plus the multiplexer delay. It happens that in no other part of the circuit there is a longer path. Examining the other components, for example the adder that computes the U_i value, it has

$$W_2 = \lceil \log_2 \lceil \frac{Lmax}{B} \rceil \rceil + 1$$

bits,which may or may not be smaller than the first subtractor. Since the equality checkers at the bottom use this same number of bits but have a more optimized circuit, they are going to be more efficient and won't be part of the critical path. The equality checkers for the Zero shift are going to have the same number of bits than X_i, which is $W_1 - 1$, and for the same reason are going to be more efficient than the adders inside the loop.

The way flow control is implemented is rigid. We see that the stall line goes up to the previous component and stalls at the same time all the registers. This implementation has been chosen because this component doesn't allows invalid lines, as each invalid incoming header is discarded. In this way, it takes seven clock

Figure 4.14. Complete Instruction Computer



cycles for the Instruction Computer to provide useful information. This may be a problem, but it happens only once at the beginning of each frame, so it can be tolerated. The invalid headers have to be discarded necessarily, as an incorrect L_i mod B value would create an invalid X_i value and thus would cause the system to malfunction.

4.4 Header Decoder

In the previous section, the Instruction Computer obtained all the information required by the Packet Shifter and by the Packet Builder starting from three values:

- L_i, the length of each packet
- $L_i \mod B$, the reminder of the division between the length of each packet and the width of the memory port
- $\lfloor \frac{L_i}{B} \rfloor$, the quotient of the division between the length of each packet and the width of the memory port

These values have to arrive somehow to the Instruction Computer. This depends on the way we encode the data in the external memory and on the way we write the configuration program, since we are not going to compute any division on the FPGA.

In Section 2.6 we have seen some ways to encode the data packets. We arrived to the conclusion that we need to transfer headers of packets to the system if we do not want to have a huge memory space in the FPGA holding all the U_i and X_i values. These headers have a one-to-one relationship with every kind of packet, as we do know that packets can only have some predetermined lengths. This one-to-one relationship applies also to the data the Instruction Computer needs. These data are therefore provided during the transmission of the data frame. By the way the header frame is made, header data has to be stored on the FPGA and consequently decoded into the desired format. The purpose of this component is to store and decode the headers. As it is a component of the Packet Decoder it has to support the streaming architecture. This means it will support flow control, loop reduction and optimization through pipelining.

In Section 2.6 we have seen some of the problems arising from the use of headers inside the encoding. This problems happen mainly because our decoder has to distinguish which line is a header and which line isn't. It has been chosen to adopt the approach with lines of headers and lines of payloads also due to the simplicity and easy adaptation to the streaming architecture. We have chosen this approach mainly because this component. With lines of headers this component is easily adapted to the streaming architecture. When a line of header arrives, a FIFO component gets filled with all the headers of a header line, one header per every FIFO stage. In this way we can read the headers in a fast and efficient way.

The key point of this component is that it will expect from the memory a header line only when it has no headers left inside the FIFO component, and therefore it

will not expect header lines at each clock cycle as it can hold this many headers:

$$H = \lfloor \frac{Header\ width}{Bus\ width} \rfloor$$

This actually means that we have to grant that the size of the headers is the smallest possible, as the bigger is the header dimension, the more frequently this component is going to expect headers from the memory, thus interrupting the flow of payload lines to the Packet Builder. This would translate into less performance of the Packet Decoder.

The way this component is related to the encoding depends on the way header lines are stored in the frame. If we do perfectly anticipate when our header register is going to be empty, then we do not need even have to distinguish between payload and headers lines, as we would rely on the capability of the frame encoder to ensure that when we need a line of headers the line coming from memory is going to be a header line. This, however, is hard to predict as this component doesn't behave in a fully predictable way. What we can know is an approximate timing of the header lines. For example, we can say that a packet that will use three lines of payload will at least require tree clock cycles to be computed. This, however, isn't necessarily true as it depends on where the packet will start and end inside the lines of payload. Furthermore, the header has to be processed by the Instruction Computer, and this makes the predictability more complicated and less deterministic. We have to grant a small flexibility in this terms to our component.

The solution adopted to address this problem consists to simply use a bit line of the bus to distinguish between payload lines and headers lines. This will reduce the width of the bus, and since it is the bottleneck of the system this solution isn't considered optimal in our case. However, this way is the most simple and effective because if the Header Decoder runs out of valid headers it will wait until a header line appear in the bus and when this happens it can easily read it as it is signaled by a bit. The system can still continue to work for at least five clock cycles after the Header FIFO is empty, as the Instruction Computer is still filled with valid headers (the first two stages of the Instruction Computer will stall if an invalid header arrives). This means that in the frame encoding, in the case H is greater or equal to 7, we have to ensure that the first line of the data line is a header line, the second header line is placed after $H-7$ packets and before $H-2$ packets and that every other line of headers has to be placed exactly after H packets and before $H+5$ packets. In the case H is less than 7 we need a sequence of $\lceil \frac{7}{H} \rceil$ lines of headers at he beginning of the data frame, and the next line of header should be placed after $7+7 \mod H$ packets and before $12+7 \mod H$ packets. For the header lines after the $\lceil \frac{7}{H} \rceil + 1$ header line, the behaviour in the case H is less than 7 is the same as if H is greater than 7.

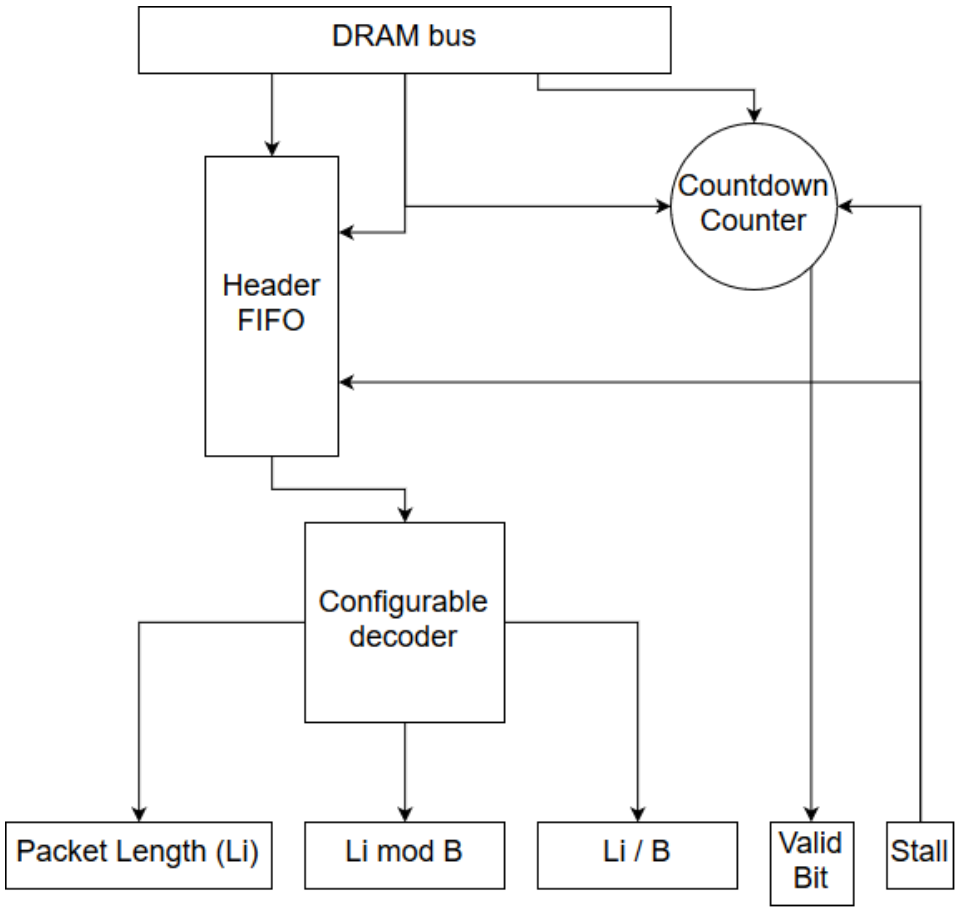
In the case a header line is placed after the higher limit of number of packets, the Packet Builder will stop requesting payload lines and it is impossible to replenish the header decoder with new headers, interrupting the Packet Decoder without possibility to recover before the end of the data frame. In the case a header line is placed before the lower limit of number of packets, at least one header will be discarded from the FIFO and so the Instruction Computer will fail to process the X_i value and consequently the whole decoding process will fail. If the header line isn't placed exactly when the Header FIFO is empty but still before the five packets limit there are going to be a number of clock cycles of invalid headers (which translate into a series of invalid packets sent to the FPGA) equal to the number of packets between the Hth packet and the line of headers. Overall a correct sequence of Header line would be: header line in the first data line, header line after $H-2$ packets, all the other header lines H packets after the previous header line. In this case there wouldn't be stalls and it also would be possible to remove the extra bit as it would be possible to determine the exact clock cycle when the Header decoder has to read the data frame.

This component has to interface with the memory and the Instruction Computer using the flow control technique. This means that this component will have a stall line going to the memory and a valid line coming from the memory, a stall line coming from the Instruction Computer and a valid line going to the Instruction Computer. The valid line coming from the memory for this component is the extra bit inside the frame. The stall line has instead to be computed and then merged with an OR gate to the payload request signal coming from the Packet Builder. The stall signal has to be generated by this component, and it is equal to the valid bit going to the Instruction Computer. There are more approaches to generate the stall signal. One approach consists of filling the FIFO with ones or zeros as every header is sent to the Instruction Computer. When all the FIFO is filled with the respective number, it will signal the memory that it is ready to accept a new line of headers, thus setting the stall line to not active. This is a simple and reliable way to perform this control action, however it has a drawback, that is that we need to reserve a header case, that is or all zero headers or all one headers. This is not a problem unless the number of possible packet lengths is exactly a power of two. In this case, the header is going to be one bit larger and thus the perfomance will lower. Another way, that is the way we adopted, is to use a countdown counter. This counter is set to the H value whenever it is replenished with headers, and each time the Instruction Computer will ask for a header, the counter will decrease. When the counter reaches zero, this component will both signal the memory that we are ready to accept a new header line, and both the Instruction Computer that now until the Header Decoder is replenished the headers will not be valid. Furthermore, the stall signal coming from the Instruction Computer will also stall this

component in a rigid way and it will also affect the stall signal going to the memory.

In this way we have assured that this component actually behaves like all the other components of the streaming architecture: it has flow control, no loops and it is optimized. Figure 8.1 shows the implementation of the Header Decoder.

Figure 4.15. Header Decoder and decoder



The Header Decoder has also to perform the task to decode the headers into the data required by the Instruction Computer. This happens because as previously said the headers have to be encoded in order to optimize the header frame. This

task if performed by a small ROM memory. This memory is written during the configuration of the Packet Decoder. This memory contains the L_i value, the L_i mod B value and the $\lfloor \frac{L_i}{B} \rfloor$ value for each possible length of the packets. This is possible as the configuration file (see Section 3.4) holds all the possible lengths and the memory port width. The way we fill the content of this memory is by using the configuration program to compute all the requested values and to directly write the VHDL file each time the configuration program is executed. In this way we achieve the goal of computing all the divisions on the host.

4.5 Configuration program

The configuration program has two main tasks: to write the header decoder directly on a VHDL file and to compute all the parameters necessary to the components and write them on a VHDL packet file. The first task has been described in the previous section. The program perform the $L_i \mod B$ and $\lfloor \frac{L_i}{B} \rfloor$ for every possible packet length inside the configuration program and writes the results on the corresponding VHDL file in the form of unsigned binary numbers.

The second task the configuration program has to perform is to compute all the parameters necessary to the components. These parameters are the width of the ports of each component and of the Packet Decoder, the k value and the B value for the Instruction Computer (see Subsection 4.3.3). However, most of the port widths depend on some basic values. It is not necessary to compute all the necessary widths, but it is necessary to compute and write only the basic values from which the others can be computed in the VHDL during the compilation of the design. As reported in Section 3.5, this part is necessary as in this way we avoid that the user inserts incorrect generic values during the compilation of the design.

The values necessary to be computed and written by the configuration program are:

- The memory bus width, B.
- The maximum length of a packet, $max(L_i)$.
- The base 2 logarithm of the number of possible packet lengths.
- The maximum number of Used Lines, $max(U_i)$.
- The base 2 logarithm of the memory bus width, k.

The first parameter doesn't need to be computed, as it is written inside the configuration file. However, it is necessary for the design as the Packet Shifter, the Packet Builder and also the Packet Decoder as module need this information. The Instruction Computer also needs this information as an input for the first subtractor (see Figure 4.14). The maximum length of a packet, together with the bus width determine the width of the output port of the Packet Shifter and of the Packet Decoder. The logarithm of the number of possible packet lengths determines the width of the headers, which is necessary to determine the width of the Header Decoder. The maximum number of Used Lines is necessary for the Packet Builder as it determines the number of states of the Build Sequencer and the number of step registers of the Payload Placer. The k value is requested by the Instruction Computer to compute the $Xreal_i$ and $Kline_i$ values and also by the Packet Shifter to determine its number of stages.

Chapter 5

Testing and results

The system described in the previous chapter complies with the specifications presented in Chapter 2. In this chapter it is going to be described how the Packet Decoder's implementation has been tested and verified in order to check if the design supports all the points addressed in the specifications. There are two main tests to be performed: one is the functionality test, which is a logic verification of the system. It ensures that our system works correctly in every possible scenario. The second test is the area and performance test. This test has been performed to provide an estimation of the performance and the resource utilization of the Packet Decoder.

5.1 Functionality test analysis

The first test is the functionality test. The way to go to run this test is to produce testbenches which will provide the logical verification of the system. The testbenches have the primary purpose to simulate the behaviour of the external modules the Packet Decoder is going to interface with. This two modules are going to be

- The external memory
- The FPGA memory

This two modules will be modeled, as explained in Chapter 3, as FIFO modules. The external memory is going to hold the data frame in the format explained in Chapter 2. The FPGA memory will consist of two FIFOs, one for packets and the other for the packets lengths; both will have the same stall line and the same valid

line.

With these testbenches we have to ensure that our system would work in every possible scenario and with every possible frame. The user interface has two input main parameters: the width of the bus and the number of possible lengths the packets can have in the frame. All the possible cases to be tested are therefore described as ranges of numbers for these values.

The core variable to be tested is the width of the bus. From this variable all the components assume different behaviours. For example, the number of stages of the Packet Shifter depends on the logarithm of this value, and therefore also the Instruction Computer has to adapt to this configuration of the Packet Shifter. The Packet Builder will change structure depending on the maximum Used Lines value: the number of step registers inside the Payload Placer and the number of states of the Build Sequencer depend on this number. The maximum Used Lines value depends on the maximum length of a packet and on the width of the bus. This value will not however produce drastic changes in the design. As explained in Section 4.2.5, every state beyond the first two works in the same way. Therefore, it is redundant to make a test for each one of those possible configurations. Instead it is interesting to see how the design may change when the maximum number of Used Lines is greater than two or less or equal to two. However, it is even more important to create various frame scenarios to test the functionality of the Packet Builder. The different possible lengths aren't pivotal cases as in fact they will mostly impact the Header Decoder, as the number of possible cases determines the width of a header, and the port width of many components, just like the biggest packet will determine the width of the Packet Shifter, and therefore also the width of the Packet Builder. This, in a way, isn't as problematic as the width of the bus, since it doesn't affect the Instruction Computer, or any particular control logic.

The testbenches have also to ensure that every frame generated from a single configuration can be decoded correctly. In every frame there are possible corner cases that will test both the Instruction Computer and the Packet Builder. This corner cases depend strictly on the way the Packet Builder has been implemented and also on the Instruction Computer.

The functionality testbenches have the purpose to ensure that the system works for every possible memory port width and for every possible payload frame. To achieve this goal we have decided to make tests for two cases in particular, as there are two cases where the system is structured in a significantly different way depending on the memory bus width. These cases are :

- Bus width equal to a power of two
- Bus width different from a power of two

The first case mostly influences both the Packet Shifter and the Instruction Computer. The Packet Shifter in this case doesn't have a first shifting stage as explained in Section 4.1.3. Therefore, the Instruction Computer doesn't have to compute the $Xreal_i$ value shown in Section 4.3.2, as it just has to forward the X_i value.

The generated testbenches provide the verification of the system for this bus widths :

- Bus width equal to 44 bits
- Bus width equal to 128 bits

These values have been chosen because they represent two realistic port width scenarios of the FPGA's port width. By looking at the IP cores of the FIFOs provided by Quartus, these two values are inside the set of possibles port widths.

The other core parameters of these testbenches are: For the 44 bits port width testbench

- Maximum packet length = 151 bits
- Number of possible packet lengths = 19
- Maximum Used Lines = 4

For the 128 bits port width testbench

- Maximum Packet length = 293
- Number of possible packet lengths = 19
- Maximum Used lines = 3

This data can be extrapolated from the configuration files, which are included in Sections B.3 and B.4 of Appendix B, where the numbers are reported in hexadecimal format.

In order to simulate correctly the system we have to simulate the behaviour of the external modules the Packet Decoder has to interface with, that is the external memory and the FPGA memory As we are supporting the streaming architecture concept, the way to represent these modules is to model them with FIFOs, as explained in Section 3.3. The first FIFO is the one containing the input frame, and will have a data structure written inside as the one described in Chapter 2. The FPGA memory, on the other hand, will have to get both the lengths of the packets and the decoded packets, that should have a structure like the one shown at the

bottom of Figure 3.3.

The results of the simulation are in Appendix B. Section B.1 shows the results for a simulation with a 44-bit port width; Section B.2 shows the results for a simulation with a port width of 128 bits. As it can be seen, the input data are encoded like shown in Chapter 2: each input line has an additional bit at the end of the frame that tells the Packet Decoder whether the line is made of payload data or header data. With payload data the final bit is equal to 1 and with header data the final bit is equal to 0. The headers are encoded. Both testbenches have header widths of five bits, ($\lceil log_2 19 \rceil = 5$) The payload data are generated in a way that is clearly detectable where a packet begins and where a packet ends. Each packet starts with a sequence of three consecutive ones, and ends with a sequence of a one and a zero. The in-between bits are all zeros. If we take a look at the output of the simulations we see that each decoded packet begins with three ones, and if we want to count the number of bits between the beginning and the end of each packet, we see that it is always the number shown at the beginning of the decoded packet, which stands for the packet length. This doesn't apply for 0 length packets as the information necessary is already inside the packet length. At the end of both frames there is a sequence of zero packets which stands for the external limit of the mesh. The output width of each output frame is equal to $Maximum\ packet\ length\ +\ Memory\ port\ width - 1$. In the first case is $44 + 151 - 1 = 194$ bits. In the second case is $128 + 293 - 1 = 420$ bits. The data shown in the appendixes have been obtained with the VHDL textio package by writing the content of the FIFO on a file at the end of the simulations. It has to be noticed that the FIFOs gather lines only when the valid line is active.

If we look at the frame configurations of the two testbenches we see that there are some corner cases taken into consideration. In the first frame, there is a packet which starts exactly at the beginning of a new line. This packet stands inside two lines of payload, and it is 82 bits long. This packet would stand inside case four of the first type instruction of the Packet Builder. There are also packets smaller than the memory line which stand on more than one payload line. In the 44 bits frame this case is represented by the second packet, whose length is 18 bits. This case is peculiar as the Instruction Computer will have to choose inside the loop a X_i value coming from the $X_{i-1} + L_{i-1} - B$ adder. It is also representative of the third case of the first instruction type in the Packet Builder. The second case of the first instruction type in the Packet Builder is represented by the sixth packet in the first frame, whose length is equal to 7. The fifth case of the first instruction type of the Packet Builder is also represented inside the test frames by each packet which extends over more than two payload lines. An example is the third packet of the first frame, which extends ever three payload lines and it is 102 bits long. This also goes inside the second type of instruction states as it is distributed on

more than two payload lines.

Figure 5.1 shows the result of the RTL simulation made with a memory port width of 44 bits on Modelsim. We see that there are only three header lines, two stalls, one lasts 8 clock cycles and another 7 clock cycles. Due to the elastic design of the Packet Shifter, after the stalls there is a sequence of consecutive decoded packets, that lasts exactly 7 clock cycles, the depth of the Packet Shifter. After this sequence there is a sequence of large packets which take more time to be decoded. Overall there are 15 packets to be decoded and the total simulation lasts 141 clock cycles, from the end of the reset line to the last packet decoded. It is also interesting to see that the request payload line isn't always active. This happens because the frame has a lot of cases where the packet spreads on more than three lines of payload, thus going inside the second type of instruction of the Packet Builder, where the payload request is not active. The payload request line functions also as header request line. The frame decoded is shown in the Appendix B.1.

Figure 5.2 shows the result of the Instruction Computer loop for the same simulation of the first 8 packets. The result of the operation is the top line of the chronogram. The line below shows the result of the adder which performs the $X_{i-1} + L_{i-1}$ addition and the third line shows the result of the adder which performs the $X_{i-1} + L_{i-1} - B$ addition. The results of these operations are shown as unsigned decimal numbers for the first line and as signed decimal numbers for the second and third lines. By analyzing the payload frame of the 44-bit port testbench, we can see that the results are correct. The first value is zero (not shown), the second value is 27, which is equal to $(0 + 72 \mod 44)$, the third value is equal to $(27+18 \mod 44-44) = 1$, the fourth value is equal to$(1+102 \mod 44)$ which is equal to 15 and so on. It is interesting to see that each time the third line is less than zero, the result of the operation is the second line. Otherwise the result is the third line itself. It is also interesting to notice than there are cases where the first adder has a result higher than B -the second value in the chronogram- which should be impossible by the structure of the Packet Builder. In this cases, the other adder always has a positive number. This happens because the quantization of the adder in this case is equal to $\lceil log_2 44 \rceil + 1 = 7\ bits$ and this number of bits is enough to represent signed decimal values up to 63. If we also compare the results of the operation with the frame shown in Appendix B, if we count the bits from the beginning of each line where a packet begins up to the beginning of the packet we see that the result of the loop corresponds to the X_i value of the first 8 packets.

Figure 5.1. Packet Decoder simulation for 44 bits bus

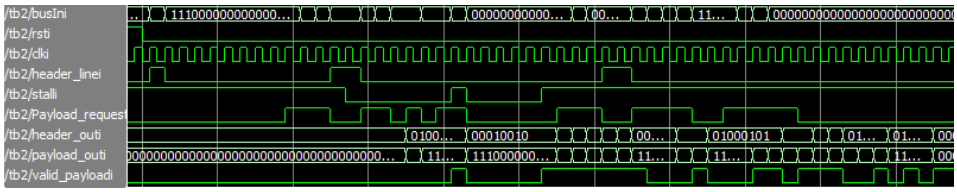
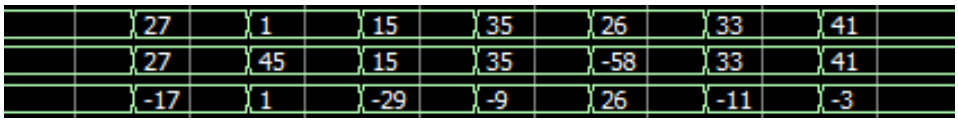


Figure 5.2. Loop simulation for the 44 bits bus



5.2 Performance and area tests

In the previous section, we have proven that this system works correctly in all the corner cases of the inputs, and this ensures that the Packet Decoder complies with the requirement of decoding the packets, being portable for multiple kinds of FPGAs and functioning correctly for every possible frame.

What is also required from the specification is that the system is optimized in terms of performance, and also that the amount of used resources isn't too high. In order to test the performance of the Packet Decoder, a timing analysis for a specific FPGA has to be performed. In our case, we have chose the Stratix 10 FPGA, which is one of the two FPGAs where the Packet Decoder is going to be implemented. The way to go for this test is to add, as in the previous test, two FIFOs at the beginning and at the end of the system, but while in the previous test the only purpose was to test the logical functionality, in this test there is a need to consider the physical delays of every circuit. The synthesis and implementation of the design on the FPGA are therefore also required.

The three FIFOs at the beginning and at the end of the system need to have two different clocks for the input and the output. The input of the FPGA FIFO and the output of the external memory FIFO have to have the same clock as the memory decoder, while the output of the FPGA memory and the input of the external memory FIFO have to be different from the one of the FPGA. To grant this, after the generation of the clock we added a PLL that changes the frequency of the clock, and then we connected the clocks to their correspondent ports.

The synthesis and implementation also has an influence on the test, as depending on where the components are placed inside the FPGA, there may be some additional delays in the system. Since the synthesis tool cannot be customized completely for our system, what has been done is to set the syntheses tool for maximum performance optimization. The width of the ports have been set to 44 bits at the input and 194 bits at the output for the first test, like in the one used in the previous section. For the second test we used 128 bits at the input and 420 bits at the output, once again like the case used in the previous section. The frequency of the clock has been set to 800 MHz and 300 MHz for the input and output FIFOs.

The results are shown in Table 5.1. Inside the table it is reported the maximum clock frequency, in Megahertz, the critical path, the resources occupied by each component and in total, in the form of ALUTs (Asynchronous Look-Up Tables) and LRs (Logic Registers). The ALUTs represent the combinational logic resources, while the LRs represent the sequential logic resources. We firstly see that our system has as critical path the stall line. This was predictable as it has to go from the output FIFO straight up to the input FIFO passing through all the logic gates to grant the elasticity of both the Packet Builder and the Packet Shifter, plus the extra logic for joining the header and payload FIFO. The frequency obtained is therfore higher for the smaller design, and lower for the bigger design. Nevertheless, frequencies are above 400 MHZ in both cases. The resources occupied by the modules are 1379 ALUTs and 1798 logic registers for the 44 bits implementation and 3319 ALUTs and 4097 logic registers for the 128 bit implementation. Considering that a Stratix 10 GX 400 has 378000 logic elements [7], the larger design with the 128 port width would occupy 0,878% of the overall logic resources. As expected, the largest component is the Packet Shifter, followed by the Packet Builder, and then the Header Decoder . The Instruction Computer is the smallest component. In all components except the Instruction Computer the components are larger in the 128-bit port design than in the 44-bit port design. The decrease of used ALUTs in the Instruction Computer may happen due to the fact that the maximum $\frac{L_i}{B}$ value is lower in the 128-bit port design than the 44-bit port design.

By taking a look at Table 5.1 we can see that the resource occupation results are consistent with the design. We see that the biggest component is the Packet Shifter. From this component, we know that the width of each register in the case of the 128-bit input port Packet Decoder is equal to the output port of the Packet Decoder, which in this case is 420 bits. Since the Packet Shifter number of pipeline stages is equal to $log_2 B$, in this case it is equal to 7. It is expected, therefore, to have 7 registers of 420 bits each, occupying therefore at least 2940 logic elements, which is very close to the results shown in the Table 5.1, which also includes the

registers for the header and the valid bits. The same reasoning can be applied to the Packet Builder, where the Payload Placer has an extended register of 128*3 = 384 bits, and the Payload Reader has a 128-bit width for the paylaod lines. The Packet Builder should also occupy at least 512 Logic registers, without considering the Build Sequencer and the flow control, the header and the shift value registers. In fact, we see that the value for the Packet Builder is larger than the computed value, making the Packet Builder the second largest component of the system.

Table 5.1. Performance evaluation and resource occupation of a 44 bits input port width Packet Decoder and of a 128 bits input port width Packet Decoder

44 bits input port width	194 bits output port width			
Maximum frequency	450,65 MHZ			
Critical path	Stall line			
Component's resources	# of ALUTs	%	# of LRs	%
Packet Builder	94	6,81%	364	20,24%
Packet Shifter	1170	84,84%	1233	68,57%
Instruction Computer	57	4,13%	147	8,17%
Header Decoder	56	4,06%	44	2,44%
Total	1379	100%	1798	100%

128 bits input port width	420 bits output port width			
Maximum frequency	426,99 MHZ			
Critical Path	Stall line			
Component's resources	# of ALUTs	%	# of LRs	%
Packet Builder	180	5,42%	764	18,64%
Packet Shifter	2947	88,79%	3031	73,98%
Instruction Computer	47	1,41%	163	3,97%
Header Decoder	143	4,30%	139	3,39%
Total	3319	100%	4097	100%

Our final performance issue to consider is the actual gain that the approach taken in the design of the Packet Decoder might provide. The common approach for a high-performance implementation that has to deal with irregular data (variable length packet in our case) is try to achieve a uniform processing of the irregular data. In our case, this would translate into transforming the stream of irregular packets into a stream where all packets have the same length, namely the length of the largest irregular packet. This would allow uniform, and therefore, simple processing of all packets.

However, the Packet Decoder presented here targets specific applications (for example CFD algorithms) whose FPGA implementation is clearly memory-bound. This means that, despite a lot of processing has to be performed in the FPGA, a deeply pipelined implementation supported by elastic interfaces will allow high frequency clocks. Consequently, the bottleneck of the system is not the processing in the FPGA, but rather the transfer of the data from external memory to the FPGA. Thus the memory-bound descriptor for this type of applications.

In this context, transforming a stream of irregular data into a stream of uniform data packets has negative consequences in terms of performance, as the overall computation time is basically determined by the time it takes to transfer data from memory to the FPGA. This is why a novel approach has been taken in the design of the Packet Decoder, where the whole design aims at dealing with streams of irregular data packets.

In order to determine the gain provided by this novel approach, it is clear that complete systems with irregular and regular data should be implemented so their performance (including real stall cycles) could be computed for a relevant amount of datasets. Naturally, this falls out the scope of this thesis. However, it is still possible to have an estimation of such gain by considering basic analytic models of both implementations.

Let's consider frames with N packets decoded in the 44-bit and 128-bit implementations considered so far. If uniform length packets are forced, the maximum packet lengths of 151 and 193 bit, respectively, must be considered. The simplest uniform approach is to assume that a packet takes three cycles to be issued from memory to the FPGA, as this is what it takes to send the largest packet in both cases. So, assuming that decoding is simple and perfect and no stalls are chased by the decoder, the time it takes to trim for and decode the frame can be approximated to 3N cycles.

On the other hand, if the proposed implementation is evaluated, some assumption must be made. First, as flow control supports elastic interfaces, it is assumed that any stall caused by packets using more than one bus line are compensated by situations where several packets fit in a single line, so overall the assumption is that no stalls propagate to the inputs of the decoder. Therefore, the processing time only depends on the length of the data frame. The second assumption relates to the numbers of packets of each size considering the sizes listed in the configuration files included in Appendix B which have the structure described in Section3.4. Even though large sizes tend to be rare in a real CFD mesh, we consider the pessimistic approximation of considering that there is the same amount of packets of each size. Since there are 19+1 (zero length) = 20 different sizes in both designs,

there are $\frac{N}{20}\sum_{0}^{19} sizes$ bits. This value must be divided by the bus width (44 or 128) to determine the cycles required to transfer and decode the frame. In particular, for the 44-bit port width design, this value is $\frac{1095N}{20*44} = 1{,}24N$ and for the 128-bit port width design is $\frac{2049N}{20*128} = 0{,}79N$. Therefore, the proposed approach estimated speedups are 2,42 and 3,80 respectively.

Chapter 6

Conclusions

6.1 Overall conclusions

This thesis has as main purposes to create a frame encoding capable of optimizing the stream of packets of variable lengths, and to design a high performing Packet Decoder capable of decoding the encoded frame. These two parts have to comply with additional requirements: the frame has to be optimized, and has to provide the decoder all the necessary information to decode the packets. The Packet Decoder has to function for every possible frame, be high performing, be portable for multiple FPGAs, it has to comply with the streaming architecture paradigms, and doesn't have to use excessive resources. An additional objective is to define clearly the specification of the Packet Decoder and of all its components. All this points have been proposed in Section 1.2.

The Frame design proposed in Chapter 2 complies with its main objectives, which are the optimization of the density of the payload frame and the fast decoding of the packets. The payload encoding is optimized so each packet inside the paylaod lines begins exactly at the end of the previous packet. Therefore no bits are wasted in this regard. The header encoding is also optimized so each header occupies $Log_2\#\ of\ possible\ packet\ lengths$ bits thanks to the binary encoding. The fast decoding of the packets is granted by the division of the payload and header lines and by the additional bit in each line.

The decoder design also complies with its specification as demonstrated in the previous chapter. It decodes every possible packet inside the values declared in the configuration file. This means that the Packet Decoder can decode every possible unstructured mesh. The Packet Decoder also grants high performance. This is

shown in the previous chapter, as the versions with 128-bit input port and 44-bit input port have maximum operating frequencies above 400 MHz. It does so using limited resources, as these versions occupy only the 0,878% (128-bit input port) and the 0,365% (44-bit input port) of the logic elements of a Stratix 10 GX 400. The design has been made also portable for different bus widths. This is demonstrated by the two tests made in the previous chapter, where we proved that the system works for two different input port widths. The system also complies with the streaming architecture. Each component has its flow control and there aren't any feedbacks between different pipeline stages. It is also deeply pipelined as for the 128-bit input port version the design has 17 pipeline stages.

This system is, therefore, an example of how the FPGAs can be used to implement high performance algorithms that process irregular data. The streaming architecture for FPGAs has been proven to be effective for the decoding of packets of variable lengths, and thus capable of accelerating CFD algorithms based on unstructured meshes. We have seen how starting from a complex and fuzzy definition of the problem we have slowly refined the specification and the requirements of this system, and how from a problem related to fluid dynamics we have arrived to a solution abstracted and related to general FPGA design and digital design. The core problem of unstructured meshes, that is to process a mesh where every vertex has a variable number of neighbouring vertices using FPGAs, requires the use of a decoder capable of handling at high speed the data frame describing a mesh like the one presented here.

The most common practice to accelerate algorithms using FPGAs is to regularize every information possible in order to increase the performance. If we take a look at the Header datapath, this is the approach taken with headers that have fixed lengths. This allows us to have a more simple and straight forward design with respect to the packet's datapath. The headers are stored inside the Header Decoder, and then they are directly processed without any further operations, and their optimization inside the frame is granted by their encoding. On the other hand, the packets we are decoding require this whole module to make their compression and optimization inside the frame viable. The fact that packets are not fixed in length is the core issue when optimizing algorithms that use unstructured meshes using FPGAs, but with this thesis we have shown that this problem can be addressed and can be proven to be effective and high performing.

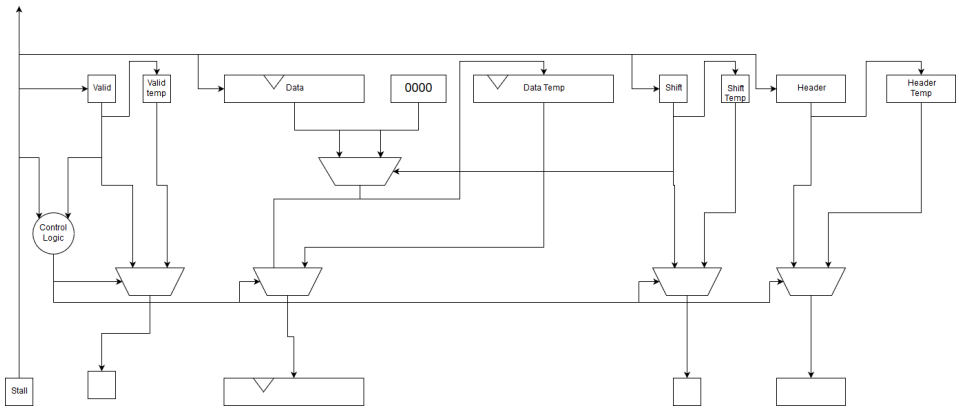
6.2 Further improvements

In Chapter 5 we have presented the performance report of the timing analysis. This analysis has shown that the critical path goes through the stall line, as it is

the only line that cannot be pipelinend as it goes in the opposite direction with respect to the dataflow. This problem gets critical when using deep pipelining, such as in our case.

There is, however, a solution to this apparently insolvable problem. The issue when pipelining the stall line is that we inherently loose a whole line of data held inside a pipeline stage when the stall happens, as all the previous sections of the design will continue to work for that clock cycle wasted to pipeline the stall line, and thus exactly one complete line of data is lost for each pipeline stage we add to the stall line when a stall occurs. The way to go to address this problem is therefore to store all the information of a pipeline stage inside a cache line whenever we want to add a pipeline stage to the stall line. In this way, when a stall occurs the flow of data will still continue for that one clock cycle, but our cache line will grant that the data previously computed aren't lost. Figure 6.1 shows the implementation of this concept in the Packet Shifter. We see that for that exact stage there is an added multiplexer for every incoming data. This multiplexer should be controlled by a specific bit that depends on the stall line and on the valid line.

Figure 6.1. Pipeline stage for the stall line of a rigid Packet Shifter



The drawback of this approach is that we add a complete line of registers to the design, and this may prove to be expensive if we want to add several pipeline stages to the stall line. This means that the stall line shouldn't be pipelined too much, otherwise the design would become too big. The number of pipeline stages of the design should be larger than the number of pipeline stages of the stall line, or it would be uselessly redundant. Also it is more clever to place this stage in places of the pipeline where the registers are small; in this design would be clever to put

them in the Instruction Computer, or inside the Packet Builder. Furthermore, the addition of a multiplexer at the end of a stage may decrease the performance of the component locally.

Apart from improving the design of the stall line to support a design with many pipeline stages and achieve even higher frequencies, the next phases of this project would be to actually implement it inside the FPGA and to test it inside the complete CFD design. For this, it would be necessary to include the implementation of the actual CFD algorithm. Although this design is being performed in the LSI research group and it is in its final stages, it is still not yet available. Therefore, the testing of the complete system will be finished in the near future.

Appendix A

Demonstration of modulus operation properties

A.1 Demonstration of modulus property 1

This demonstration gives the proofs of the equivalence of Equations 4.7 and 4.9. What we want to demonstrate is that

$$(A+B) \mod D = (A \mod D + B \mod D) \mod D \qquad \forall A, B \in \mathbb{N}; D \in \mathbb{N} - \{0\}$$

First, by the definition of the module operation [6]

$$A \mod D = A - \lfloor \frac{A}{D} \rfloor * D \tag{A.1}$$

The equation becomes

$$(A+B) \mod D = A + B - \lfloor \frac{A+B}{D} \rfloor * D \tag{A.2}$$

Since the division has the distributive property

$$(A+B) \mod D = A + B - \lfloor \frac{A}{D} + \frac{B}{D} \rfloor * D$$

A and B can be rewritten like in Equation A.1

$$(A+B) \mod D = A \mod D + \lfloor \frac{A}{D} \rfloor * D + B \mod D + \lfloor \frac{B}{D} \rfloor * D - \lfloor \frac{A}{D} + \frac{B}{D} \rfloor * D \tag{A.3}$$

The term

$$\frac{A}{D}$$

can be rewritten as the next formula for the definition of module In Equation A.1

$$\lfloor\frac{A}{D}\rfloor + \frac{A \mod D}{D}$$

Thus the term

$$\lfloor\frac{A}{D} + \frac{B}{D}\rfloor * D$$

becomes

$$\lfloor\lfloor\frac{A}{D}\rfloor + \frac{A \mod D}{D} + \lfloor\frac{B}{D}\rfloor + \frac{B \mod D}{D}\rfloor * D$$

since the approximation of a number already approximated is redundant, the previous term is equal to

$$\lfloor\frac{A}{D}\rfloor * D + \lfloor\frac{B}{D}\rfloor * D + \lfloor\frac{A \mod D}{D} + \frac{B \mod D}{D}\rfloor * D$$

Eqation A.3 thus becomes

$$A \mod D + \lfloor\frac{A}{D}\rfloor * D + B \mod D + \lfloor\frac{B}{D}\rfloor * D - (\lfloor\frac{A}{D}\rfloor * D + \lfloor\frac{B}{D}\rfloor * D + \lfloor\frac{A \mod D}{D} + \frac{B \mod D}{D}\rfloor * D)$$

and therefore

$$A \mod D + B \mod D - \lfloor\frac{A \mod D + B \mod D}{D}\rfloor * D$$

that is equal to Equation A.2 with $A \mod D$ as the first term (A) and $B \mod D$ as the second term (B). By substitution, the previous operation is equal to

$$(A \mod D + B \mod D) \mod D$$

Thus the thesis is demonstrated

A.2 Demonstration of modulus property 2

The second demonstration we want to proof the inequality

$$0 \leq A \mod D < D \quad \forall A \in \mathbb{N}, D \in \mathbb{N} - \{0\} \tag{A.4}$$

First we start from the properties of approximation by defect and excess [6]:

$$\lceil A \rceil \geq A \geq \lfloor A \rfloor \ \forall A \in \mathbb{R}+ \tag{A.5}$$

We have asserted that for the properties of the approximations that

$$\lceil A \rceil - \lfloor A \rfloor = 1 \ \forall A \in \mathbb{R}+ - \mathbb{N} \tag{A.6}$$

and that

$$\lceil A \rceil - \lfloor A \rfloor = 0 \ \forall A \in \mathbb{N} \tag{A.7}$$

Here there are two properties of the division that will be used later

$$\frac{A}{D} \in \mathbb{R}+ - \mathbb{N} \ \forall A \neq kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N}$$

$$\frac{A}{D} \in \mathbb{N} \ \forall A = kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N}$$

Therefore applying this to Equation A.5

$$\lceil \frac{A}{D} \rceil > \frac{A}{D} > \lfloor \frac{A}{D} \rfloor \ \forall A \neq kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N} \tag{A.8}$$

and

$$\lceil \frac{A}{D} \rceil = \frac{A}{D} = \lfloor \frac{A}{D} \rfloor \ \forall A = kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N} \tag{A.9}$$

We can transform the Equation A.9 in a similar way we did to the Equation A.1 : we see that

$$\frac{A}{D} - \lfloor \frac{X}{D} \rfloor = 0 \tag{A.10}$$

and thus

$$A \mod D = 0 \ \forall A = kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N} \tag{A.11}$$

In all the other cases the Expression A.8 is valid. This means that

$$\lceil \frac{A}{D} \rceil > \frac{A}{D} \ \forall A \neq kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N}$$

applying Equation A.6 we get

$$\lfloor \frac{A}{D} \rfloor + 1 > \frac{A}{D}$$

thus

$$\lfloor \frac{A}{D} \rfloor - \frac{A}{D} > -1$$

thus

$$\frac{A}{D} - \lfloor \frac{A}{D} \rfloor < 1$$

and finally

$$A - \lfloor \frac{A}{D} \rfloor * D < D$$

That from the definition of the reminder (Equation A.1) operation means

$$A \mod D < D \; \forall A \neq kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N} \tag{A.12}$$

The same procedure can be applied to the remaining part of Equation A.8

$$\frac{A}{D} > \lfloor \frac{A}{D} \rfloor \; \forall A \neq kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N}$$

This means that

$$A - \lfloor \frac{A}{D} \rfloor > 0$$

and thus

$$A - \lfloor \frac{A}{D} \rfloor * D > 0$$

From the definition of reminder (Equation A.1) this becomes

$$A \mod D > 0 \; \forall A \neq kD; A \in \mathbb{R}+; D \in \mathbb{R}+ - \{0\}; k \in \mathbb{N} \tag{A.13}$$

By adding to the dominion all the cases when A is equal to kD we get according to Equations A.11, A.12 and A.13 we get

$$0 \leq A \mod D < D \; \forall A \in \mathbb{R}+, D \in \mathbb{R}+ - \{0\} \tag{A.14}$$

Since the natural dominion of numbers is a subset of the positive real dominion of numbers, the property is demonstrated.

A.3 Demonstration of modulus property 3

This section includes demonstration of the property of the module shown in section 4.3.3, which is:

$$\begin{gathered} A \mod D = A - kD \\ if \; kD \leq A < (k+1)D \\ \forall A \in \mathbb{N}, D \in \mathbb{N} - \{0\}, k \in \mathbb{N} \end{gathered}$$

First of all let's set once again the definition of the module operation, as already done in Section A.1

$$A \mod D = A - \lfloor \frac{A}{D} \rfloor * D \tag{A.15}$$

To this definition, we will add and delete from the dividend of the division the term kD, with both k and D belonging to the natural numbers

$$A \mod D = A - \lfloor \frac{kD + A - kD}{D} \rfloor * D \tag{A.16}$$

Since k is a natural number, the approximation by defect of k is equal to k itself. This wouldn't be the case if k was a rational number and in fact this property only applies if k is an integer number, which it is by definition. So the Equation A.16 becomes

$$A \mod D = A - kD - \lfloor \frac{A - kD}{D} \rfloor * D \tag{A.17}$$

From the definition in the hypothesis A is inside the range shown below

$$kD \leq A < (k+1)D \tag{A.18}$$

If we subtract kD from the first and the second unequation the $A - kD$ term is inside the range shown below

$$0 \leq A - kD < D \tag{A.19}$$

This means that the division of this term by D is in the range below, since B is a natural number greater than 0

$$0 \leq \frac{A - kD}{D} < 1 \tag{A.20}$$

And since this operation is always between 0 and 1, the approximation by defect of the above division is going to be always equal to 0.

$$\lfloor \frac{A - kD}{D} \rfloor = 0 \tag{A.21}$$

The division term of Equation A.17 can be deleted, and the remaining part of Equation A.17 is

$$A \mod D = A - kD \tag{A.22}$$

And thus the property is demonstrated. This property only applies if k is a natural number and if A is inside the range in Equation A.18 , which it is in our case.

Appendix B

Testbench results

B.1 Input and output frames for 44 bit memory port width testbench

In this section are reported the results of the RTL simulation for the testbench of the 44-bit port design defined in Chapter 5. The first frame represents the input frame stored inside the external memory before the execution of the Packet Decoder. The input frame is composed by header lines and payload lines, which are distinguished by the last bit of every line: 0 means a header line, 1 means a payload line. The extra bit at the end of the packet makes the port width of 45 bits. However, for the way the Packet Decoder is made it is more important to point out the width of the line with useful information, which is 44 bits. Headers are encoded and have a fixed length of 5 bits. Packets are readable: the first bits of every packet are equal to three ones, followed by a sequence of zeros, and it ends with a sequence of a one and a zero or with a sequence of two ones and two zeros. The packet that starts at the beginning of a payload line is at the 17th line.

00001010100110100111000001111011010010100010 0
11100000000000000000000000000000000000000000 1
00000000000000000000000010111000000000000011 0
01110000000000000000000000000000000000000001 0
00010101001101101110010001110100011000100011 0
00000000000000000000000000000000000000000001 0
00000000000001011100000000000000101110000001 0
00000000000000000000000010111001011100000001 0
00000000000000000000000000000000000000101111 0

00001011100000000000000000000000000000000001
00000000000000000000000000000000000000000001
00000000000000000000000000000000011001110001
00000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000
00000000000000110011100000000000000000000001
00000000000000000000000000000000000000011001
11100000000000000000000000000000000000000001
00000000000000000000000000000011001110011
11000000000000000000000000000000000000000001
00000000000000000000000000000000000000000001
00000000011001110000000000000000000000000001
00000000000000000000000000000000000000000001
00000000000000000000000000000000000000000001

Here is reported the output frame of the 44-bit port testbench. At the beginning of each packet is reported the length of each packet in the form of a decimal number. Then there is the output line which is spread over 194 bits. We see that every packet starts with the sequence of three ones, and thus each packet is decoded correctly.

71
11100000000000000000000000000000000000000000000000000000000000000000010111
00000000000001000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
18
11100000000000010111000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
102
11100000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000010111000000000000010111000000000000000000000000000
00000000000000000000000000000000000000000000
20
11100000000000000101110000000000000000000000000000000000000000000000000000
00000000000010111000000000000010111000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
35
11100000000000000000000000000010111001011100000000000000000000010111000

00000000000010111000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
7
11100101110000000000000000000000000000001011100101110000000000000000000000
01011100000000000000010111000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
52
11100000000000000000000000000000000000000000000010111000000000000101110
00000000000000101110000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
9
11100001011100000000000000000000000000000000000000000000010111000000000
00000010111000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
120
11100000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000001100111000000000000000000000000000000000
00000000000000000000000000000000000000000000
69
11100000000000000000000000000000000000000000000000000000000000000000110011100
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
69
11100000000000000000000000000000000000000000000000000000000000000110000000
00000000001100111000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
82
11100000000000000000000000000000000000000000000000000000000000000000000000000000
00001100111001000000000000001100111000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
7
11100100000000000000000000000000000001100111001000000000000000110011100
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
0
10000000000000000000000000000000001100111001000000000000001100111000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
102
11100000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000001100111000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000

101
11100000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000011001110000000000000
0000000000000000000000000000000000000000000000
0
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000110011100000000000000000000000000
0000000000000000000000000000000000000000000000
0
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000110011100000000000000000000000000
0000000000000000000000000000000000000000000000
0
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000110011100000000000000000000000000
0000000000000000000000000000000000000000000000
0
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000110011100000000000000000000000000
0000000000000000000000000000000000000000000000

B.2 Input and output frames for 128 bit memory port width testbench

Here is presented the input and output frames of the 128-bit port testbench. The packet format is the same of the previous testbench and the last bit determines the difference between payload lines and header lines. In this case, the last bit makes the port width be equal to 129 bits, but as said previously the behaviour of the Packet Decoder depends on the width of the line with only useful bits (payload and header related bits), and thus 128 bits are considered.

00000000000000000000000000000000000000000000000000111000110111100001001001
0000001111000110000100000111111000001100011100011000010
11100000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000011001110000000000000000000000000000000000
0000000001100111000000000000000000000000000000000000001

00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000011001110000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000011001110000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000110011100000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000011001111011000111001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000001100111001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000011001110000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000001100111000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001

00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000001

Here is presented the output of the 128-bit port testbench. The output port is equal to 420 bits. At the beginning of each packet there is a number defining the length of each decoded packet. All the packets begin with a sequence of three ones, and thus the packets are all correctly decoded.

293
11100000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000001100111
00000000000000000000000000000000000000001100111000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
50
11100000000000000000000000000000000000000000001100111000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000001100111000000000000000000000000000000000000000
00000011001110000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
256
11100000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000001100111000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
144
111000000000000000000000000000000000000000000000000000000000000000000000000000000000

00000000000000000000000000000000000000000000000000000000000000000000000000000000000011001110
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000110011100000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
0
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
162
11100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000001100111000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
180
11100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000110011100000000000000000000000000000000000000000000000000000
00000000000000000000000000000000110011100000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
180
11100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000110011110110001110000000110011100000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
10
11110110001110000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
11001110000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
256
11100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000011001110000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
180
1110000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000011001110000000000000011001110000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
21
1110000000000000011001110000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000011001110000000000000011001110000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
293
1110000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000001100111
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
162
1110000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001100111000000000000000000000000000000000000000000000000000011001110000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
50
1110000000000000000000000000000000000000000000000000000011001110000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000011001110000000000000000000000000000000000000000000000000
0000011001110000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
256
1110000000000000000000000000000000000000000000000000000000000000000000000000000000000000

00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0
11100000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0
11100000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0
11100000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0
11100000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000

B.3 Configuration file of the 44 bits port width testbench

In this section is presented the configuration file of the 44 bit testbench. Each number is written as an hexadecimal number. The first number represents the

memory port width, while all the other numbers are the possible packet lengths of the frame. In this case there are 19 possible lengths, the maximum packet length is 151 bits and the memory port width is 44 bits.

2C 0 58 47 78 5 12 34 51 23 65 97 29 66 52 14 45 7 8 9

B.4 Configuration file of the 128 bits port width testbench

In this section is reported the configuration file for the 128 bit testbench. As the previous configuration file the numbers are represented as hexadecimal numbers, the first number represents the memory port width, and the remaining number represent the possible packet lengths. The memory port width is 128, the maximum packet length is 293 bits, and there are 19 possible lengths.

80 0 125 46 32 15 7B 90 100 DE 1A 88 81 7D 12 33 A2 B4 A D

Bibliography

[1] C. Carreras, J.A. Lopez, R. Sierra, R. Jevtic, P. Barrio, E. Sedano and J.A. Fernandez , "Performance Evaluation of 2D-Euler in the Optimized Hardware Platform and Conclusions. Update on the Design Methodology and Tools ", Deliverable 4, project DOVRES/FUSIM-E, December 2010

[2] F. Manso Rodriguez, " Design of FPGA interfaces for DDR Memory and PCI-Express D", Trabajo fin de grado, Universidad Politécnica de Madrid, 2018.

[3] P. Barrio, C. Carreras J. A. Lopez, O. Robles, R. Jevtic, R. Sierra, in "Memory Optimization in FPGA-accelerated scientific codes based on nstructured meshes", Journal of Systems Architecture, vol. 6, issue 7, pp 579-591, Elsevier, June 2014.

[4] K. K. Parhi, "VLSI Signal Processing Systems, Design and implementation", John Wiley and Sons, December 1999.

[5] Sun Microsystems, Inc., "OpenSPARC T2 Core Microarchitecture Specification", URL https://www.oracle.com/technetwork/systems/opensparc/t2-06-opensparct2-core-microarch-1537749.html, December 2007.

[6] D. Knuth, "The art of Computer Programming, 3rd edition", Addison-Wesley, 1999.

[7] Intel®, "Stratix 10 GX/SX Device Overview", URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf, August 2018.

[8] GiDEL PROCstar III, https://www.gidel.com/PROCstar%20III. html

[9] Intel®, Quartus 16.2 User Guide, URL https://www.intel.com/content/www/us/en/programmable/documentation/sbv1513989262284.html

[10] Mentor Grafics, ModelSim® Userâs Manual ,URL https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user

[11] Pyhton, 3.6.5 release, https://www.python.org/downloads/release/python-365/