High Level Synthesis for Packet Processing Pipelines

Cristian Soviani

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2007

ABSTRACT

High Level Synthesis for Packet Processing Pipelines

Cristian Soviani

Packet processing is an essential function of state-of-the-art network routers and switches. Implementing packet processors in pipelined architectures is a well-known, established technique, albeit different approaches have been proposed.

The design of packet processing pipelines is a delicate trade-off between the desire for abstract specifications, short development time, and design maintainability on one hand and very aggressive performance requirements on the other.

This thesis proposes a coherent design flow for packet processing pipelines. Like the design process itself, I start by introducing a novel domain-specific language that provides a high-level specification of the pipeline. Next, I address synthesizing this model and calculating its worst-case throughput. Finally, I address some specific circuit optimization issues.

I claim, based on experimental results, that my proposed technique can dramatically improve the design process of these pipelines, while the resulting performance matches the expectations of hand-crafted design.

The considered pipelines exhibit a pseudo-linear topology, which can be too restrictive in the general case. However, especially due to its high performance, such an architecture may be suitable for applications outside packet processing, in which case some of my proposed techniques could be easily adapted.

Since I ran my experiments on FPGAs, this work has an inherent bias towards that technology; however, most results are technology-independent.

Contents

List of Figures

List of Tables

Acknowledgements

I thank my advisor prof. Stephen A. Edwards for his constant guidance and help during my entire graduate school. Not only was his liberal advice essential for my academic progress but he actively contributed to the research that led to this thesis. Doubtlessly, most of the material presented below as original work was published before at several academic venues; with no exception Stephen has co-authored these publications. But scarcely was Stephen's help strictly academic: he is my friend.

I thank Dr. Ilija Hadzic, my mentor during my two Bell Labs internships. The research started together during the summer of 2005 represents the keystone of this thesis. Without his suggestions and comments, based on a vast engineering experience, my research could have easily strayed away from this physical world.

I thank Dr. Olivier Tardieu for his active contribution to my research. His touch can be easily recognized in the last technical chapter, where a very rigorous mathematical formulation was required.

I thank prof. Luca Carloni for his insightful suggestions and comments on some of the most delicate and convoluted topics of my research.

I thank prof. Jonathan Gross for teaching the most interesting two courses I have ever taken. His teaching is divinely inspired.

I thank the members of my defense committee for their interest in my modest work and for the unforgettable moments they have offered me during my defense.

I thank all faculty of our department for their inestimable advice and help.

Last but not least I wish to thank my family for the irreplaceable and unconditional support during the last thirty-three years of my life.

To Lizi and Valona

Chapter 1

Introduction

1.1 Packet processing inside a Switch

A packet switch (or router) is a basic building block of data communication networks. Its primary role is to forward packets based on their content, specifically header data at the beginning of the packets. As part of this forwarding operation, packets are classified, queued, modified, transmitted, or dropped.

Figure 1.1 shows the simplified architecture of a generic packet switch. Its central

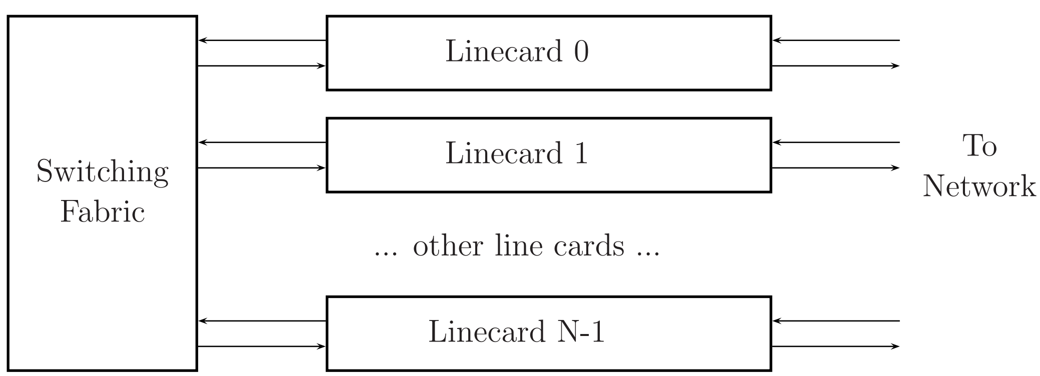


Figure 1.1: Generic switch architecture

component is a switching fabric, which performs the actual switching operation among its N ports. In addition, the switch contains N linecards, one for each fabric port. As can be seen, they are the interface between the switching fabric and the physical network. These linecards are responsible for a large class of operations, such as packet classification, next destination computation, metering, and traffic management.

Packet processing is a very important task for a linecard. According to the network protocol(s) supported by the switch, some header fields of the packets have to be updated, such as decrementing the TTL (Time to Live) count in the IP header (Internet protocol), inserted or removed, such as the MPLS (Multiprotocol Label Switching) labels used by the Martini encapsulation protocol. In addition, some packets have to be dropped (e.g., an IP with a zero TTL value) or forwarded internally to a dedicated component inside the switch, e.g., discovery PPPOE (Point-to-Point over Ethernet) packets. These operations are very common, but this list is far from being exhaustive: the important issue to be noticed is that each network protocol requires its own set of such packet transformations.

The actual operations that can be performed by the packet processor define the overall switch functionality. The other switch components, such as the switching fabric, perform generic tasks. It is mainly the packet processor that makes the difference between a simple Ethernet switch and a more sophisticated one that also supports VLAN (Virtual Local Area Network) and PPPOE traffic.

Figure 1.2 shows a more detailed view of a typical linecard. It consists of an ingress path and an egress one, which handle the incoming and the outgoing traffic. Each path contains a packet processor, which performs the abovementioned operations.

As mentioned earlier, the packet processors encapsulate the switch's knowledge of network protocols. Consequently, the complexity of these processors is high, and tends to increase with the deployment of new protocols and multi-protocol network devices. Moreover, these components are the first to require changes due to protocol updates, additional functionality required, or simply because of bugs discovered in the

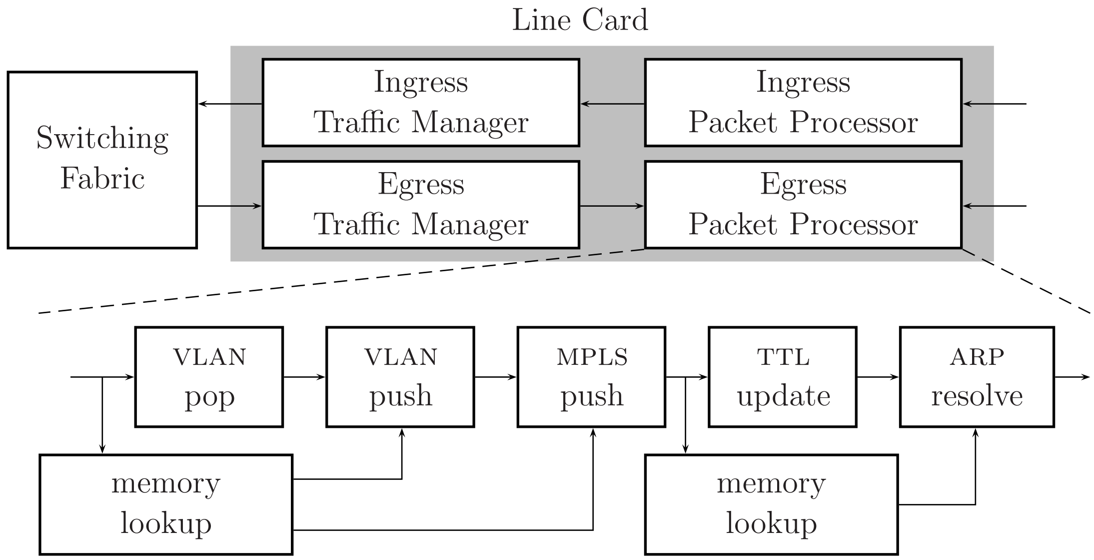


Figure 1.2: A switch with detail of a packet processing pipeline.

current implementation. It is critical that these changes be done in the field to avoid the physical replacement of the existing product with a new one, and preferably without interrupting the network operation for a long time. The above issues are known as maintainability.

Traditionally, packet processing was done in software, either by general purpose CPUs, or, for increased performance, by specialized NPUs (Network Processor Units). As expected, this approach successfully addresses the maintainability issue. Although the solution is still feasible for low-end network devices, high-end switches, such as those found on network backbones or in data centers, operate today as speeds hard to imagine two decades ago.

In the spring of 2007, it was not uncommon to find linecards working at 40 Gbps. Since the minimum Ethernet packet size is 64 bytes, these cards must work at the incredible rate of 80,000,000 packets per second: each packet has a header that must be handled by the packet processor.

Implementing the packet processors in hardware becomes a very attractive choice, if not the only reasonable one, for any modern high-end networking device.

The imminent drawback is that for classical technologies such as ASIC, the maintainability advantages offered by software are lost. Some vendors (Cisco, Juniper, etc.) build hard-wired ASICs with generic sequences for state machines to achieve field upgrade and maintenance capability, but this is limited.

Fortunately, with the recent growing popularity of FPGA (Field Programmable Gate Array) devices, a compromise solution has been found. Although slower than ASICs, they are much faster than software, and allow the same maintainability as software. Asics indeed requires less physical resources (in terms of transistors), but due to commercial issues (mainly the quantities in which they are manufactured) FPGAs are often more economical.

1.1.1 FPGA particularities

Although an overview of FPGA technology is outside the scope of this thesis, I will emphasize some of its particularities as they trigger important decisions in the rest of this work.

Due to its programmability, an FPGA uses far more silicon resources than an equivalent ASIC design. For example, a 4-input AND gate is usually mapped in a Xilinx Virtex 4 device to a 4-input LUT (Look-Up Table), which consists mainly of 16 SRAM (Static RAM) cells (used for configuration) and a 16-to-1 multiplexer.

Another particularity, impossible to ignore, is that the "wires" connecting the various logic elements are in fact complex circuits that include buffers and multiplexers.

Consequently, FPGAs are less efficient than ASICs in both area and delay. The difference is not minor: an FPGA is roughly ten times larger and ten times slower than the equivalent ASIC.

It follows that FPGA designs will work at lower clock frequencies (hundreds of MHz), and most operations have to be pipelined and performed on wider data words to be able to perform the same tasks. All FPGAs provide a disproportionately large number of registers relative to the number of combinational logic primitives. For example, all

FPGAs from the Xilinx Spartan and Virtex families provide a register for each four-input LUT. Consequently, adding several pipeline stages to a combinational path is inexpensive.

Applying pipelining and using wide data words leads to very complex sequential networks. Again, an example: if we are given four modules instead of one simple FEC (forward error correction) module, each with four pipeline stages, it is anything but trivial to speed up an existing application by a factor of 16. Assuming that it is possible, one may expect a complex sequential network around the modules, e.g., performing load balancing. To conclude, FPGAs usually require more complex designs than ASICs for solving the same problem.

However, the real advantage of FPGAs is their programmability. Much like software, an FPGA may be updated in a matter of seconds, even when the device is deployed in the field. Thus, not only is the development and testing process dramatically sped up (e.g., the physical boards may be manufactured before the design is complete), but the systems may be maintained dynamically with minimal overhead. This proves to be a crucial advantage for applications such as network devices, when new features and fixes have to be added on a regular basis.

In fact, the bottleneck in the update process is the design time itself. If adding a new feature to the design takes several weeks, the main advantage of this technology could not be exploited to its maximum potential.

There are two different, contradictory issues: FPGA designs require complex designs and design time is critical.

The whole point of my thesis is to balance these issues and the requirement of high performance in modern networking devices.

1.2 Packet processing pipelines

The algorithms executed by an ingress or egress packet processor in most switches are very suitable to a pipelined hardware implementation. Figure 1.2 shows an example: a (simplified) pipelined realization of an egress packet processor in a commercial switch.

The core of the packet processor consists of a linear pipeline. The logical flow may have forks and joins but they can be emulated with a linear physical topology. This packet processing pipeline is traversed by all the packets to be processed. Note that there are no forks and no joins. The main property of this linear architecture is that the order in which packets enter and exit the pipeline is exactly the same. For example, if a packet is to be dropped, it is simply marked as such: the actual dropping will be done at the exit of the pipeline. We will argue for the advantages of this simplified architecture in the next chapter.

As a general rule, the overall complex functionality of the pipeline is divided between several modules, each of them performing tasks of small or moderate complexity. These modules are known as packet header editors because they only modify packet headers, not their payloads. In the sequel, I will often refer to them using shorter names such as packet editors, editing modules, or simply modules.

The packet processing pipeline also contains several auxiliary elements, such as the two lookup modules in Figure 1.2. The main traffic does not not traverse them; instead they function as clients of the main pipeline. Such an element takes requests from a pipeline module, performs a specific operation, and returns the results to another downstream module.

Memory lookup elements are probably the most common type. For example, the one in the right of Figure 1.2 performs an ARP (Address Resolution Protocol) lookup: it takes the packet IP destination from the MPLSPUSH module and returns the Ethernet address of the destination interface to ARP resolve. The latter uses this result to update the relevant field of the packet header.

The communication between the pipeline elements, packet editors and auxiliary elements is realized by FIFOs. For simplicity, I represent them as arrows in Figure 1.2.

Since the packet editors can insert or remove header fields, the data flow rate through the pipeline is not constant. Such non-constant rates also arise because some modules or auxiliary elements take extra time to perform some computation.

Thus, a communication link between two components may be active and carry valid information in only certain clock cycles and not on others. Therefore, these FIFOs perform the essential task of compensating for the non-continuous operation of the various components.

1.3 Packet editing modules

As seen above, the packet editing modules are critical components of a packet processing pipeline. They are neither more nor less important than the other components (lookup tables, etc.) but they form the backbone of the pipeline.

While the other elements are either simple connecting elements, e.g., FIFOs or perform important but standard operations, e.g., memory look-up modules, the packet editors are the "active" elements that encapsulate the functionality of the pipeline, i.e., its ability to process various types of packets according to different network protocols.

From a high-level view, these modules perform simple transformations on each packet header. They perform simple arithmetic or logical operations, such as decrementing the IP TTL field, inserting or removing data, such as the MPLS labels, setting or reseting some flags, or any combination of those. I will describe the possible operations in Chapter 2. The overall functionality is split among various packet editors in the pipeline, both to keep each module simple and to enable the interaction with auxiliary elements.

To keep these critical modules simple, they are not allowed to perform arbitrary operations such as dropping packets or creating new ones. Even if these operations are required, they are usually done outside the packet processing pipeline for performance reasons. As a result, the behavior of such module looks simple: read packet, make some simple modifications, write packet, repeat.

However, these modules are not trivial when expressed at the register transfer level in a language such as Verilog or VHDL and do not look similar to each other. Even a simple operation such as adding two fields may have radically different implementations depending on where the fields are in the header, the speed of the add operation, etc.

The RTL design of such a module is not a trivial task; it takes a lot of time even for an experienced engineer, and is prone to errors; their high-level simple and uniform functionality disappears in the RTL domain, where even trivial modules require complex implementations.

1.4 Main challenges

Below, I summarize the main challenges in designing a high-performance packet processing pipeline. After this short summary, I will expand each topic, adding a brief description of my contribution in surmounting the respective issues. The next sections are the outline of my thesis.

First, I argue that the design flow for packet editing modules has to start from a high-level specification instead of RTL because the difficulty of RTL design and the big advantages of speeding up the design process. I introduce such a specification model: the PEG (Packet Editing Graph). I show how to translate a PEG specification into an RTL model such as VHDL.

Next, I argue for the importance of analyzing the worst case performance of a packet processing pipeline. I propose a fast method for this analysis. I also argue for the importance of computing the minimum sizes of the interconnecting FIFOs to guarantee the desired pipeline performance. I also present a model-checking-based algorithm to answer this issue.

Finally, I argue the real performance bottlenecks are the critical loops, which usually spread across the design in a complex sequential system such as one resulting from my synthesis procedure. To address this issue, I propose an algorithm based on a combination of Shannon decomposition and retiming.

1.5 Challenge: Avoiding RTL

The specific challenge in designing a packet processing pipeline is the design of its packet editing modules. This is because FIFOs are usually instantiated from an existing library or synthesized by a specialized tool, while auxiliary elements are usually designed using standard techniques, albeit not always simple. Once the pipeline components are available, putting them together according to the pipeline topology is straightforward.

Therefore, I focus mainly on the design of the packet editing modules throughout the rest of this thesis.

First, I identify several particularities of these modules. They are deeply pipelined circuits that operate on many bits in parallel (e.g., 64, 128 or 256 bits). This tendency becomes very visible in FPGA implementations because of their relative lack in performance compared to ASIC. Second, they interact with the adjacent components through FIFO interfaces. To achieve the desired performance, the handshaking signals require very careful handling. The arising problems have a good reputation of being "brain-twisters" for the design engineer. Third, the specifications of these modules are usually long documents written in plain English, designed by network protocol architects with moderate hardware design experience; traditionally, they have a software-centric view of the issue.

Consequently, these modules are tedious to code at the RTL (register transfer level) by using a hardware description language such as Verilog or VHDL: designing these modules is usually reserved to experienced engineers and takes a lot of time. Since high performance is needed, very aggressive design techniques are often employed. Bugs appear inherently in corner cases, are hard to fix and even harder to detect.

The critical drawback of RTL design is not a purely technical one. I have mentioned the importance of quick development and maintainability. The difficulty of RTL design tends to undermine the advantages brought on by the use of FPGA technology.

1.5.1 Contribution: high level packet editor description

The first contribution of this thesis is PEG, a new high-level DSL (domain specific language) for specifying the functionality of packet editing modules. Using this language, the designer can specify any practical module in a simple and intuitive manner, staying close to the top-level view of the problem that is familiar to the network protocol architect. Since in PEG all the low-level implementation details are abstracted, RTL design experience is unnecessary; the language encourages the designer to express the problem in terms of high-level functionality.

The proposed language does not have a hardware flavor and I also consider it a desirable goal to remove any software flavor as well. It could be easily argued that an abstract view of packet header editing is completely orthogonal with low-level RTL specifications, the latter being a means to physically accomplishing the former on a hardware target. I argue with the same confidence that a software program is also a mean of implementing the same task on a software target: the correct abstraction for packet editing has little in common with the RTL model or with the software sequential model.

The proposed abstract model can be efficiently translated into both software or hardware, depending on the chosen implementation of the packet processor. The hardware translation forms a significant part of this thesis.

The PEG language has a graphical flavor: the operands, computations, results, etc., are represented as nodes in a graph and the dependencies between them are modeled by the graph's edges. PEG is chosen to be general enough for describing all practical header editing operations and particular enough to allow the synthesis procedure generate efficient circuits by taking advantage of some domain-specific properties.

1.5.2 Contribution: packet editor synthesis

Starting from a packet editing module described in PEG, the following task consists in synthesizing a RTL model of the module. In particular, my existing implementation generates synthesizable VHDL code.

This RTL code has to be further synthesized for a specific target, by using a traditional low-level synthesis flow. My primarily considered target was FPGA: the generated RTL code is biased towards this technology in some degree. However, this bias is minor and the flow could easily be re-targeted towards different technologies such as ASICs.

To a first approximation, my synthesis steps resemble traditional high-level synthesis. However, the underlying architectural and computational model are totally different from those found in the traditional approach: the resemblance to classical high-level synthesis is helpful for understanding the general issues but proves shallow at the level of technical details.

In traditional high-level synthesis the RTL design looks like a simple CPU: various computational and memory elements are interconnected by buses, multiplexers, etc. Computational units have "random access" to all operands and destinations. Therefore, computation can be divided between different elements and different clock cycles in an a very flexible manner, given that the causality constraints are respected (i.e., an operand is never used before it is produced).

In this thesis, the architecture is a deep pipeline: data flows through all stages and is locally modified by each stage. Since stages are usually light-weight elements, correlated data (e.g., a packet header) may appear simultaneously in several stages or even in several modules.

Since data enters (and exits) each stage word by word, ordered as they appear in the packet header, not all operands are available exactly when needed: some buffers are sometimes needed to store useful data within a stage for further use and to store already computed results that have to be output later.

Ideally, data passes constantly through the pipeline and is modified on the fly. In practice, the modules use only some fraction of the clock cycles to read or write data: the pipeline works at lower-than-unit efficiency. Obtaining the best possible efficiency becomes the primary synthesis goal.

My experimental results show that packet editing modules synthesized by my proposed technique meet the performance and area requirement that is expected from a hand-written RTL design. I produce circuits targeting Xilinx Virtex4 FPGAs that can sustain 40 Gbps throughput on industrial examples, equal to state-of-the-art switches, and are vastly easier to write and maintain than classic RTL designs.

1.6 Challenge: Pipeline worst case performance

Performance-critical pipelines are generally built from a sequence of simple processing modules, connected by FIFOs. I focus on packet processing pipelines such as those found inside a networking device. Here, the processing modules are the packet header editors, which can be synthesized using my synthesis technique above. However, the issues described in this section are more general and can be applied to a broader class of pipelines, as well as my proposed solution.

Given a complex pipeline, an essential issue is to find the overall pipeline throughput. Ideally, all pipeline modules work at full efficiency. Since a valid data-word is transferred between any two stages every cycle, computing the throughput amounts to multiplying the pipeline clock frequency with its data width.

This ideal case does not occur in practice: real modules insert and remove data, or require several clock cycles for an internal computation.

For real pipelines such as those I encountered, actual throughputs of 30–70% of ideal are not uncommom. Finding out the real pipeline becomes a critical issue, since the ideal throughput can not be used as a realistic estimation, not even in the coarsest sense. A widely-used rule of a thumb estimates the actual throughput to be 50%. This number can be either too optimistic or too conservative. The latter case leads to over-design, the former to a low service quality (i.e., dropped packets).

A widely used performance analysis technique is simulation. The main drawback is that the modules' behavior and consequently the whole pipeline's behavior depends on the processed data itself. The natural response to this issue is to run extensive simulations with different data patterns. They are chosen to cover, as much as possible, all realistic data patterns. This approach is not very different from the well-known technique of estimating the performance of general purpose CPUs by running various benchmarks.

However, in contrast to a CPU, a packet processor is a hard real-time system. For this class of systems the worst case performance is generally more important than the average one. Most benchmarks are designed to cover all possible corner cases, since it is expected that the worst case behavior would correspond to a corner case. For example, a simple Ethernet linecard would be simulated using sequences of minimum size (64 bytes) and maximum size (1500 bytes) packets.

For a modern packet processor, which is more complex, the number of these corner cases is very large and their simulation requires extensive running time. Practically, the simulation is restricted to a limited number of patterns. Consequently, "the field tests always find the bugs in the simulation setup," i.e., often data patterns not considered in simulation occur in practice. The development task becomes an interactive process between the designer, field tester, and worse, the customer.

Naturally, one would be interested in covering all possible cases by using an analytical, surprise-free approach that provides a system performance guarantee.

A related issue is the required sizes of the FIFOs interconnecting the pipeline modules. Their size influences the overall pipeline performance because a full or an empty FIFO stalls the adjacent modules for one or several clock cycles. Large capacity FIFOs successfully decouple the pipeline modules but waste a lot of resources. One would like to know the minimum FIFO sizes for which the pipeline achieves its highest performance.

One approach is to size the FIFOs by using some rough heuristics and to refine the analysis by extensive simulation. This method has the same drawbacks mentioned above.

1.6.1 Contribution: pipeline performance analysis

Computing the worst case throughput of packet processing pipelines is a difficult task because of their complex sequential behavior.

I propose a technique to compute the maximum possible pipeline throughput assuming that the FIFOs are unbounded. First, I analyze each module in isolation: I define some performance metrics for these modules and I show how they can be computed. Second, I focus on the entire pipeline: I assemble the numbers already computed for each individual module to determine the pipeline's worst case throughput.

This proposed algorithm is very fast. As unbounded FIFOs can not be realized in practice—the considered pipelines are not realistic but the computed throughput is known to be achievable by using FIFOs with finite sizes, although the algorithm returns no actual numbers for these sizes.

From the system designer's point of view, the same pipeline functionality can be achieved by using more or fewer modules, by dividing the task between modules in different ways, or by choosing different implementations for each module. Different functionally equivalent designs can be manually or automatically generated and a performance analysis step is required in every case. My proposed algorithm can be run inside this design exploration loop because of its small running time. Therefore, the algorithm can help indirectly the design and synthesis steps.

1.6.2 Computing minimum FIFO sizes

FIFOs consume a lot of resources: their area sometimes exceeds that consumed by the processing modules themselves. Additionally, large FIFOs require specific implementations such as instantiating block-memory primitives on a FPGA. This decreases the maximum clock period. Since reducing the FIFO sizes helps the design in terms of both area and performance, I treat FIFO sizing as an important topic of my thesis.

I compute the minimum FIFO sizes required by the pipeline for achieving the ideal throughput computed for the case when the FIFOs are unbounded. I propose two algorithms based on model checking: one exact and one heuristic. The heuristic algorithm is faster. This is an advantage for big pipelines where the running time grows rapidly with the design complexity.

Experimental results suggest that my algorithms are applicable to pipelines of at least five modules, for which an optimal or very good solution is found in minutes or less. I consider them practical since they are not invoked repeatedly during a normal design process but only once.

1.7 Challenge: Sequential RTL issues

Retiming is a well-known sequential optimization technique. It changes the positions of the registers inside a sequential network by moving them forward or backward. The system functionality is preserved but the resulting performance may be better than the original one.

Although the optimization potential of retiming exists for any sequential network, it becomes immediately visible if a linear pipeline is considered. If each stage takes a certain amount of time for computation, the maximum clock period will be given by the slower stage. Equalizing the time taken by the pipeline stages by changing the frontiers between them (i.e., the position of the registers) leads to the best outcome that can be achieved by retiming alone: the best performance is the ratio between the total computational time and the number of pipeline stages.

For a high-performance pipeline such as the packet processors considered in this work, retiming is a natural transformation in the optimization step. The RTL designer usually does not spend much time hand-placing the registers carefully but leaves the task to the optimization tool since at the moment of writing the RTL code he does not know exactly the time taken by each pipeline stage.

It can not be overemphasized that the maximum clock period achievable *before* retiming is irrelevant, while the relevant merit figure is the clock period *after* retiming. For example, to correctly compare two equivalent circuits, one has to retime them first: only after retiming is comparing the two clock periods meaningful.

The bottleneck which affects the performance of a high-speed pipeline is generated by the feedback loops. Generally, data flows forward through the pipeline but control and status signals often travel in the reverse direction. Consequently, the pipeline is no longer a simple loop-free sequential network but a complex one.

The performance of a simple linear pipeline may be increased by adding several extra stages and retiming the resulting pipeline. This is not possible if feedback loops are present. Retiming can only move registers, so the number of registers on any loop remains constant: therefore it is not possible to add extra registers inside a loop without altering the network functionality. Retiming remains a powerful technique but it has to be combined with other techniques that also modify the combinational logic.

To summarize, if for a combinational network the optimization priority is to reduce the length of the critical paths, for a high performance sequential network the goal is the reduce the length of the critical feedback loops. This task is more difficult in the sequential case.

1.7.1 Contribution: combining Retiming and Shannon

Optimizing sequential cycles is essential for many types of high-performance circuits such as pipelines for packet processing. Retiming is a powerful technique for speeding up the pipelines but it is stymied by tight sequential cycles.

The class of techniques that addresses this issue is known as retiming and resynthesis. Theoretically, R&R performs arbitrary logic optimizations plus retiming; it can be shown that this technique can transform a circuit into any equivalent circuit.

Exploring all possible transformations is not practically possible. Real algorithms usually choose some generic transformations and run them iteratively, in a heuristic manner.

Shannon decomposition is a generic transformation that may reduce the length of a critical cycle. It can be seen as a form of speculation. Designers seldom attack such critical cycles by manually combining Shannon decomposition with retiming, since such a manual approach is slow and error-prone.

It can be seen that combining retiming with Shannon decomposition is a special case of the more general R&R technique.

I propose a sequential optimization algorithm that restricts the class of allowable transformations to Shannon decomposition and retiming. The algorithm performs a systematic design space exploration: it finds exactly the best performance circuit that can be obtained by applying these two transformations.

While the algorithm is only able to improve certain circuits (roughly half of the benchmarks I tried), the performance increase can be dramatic (7%–61%) with only a modest increase in area (1%–12%). The algorithm is also fast, making it a practical addition to a synthesis flow.

1.8 Contribution summary. Thesis outline

In this thesis I propose a complete design flow for specifying, synthesizing, and analyzing packet processing pipelines, which are performance critical components of a modern network device such as a packet switch.

I propose PEG, a high level domain specific language for specifying the active pipeline elements, the packet header editors. Next, I propose a synthesis technique for transforming PEG into actual RTL code, which can be later mapped on a specific target such as an FPGA.

I claim, based on experiments, that the proposed synthesis flow is practical: using the presented methodology, entire packet processing pipelines belonging to industrial grade networking devices were successfully modeled and synthesized.

The proposed language, as well as the synthesis technique, are described in Chapter 2 of this thesis. Part of the material was published at the Design Automation Conference in 2006 [SHE06].

I introduce an analytic approach for the analysis of packet processing pipelines that offers a worst-case performance guarantee. As a related topic, I address the issue of computing the minimum FIFO sizes in a packet processing pipeline.

The performance analysis and FIFO sizing techniques are detailed in Chapter 3 of this work. Results were published at the International Workshop on Logic and Synthesis in 2007 [SE07].

Finally, I present a low-level sequential optimization algorithm, based on the combination of Shannon decomposition and retiming, which optimizes a certain class of sequential networks such as the high performance packet processing pipelines.

The algorithm is presented in detail in Chapter 4. I published a condensed description at Design, Automation, and Test in Europe in 2006 [STE06] as well as a more complete version in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems in 2007 [STE07].

In the last chapter I summarize the results of the present thesis.

Chapter 2

Synthesis of Packet Processing Pipelines

2.1 Overview

Packet processing pipelines are fundamental blocks of many data communication systems such as switches and routers. For example, the linecard in Figure 1.2 contains two such pipelines: an ingress and an egress one, which handle the incoming and outgoing traffic, respectively.

The core of such a structure is a linear pipeline containing several chained modules: the packets flow from left to right and are transformed on the fly. The function of each module and their order connection give the overall pipeline functionality.

Each module performs some specific transformation of the packet headers. For example, the TTLUPDATE module decrements the TTL field of each header and marks the packet to be dropped if the value reaches zero. The operation performed by a module consists in modifying, adding, and/or removing various fields of the packet header and is hence called packet header editing.

The sequential network implementing such a packet header editor is complex. I have encountered over a dozen of real modules during my experiments (Section 2.7) and I concluded that an efficient RTL implementation is very convoluted even for modules performing simple operations. A high-level approach for synthesizing these modules brings a lot of advantages compared to manual RTL design (Section 1.5).

However, generic high-level approaches such as synthesizing C-like behavioral models give poor results. It is sufficient to notice that real projects still use traditional RTL design even if behavioral synthesis has been proposed for several decades. The main drawback of a generic method is the low performance obtained after synthesis. In contrast, a custom method can take advantage of the specific particularities of a small class of problems and give good results.

In this chapter, I propose a high-level model, PEG, that can specify the functionality of a given header editing module. I also propose an automated high-level synthesis algorithm that starts from this model and translates it into synthesizable RTL code. This algorithm takes advantage of the specific properties of a header editing module and generates a result comparable to a manual RTL implementation.

The drawback is that my approach has a limited scope. However, a real network device contains several packet editing modules, so my technique covers an important share of the entire design. Moreover, these modules are the components most likely to be altered when specifications are changed. Therefore, it is the synthesis of these modules that gets the most benefits from a high-level approach.

It may be argued that it is desirable to have a uniform high-level description of the entire system. This is not realistic today. It is unlikely that critical components such as memories, transceivers, DDR interfaces, etc., could be synthesized by a generic technique. Practically, each of these has its particularities and is designed manually or using a specialized tool. This observation shows the place of my synthesis technique in the overall picture.

2.2 Packet Processors and Editing

Figure 1.2 shows the block diagram of a packet switch consisting of line cards (we only show one) connected to a switching fabric that transfers packets. I focus on designing the line cards, which provide network interfaces, make forwarding and scheduling decisions, and, most critically, modify packets according to their contents.

My synthesis technique builds components in the ingress and egress packet processors. A packet processor is a functional block that transforms a stream of input packets into output packets. These transformations consist of adding, removing, and modifying fields in the packet header. In addition to headers defined by network protocols, the switch may add its own control headers for internal use.

Packet processors perform complex tasks through a linear composition of simpler functions. This model has been used for software implementations on hosts [OP92] and on switches [KMC+00]. Another proposed architecture for hardware is a pool of task-specific threads that process the same packet in parallel without moving the packet [BJRKK04].

I use a unidirectional, linear pipeline model that simplifies the implementation without introducing major limitations. For example, Kohler's IP router [KMC+00] uses loops only to handle exceptions. I would do this with a separate control processor.

While the logical flow can fork and join, I implement only linear pipelines that can use flags to emulate such behavior. Non-linear pipelines are more complicated and would not improve throughput. If a packet needs to be dropped or forwarded to the control processor, I set flags in the control header and perform the action at the end of the pipeline. This guarantees every processing element sees all the packets in the same order; packet reordering is usually done in a traffic manager, a topic beyond the scope of this thesis.

Figure 1.2 shows a packet processing pipeline that edits the Virtual Local Area Network (VLAN) tag of an Ethernet packet [IEE03] and adds a Multi Protocol Label Switching (MPLS) label [RVC01], based on unique flow identification (FlowID). I assume a previous stage has performed flow classification and pre-appended a control header with the FlowID.

Both the VLAN push and MPLSPUSH modules insert additional headers after the Ethernet header, while the Time-To-Live (TTL) update and Address Resolution Protocol (ARP) resolution modules only modify existing packet fields. The VLAN pop module removes a header field. While this pipeline is simple, real switches just perform more such operations, not more complicated ones.

Thus, packet processing amounts to adding, removing, and modifying fields. Even the flow classification stage, which often involves a complex search operation, ultimately produces a modified header. I refer to these operations as *packet editing*; it is the fundamental building block of a packet processor.

In addition to the main pipeline, Figure 1.2 shows two memory lookup blocks. These blocks store descriptors that define how to edit the headers (e.g., how many MPLS labels to add). Here, the FlowID is an index into descriptor memory. A memory lookup module is any component that takes selected fields and produces data for a downstream processing element (e.g., IP address search, present in all IP routers, is a form of generalized memory lookup). Flow classification is thus packet editing with memory lookups.

Modules that use memory lookup assume a previous pipeline stage issued the request, which is processed in parallel to hide memory latency. I do not synthesize memory lookup blocks, but can generate requests and consume results. Because our pipelines preserve packet order, simple FIFOs suffice for memory interfaces.

Hence, I model packet processors as linear pipelines whose elements have four types of ports: input from the previous stage, output to the next, requests to memory, and memory results. Each processing element must be capable of editing packets based on their content and data from memory.

2.3 Related work

Kohler et al. [KMC+00] propose the CLICK domain-specific language for network applications. It organizes processing elements in a directed dataflow graph. CLICK specifies the interface between elements to facilitate their assembly. Although originally for software, Kulkarni et al. [KBS04] propose a hardware variant called CLIFF, which represents its elements in Verilog. Schelle et al.'s [SG05] CUSP is similar. Brebner et al. [BJRKK04] add speculative multi-threaded execution.

Unlike CLIFF/CUSP, I fully synthesize my modules instead of assembling library components. CLICK, furthermore, defines fine-grained elements whose connection has substantial handshaking overhead. My method synthesizes bigger modules; as fewer handshaking signals are necessary, the above mentioned overhead decreases.

My approach differs from classical high-level synthesis (c.f., De Micheli [De 94]) in important ways. For example, I always use a flavor of as-soon-as-possible scheduling for speed; classical high-level synthesis considers others. Furthermore, most operations are smaller than the muxes needed to share them, so I do not consider sharing.

The main difference is my choice of computational model. Rather than assume data are stored and retrieved from memories, I assume data arrive and depart a word at a time: my scheduling considers the clock cycle in which data arrive and can leave.

2.4 The Packet Editing Graph

2.4.1 What to expect from the PEG

The Packet Editing Graph PEG is a high-level model designed to specify the operations performed by a packet header editing module. PEG is the format taken as input by my synthesis procedure, which is described in the next sections.

I have designed PEG following two goals. First, it has to offer the right level of abstraction. That is, to offer a familiar view of the problem to the designer who is most likely to use it. Usually, he has good expertise in system architecture and network protocols but may lack RTL design skills. He sees the packet header editing as a group of arithmetic and logical operations performed on the packet fields. More abstractly, he sees the output data as a function of the input, not as a sequence of C or RTL statements including loops, jumps, pauses between clock cycles, etc.

The model has to be general enough to express any header editing operation. The reverse is more difficult to argue for but it is not less important. Any feature added without a good reason tends to cancel most of the advantages brought by a high-level language: clarity, easiness to debug, etc. Rich languages such as C++ or VHDL have proven this to be true.

Second, performance is critical for many applications such as packet processing pipelines. That is, a good model has to allow an efficient automatic synthesis technique. An elegant approach such as RTL synthesis from a software (C) model may indeed hide all implementation details but existing automatic synthesis algorithms are not very efficient. Even if there is no theoretical barrier against developing efficient algorithms in the future, such an approach is uninteresting today. Another reason for hiding the implementation details is the desire to synthesize to module on different targets with minimum effort.

I believe that PEG answers both issues well, as concluded at the end of this chapter, where experimental data are presented.

2.4.2 A PEG Example

Figure 2.1 shows a PEG for a simplified MPLSPUSH module. The MPLS protocol adds a label to the beginning of a packet that acts as a shorthand for the IP header. When another switch receives the packet, it uses separate rules to forward the packet. As the technique allows stacking, a packet may contain more MPLS labels.

As seen by the MPLSPUSH module a packet starts with a control header, which contains various fields, such as FLOWID, FLAGS, and LCT. This is followed by zero or

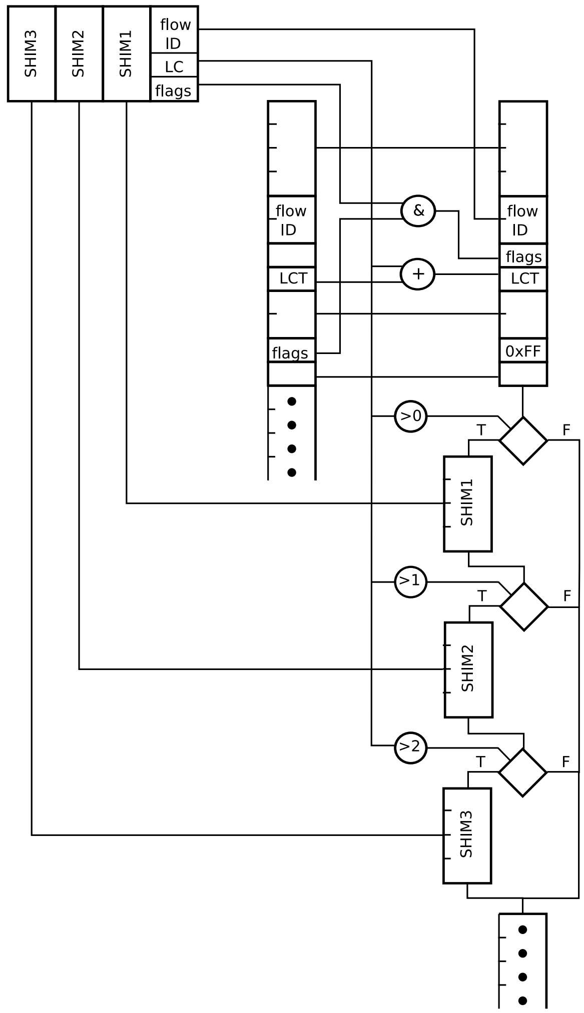


Figure 2.1: Graphic PEG specification for the MPLS module

more MPLS labels and the packet payload, which can be arbitrary data.

The module in Figure 2.1 inserts up to three MPLS labels, immediately after the control header according to a transaction descriptor (TD). The packet processor needs one TD for each processed packet: in this example, it obtains them by issuing requests to a memory lookup engine in a previous pipeline stage. The MPLSPUSH module has accordingly an auxiliary input to receive this TD from the memory lookup.

The module also updates the label count (LCT) of the control header by adding to its current value the label count field (LC) of the TD, so as the new header reflects the correct number of labels. Additionally, it overwrites the FLOWID field with the one from the TD and overwrites a reserved header field with the FLAGS, after they are anded with a mask taken from the TD; the original FLAGS field is overwritten with 11111111. Functionally, these operations map a set of MPLS tunnels to the set of next-hop destinations.

The reader can follow these operations in the PEG graphical representation from Figure 2.1. He can notice an acyclic, directed graph consisting of inputs, arithmetic and logical operators, and outputs, and the connections among those. The inputs are the packet itself and TD coming from the memory lookup engine: they are drawn as rectangles in the top left section. The operators are represented by the circular nodes in the middle of the figure. The output packet corresponds to the packet map (the control-flow graph on the right).

Looking at the figure, the reader can assume that time flows from top to bottom and data flows from left to right.

The packet map is the novel aspect of a PEG. The bits of the output packet are assembled by following a downward path. A diamond-shaped node is a conditional: control flows to one of its successors depending on the value of the predicate. Conditionals allow bits to be inserted and deleted from the output packet. The final node, marked with dots, copies the remainder of the input packet to the output.

2.4.3 PEG description

In this section I give a detailed description of PEG. A PEG is an acyclic, directed graph. It consists of three subgraphs: input, computation, and output. Each subgraph contains nodes of several types. In the above example, various types were represented by different shapes. A systematic list of these types and their exact meaning is given below.

Although PEG is conveniently depicted as a graphical model, I introduce in the sequel a text format for rigorously defining the type of the nodes, their attributes

```
#the input subgraph
pktin PIN 96
auxin TD 128

#the computation subgraph
alias H1 32 PIN 0 31
alias FLOW_ID 16 PIN 32 47
alias H2 8 PIN 48 55
alias LCT 8 PIN 56 63
alias H3 16 PIN 64 79
alias FLAGS 8 PIN 80 87
alias H4 8 PIN 88 95
alias SHIM3 32 TD 0 31
alias SHIM2 32 TD 32 63
alias SHIM1 32 TD 64 95
alias TD_FLOWID 16 TD 96 111
alias TD_LC 8 TD 112 119
alias TD_FLAGS 8 TD 120 127

arith NEW_FLAGS 8 and FLAGS TD_FLAGS
arith NEW_LCT 8 + LCT TD_LC
const FILL_FF 8 X''FF''
alias NEW_HDR 96 \
  H1 0 31 \ TD_FLOWID 0 15 \
  NEW_FLAGS 0 7 NEW_LCT 0 7 \
  H3 0 15 FILL_FF 0 7 H4 0 7

const C0 8 0
const C1 8 1
const C2 8 2
arith GT0 1 > TD_LC C0
arith GT1 1 > TD_LC C1
arith GT2 1 > TD_LC C2

#the output subgraph
pktout POUT OHDR
odata OHDR NEW_HDR BR0
cond BR0 GT0 OSH1 ! OPD
odata OSH1 SHIM1 BR1
cond BR1 GT1 OSH2 ! OPD
odata OSH2 SHIM2 BR2
cond BR2 GT2 OSH3 ! OPD
odata OSH3 SHIM3 OPD
payld OPD 96
```

Figure 2.2: Textual PEG specification for the MPLS module (2 columns)

and their functionality. This textual format is not meant to be human writable or readable. It is easy to imagine a GUI for designing a PEG graphically but I consider this beyond the scope of my thesis.

The syntax is trivial. Each line describes a node: it begins with the node type and its name, followed by some fields, according to the node type. The quickest way to understand the PEG syntax is to compare the graphical representation of the MPLS module in Figure 2.1 which the equivalent textual specification in Figure 2.2.

2.4.3.1 The input subgraph

This subgraph is trivial, and consists of a PKTIN node, and an optional AUXIN node, which specify the main packet input and the possible auxiliary input, respectively. They are described below:

Syntax	pktin	*name*	*size*
Example	pktin	N	128

Outcome: N represents the input packet, which has a minimum length of 128 bits.

Any PEG has one node, of type PKTIN, which represents the input packet. The *size* parameter is the minimum packet length.

Syntax	auxin	*name*	*size*
Example	auxin	N	112

Outcome: N represents the auxiliary input, which has 112 bits.

This node is optional, i.e., is present only for modules that require an auxiliary input such as a reply coming from a memory lookup engine. This input remains constant for all the time a packet is processed and changes only between packets. At most one AUXIN node may be present in a PEG. If several auxiliary inputs are required, they can be merged into a single AUXIN node: the RTL designer must combine the appropriate handshaking signals.

2.4.3.2 The computation subgraph

This subgraph is itself a DAG. Its nodes perform various computations such as arithmetic or logic operations, extracting and concatenating bit fields. Custom operations are also supported, assuming that corresponding RTL code (e.g., VHDL) is provided.

Each of the nodes in the computational subgraph take a number of inputs according to their functionality; these inputs may belong to the input subgraph or may be computational nodes themselves.

To avoid a series of ambiguities, PEG requires that each computational node to be assigned a size, which is the size of the computation result. Even if for some operation this size can be inferred from the size of the input operands, this cannot be safely done in the general case.

Below, I describe the type of nodes supported by the computational subgraph:

Syntax	const	*name*	*size*	*const*
Example	const	N	12	7

Outcome: N is the constant 7 represented with 12 bits

The CONST node is used to represent a constant.

Syntax	arith	*name*	*size*	*op*	I_0	[...]
Example	arith	N	8	and	A	B

Outcome: N is the bitwise AND of the two inputs A and B, extended or truncated to 8 bits.

The ARITH node takes as many inputs as needed by that particular operation. The operation *op* can be any arithmetic or logic operation supported in VHDL. Note that the size of the result is unambiguously specified; it is not inferred from the size of the inputs. This avoids a lot of confusion, such as the size resulting from adding two 8-bit numbers.

Syntax	alias	*name*	*size*	I_0	msb_0	lsb_0	[...]		
Example	alias	N	10	A	0	3	B	10	15

Outcome: N is the concatenation of 4 bits (0:3) of A and 6 bits (10:15) of B.

The ALIAS node concatenates subfields from one or more inputs. The size has to match the sum of the subfield sizes. In particular, this node can have only one input, so it may be used to extract a subfield of a node.

Syntax	extern	*name*	*size*	*expr*
Example	extern	M	14	A

Outcome: the computation for M is done by an extern module (M.VHDL)

An external module, from file NAME performs the computation. The input of the module is *expr*. The module has to provide an input *req* and an output *rpy*, whose sizes have to match the size of *expr*, and of the current EXTERN node, respectively. The module may have additional ports, which have to be handled manually. The module may contain sequential logic, but from our point of view it has to behave like a purely combinational node, i.e. its *rpy* output has to reflect the changes of its *req* input in the same clock cycle.

If *expr* is the special symbol "!," the module has no *req* input. This is useful, for example, for storing parameters, which from our point of view behave like constants but can be modified by the configuration circuitry that assists the pipeline.

2.4.3.3 The output subgraph

The output subgraph specifies the data that the module writes to the main packet output and, optionally, to the auxiliary output. It contains a PKTOUT node and an optional AUXSC one.

The PKTOUT node is in fact the root of a control-flow subgraph. For each packet, this subgraph is traversed, starting from the root down to one of its leaves. Once a leaf is reached, the process is implicitly repeated to handle to next packet.

When the control flow reaches an ODATA node, the module writes some data to the packet output; this node makes reference to a node in the computation subgraph, which specifies the data to be written. After that, the control is passed to the unique succesor node.

The output subgraph also contains COND nodes, which write nothing to the output but pass control flow to one of their successors, depending on some conditions. These conditions are also nodes from the computation subgraph.

The leaves are always PAYLD nodes, which copy the rest of the input packet (usually the payload) to the output.

Below I describe the nodes supported by the output subgraph.

Syntax	pktout	*name*	*dest*
Example	pktout	N	N1

Outcome: N is the root of the output packet subgraph.

A PEG contains one PKTOUT node, which represents the output packet. The *dest* points to the first active node in the output packet subgraph.

Syntax	auxout	*name*	*size expr*	
Example	auxout	N	48	A

Outcome: the auxiliary output N, of size 48, will output the expression A.

At most one AUXOUT node may be present in a PEG. The node is optional, i.e. is present only for modules which have an auxiliary output, such as a request going to a memory lookup engine. If more auxiliary outputs are needed, please see the description of the AUXIN node.

Syntax	odata	*name*	*size*	*expr*	*dest*
Example	odata	N	16	A	N1

Outcome: if N is reached, the module outputs 16 bits corresponding to expression A, then passes the control to node N1

The ODATA node instructs the module to write some data to the output packet.

The data to be written is taken from *expr*, which is a node from the computational subgraph, such as an arithmetic one. After that, the control is passed to node *dest*

Syntax	cond	*name*	*expr*	*dest*	[...]	!	*dest*
Example	cond	N	A	N1		!	N2

Outcome: when reached, node N transfers the control to N1, if $A \neq 0$, and to N2, otherwise.

The COND node behaves like an *if ... elsif ...* construct. Note that the condition for the last destination is marked by "!," which I use to denote "always." Indeed, N is required to pass control to some of its successors, for any values of the input conditions.

The main purpose of the COND node is to model conditional field insertions and removals, but its use is not restricted to these operations.

Syntax	payld	*name*	*offset*
Example	payld	N	128

Outcome: when reached, N copies to the output the input packet, starting from offset 128, up to the end.

The PAYLD node is used to handle the payload of the packet, which may have a variable size. Usually, the *offset* points immediately after the packet header. It has to be less than or equal to the minimum input packet size, as specified by the PKTIN node.

2.5 The Synthesis Procedure

PEG is an abstract model and can be synthesized to various hardware or software targets. I address the hardware synthesis of modules that belong to a pipeline such as in Figure 1.2 and interact with the environment using the protocol described in Section 2.5.1. This target is an attractive choice for high-performance network devices.

The proposed technique translates the PEG specification to an RTL model, i.e., synthesizable VHDL code. Since all modules and the inter-connecting FIFOs respect the mentioned I/O protocol, assembling the pipeline from its components becomes a mechanical operation.

The synthesis starting point, PEG, offers a flat, implementation-independent view of the module. Inputs and outputs can have different sizes and they are considered to be accessible as operands or destinations at any time. The challenge is to translate this model into RTL, where packets enter and exit the module on the fly, as a sequence of words of fixed size.

Consequently, different fields of the packet are present inside the real module on various clock cycles and the module has to perform the right operations at the right moments.

Fields do not generally fall on word boundaries: some fields of the input packet have to be assembled over several clock cycles and some fields of the output packet have to be output in several clock cycles. Moreover, since PEG allows conditional insertions and removals, there is not always a simple way of deciding when and where a field in the output graph has to be sent to the output.

Inserting and removing data changes the size of the packet. For some clock cycles, the input has to be stalled, the output has to idle, or both. Moreover, some fields in the output packet may depend on fields that arrive later in the input packet: the corresponding output word cannot be generated before the needed input arrives. The data does not flow through the pipeline module uniformly; valid data tokens are interleaved with idle ones.

The main goal of the synthesis technique below is to keep this data flow as close as possible to the ideal case, i.e., to insert as few input/output idle cycles as possible. However, the overall performance depends not only on the number of cycles but also on the cycle period. Therefore, concentrating complex operations inside a single clock

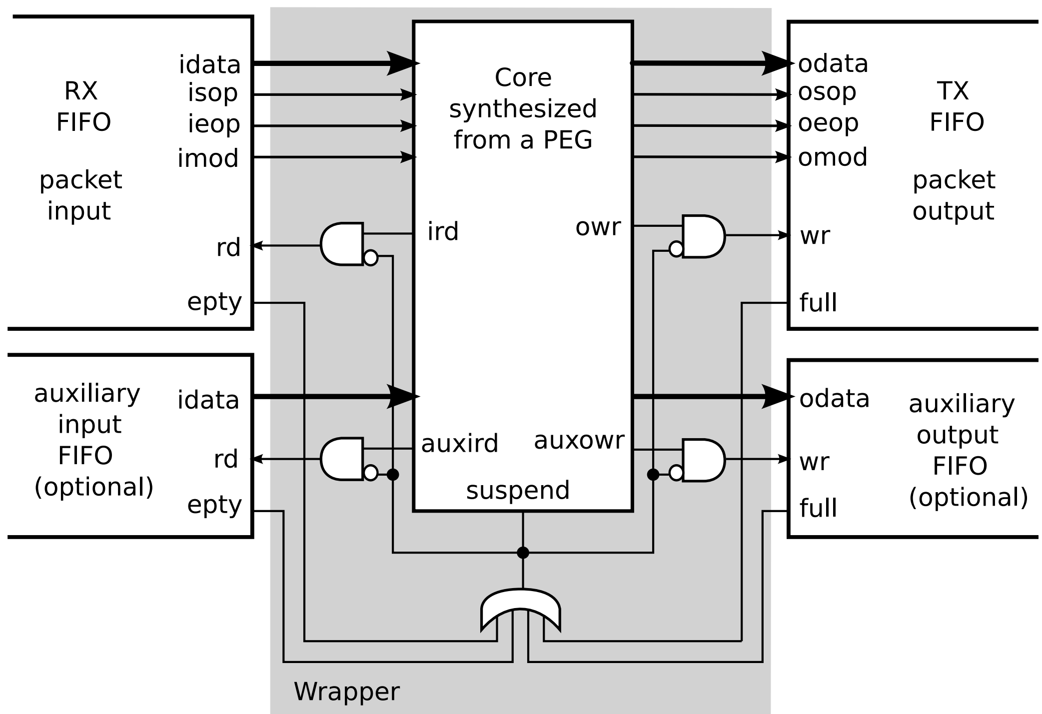


Figure 2.3: A module I/O signals and its two components

cycle may lead to a different outcome than intended.

2.5.1 The module I/O protocol

Within the pipeline, the packet editor module interacts with the adjacent FIFOs through four interfaces: the packet input and output, and the optional auxiliary input and output (Figure 2.3, where the module is represented by the gray area).

The module sees the input packet as a sequence of w-byte words arriving on the *idata* port, where values of 4, 8, 16 (i.e., 32, 64, and 128 bits) are typical for w. Similarly, the output is generated as a sequence of w-byte words on the *odata* port.

Additionally, data words are accompanied by two framing flags that identify the beginning and the end of the packets. The signal *sop* is asserted for the first word of the Packet and *eop* for the last. Both signals are asserted in the same cycle if the packet is small enough to be included within one word. All modules assume these signals arrive in a consistent order and generate them as such.

Since the packet may contain a number of bytes that is not multiple of w, the last data word may contain from 1 to w significant bytes. The *mod* signal, of size $\log_2(w)$, gives the number of valid bytes in the last data word minus one. This signal is meaningful only when the *eop* signal is asserted; any pipeline element is free to output undefined data on *mod* otherwise.

The input interface consist of the *idata*, *isop*, *ieop*, and *imod* signals described above, plus two handshaking signals: *val* and *rd*.

The *val* signal, generated by the input FIFO, tells the module if data is available at the input: when *val* is asserted then the $i*$ signals carry valid values, otherwise these signals have to be ignored.

The *rd* signal, generated by the module, instructs the input FIFO to show the next values on the $i*$ port in the next cycle; if de-asserted, and *val* is active, the FIFO has to keep the values unchanged. The value of *rd* is ignored by the FIFO if *val* is de-asserted; in this case, the FIFO will present valid data on the $i*$ port as soon as it becomes available, even if there is no explicit request by the module.

The output interface consist of the *odata*, *osop*, *oeop* and *omod* signals, plus two handshaking signals: *wr* and *full*.

The *wr* signal, generated by the module, is asserted when the values on the the $o*$ port are written to the output FIFO in the current cycle. If the *full* signal, generated by the output FIFO, is asserted, the *wr* is ignored and no data is written.

The optional input and output auxiliary ports use the same handshaking scheme as above, i.e., they use the *val*, *rd* signals and *wr*, *full*. The only difference is that these ports may have arbitrary bit-widths and that the framing flags *sop*, *eop*, and *mod* are missing.

The module makes one and only one access to each of these auxiliary ports for each packet. This differs from the main packet input and output ports, which transfer each packet as a sequence of words, in multiple cycles. For the auxiliary input, the read request *auxird* is asserted done at the start of each packet; therefore, the data value on this port remains stable while processing the entire current packet. For the auxiliary output, the write request *auxowr* is asserted as soon as the data to be written becomes available.

2.5.2 Core and Wrapper

The module can not continue its normal operation in two cases: when input data is not available, i.e., the input FIFO is empty, and/or when output data can not be written, i.e., the output FIFO is full.

To address this issue systematically, I divide the module in two parts, as depicted in Figure 2.3. The main part is the core module, which implements the actual module functionality assuming that two above exceptions never occur, i.e. that input data is always available and writing to the output is always possible.

The second part is a wrapper, consisting of simple logic, which stalls the core module in the above situations. To facilitate this process, the core module provides a special input, *suspend*, that freezes the state of all its sequential elements. This signal behaves as a general clock gate for the core module. Consequently, the synthesis of the core module can ignoring the above input/output exceptions, provided that trivial stalling logic is added later.

The wrapper also masks the *rd* and *wr* signals generated by the module in a stall cycle. This is necessary for correct sequential behavior.

The module may be stalled unnecessarily. This is the case when the *full* signal is asserted but the module does not try to write that cycle. The module would be able to continue its operation in this situation but my wrapper does not allow it.

I choose this behavior for performance reasons. I mentioned that not only the number of cycles has to be minimized but also the clock period. The core module may generate the *ird* and *owr* signals very late within the clock cycle. Often they belong to the critical loops of the overall sequential system. As a result, one would avoid connecting these signals to inputs to complicated logic functions; it is better to keep the surrounding logic as simple as possible, as my wrapper does.

One the other hand, the advantage of a more aggressive scheme is minor. In the above scenario, if the output FIFO is full, very likely this happens because the downstream module can not sustain the same data rate as the current module provides. Since the current module is not the bottleneck, saving one clock cycle at this point would not improve system performance.

The wrapper is generated by a simple script because the differences between wrappers to be generated for different modules are minor. These differences, e.g., the presence or absence of the auxiliary input and output ports, and their sizes, can be represented by several parameters that are passed to the script as arguments.

The rest of this chapter focuses on the synthesis of the core module itself.

2.5.3 Splitting Data into Words

The synthesis procedure begins by dividing the input and output packets on word boundaries using the procedure listed in Figure 2.5. Dividing the input packet is straightforward; dividing the output packet map is more complicated because of conditionals nodes. Figure 2.4 shows the result of applying this step on the MPLSPUSH example from Figure 2.1.

I restructure the packet map such that conditions are only checked at word boundaries. This assures that only complete words are written to the output every cycle. For example, the > 0 condition in Figure 2.1 has been moved four bytes earlier in Figure 2.4 and the intervening four bytes have been copied to the two branches under the conditional node.

The algorithm in Figure 2.5 recursively walks the packet map to build a new one whose nodes are all w bytes long (the word size). Each node is visited with a vector v that contains bits that are "pending" in the current word. Output nodes are added

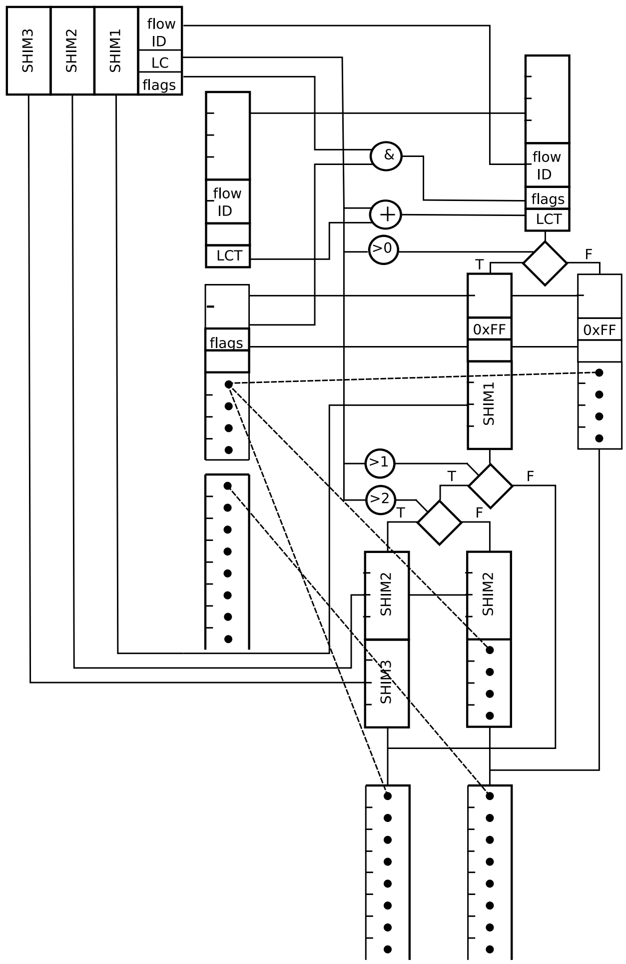


Figure 2.4: Split into 64-bit words

to this vector until $w * 8$ bits are accumulated: at this point a new output node n' is created by *build-node* by putting together the bits pending in v. When a conditional nodes is reached, the algorithm copies it to a new node n' and visits the two successors under the conditional. The same v is passed to each recursive call because the bits that appeared before the conditional and are not written in this point have to be written later in both cases.

This can generate an exponentially large tree but in practice this never happens. For example, there are four paths in Figure 2.4 but they lead to only two different final states: the one- and three-label cases converge since they require the same alignment;

function Restruct(node n, pending bits v, word size w)
  clean-visit $\leftarrow$ true if v is empty
  if *clean-visit* and *cache* contains n **then**
    return $cache[n]$
  case type of node n **of**
  output data **:** {≥ 1 bytes, one successor}
    append n to v {put n in current word}
    if v is $w * 8$ bits **then** {finished word}
      $n' \leftarrow$ build-node(v) {next word node}
      $n'' \leftarrow$ Restruct(successor of $n, (), w$)
      make n'' the successor of n'
    else
      $n' \leftarrow$ Restruct(successor of n, v, w)
  conditional **:**
    n' = copy of the conditional n
    for each successor s of n **do**
      n'' = Restruct(s, v, w)
      add n'' as a successor of n'
  if *clean-visit* **then**
    $cache[n] \leftarrow n'$
  return n' {the restructured node for n}

Figure 2.5: Structuring a packet map into words

the zero- and two-label cases are similar.

I handle reconvergence by maintaining a cache of nodes that have been already visited. The cache mechanism is activated only if a node visit is "clean," i.e., the pending vector v is empty. In these cases, if the node has already been visited (so it's in the cache) the graph traversal is stopped and the restructured subgraph already computed for the previous visit is used. If the node has not been visited (so it's not in the cache) then the traversal continues and the restructured subgraph is added to the cache so further visits of this node might use it. Consequently, the paths that use the same cache entry reconverge to the same subgraph.

2.5.4 Assigning Read Cycle Indices

After splitting the original graph into words, the resulting graph still respects the PEG semantics with two important differences, as shown in Figure 2.6, First, the PKTIN node in the input subgraph is split into several nodes of w bytes each. Second, all the ODATA nodes in the output subgraph have a fixed size, i.e., w bytes each.

I proceed now to assigning read cycle indices to each node in the graph. Figure 2.6 depicts these indices inside small black squares. To a first approximation, these indices represent the clock cycle in which each node is handled but they have a slighter different meaning.

A node's read cycle index tells how many words have to be read from the input FIFO to be able to compute the node's value, minus one.

The nodes representing the input packet are trivially labeled with consecutive indices, starting from 0. The optional AUXIN node is labeled with 0, because its value is known simultaneously with the first packet word.

For the computational subgraph nodes, I assign these indices in topological order: a node's index is the maximum value from its ancestors' indices. For example, the & node in the figure has two operands: the FLAGS field from the auxiliary input, and the FLAGS from the input packet. They have indices 0 and 1, respectively, so the &

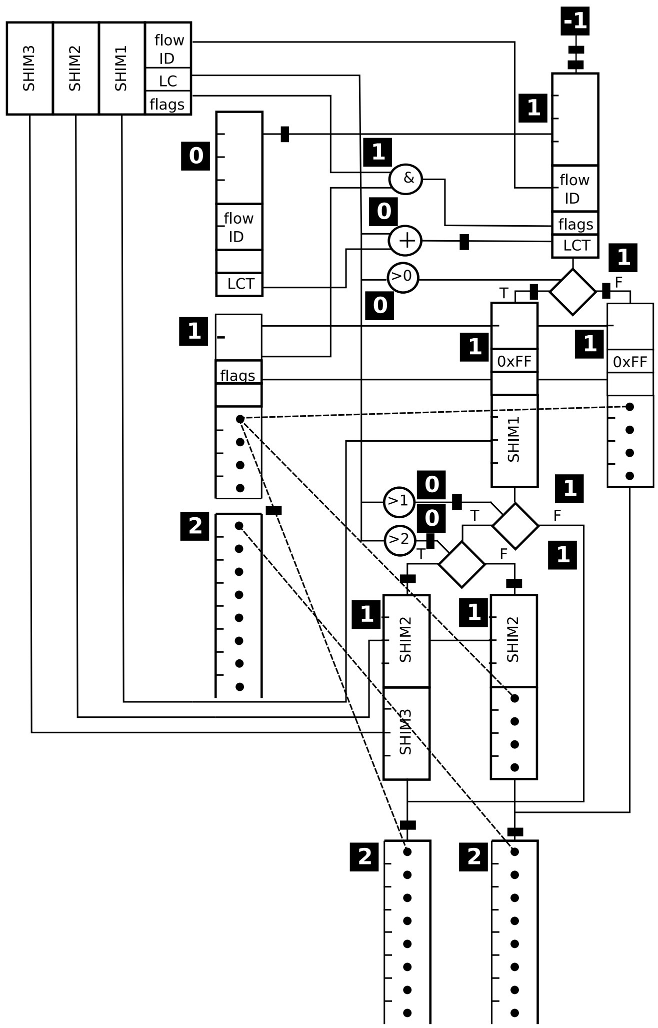


Figure 2.6: Read cycle indices and delay bubbles added

node is assigned index 1. Indeed, the module has to read at least two words from the input packet before computing this operation because the second operand belongs to the second word of the packet.

These indices do not have a direct correspondence to physical clock cycles: this correspondence is computed in a scheduling step described in the next section. Intuitively, the indices reflect the causality relations between the nodes inside the module.

By definition, the PKTOUT node, the root of the output subgraph, is assigned index -1. The rest of the nodes in the output subgraph are assigned indices as described above, taking the maximum value from their ancestors. Nodes generally have two kinds of ancestors: nodes from the computational subgraph and from the output subgraph. The current procedure makes no difference between these ancersors.

I exemplify this for an ODATA node. Two pre-conditions are necessary for this node to write a word to the output. First, the value to be written has to be known: this value comes from the ancestor in the computational subgraph. Second, all COND nodes above it have to be evaluated to assure that the current node is on the active path: this information comes from the ancestors in the output subgraph.

2.5.5 Scheduling

Once read cycle indices are assigned, the next step is to schedule the graph, i.e., to assign a physical clock cycle to each operation.

Each read cycle index corresponds to one or more physical clock cycles. The number of physical cycles corresponding to an index can vary depending on the active path in the output subgraph, i.e., depending on the direction in which the COND nodes are evaluated. Therefore, a given node is logically processed at a fix time, according to its index, but physically this can happen in a different clock cycle.

A node labeled with index k will have a valid value in all clock cycles corresponding to its index k. In particular, as the input words are labeled with consecutive indices 0, 1, 2, etc, the *idata* input will display the k-th packet word during all clock cycles corresponding to index k.

In the scheduling step, the algorithm inserts "bubbles" in the graph; each bubble corresponds to a transition between two clock cycles. Figure 2.6 depicts them as small black rectangles. This operation can done in several ways but the result has to respect the following properties:

- at least k bubbles are present between any two adjacent nodes whose indices differ by k: this rule states that a physical clock cycle can not be shared by two read cycle indices and that each index corresponds to at least one clock cycle.

- at least one bubble is present on each possible path between any two ODATA nodes in the output subgraph: this rule states that two data words can not be written to the output within the same clock cycle

Following the first rule, two bubbles were inserted between the top-most node and the first output node because the difference between their indices is two.

To comply with the second rule, bubbles were added on the arcs below the first conditional because they are connecting ODATA nodes with the same index. For the same reason, bubbles were added below the group of the remaining two conditionals. Since bubbles mark clock edges, it becomes clear that the same index may correspond to several clock cycles.

2.5.6 Synthesizing the controller

After read cycle indices and bubbles have been added, the module's control FSM (Finite State Machine) and its data path can easily be synthesized: the input and computational subgraphs generate the data path, and the output subgraph generates the controller.

The state diagram of the control FSM closely follows the structure of the output subgraph of the scheduled PEG, as can be seen in the ASM (Algorithmic State Machine) chart depicted in Figure 2.7.

The bubbles that were inserted in the output subgraph during the scheduling step become states of the resulting control FSM.

The bubble that was inserted after the PKTOUT node is the initial state of the FSM; this is labeled INIT in Figure 2.7.

The bubbles adjacent to the leaves of the subgraph are treated in a special way: all are merged in a common state, labeled REP, that handles the variable length payload. When the module reaches this state, it copies the packet input to its output, with no transformation, until the end of the packet. After the packet ends, the machine goes to the INIT state and starts to process the next packet.

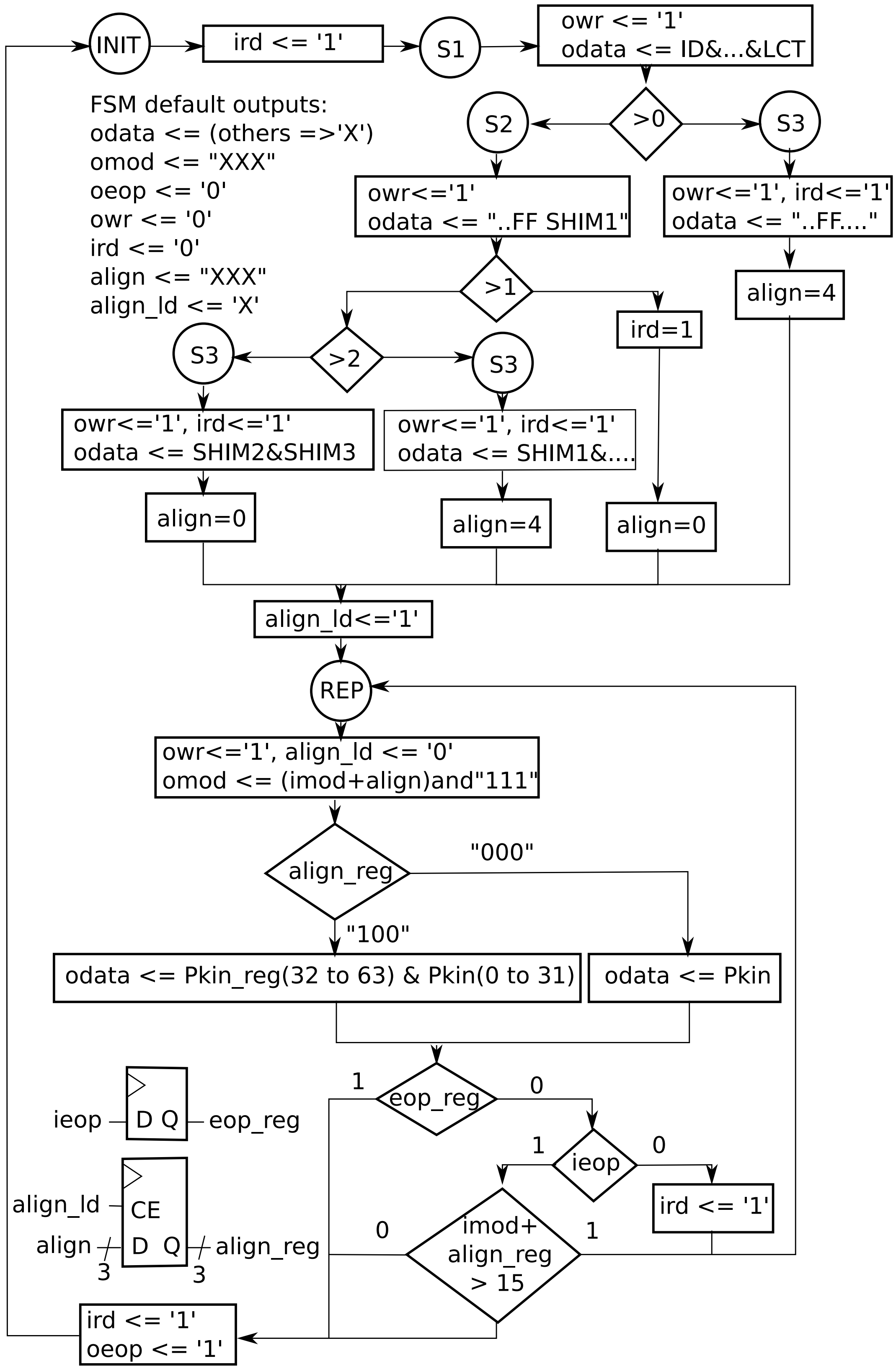


Figure 2.7: ASM chart for the controller synthesized from Figure 2.6

Each remaining bubble is simply translated into a regular state.

The COND nodes are trivially translated into decision nodes in the resulting ASM chart. The signal carrying the condition(s) to be evaluated comes from the data-path.

The ODATA nodes become control signals that go to the data-path and select the source of the data to be output to the *odata* port. In a practical implementation, they act as select signals driving a multiplexer. An ODATA node also activates the *owr* control signal, instructing the output FIFO that a valid data word has to be written.

The second scheduling rule ensures that at most one output node can be encountered on any path between two states. Therefore, it is guaranteed that the module never tries write two words to the *odata* port in the same cycle, which is impossible.

The signal *owr* remains de-asserted for state transitions where no ODATA node is encountered and consequently the module output will idle for that clock cycle. The value on the *odata* port is specified as don't care to create opportunities for further logic optimization.

The *ird* signal is asserted for each arc that joins two nodes with different arc indices. The first scheduling rule ensures that at most one such arc is present between two states: in this case the index increases by exactly one. Therefore, it is guaranteed that two words are not read from the input FIFO in the same cycle, which is also impossible.

The *ird* signal remains de-asserted for state transitions where the index remains constant and consequently the input FIFO is stalled for that clock cycle.

2.5.7 Handling the end of a packet

The leaves in the PEG output subgraph correspond to a payload processing state that copies the module input to its output until the packet ends.

Two such leaves are present in Figure 2.6. The one on the right corresponds to the cases where no label or two labels, i.e., 0 or 8 bytes, are inserted in the header. Since in this example $w = 8$, the payload in the output packet has in both cases the same alignment as in the input packet. The copy operation is trivial in this case since each payload output word corresponds to a word from the input payload.

The left leaf is slightly different: it corresponds to the cases where one or three labels are inserted in the header, i.e., 4 or 12 bytes. Therefore, the payload data in the output packet has a misalignment of 4 bytes relative to the word boundaries; the module has to assemble each payload output word from two consecutive input words.

After restructuring the PEG into words of w bytes, all paths in the output subgraph finally reconverge to no more than w different leaves, corresponding to different misalignments, from 0 to $w-1$ bytes.

Instead of using a different state for each leaf, the algorithm merges all of them in a common REP state. This state is accompanied by an auxiliary *align* register of size $\log_2(w)$ that encodes the misalignment between the input and output payload when the controller operates in the REP state. The *align* register is loaded with the correct value on each path that terminates with a leaf by asserting the *align_ld* signal.

First, the *align* register is used to generate the correct payload output words. I have mentioned that, in the general case, an output word is a combination of two consecutive words, i.e., the current input word and the previous word. Using *align*, the correct bits are selected from each. In the particular case where the input and output are aligned, no bits from the previous word are used. In Figure 2.7, the decision node below the REP state selects the correct bits to be used for each alignment.

Second, *align* is used to handle packet termination. The value of *align* can be seen as the number of pending bytes from the previous word. Indeed, in our example, if the data is perfectly aligned then $align = 0$, meaning that all the bits from the previous word have already been written. If $align = 4$, it follows that four bytes from the previous word are pending.

When *ieop* is detected, the module knows that the current input word is the last in the packet. In this point, the module has to output the pending bytes plus the valid bytes from the current word before going in the INIT state. Two cases are possible:

if the remaining data can be written in a single output word, i.e., $imod + align \leq 15$, then the module goes directly to the INIT state; otherwise, a second cycle is required to finish the transfer.

The *eop_reg* one-bit register is used for handling the second case. It is loaded with one in the first cycle when *ieop* is detected and it forces the controller to move to the INIT state in the next cycle. This prevents the machine from looping forever.

The operations described in Figure 2.7 are performed by the two decision nodes in the bottom right. The *oeop* signal is asserted when the last word is written. The *ird* signal is asserted together with *oeop*: this fetches the first word of the next packet, so the controller can continue its operation without pause between consecutive packets.

The *sop* signal, not covered by the figure, is asserted together with the first data word written to the output after the start of the packet signal *isop* is detected.

2.5.8 Synthesizing the data path

The nodes in the computational subgraph of the PEG are translated directly into combinational logic to form the data-path. Additionally, the data-path also includes the output multiplexer that generate the output word *odata*.

The algorithm replaces the bubbles in the computational subgraph with registers; their width corresponds to the data width of the corresponding arc. The resulting data-path has the property that a node has a valid value during the clock cycles corresponding to its read cycle index, as described in Section 2.5.4.

A read cycle index may contain several clock cycles. Therefore, a register has to be loaded on the last clock cycle contained in the read cycle index that precedes its own and to hold its value during its own cycle index. Otherwise, the value of the register is undefined and I treat its clock enable signal as don't-care. Generally, the combinational logic that produces these signals becomes trivial after minimization.

Many of the data-path registers could be removed during this step but I have chosen not to do so. The rationale is that the resulting sequential network is not the final product but the input for a low level tool synthesis and optimization tool that has to be invoked with the retiming option enabled. Having more registers on a combinational path helps the retiming procedure. Retiming is critical since the bubble positions within the PEG computational subgraph is computed in the scheduling step without considering the time taken by the various operations: the delays of different components can not be known before logic minimization.

Consequently, minimizing the number of registers can deteriorate the module performance. Since I have targeted FPGA devices, where the area cost of a register is very small, I have not explored any more sophisticated technique. However, it is theoretically possible to obtain a better result in terms of both area and performance by removing the extra registers and augmenting the corresponding paths with multicycle timing constraints.

2.5.9 Comments on scheduling and binding

The above scheduling technique may be seen as special case of ASAP scheduling combined with a trivial binding step that shares no resource [De 94]. However, the present approach differs from the the standard techniques used in high level synthesis because of the different computational model involved and of the different architecture of the module.

Nevertheless, I believe that looking at the current issues from the classical perspective, pointing out the similarities and the differences, could be interesting. Strictly looking from that perspective, one may doubtlessly argue that a more sophisticated binding/scheduling technique might lead to better results in terms of performance and area.

Classical scheduling and binding assume that the data-path consists of computational units and storage elements connected by buses, resembling a small custom CPU [De 94]. In this framework, the scheduling and binding steps try to reduce the total number of clock cycles for a complex operation, the amount of resources used, or both, with the only constraint that operations have to be done in causal order. An important assumption is that data can be read/written from/to the storage elements on any clock cycle. The only drawback is that multiple accesses in the same clock cycle increase the complexity of the data-path components and the number of buses.

The framework considered in this work is different. Data enters and exits the module as a sequence of words, in a fixed order that is given by the format of the packet header and is not necessarily the best for scheduling the computation. For performance reasons, the module has to operate on the data stream on the fly, letting the data flow through as uniformly as possible: the data-path becomes a pipeline.

Consequently, the order of performing the computations has not only to respect the causality constraints but heavily depends on the position of the operands and results in the input packet and the output packet, respectively. If a result has to be written before its operands are available, the module has to stall for one or several cycles. This reduces the actual throughput of the pipeline, as shown in Chapter 3. Avoiding unnecessary stalls becomes the first priority for synthesis.

The individual operations of a typical header editing module consist of simple logic, as illustrated in Figure 2.6; many of them are shifts and concatenations, and require a zero logic cost. Intuitively, these modules are not number-crunching engines but data-shuffling components: this is probably what one expects from a module inside a network switch or router.

Most individual operations take much less than a clock cycle and their area is smaller than the one occupied by multiplexer of the same width. Moreover, two or more adjacent simple operations can be successfully simplified by logic synthesis.

Therefore, there is no advantage in sharing computational units, since this eventually leads to both lower performance and increased area. I simply instantiate a computational unit for each operation found in the PEG. This corresponds to a trivial binding with no resource sharing.

Because of this choice, there are no conflicts between different operations. All computations can be performed as soon as their operands are available. Practically, computations are performed on every clock cycle, regardless of the validity of their operands. It is the responsibility of the control FSM to use their results only when they are consistent. From the classical perspective, this corresponding to an ASAP scheduling; indeed, if there are no resource constraints, an ASAP scheduling brings no penalty compared to any other possible technique.

To conclude, classic high-level synthesis considers various binding and scheduling techniques, each with its advantages and disadvantages. By contrast, the present problem is formulated in a different framework, targets a different architecture and requires different goals. I have found a simple, good — even if not optimal — solution for this problem.

2.5.10 Comments on FSM encoding

Starting from the ASM chart, the FSM can be encoded using any technique. I leave this encoding step to be performed by the low-level synthesis tool.

However, I noticed experimentally that the tool evaluates several alternatives but usually chooses a one-hot encoding, except for very small machines.

It can be argued that a more efficient encoding could be achieved by exploiting the high-level structure of the FSM, which is hidden to a low-level synthesis tool. This is correct.

Grouping the leaf states in a common REP state and introducing the *eop_reg* and *align_reg* registers does a *de facto* partial encoding of the FSM. In a prior implementation, I treated the leaf states almost like regular states, i.e., I created creating a different state for each bubble: the performance increased when I switched to the present scheme. The grouping trick was possible because these final states have a very close functionality; therefore, a better encoding could be found by exploiting high-level information.

However, the regular states have few properties that can be immediately exploited.

A better encoding depends on the Boolean properties of the whole module, including the control FSM and the data-path. This information is available only after the first steps of logic synthesis and optimization. Consequently, the best choice is to rely on an existing encoding and sequential synthesis tool.

2.6 Implementation

I implemented the proposed synthesis technique on a Linux Fedora Core 3 platform, coding the algorithms in C++ and using PERL scripts to put them together.

2.6.1 Textual frontend

The PEG model is the starting point for the whole synthesis procedure; moreover, all test cases I have used in the process of writing and debugging the software were written in PEG. However, PEG was not designed with the intention of being human writable or readable. The complex modules used for the final testing were not written in PEG but in a human friendly textual language developed independently by another team.

I implemented a simple translator that converts the text specification into the PEG model. This translator is not a mandatory part of the synthesis flow but it proved very helpful: I could test complex modules that belong to real world applications and are written by different engineers. This last aspect was a very useful sanity check.

2.6.2 Synthesis steps

The tool consists of several programs, written in C++, which are invoked in succession from a PERL script. These sub-programs perform the tasks mentioned above: translate the textual specification into PEG (optional), split the PEG into fixed-size words, schedule the resulting graph, generate the main VHDL code, and generate the VHDL code for the module wrapper.

The programs make extensive use of C++'s STL (Standard template library) to handle the complex data structures required, principally graphs. I have avoided any obscure or platform dependent feature and, a result, the code is highly portable.

Each sub-program generates an intermediate file that is in turn read by the next. These files are human readable, which proved to be a major advantage for debugging. Moreover, I am able to display graphically, at each intermediate step, the internal data structures by using the DOT graphical package and a PostScript viewer.

I also considered important generating easily readable VHDL files. For example, I preserved (as much as possible) the names of the identifiers, indented the VHDL code according to the common standards, described the control FSM in a textbook-like manner, etc.

The whole project consists of 8500 lines of C++ code, 2500 lines of VHDL and almost 1000 lines of BASH and PERL scripts.

The running-time of all synthesis steps, taken together, is negligible. This is not a surprise since all the presented algorithms run either in linear time or have at most a quadratic complexity.

2.6.3 Simulation

I have written a PEG simulator to check the correctness of the synthesis flow. This simulator takes a module written in PEG and an an input packet and generates an output packet according to the PEG semantics. It also supports auxiliary inputs and outputs.

The simulator reads the entire input packet into an internal array and then performs all the operations in topological order. Finally, the output packet is assembled by processing the output subgraph: following the control flow, each output node appends its current value to an initially empty packet. I took this most trivial approach since my goal was to avoid any error, both in the way I interpret the PEG semantics and in the simulator code itself.

For simulation, I first generate a large number (millions) of random input packets and feed them to the simulator, one by one: the resulting output packets are collected as reference. Then, I synthesize the PEG model for three different pipeline widths (32, 64, and 128 bit). Because of the different widths, the synthesis generates very different circuits in the three cases: the three control FSMs have a quite different number of states and the data-paths are likewise very different.

I simulate the resulting VHDL code by ModelTech's VSIM after I include it inside a small custom wrapper that reads the input stimuli from an input file and dumps the results to an output file. This wrapper can randomly insert idle input tokens in the data stream and also backpressure the module from the output, modeling a realistic environment.

Finally, I compare the three results with the initial reference. I have simulated in this way over two dozen modules, for a total simulation time of over 100 hours. Although the perfect concordance of the results does not prove the correctness of my implementation, at this point I am confident in the quality of my implementation.

2.6.4 RTL and Low-Level Synthesis

Although not a part of the current research, I insert these paragraphs since they bring a natural conclusion to my proposed synthesis flow and also make the experimental results presented in the next section easier to follow.

I performed the RTL synthesis by using the Synplify-Pro tool, which I run with the optimization flags tuned towards maximum performance, i.e., clock frequency. In particular, I have enabled the sequential optimization steps such as FSM encoding exploration and retiming. Since I targeted the FPGAs from the Xilinx Virtex 4 family, I used the Xilinx ISE tools for technology mapping, placing, and routing.

For a realistic performance evaluation, as well as for a correct count of the used resources, I have placed my synthesized module inside a wrapper (slightly different from the one used for simulation). Consequently, I synthesize a small fragment of a

Table 2.1: Synthesis results for selected modules

Module	**Core size**		**Delay**	**Throughput**
	LUTs	**FFs**	**ns**	**Gbps**
MPLSpush	556	107	3.8	33
TTLEXPupdate	43	20	2.9	44
VLANfilter	11	12	2.9	44
VLANedit	505	125	4.0	32
PPPoEfilter	410	151	3.7	34
PPPoEterm	819	322	4.0	32

packet processing pipeline that includes the analyzed module. I found that my initial timing and area estimates are correct.

2.7 Experimental Results

I synthesized several modules from an industrial design. The chosen pipeline width was 128 bits. I have chosen as target a Xilinx Virtex 4 xc4vlx40–ff668–10 FPGA. Both choices are representative: they were not made by the author but are the choices made by system architects for a real system which includes two packet processing pipelines.

The synthesis flow used for testing follows the description from the previous section, with the exception that I have used the Xilinx Xst RTL synthesis tool instead of Synplify-Pro; this choice was made for non-technical reasons. I do not expect significant differences in the results.

I have described the MPLSPUSH module in the previous sections, where it was used as an example (Figure 2.1, Figure 2.2). The rest of the modules, although from the system point of view perform very different tasks, can be seen for the purpose of this work as variants, more or less complex, of this module.

To realistically estimate the size and performance of the synthesized modules, I have synthesized each module inside a wrapper that models the module's neighborhood in a real packet processing pipeline.

Table 2.1 shows the size and performance of each module. To measure the performance, I use the post-routing static timing analyzer from the Xilinx ISE package. This is the delay of the longest register-to-register path, plus a register's setup and hold times. For area, I report the number of flip-flop and lookup table primitives required by a the Virtex 4 device minus the number of resources used by my wrapper. Since the synthesis tool makes some optimizations across the hierarchical boundaries, the results can slightly vary if a different wrapper is used.

The last column shows the estimated module throughput, i.e., the product between the module bit-width and the frequency from the previous column. This number is a rough approximation: computing the real worst-case throughput of a packet processing pipeline is more complicated. I treat this topic in detail in Chapter 3.

The real pipeline that contains these modules is clocked at 166 Mhz, i.e., the clock period is 6ns. All six synthesized modules can work considerably faster.

2.8 Conclusions

Establishing a strict formalism for describing packet editing operations (the packet editing graph) allowed me to construct a high-performance hardware synthesis procedure that can be used to create packet processors.

The performance of circuits synthesized by my procedure is comparable to the performance of circuits in state-of-the-art switches, while the design entry is done at a much higher level of abstraction than the RTL usually used.

The direct benefit is improved designer productivity and code maintainability. Experimental results on modules extracted from actual product-quality designs suggest that my approach is viable.

Chapter 3

Packet Processing Pipeline Analysis

3.1 Overview

High performance pipelines are critical components in many modern digital systems. The packet processors which can be found on the line cards of modern network switches, the central topic of the present thesis, are typical instances of such critical pipelines.

Analyzing the performance of a packet processing pipeline is an essential component of the design process. Without an accurate performance estimation, a network device can be over-designed or can function below its required parameters, with the consequence that packets are dropped even if the communication link is not saturated.

To help the reader understand more clearly the issues I address in the sequel, I summarize below the particularities of a packet processing pipeline, as they were found in the previous chapter.

First, these pipelines have a linear structure. There are no data dependency loops as those encountered, for example, in a CPU core. This simplifies their architecture a lot and offers a framework for obtaining a very good performance.

Additionally, the function of individual pipeline modules tends to be very simple because of their high performance requirements (perhaps tens of gigabits per second); the overall pipeline complexity arises from chaining modules together. This design style is also encouraged by the fact that pipeline latency is not very important: the interesting figure of merit is the throughput. Doubtlessly, the total switch latency is important, but the pipeline latency is dominated by that introduced by queueing and switching, so spending a few extra cycles in the pipeline is not a major drawback.

Another particularity of the pipelines I am considering is the non-constant data flow through each module: modules can insert and/or remove data. Specifically, each module has state and is free to perform different computations at different times depending both on its state and the data fed to it. As such it is not always ready to produce or receive more data. The data flow through the pipeline would contain empty data tokens in addition to the actual data words.

Packet processing pipelines consist of modules connected by FIFOs to mitigate the effect of variable data rates. Such FIFOs decouple the behavior of the modules, maintaining a consistent data flow through the pipeline.

However, the most important issue, briefly mentioned above, is that a module may perform various classes of computation, depending on the packet data itself. For example, some module inserts or removes data from a VLAN packet but leaves the PPPOE packets unchanged. Consequently, the input/output behavior of a module can vary substantially from one packet to the next. Moreover, the complexity of adjacent modules may vary dramatically: not seldom a complex module is followed by a trivial one or vice-versa.

It it easier to start the discussion about analyzing the performance of packet processing pipelines with a simple motivational example.

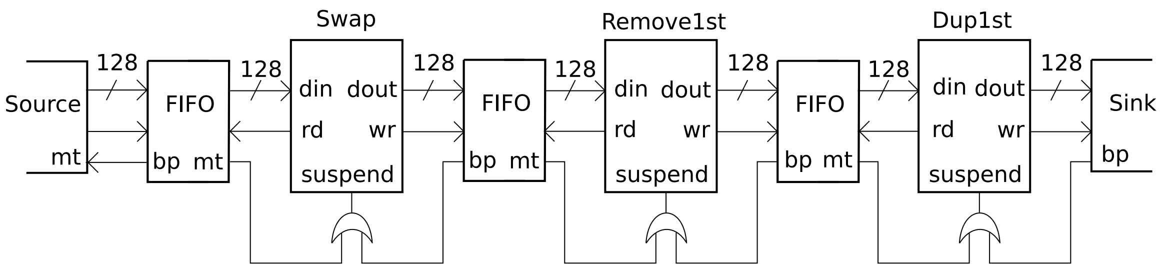


Figure 3.1: A packet pipeline: how big should the FIFOs be?

3.1.1 Motivational example

Consider Figure 3.1 — a pipeline of a network switch. It is intended to process packets that start with a number of 112-bit Ethernet headers followed by a payload.

The first module swaps the first two headers, the second removes the first header, and the third duplicates the first header. Each module performs its specific operation on certain packets (those with a matching TYPE field) and leaves the others intact.

This pipeline can modify packets in eight different ways. Although this simple pipeline could be replaced by a single module, chaining these operations is more representative of a real packet processing pipeline.

The first module leaves the packet length unchanged but the second and third may shrink and expand it respectively: the flow through the pipeline is not constant. Moreover, this flow varies according to the packets' content.

The combination of data- and state-dependent computation in the modules plus the presence of the FIFOs make this pipeline a complex sequential system. This chapter addresses two related questions.

First, what is a pipeline's worst-case throughput? One rule of thumb is that typical pipelines work at 50% throughput, so doubling the clock frequency should be enough to process any data pattern. My experiments show that this approximation is often wasteful and may even be incorrect for many real pipelines.

Second, given an achievable throughput, what are the minimum FIFO sizes for which we can guarantee that performance? Here, the designer usually over-estimates these sizes using his experience. However, many modules are simple and consist mostly of combinational logic, while FIFOs contain many sequential elements. Over-designed FIFOs waste chip area and power.

Even for such a simple pipeline, answering these questions about throughput and FIFO size is challenging and not something one can easily do by hand.

The methods presented in this chapter take less than thirty seconds to tell us that the worst-case throughput of this pipeline is 0.6 and that this can guaranteed with a minimum total FIFO capacity of fourteen (specifically, 4, 5, and 5 for the three FIFOs in Figure 3.1). Furthermore, if we reduce the throughput requirement to 0.50, a total size of twelve is sufficient: 4,4,4 and 4,5,3 are both valid solutions.

Below, I first define some metrics to characterize the sequential behavior of a single pipeline element (Section 3.3). Then I analyze the worst-case aggregate throughput of the pipeline assuming ideal FIFO sizes (Section 3.5.1). Next, I propose two algorithms that can determine the minimum FIFO sizes; one algorithm is exact, the other is heuristic (Section 3.7). I support these results by experiment (Section 3.8).

3.2 Related Work

Traditional queuing theory is not able to answer these questions. First, it assumes arrival rates can be modeled stochastically, yet network traffic rarely follows standard distributions such as the Poisson. Second, even if we had appropriate distributions, queuing theory only provides stochastic (perhaps average-case) results. Here we are concerned with the worst case, not a distribution.

Le Boudec and Thiran's network calculus [LT01] provides a helpful methodology for analyzing the pipeline throughput and FIFO sizing, given modules with variable delays, but it requires that each module has a fixed read and write throughput. This is not the case here, where each module may have highly variable, data-dependent input/output behavior.

Many have addressed analyzing and reducing buffer memory consumption for synchronous dataflow graphs (SDF [LM87]). While the arbitrary topology of SDF graphs is richer than our linear pipelines, they assume their modules produce and consume data at a fixed rate. As such, SDF admits fixed schedules and very aggressive analysis. For example, Murthy and Bhattacharyya [MB04] show how input and output buffer space can be shared when exact data lifetimes are known. Such precise analysis is generally impossible for our models since they allow data-dependent production and consumption rates.

Like SDF, latency-insensitive design [CMSV01] allows richer topologies than our pipelines, but insists on simple module communication patterns. Lu and Koh [LK03, LK06] attack essentially the same problem as we do, but in a semantically different setting. They propose a mixed integer-linear programming solution. Casu and Macchiarulo [CM05] is representative of more floorplan-driven concerns: their buffer sizes are driven by expected wire delays, something I do not consider.

The asynchronous community also considers buffer sizing (e.g., Burns and Martin [BM91]), but again consider richer topologies, simpler communication, and fancier delay models.

Answering to this question can not be avoided: is worst-case analysis justified for real-world packet processing pipelines or would a statistical analysis suffice? Are the extreme cases significant in real data patterns?

The answer is yes. Even though most extreme cases never occur, I have noticed that real data patterns are generally far from the statistical "average case." For example, Ethernet linecards are often tested with sequences of all smallest-length and all longest-length packets. These traffic patterns are not realistic but these tests became standard, vendors are advertising these benchmarks and customers are asking for these numbers. Of course, these tests are easy to perform but their popularity has a second reason: they are acceptable approximations of the most extreme traffic patterns for a simple device: they provide a worst case guarantee.

Pipelines to be found today inside a commercial linecard are far more complex. For example, they can simultaneously process native Ethernet, VLAN and PPPOE packets. It becomes more difficult if not impossible to enumerate all corner cases.

A folklore rule says that the field traffic pattern never matches any of the patterns used for development and testing, no matter how carefully the test set was designed. Unfortunately, this is too well confirmed by practice: many network devices operate much under the specified performance in critical real world situations; most of them behave properly on the test bench.

I conclude that the worst case performance analysis of a network device, in general, and of a packet processing pipeline, in particular, is not a strictly academic challenge but has a very important impact in the real world.

3.3 Single module analysis

Packet processing pipelines consist of a linear array of FSMs (Figure 3.1). I assume the whole pipeline runs synchronously with a single clock — a typical implementation technique. While I could consider an asynchronous implementation, the analysis of the individual modules would be much more complicated (see, e.g., Burns and Martin [BM91]).

The first step in addressing the pipeline performance is to analyze each module in isolation. My first goal is to define an exact metric for measuring the performance of such a complex module; additionally, these partial results can be rapidly combined for a worst case estimation of the throughput of the entire pipeline (Section 3.5.1).

As shown in Figure 3.5, data for each module comes from the source and goes to the sink through the *din* and *dout* signals. For efficiency, these are wide buses, e.g., 128 bits. The module interacts with source and sink through two control signals, *rd* and *wr*, which the module asserts when it wants to read and write 128-bit blocks of data. None, either, or both of these signals may be asserted each clock cycle.

Each module also has one input signal, *suspend*, which stalls the module when the input data is unavailable to read, or when there is no place to write output data. The details about the handshaking between neighboring pipeline elements will be described in the next section.

At this point, I simply assume that the module to be analyzed in isolation is placed between ideal source and the sink, i.e., both *mt* and *bp* signals are never asserted. In this case, the FSM is never stalled.

3.3.1 Defining module performance

A module does not always assert the *rd* and *wr* signals. Hence, idle cycles may occur at the module's input and/or output, and the overall throughput is less than unity in general.

The sequence of the values observed on the *rd* and *wr* signals depends on the module functionality and on the data being fed to the module: different sequences will be observed for different data patterns.

Let r_i^p, w_i^p, $i = 0, 1, 2, \ldots$ be the sequences of values observed on the *rd* and *wr* signals for a given input data pattern p.

It is desirable to quantify the worst case throughput of the module for any data pattern. However, since modules insert and remove data, the amount of data entering the module is not necessarily equal to the amount that leaves it. Therefore, I define a worst-case read and a worst-case write throughput, which I denote with R and W, respectively.

For the input, the worst case corresponds to sequences r_i^p with the smallest number of 1s in a given time. Formally,

$$R = \lim_{t \to \infty} \left(\min_p \frac{\sum_{i<t} r_i^p}{t} \right). \tag{3.1}$$

Intuitively, such a limit exists because I am only considering finite systems that ultimately exhibit repeating behavior. I present a more formal argument later.

R is interesting for pipelines such as ingress packet processors when one wants to guarantee that a module can process any input flow at a given rate. For example, if one finds $R = 0.4$, then to guarantee a 200 MS/s input throughput for any data pattern, the pipeline must be clocked at 500 MHz (as $500 \times 0.4 = 200$).

I similarly define W, the worst case output throughput:

$$W = \lim_{t\to\infty} \left(\min_p \frac{\sum_{i<t} w_i^p}{t} \right). \tag{3.2}$$

This is interesting for pipelines such as egress packet processors when one wishes to guarantee a certain output rate. In the sequel, I will focus on input flow rates. The other case is symmetrical.

Another interesting metric for a module is the minimum read/write ratio, denoted by T, defined as:

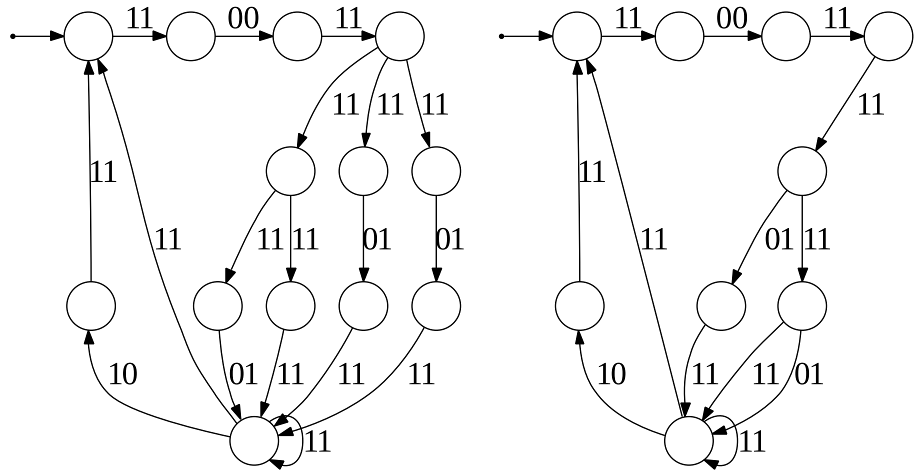
$$T = \lim_{t\to\infty} \left(\min_p \frac{\sum_{i<t} r_i^p}{\sum_{i<t} w_i^p} \right). \tag{3.3}$$

Since a module's input and output data rates can be different, T is defined as the minimum ratio between the input and output rates, for any data pattern. The rationale of introducing T becomes visible in Section 3.5.1.

A module's input/output behavior is not completely determined by these three values, R, W, and T. However, they encapsulate a lot of information about the module behavior. Having these values computed for each module is sufficient for computing a bound of the overall throughput for a linear pipeline (Section 3.5.1).

3.3.2 Computing R,W, and T from the STG

I start with the STG (State Transition Graph) of the module's FSM. I build a graph $G = (V, E)$, where V is the set of states and $E \subseteq V \times V$ is the set of transitions. It is critical to know on what transitions the *rd* and *wr* signals are asserted, so I assign to each edge $e \in E$ two labels: x_e^r and $x_e^w \in \{0, 1\}$.



(a) original (b) simplified

Figure 3.2: STGs for the DwVLANproc module

Figure 3.2a shows a typical STG of the control FSM belonging to a packet editing module. The reset state in the top-left is the INIT state (see Section 2.5.6); the state with the small self-loop at the bottom is the REP state; the four paths on the right correspond to the different operations that can be performed by the module. Although this example FSM is generated by the synthesis technique from Chapter 2, I address in this chapter any possible implementation.

The FSM input signals are not relevant at this point; the important ones are the output control signals *rd* and *wr*. Therefore, instead of attaching to the arcs some labels such as "xxx/01" I simply write "01," i.e., the values of x_r and x_w for the corresponding transition. This explains the situation where multiple outgoing transitions are shown to start from the same state such as the bottom-most state REP. The system is completely deterministic but since the conditions for each transactions are ignored, I treat the system as a non-deterministic one.

The above STG can be thought of as a nondeterministic finite automaton with four output symbols that correspond to the four possibilities for reading and writing data, i.e., 00, 01, 10, and 11, and all states accepting. It is possible to simplify it using an heuristic algorithm to produce the slightly simpler STG in Figure 3.2b but I did not implement this optimization.

Any input data pattern p corresponds to an infinite path in the STG that starts from the reset state. All possible sequences of r and w signals can be enumerated by considering all possible paths, but this is not practical.

I show the module metrics R, W and T can be computed by inspecting the above mentioned STG by inspecting its cycles instead of enumerating all its infinite paths, which would be impossible.

A simple cycle c is defined as a sequence of edges that forms a non-intersecting path whose tail connects to its head. Let C be the set of all simple cycles in the graph G. I can compute R, W, and T by considering every simple cycle $c \in C$:

$$R = \min_{c \in C} \frac{\sum_{e \in c} x_e^r}{|c|} \tag{3.4}$$

$$W = \min_{c \in C} \frac{\sum_{e \in c} x_e^w}{|c|} \tag{3.5}$$

$$T = \min_{c \in C} \frac{\sum_{e \in c} x_e^r}{\sum_{e \in c} x_e^w} \tag{3.6}$$

where $|c|$ is the number of edges in cycle c.

The rationale is as follows. R, as defined by equation (3.1), corresponds to the input data patterns which generate x^r sequences with the smallest average read/cycle ratio. Any such pattern corresponds to a path in the STG.

First, I consider the paths that consist of simple cycles $c \in C$ that repeat periodically. I immediately arrive at equation (3.4), where the hard limit R is reached when a STG cycle with minimum read/cycle ratio is repeating ad infinitum.

More complex paths can be decomposed in a sequence of simple cycles because the number of states in the STG is finite. Let this sequence be denoted $S = (c_1, c_2, \ldots)$, where $c_i \in C$ are simple cycles; I emphasize that this sequence is not periodic in the general case. Equations 3.4–3.6 are somehow misleading as they suggest that only periodic patterns are considered for analysis.

As defined by equation (3.1), R becomes:

$$R = \frac{\sum_{c_i \in S}(\sum_{e \in c_i} x_e^r)}{\sum_{c_i \in S} |c_i|} \geq \min_{c \in C} \frac{\sum_{e \in c} x_e^r}{|c|}$$

which proves the equivalence between the two formulations, if we consider the well-known inequality:

$$\frac{\sum a_i}{\sum b_i} \geq \min(\frac{a_i}{b_i})$$

A similar argument holds for W and T.

The next step is to compute R, W, and T using the above equations.

3.3.3 Using Bellman-Ford

Graph metrics such as those defines by equations (3.4), (3.5), and (3.6) are known as minimum cycle means [Kar78]. Dasdan [Das04] uses slightly different notation but otherwise our formulation is identical.

Dasdan assigns each edge two weights, $\omega(e)$ and $\tau(e)$, which simply correspond to our x_e^r and x_e^w:

- for R: $\omega(e) = x_e^r, \tau(e) = 1$
- for W: $\omega(e) = x_e^w, \tau(e) = 1$
- for T: $\omega(e) = x_e^r, \tau(e) = x_e^w$

Here, I show how to compute the metrics R and W using a method proposed by Lawner (see Dasdan [Das04]). I chose it for its simplicity.

Let $G = (V, E)$ be a directed graph with edge weights w_e for $e \in E$. The $O(VE)$ Bellman-Ford algorithm checks if all cycles in G have positive weights. If this is true, the algorithm returns the minimum paths from a given source to each node.

I ignore any computed path lengths and use Bellman-Ford only to check whether $\forall c \in C, \sum_{e \in c} w_e \geq 0$.

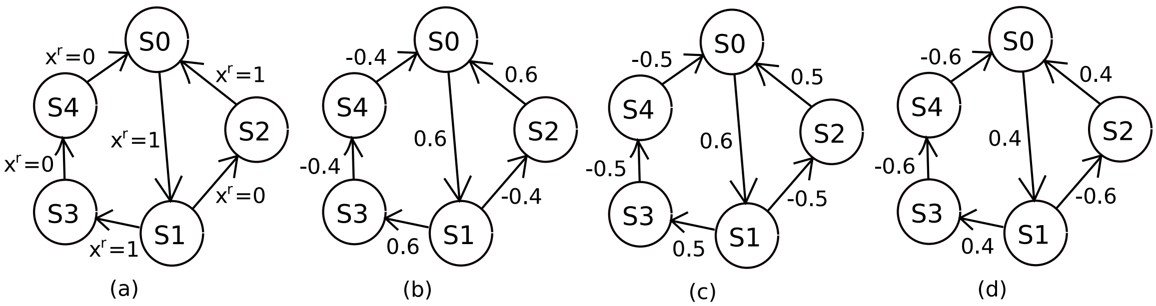


Figure 3.3: Sample STG to illustrate section 3.3.3 (a) x^r labels (b), (c), (d) label weights w_e for $\alpha = 0.4, 0.5, 0.6$ respectively

Using some simple arithmetic tricks, Bellman-Ford can be used to inexpensively compute the R, W, and T metrics.

To compute R, it follows from (3.4) that

$$R = \max(\alpha) \text{ s.t. } \forall c \in C, \alpha \leq \frac{\sum_{e \in c} x_e^r}{|c|}$$

Assigning $w_e = x_e^r - \alpha$, we have

$$\frac{\sum_{e \in c} x_e^r}{|c|} \geq \alpha \leftrightarrow \sum_{e \in c} x_e^r - |c| \cdot \alpha \geq 0 \leftrightarrow \sum_{e \in c} w_e \geq 0.$$

Therefore, we have to find the maximum α such that all cycles are positive. For a given α, Bellman-Ford can be used to check this condition, α can be approximated as accurately as desired by binary search.

Similarly, to compute T —see equation (3.6)— I assign $w_e = x_e^r - \alpha \cdot x_e^w$, which gives

$$\frac{\sum_{e \in c} x_e^r}{\sum_{e \in c} x_e^w} \geq \alpha \leftrightarrow \sum_{e \in c} x_e^r - \alpha \cdot \sum_{e \in c} x_e^w \geq 0 \leftrightarrow \sum_{e \in c} w_e \geq 0.$$

Figure 3.3 illustrates this technique by computing R on a small STG. Figure 3.3a shows the original STG and its edge labels x^r. Two simple cycles can be found by inspection: (S0, S1, S2) and (S0, S1, S3, S4). Their read/time ratios, i.e. $\sum_{e \in c} x_e^r / |c|$, are $2/3 \approx 0.666$ and $2/4 = 0.5$. According to (3.4), $R = 0.5$.

Since the number of cycles can be exponential in the size of the graph, this straightforward approach is not feasible.

I will approximate R as described above. In Figures 3.3b, c, and d, I assign $\alpha = 0.4, 0.5$, and 0.6, respectively. Next, I compute the edge weights $w_e = x_e^r - \alpha$ as shown above.

In Figure 3.3b, both cycles are positive: the Bellman-Ford algorithm accepts this graph and I conclude that $R \geq 0.4$. In Figure 3.3d, the small cycle is positive but the second is negative: Bellman-Ford rejects this graph and I conclude that $R < 0.6$.

Bellman-Ford accepts α values greater or equal than the true R and rejects those less than the true R (here, 0.5). Binary search can be used to approximate R with arbitrary precision.

The case in Figure 3.3c is the limit since $\alpha = R$. Since this is the threshold between *all-cycles-positive* and *at-least-one-cycle-negative*, it is not surprising to find one cycle has exactly zero length.

3.3.4 Abstracting the data path

The FSM of a packet processing module can have an enormous number of states. This happens because the number of states depends on all registers, both in the controller and in the data-path. The number of registers in the controller is usually small but the number of data-path registers is much bigger: the latter is of the order of magnitude of the pipeline width. Since widths of 128-bit or 256-bit are common, the total number of states becomes very large and renders the above method infeasible.

The problem can be surmounted by dividing the FSM into its data-path and control components. The controller includes the *rd*, *wr*, and *suspend* signals (Figure 3.4). I only consider for analysis the states of the controller and treat the data-path as a black box. The STG to be analyzed has a small number of states: the above algorithm becomes practical.

There is no theoretical limit between the control and the data-path of a module. Since I used for experiments modules synthesized by the technique in Section 2.5, I choose the natural boundary given by the above synthesis procedure.

This control STG can be seen as an abstracted version of the original one. It is important to known how well this abstracted version represent the original input/output behavior.

All signals from the data-path to the control part of the original module, which are truly internal signals, are treated as independent primary inputs for the abstracted STG. Therefore, the abstracted STG can have extra states and transitions but it never lacks the real ones, i.e., those present in the original STG.

The input/output behavior of the abstracted STG cannot be better than the original: any worst-case performance guarantee obtained for the abstracted STG also holds for the original. Ignoring the module data-path leads to an approximation but this approximation is always done towards the safe side. I lack a theoretical result for the quality of this approximation but experiments suggest the difference is small for practical modules.

In the rest of this chapter, I only consider the abstracted STGs of the modules.

3.4 Connecting modules

In the previous section, I have shown how the throughput can be analyzed for each isolated module. The next step is to put these partial results together to analyze the complete pipeline.

First, I attempt to compute the worst case-pipeline throughput in a theoretical scenario that assumes the interconnecting FIFOs are large enough to decouple the processing modules.

Next, I propose a model-checking based technique to analyze the pipeline performance for practical systems with finite sized FIFOs.

Finally, I use the above results to compute the minimum FIFO sizes for which the pipeline performance is not compromised. This follows the observation that the performance decreases if the FIFO sizes drop below a certain threshold.

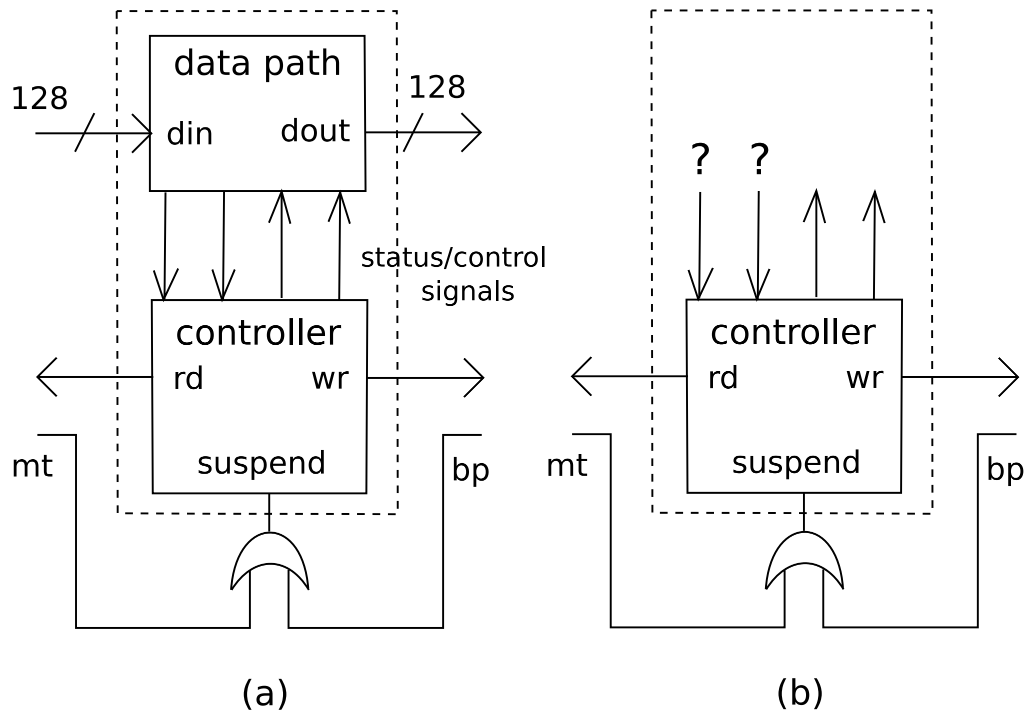


Figure 3.4: (a) Original module FSM (b) simplified FSM with abstracted data-path

3.4.1 The module I/O protocol

I describe the handshaking scheme used by a processing module to communicate with its neighbor pipeline elements, i.e., the input and output FIFOs. Figure 3.5 shows the signals used for handshaking and their interconnection.

The data travels the pipeline through a wide data-path. it enters and exits a processing module by the *din* and *dout* ports; the FIFOs have matching ports.

Since the module does not necessarily read or write data on each cycle, it is responsible to generate two control signals, *rd* and *wr*. They enable reading and writing from the upstream FIFO and to the downstream FIFO, respectively.

The source (e.g., the input FIFO) asserts the *mt* ("empty") status signal when no data is available (e.g., the FIFO is empty). Likewise, the sink asserts the *bp* ("back pressure") status signal when the it can not accept data.

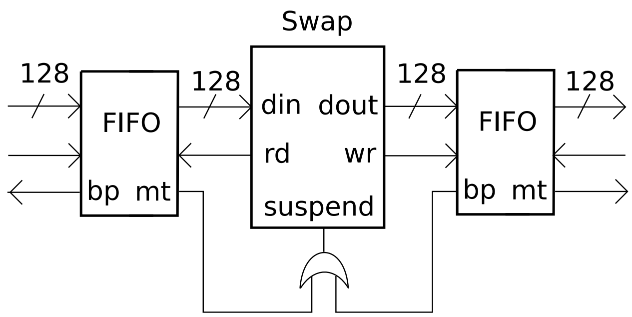


Figure 3.5: Handshaking between module and adjacent FIFOs

These two status signals are ORed to form a *suspend* signal that stalls the module for a clock cycle when asserted. When stalled, the module holds its state for the current cycle and de-asserts the *rd* and *wr* controls. Therefore, the module is stalled regardless of its intent to read or write in a particular cycle: the stalling depends exclusively on the FIFOs' status.

It would be possible to carefully analyze the two control signals *rd* and *wr*, and the two status signals *mt* and *bp* to stall the module only when strictly necessary. If the module does not read in the current cycle it can proceed even if the input FIFO is empty. Similarly for writing.

However, I am advocating this simple handshaking scheme for performance reasons. In the present framework, the *suspend* signal is quickly computed by ORing the *mt* and *bp* signals. On the contrary, in a more complex approach the *suspend* signal also depends on the module's intention to read or write, which in turns depends on the current state and on input data. The extra complexity tends to increase the length of the critical paths and affects the clock frequency: this ultimately leads to lower performance.

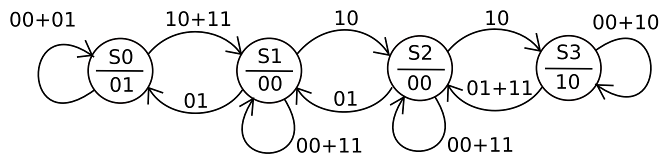


Figure 3.6: STG of a 3-place FIFO: $I = (wr, rd)$, $O = (bp, mt)$

3.4.2 The FIFO implementation

I define the STG of the FIFOs used for interconnecting the pipeline elements and the exact function of their input and output signals. There are a lot of FIFO variants, very close in functionality but different from the RTL point of view. An exact model is necessary for a cycle-accurate analysis.

Figure 3.6 shows the STG for a FIFO with three storage locations. Only the control part of the FIFO is represented; the data-path is abstracted for the reason described in Section 3.3.4.

This STG contains four states, corresponding to an empty FIFO, and to 1, 2, and 3 occupied elements. It is easy to extrapolate this diagram for a FIFO with a different number of storage locations.

This FIFO variant is a Moore machine. Both the control signals (*bp* and *mt*) and the data outputs (not shown in the abstracted STG) depend only on the current state (and the memory content). There is no combinational path from any input to any output.

While this means it always takes at least one clock cycle for a FIFO to react (e.g., by asserting the backpressure signal), such an approach greatly improves the performance of the system, exactly because long combinational paths usually create performance bottlenecks. Intuitively, my FIFO choice improves the performance (i.e. clock period) of the system at the expense of input-output latency, measured in clock cycles: for current packet processing applications, the latency is hardly a constraint, while throughput is critical.

Another advantage of choosing this FIFO variant is that the logic synthesis problem is simplified since optimizing a sequential network with relatively disjoint components is much easier than a tightly coupled one. As a disadvantage, in some cases an additional stage may be needed to achieve the same system performance.

3.5 The Big FIFO scenario

In this section, I attempt to compute the pipeline's worst-case throughput under the assumption that the FIFOs are "large enough." Specifically, I want to find out what minimum throughput can be guaranteed when the FIFOs are large enough so that they are never a bottleneck.

This is a different scenario than assuming that the FIFOs are infinite. In this case, the "large" FIFOs provide sufficient "elasticity" to compensate for the short-term variations in the input and output rates of the two connected modules. However, if the upstream module consistently writes faster than the downstream module reads, the FIFO will eventually fill and the *bp* signal will be asserted. Similarly, in the opposite situation, the FIFO will become empty and the *mt* signal will be asserted.

Intuitively, I would like to analyze the pipeline operation in steady state, where the modules work at a fraction of their ideal throughput (i.e., the one achievable in isolation), such that the data rates entering and exiting each FIFO match. Since the behavior of each module is data-dependent, one may imagine many such possible balance states. My proposed technique can consider all these behaviors simultaneously, in an implicit manner; moreover, it is very fast.

This ideal worst-case throughput is an upper bound on the real throughput achievable by a real pipeline with finite FIFOs. I consider the effects of limiting the FIFO sizes later in this chapter.

3.5.1 Computing the worst-case throughput

I compute the worst-case read throughput of a linear pipeline in the "big FIFO scenario" presented above. I use the pipeline in Figure 3.1 to exemplify the algorithm.

I denote this throughput with R_{123}, since it corresponds to the modules M_1, M_2, and M_3 taken together: they can be seen as a bigger module, M_{123}.

As a first step, I compute the individual R and T metrics for each module using the method in Section 3.3.3. I denote them R_1, R_2, R_3, T_1, T_2, and T_3.

I compute the desired result R_{123} iteratively.

Consider a module M_{23} constructed by merging M_2, M_3, and the interconnecting FIFO. I compute the worst-case read throughput of M_{23}, denoted by R_{23}.

As mentioned above, I assume that each FIFO is big enough to compensate for any spurious activity of the modules it connects but I do not consider the FIFOs to be infinite. Therefore, the average data production and consumption rates of each two adjacent modules in the pipeline will be balanced in the long term.

Intuitively, two cases may be encountered, depending whether the upstream module M_2 or the downstream module M_3 forms the bottleneck.

For some data patterns, M_2 writes slower than M_3 can read. In this case, M_2 is the bottleneck and the FIFO between them becomes empty in some clock cycles, causing M_3 to stall. M_2 is never backpressured in this case and it behaves exactly as in the case where M_3 was an ideal sink. Therefore, in this first case $R'_{23} = R_2$.

For other patterns, M_2 may want to write faster than M_3 reads. In this case, M_3 is the bottleneck and the FIFO becomes full in some clock cycles, causing M_2 to stall. The fraction of time M_2 is active (not stalled) is exactly the ratio between the read rate of M_3 and the write rate of M_2 for that particular pattern. Multiplying the read rate of M_2 with the above ratio, we find that in this second case $R''_{23} = T_2 \cdot R_3$. Considering the worst of both worst cases, we find $R_{23} = \min(R'_{23}, R''_{23})$.

This step can be repeated, considering the two modules M_1 and M_{23}. The result is R_{123}, which is the answer to the original problem.

function Ideal-Throughput
    $r \leftarrow 1$
    for $i \leftarrow n \ldots 1$ **do**
        $r \leftarrow \min(R_i, r \cdot RW_i)$
    return r

Figure 3.7: Computing R assuming optimally-sized FIFOs

The complete algorithm, for an arbitrarily long pipeline, is listed in Figure 3.7. It simply considers all the pipeline modules from right (output) to left (input), and performs at each step the computation described above.

3.5.2 Performance issues

Computing R_i and T_i requires building an explicit STG for each module; practically they seldom exceed twenty states and the observed running time is negligible. The loop in figure 3.7 takes no time.

This fast algorithm is well-suited to a high-level synthesis design-space exploration loop. Generally, each module admits several implementations that are not sequentially equivalent and have different costs.

It is critical to have a quick method to compute the pipeline performance for each such design point. This is not a trivial task because of the complex interaction between the pipeline elements. As shown in the next section, finding the exact performance for a pipeline with finite sized FIFOs is computationally expensive.

However, the "big FIFO scenario" and the present algorithm allow a very quick computation of the pipeline performance. This performance can be achieved by choosing the right FIFO sizes. Consequently, it is guaranteed that once a point in the design exploration space is selected, the reported performance can be achieved in practice.

3.5.3 Module independence

When I have abstracted the data-path for each module (Section 3.3.4), I assumed that the inputs coming from the data-path to the controllers behave like independent inputs. I have argued that this is an conservative approximation, i.e., it always underestimate the performance.

A new challenge arises when considering several connected modules.

Modules' behavior depends heavily on the processed data. For a packet processing pipeline, the criterion for triggering one behavior or another is usually the packet type and some other header flags. For example, one modules processes VLAN tagged packets but leaves the others unchanged; the next module processes PPPOE tagged packets but leaves other packets unchanged, including the VLAN packets.

My method assumes the worst case scenario at each step, without taking into account that what is worst for one step is not necessarily the worst case for the next. Therefore, the computed worst-case throughput is again an under-estimation.

3.6 Verifying throughput using model checking

In this section I present a model-checking-based technique to check if a given throughput is feasible for a pipeline. More exactly, the inputs of the algorithm are a given pipeline (i.e., the FSM of each module and the size of each FIFO) and the desired throughput. The output is a Boolean value that is true if the pipeline can sustain the given throughput for any possible input data pattern.

Although the method can be used for any purpose, I use it in the sequel to compute the minimum FIFO sizes. The FIFO sizing algorithm is a search algorithm that iteratively calls this verification method with different parameters to find the minimum cost solution. I present this algorithm in Section 3.7.

To use the model checker, I slightly modify the pipeline as shown in Figure 3.8. I replace the actual sink with an ideal one, which never asserts backpressure — the

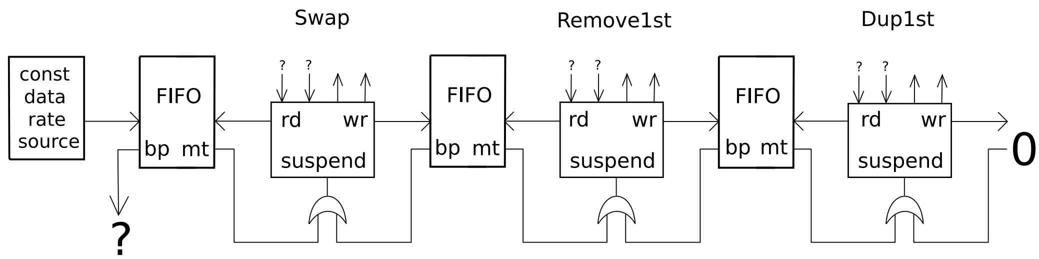


Figure 3.8: slightly modified pipeline for the model checker

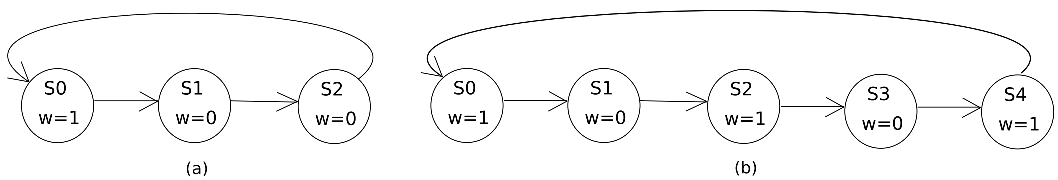


Figure 3.9: Data source FSMs (a) T=1/3 (b) T=2/5

big "0" on the right. Next, I replace the source with a small FSM that generates data at a constant rate — the throughput I wish to test.

It follows that I can model only throughputs that are ratios of integers. I typically use ratios of small integers: this improves the running time. Figure 3.9 depicts two sources, generating data streams with data rates of 1/3, and 3/5.

The data source FSM has no inputs: it is a free-running counter that cannot be backpressured. The backpressure signal coming from the first FIFO, labeled with the big "?" in the figure, is always zero if the pipeline is able to accept the desired throughput. This signal becomes one if the data source is too fast.

In this point, I run a model checker to verify the following property: the "?" signal is always zero. Proving or disproving this answers the original question.

I detail the whole procedure. Let us denote the n pipeline modules with $M_i, i = 1, 2, \ldots, n$. I treat the constant rate source FSM as an additional module M_0.

Between the $n+1$ modules there are n FIFOs denoted with $F_i, i = 1, \ldots, n$. Their sizes are denoted by f_i. The signal labeled with "?" in the figure is in fact bp_1.

I start with STGs of the pipeline elements M_i and F_i, described by KISS models [DBSV85]. I encode each of them (both modules and FIFOs) using the one-hot algorithm in SIS [SSM+92]. I now have a BLIF file for each pipeline component. The chosen encoding method is irrelevant. This choice can influence the model checker running time but not the algorithm outcome.

I assemble all the BLIF files by connecting the handshaking signals according to Figure 3.8. The resulting BLIF network has a number of primary inputs and outputs, which are placed on the cut between every module's controller and data-path. Additionally, it has more two primary outputs: wr_n and bp_1.

In this point I use the *check_invariant* algorithm from the VIS package [BHSV+96] to verify that, regardless of the current state of the system, and regardless of the input values (i.e., regardless of the data flowing through the data-path), the property $bp_1 = 0$ holds.

Doubtlessly, a more advanced model than VIS can be used. Since this changes only the running time, not the final outcome, I did not explore this possibility in my experiments.

3.6.1 Comparison with explicit analysis

The simplest way to analyze the complete pipeline is to build its product machine, starting from all its components, both modules and interconnecting FIFOs. This is a simple operation. The pipeline worst-case throughput can be computed from the explicit product machine STG using the algorithm from from Section 3.3.3.

The size of this product machine grows exponentially with the complexity of the pipeline, rendering this method infeasible in practice. I have nonetheless tried this approach on smaller examples to verify the model checking based method: the results were consistent.

Actually, the model checker internally builds the product machine but represents it implicitly, using — in the case of VIS — BDDs for representing the state transition function and the space of reachable states. This implicit representation is more compact, allowing much more complex FSMs to be analyzed.

3.6.2 Comparison with simulation

I compare the model checking approach to a widely used alternative, simulation. The main difference is that simulation covers only a limited number of possible input patterns. Its accuracy depends on the quality of the stimuli data sets and of their size, i.e., the simulation running time. Simulation gives only an experimental confirmation of the checked property.

Model checking covers all of possible input patterns implicitly: the method does not require any input stimuli. Therefore, the result is exact.

3.7 Computing the minimum FIFO sizes

In the previous sections I have shown how to compute the maximum (module-limited) throughput of a linear pipeline assuming ideally sized FIFOs. I now address the problem of finding the smallest FIFOs that can achieve that throughput.

The motivation for this study is the improvement in area and power that can be achieved by reducing the FIFO sizes. Since the goal is to achieve the same worst-case performance as in the ideal case, this improvement brings no performance penalty. The only drawback is the running time taken by the FIFO sizing algorithm at design time.

Although in practice these FIFO sizes are found to be small integers, each extra stage costs a lot of sequential elements because of the large width of the pipeline, Most pipeline modules in a typical packet processing pipeline consist of simple combinational logic: I have noticed that for practical designs the area taken by the FIFOs may easily exceed the area taken by the modules themselves. This confirms the importance of a careful selection of FIFO sizes.

This problem is more complex than the previous one. For large FIFOs, the short-term variations in the date rates were compensated by the FIFO elasticity: only the worst case *average* rates were relevant, for which reason I could complete the analysis by using only the simple R, W, and RW metrics for each module.

If the FIFOs have finite sizes, the short-term behavior of the interacting modules can no longer be ignored. The analysis has to consider the exact sequential behavior of each pipeline element, both modules and FIFOs, and their interaction.

3.7.1 Computing FIFO Sizes

Since I could not find an algorithm that directly computes the assignments of FIFO sizes of minimum cost, I take an indirect approach. It consists of two elements.

The first is a model checking based method described in Section 3.6, which can validate or invalidate a given assignment of FIFO sizes for a required throughput.

Second, a search algorithm finds the minimum cost assignment by iteratively calling the above verification method with various assignments. I propose two search algorithms with the same functionality: one of them is exact, the other one heuristic. The running time of the latter is better but the result has a lower quality.

In the sequel, I consider a very simple cost function for the quality of a FIFO size assignment: the sum of all FIFO sizes. A different cost function may be used with minor modifications to the search algorithms.

3.7.2 FIFO size monotonicity

Since I use costly model checking in the inner loop of the search algorithm to determine whether a pipeline configuration can achieve a given throughput, I make the following observation to reduce the search space.

For two pipelines with the same modules M_i, but different FIFO sizes f_i and f'_i:

$$\forall i, f_i < f'_i \text{ implies } R < R'. \tag{3.7}$$

This is because increasing the size of a FIFO can never decrease throughput: decreasing throughput requires more backpressure, but a larger FIFO never induces any.

This is not a total ordering, e.g., it does not discriminate when $f_i < f'_i$ and $f_j > f'_j$ for some $i \neq j$. Nevertheless, it helps to reduce the search space when trying to find overall minimum FIFO sizes. When $\forall i, f_i < f'_i$, I write $F \prec F'$.

3.7.3 An exact depth-first search algorithm

Here I present an exact search algorithm for determining minimum FIFO sizes. It is often too slow, so in the next section I accelerate it with a heuristic. Its runtime is sometimes practical; I also use it to evaluate my heuristics.

Figure 3.10 shows the algorithm, which is a variant of a basic depth-first search. To prune the search space, it uses the FIFO size monotonicity property (Section 3.7.2), which is checked by the GeThanAny and LeThanAny auxiliary functions.

The core function is ThroughputAchieved, which calls the VIS model checker (Section 3.6), and decides if a given assignment of FIFO sizes can achieve the desired throughput.

The algorithm first considers pipelines with all FIFOs of size one, then all of size two, etc., until a feasible one is found; this is the starting place for the search.

The algorithm maintains three lists of FIFO size assignments: GOOD, BAD, and TRY, which contain the fifo sizes which are proved to be good, bad, and not checked yet. The Succ function returns the next points in the search space to be checked if the current state is good. The depth-first behavior arises by adding and removing elements from the beginning of the TRY list in a stack-like fashion.

The MinSize function returns the best solution found; our cost metric is simply the sum of FIFO sizes, reflecting their area. The algorithm can be modified to use a more complicated metric.

```
function GeThanAny(F, list)
  for F′ ∈ list do
    if F ⪰ F′ then
      return true
  return false

function LeThanAny(F, list)
  for F′ ∈ list do
    if F ⪯ F′ then
      return true
  return false

function MinSize(list)
  s ← ∞
  for F′ ∈ list do
    s′ ← ∑_i F_i
    if s′ < s then
      F ← F′
      s ← s′

function Succ(F)
  S ← ∅
  for 0 ≤ i < n do
    S ← S ∪
        (F_0, ..., F_{i−1}, F_i − 1, F_{i+1}, ...)
  return S

function SearchMinFIFO
  s ← 1
  F ← (1, 1, ..., 1)
  repeat
    BAD.push_front(F)
    s ← s + 1
    F ← (s, s, ..., s)
  until ThroughputAchieved(F)
  GOOD.push_front(F)
  for F′ ∈ Succ(F) do
    TRY.push_front(F′)
  while TRY.size > 0 do
    F ← TRY.pop_front
    if GeThanAny(F, GOOD) then
      ok ← true
    else if LeThanAny(F, BAD) then
      ok ← false
    else
      ok ← ThroughputAchieved(F)
    if ok then
      GOOD.push_front(F)
      for F′ ∈ Succ(F) do
        TRY.push_front(F′)
    else
      BAD.push_front(F)
  return MinSize(GOOD)
```

Figure 3.10: An exact algorithm for computing minimum FIFOs

```
function GreedyMinFIFO
  s ← 1
  repeat
    s ← s + 1
    F ← (s, s, ..., s)
  until ThroughputAchieved(F)
  H ← (0, 0, ..., 0)
  while H ≠ (1, 1, ..., 1) do
    m ← −1
    for i ← 1, . . . , n do
      if H_i = 0 and (m = −1 or F_i > F_m) then
        m ← i
    F_m ← F_m − 1
    if not ThroughputAchieved(F) then
      H_m ← 1
      F_m ← F_m + 1
  return F
```

Figure 3.11: Greedy Search for minimum size FIFOs

3.7.4 A heuristic search algorithm

I find the exact algorithm of Figure 3.10 too slow. Instead, I propose the heuristic search algorithm of Figure 3.11. This does not guarantee an optimal solution, but in practice appears to be able to produce solutions close to it and runs much faster.

Like the exact algorithm, this one starts by considering FIFO sizes of all one, then all two, etc., until a solution is found. Then, it attempts to decrease the largest FIFO. When decreasing the size of this FIFO would violate the throughput constraint, I mark it as "held" and do not attempt to reduce its size further (the H array holds the "held" flags). The algorithm terminates when all FIFOs are marked as "held."

This algorithm can miss the optimal FIFO size assignment because it assumes the FIFO sizes are independent, which is not true in general. Relations between FIFO sizes can be fairly complex, for example, increasing the size of one may enable two or more

Id	Name	States	Transitions	R	RW
A	VLANedit	24	33	0.600	0.643
B	UpVLANproc	18	40	0.530	0.563
C	DwVLANproc	13	17	0.909	1.000
D	VLANfilter	1	2	1.000	1.000

Table 3.1: Module statistics

other FIFOs to be reduced (note that this does not violate my monotonicity result of Section 3.7.2). Nevertheless, I find this heuristic algorithm works well in practice.

3.8 Experimental Results

I use in my experiments four modules (A, B, C, D) taken from a packet processing pipeline in a commercial ADSL-like network linecard. I synthesized them for a 32-bit wide pipeline. In practice, 64- or 128-bit buses are more common, meaning the modules will have fewer states and require smaller FIFOs. I chose complex modules to illustrate my algorithms. Some statistics for these modules are presented in Table 3.1.

Module A is the most complex module I found in practice. It can swap, insert, or remove (or a combination of those) VLAN tags from packet headers. The operations performed on each packet depend on the packet flow, i.e., they are data dependent.

Modules B and C can be seen, for the purpose of this work, as simplified versions of A, performing a reduced set of VLAN tag manipulations. Their complexity is representative for most modules encountered in practice.

Module D is simpler, but not trivial. It generates a request to an auxiliary memory unit, but since it does not insert and remove data from the packet its input/output behavior is trivial from the pipeline point of view. This is the case with many real modules, so I believe including it in the experiments is justified.

Modules	**Ideal**	**Used**	**Greedy**		**DFS**	
1 2 3 4 5	**thruput**	**thruput**	**Size**	**Time**	**Size**	**Time**
ABC	0.329	0.250	10	6s	9	18s
CBA	0.337	0.333	11	8s	11	38s
BCD	0.511	0.500	10	2s	10	6s
DCB	0.530	0.500	8	1s	8	2s
ABCB	0.191	0.166	16	1m	13	22m
ABAB	0.123	0.111	16	5m	15	68m
ACCA	0.385	0.333	15	32s	13	8m
CBAC	0.329	0.250	11	11s	11	1m
BBCB	0.167	0.166	17	7m	15	88m
AAAA	0.159	0.142	18	19m	15	326m
ABCD A	0.217	0.200	20	3m	17	150m
DCBA D	0.337	0.333	13	7s	13	1m
AABB C	0.119	0.111	24	409m		
BBCC D	0.288	0.250	17	1m		
CCDD A	0.600	0.500	10	1s	10	1s
DDAA B	0.219	0.200	13	31s	13	5m

Table 3.2: Experimental results

To produce different examples for my experiments, I randomly combined up to five modules (Table 3.2). These pipelines do not perform a useful function, but their complexity is representative.

The "ideal thruput" column lists R as computed by the Ideal-Throughput algorithm (Figure 3.7) for each example.

I run the greedy (Figure 3.11) and exact (Figure 3.10) algorithms are with the slightly smaller throughput listed in column "used thruput." This a small integer ratio (see Section 3.6). The solution, i.e., the configuration with the minimum total FIFO capacity, and the running times are shown in the last four columns.

All the results (throughput, total FIFO size, and running time) vary substantially. In general, shorter pipelines and those containing simpler modules have a higher throughput and smaller FIFOs; the algorithms' running time is also faster.

The results clearly show module sequence is important, i.e., the interaction of adjacent modules is a critical factor that cannot be ignored for an accurate analysis.

3.9 Conclusions

I addressed worst-case performance analysis and FIFO sizing for pipelines of modules with data-dependent throughput, such as those found in network processing devices. I have shown the performance of such a pipeline depends on both its elements and their interaction.

I presented a performance estimation technique for non-interacting modules. It assumes large FIFOs and runs fast enough to be used inside a high-level design exploration loop.

Interaction makes the analysis of a real pipeline, with finite FIFOs, difficult. To answer the problem, I proposed two algorithms, one exact and one heuristic, which use a model checking algorithm to evaluate the feasibility of candidate solutions.

As presented, our FIFO sizing technique uses a simple cost function, the sum of all FIFO sizes. This is a good metric for ASICs, where each FIFO can be custom-sized. However, the algorithm can be trivially modified to use a different cost function; for example, FIFO implementations on FPGA take advantage of the FPGA built-in primitives, which have fixed sizes, so the resource usage do not increase linearly with FIFO size.

The algorithms require substantial running time, but I consider them practical since they can be run in parallel with the detailed logic synthesis of the individual modules.

Chapter 4

Combining Shannon and Retiming

4.1 Introduction

I have presented in Chapter 2 a synthesis method for packet processing pipelines that transforms the PEG high-level specification into a RTL model. The following step is the sequential optimization of the resulting network.

Although any generic sequential optimization flow can be used, it is helpful to augment the flow with a technique that takes advantage of the particularities of the synthesized pipeline. I presently summarize them.

The packet processing pipelines are mainly linear, deeply pipelined sequential networks. The linear topology is a common decision taken by system architects for performance reasons. An efficient pipeline consists of a large number of stages because latency is irrelevant compared to the main performance issue, the throughput; this also follows the FPGA-specific issues described in Section 1.1.1.

I have argued in Section 1.7 that feedback loops are the main performance bottlenecks for deeply pipelined circuits. Such loops include handshaking signals between the pipeline stages and are usually multicycle. Therefore, optimizing the combinational logic outside of the sequential context is of little help. I have also argued that a retiming step is mandatory for obtaining good performance for this class of circuits.

The class of algorithms which address this issue is known as R&R (retiming and resynthesis). R&R is a general term as "shortening the critical paths" for the combinational networks, but it considers instead "shortening the critical loops." My proposed method is an instance of this generic technique. I will review soon some relevant R&R algorithms.

4.1.1 Brief description

Logic synthesis procedures typically consist of a collection of algorithms applied in sequence that make fairly local modifications to a digital circuit. Usually, these algorithms take small steps through the solution space, i.e., by making many little perturbations of a circuit, and do not take into account what their successors can do to the circuit. Such an approach, while simple to code, often leads to sub-optimal circuits.

In this chapter, I propose a logic synthesis procedure that considers a post-retiming step while resynthesizing an entire logic circuit using Shannon decomposition to add speculation. The result is an efficient procedure that produces faster circuits than performing Shannon decomposition and retiming in isolation.

My procedure is most effective on sequential networks that have a large number of registers relative to the amount of combinational logic. This class includes high-performance pipelines. Retiming [LS91] is usually applied to such circuits, which renders the length of purely combinational paths nearly irrelevant since retiming can divide or merge such paths among multiple clock cycles to increase the clock rate. However, because retiming cannot change the number of registers on a sequential cycle—a loop that passes through combinational logic and one or more registers—the depth of the combinational logic along sequential cycles becomes the bottleneck.

Shannon decomposition provides a way to restructure logic to hide the effects of late-arriving signals. This is done by duplicating a cone of logic, feeding constant 1's and 0's into the late-arriving signal, and placing a (fast) two-input multiplexer on the output of the two cones. In the case of the packet processing pipelines, feedback loops usually contain simple (i.e., one bit) late-arriving signals: the control and status signals between pipeline stages. This makes Shannon decomposition a very attractive choice for reducing the length of these cycles.

Following Shannon decomposition with retiming can greatly improve overall circuit performance. Since Shannon decomposition can move logic out of sequential loops, a subsequent retiming step can better balance the logic to reduce the minimum clock period, giving a more efficient circuit.

Combining Shannon decomposition with retiming is a known manual design technique, albeit seldom applied because of its complexity. To my knowledge mine is the first automated algorithm for it.

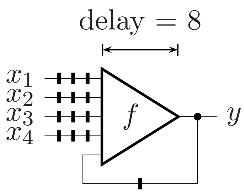
4.1.2 An example

Before giving a formal description of the proposed algorithm, I will provide the main intuition on the technique with the help of an example.

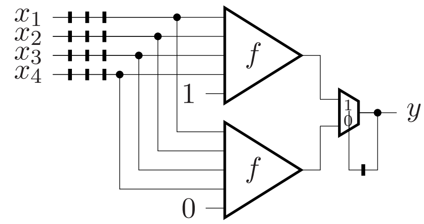
In the sequential circuit of Figure 4.1a, I assume that the combinational block f has delay 8 from any input to the output, so the minimum period of this circuit is 8. The arrival times on all f's inputs is 0; all inputs belong to a combinational critical path since they have zero slack; however, only one input belongs to a sequential critical loop.

The designer put three registers on each input hoping that retiming would distribute them uniformly throughout f to decrease the clock period; this situation often appears in a pipeline. Unfortunately, the feedback loop from the output of f to its input prevents retiming from improving the period below the combinational length of the loop, 8, since retiming cannot change the number of registers along it.

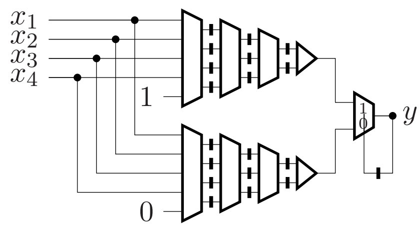
Applying Shannon decomposition to this circuit can enable retiming. Figure 4.1b illustrates how: I have made two duplicates of the combinational logic block and added a multiplexer to their outputs. While this actually increased the longest combinational



(a) Initial circuit



(b) After Shannon Decomposition



(c) After Shannon Decomposition and Retiming

Figure 4.1: (a) The single-cycle feedback loop prevents retiming from improving this circuit, but applying Shannon decomposition (b) reduces the delay around the loop so that (c) retiming can distribute registers and reduce the clock period.

path to $8 + 1 = 9$ (throughout this chapter, I assume multiplexers have unit delay), it greatly reduced the combinational delay around the cycle to the delay of only the mux: 1. This enables retiming to pipeline the slow combinational block to produce the circuit in Figure 4.1c, which has a much shorter clock period of $(1/4)(8+1) = 2.25$.

The main strength of my algorithm is its ability to consider a later retiming step while judiciously selecting where to perform Shannon decomposition. For example, a decomposition algorithm that did not consider the effect of retiming would reject the transformation in Figure 4.1b because it made the circuit both larger and slower.

procedure Restructure(S, c)
  feasible arrival times $\leftarrow$ Bellman-Ford(S, c)
  if feasible arrival time computation failed **then**
    return FAIL
  Resynthesize(S, c, feasible arrival times)
  Retime(S)
  return S

Figure 4.2: My algorithm for restructuring a circuit S to achieve a period c

4.1.3 Overview of the algorithm

My algorithm takes a network S and a target clock period c and uses resynthesis and retiming to produce a circuit with period c if one can be found, or returns failure.

Figure 4.2 shows the three phases of my algorithm. In the first phase, "Bellman-Ford" (shown in Figure 4.3 and described in detail in Section 4.2), I consider all possible Shannon decompositions by examining different ways of restructuring each node. This procedure vaguely resembles technology mapping in that it considers replacing each gate with one taken from a library, but does so in an iterative manner because it considers circuits with (sequential) loops. More precisely, the algorithm attempts to compute a set of feasible arrival times for each signal in the circuit that indicate that the target clock period c can be achieved after resynthesis and retiming.

If the smallest such c is desired, my algorithm is fast enough to be used as a test in a binary search that can approximate the lowest possible c. However, using the smallest possible c is not the only option: using greater values for c may lead to smaller circuits: my algorithm can be used to explore some performance/area tradeoffs.

In the second phase ("Resynthesize," described in Section 4.3), I use the results of this analysis to resynthesize the combinational gates in the network, which is non-trivial because to conserve area, I wish to avoid the use of the most aggressive (read: area-consuming) circuitry everywhere but on the critical paths.

As it can be seen in the example of Figure 4.1, the circuit generated after the second phase may have worse performance than the original circuit. I apply classical retiming to the circuit in the third phase, which is guaranteed to produce a circuit with period c.

In Section 4.4, I present experimental results that suggest my algorithm is efficient and can produce a substantial speed improvement with a minimal area increase on half of ISCAS89 circuits I tested; my algorithm is unable to improve the other half. However, this percentage is better for circuits taken from packet processing pipelines.

4.1.4 Related work

The spirit of my approach is a fusion of Pan's technique for considering retiming while performing resynthesis [Pan99] with the technology-mapping technique of Lehman et al. [LWGH97] that implicitly represents many different circuit structures. However, the details of my approach differ greatly. Unlike Pan, I consider a much richer notion of arrival time due to my considering many circuit structures simultaneously, and my resynthesis technique bears only a passing resemblance to classical technology mapping as my "cell library" is implicit and I consider re-encoding signals beyond simple inversion.

Performance-driven combinational resynthesis is a mature field. Singh et al.'s tree height reduction [SWBSV88] is typical: it optimizes critical combinational paths at the expense of non-critical ones. Along similar lines, Berman et al. [BHLT90] propose the generalized select transform (GST). Like GST, my technique employs Shannon decomposition, but also considers the effect of retiming. Other techniques include McGeer's generalized bypass transform (GBX) [MBSVS91], which exploits certain types of false paths, and Saldanha's exact sensitization of critical paths [SHM$^+$94], which makes corrections for input patterns that generate a late output.

My algorithm employs Leiserson and Saxe's Retiming [LS91], which can decrease the minimum period of a sequential network by repositioning registers. This commonly-used transformation cannot change the number of registers on a loop; my work employs Shannon decomposition to work around this.

Sequential logic resynthesis has also attracted extensive attention, such as the work of Singh [Sin92]. Malik et al. [MSBSV91] combine retiming and resynthesis (R&R). Pan's approach to R&R [Pan99] is a superset of mine, but my restriction to Shannon decomposition allows me to explore the design space more systematically.

Hassoun et al.'s [HE96] architectural retiming mixes retiming with speculation and prediction to optimize pipelines; Marinescu et al.'s technique [MR01] proposes using stalling and forwarding. Like me, they identify critical cycles as a major performance issue, but they synthesize from high-level specifications and can make architectural decisions. My work trades this flexibility for more detailed optimizations.

4.2 The First Phase: Computing Arrival Times

While for a combinational network arrival times can be easily computed by classical static timing analysis, the problem becomes harder if retiming is considered. Intuitively, to compute arrival times I have to also consider multi-cycle paths and sequential loops, as the current position of the registers can be changed after retiming.

To account for this, I use a modified Bellman-Ford algorithm (Figure 4.3) instead of classical static timing analysis to determine whether the fastest circuit I can find can be retimed so as to achieve period c. If the first phase is successful, it produces as a side-effect arrival times that guide my resynthesis algorithm, which I describe in Section 4.3.

4.2.1 Basics

My algorithm operates on sequential circuits that consist of combinational nodes and registers. Formally, a sequential circuit is a directed graph $S = (V, E)$ with vertices $V = PI \cup PO \cup N \cup R \cup \{\text{spi}, \text{spo}\}$. PI and PO are the primary inputs and outputs

```
function BellmanFord(S,c)
  for each register n in S do
    d(n) ← −c
  for each vertex n in S do
    fat(n) ← {(−∞)}
  fat(spi) ← {(0)}
  for i = 0 to max-iterations do
    for each vertex n in S in topological order do
      Relax(n)
    if there exists (t) in fat(spo) such that t > c then
      return FAIL
    if there were no changes in this iteration then
      return fat
  return FAIL

procedure Relax(n)
  F ← ∅
  for each p ∈ fanin(n) do
    T ← the arrival times of s_0-encoded fanins ≠ p
    for each t ∈ fat(p) do
      s_i ← the encoding of t
      for o = 0 to i do
        add AT(⟨s_i, s_i, s_o⟩, t, T, d(n)) to F
        add AT(⟨s_i, s_{i+1}, s_o⟩, t, T, d(n)) to F
      add AT(⟨s_i, s_{i+1}, s_{i+1}⟩, t, T, d(n)) to F
  while there are p, q ∈ F s.t. p ⪯ q and p ≠ q do
    remove q from F
  if F ≠ fat(n) then
    fat(n) ← F
```

Figure 4.3: My Bellman-Ford algorithm for computing feasible arrival times

respectively; N are single-output combinational nodes; R are the registers; spi and spo are two super-nodes connected to and from all PI and PO respectively. The edges $E \subset V \times V$ model the interconnect: $\mathrm{fanin}(n) = \{n' | (n', n) \in E\}$. I assume S has no combinational cycles and each vertex is on a path from spi to spo. I define weights $d : V \to \mathbb{R}$ that represent the following

$$d(n) = \begin{cases} \text{arrival time (from clock)} & n \in PI \\ \text{delay of logic} & n \in N \\ \text{required time (to clock)} & n \in PO \\ 0 & n \in R \cup \{\mathrm{spi}, \mathrm{spo}\} \end{cases}$$

Arrival times are computed in a topological order on the combinational nodes:

$$\mathrm{at}(n) = d(n) + \max_{n' \in \mathrm{fanin}(n)} \mathrm{at}(n') \tag{4.1}$$

4.2.2 Shannon decomposition

Let $f : \mathbb{B}^p \to \mathbb{B}$ be the Boolean function of a combinational node n with p inputs and let $1 \leq k \leq p$. Then

$$\begin{aligned} f(x_1, x_2, ..., x_p) &= x_k f_{x_k} + \overline{x_k} f_{\overline{x_k}} \qquad \text{where} \\ f_{x_k} &= f(x_1, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_p) \quad \text{and} \\ f_{\overline{x_k}} &= f(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_p). \end{aligned}$$

This Boolean property, due to Shannon, has an immediate consequence: modifying a node as shown in Figure 4.4 leaves its function unchanged even though its input-to-output delays change. This is known as the Shannon or generalized select transform [BHLT90].

My algorithm relies on the fact that the arrival time $\mathrm{at}(n)$ may decrease if x_k arrives later than all other x_i ($i \neq k$):

$$\mathrm{at}(n) = \max\{\mathrm{at}(f_{\overline{x_k}}), \mathrm{at}(f_{x_k}), \mathrm{at}(x_k)\} + d_{\mathrm{mux}}$$

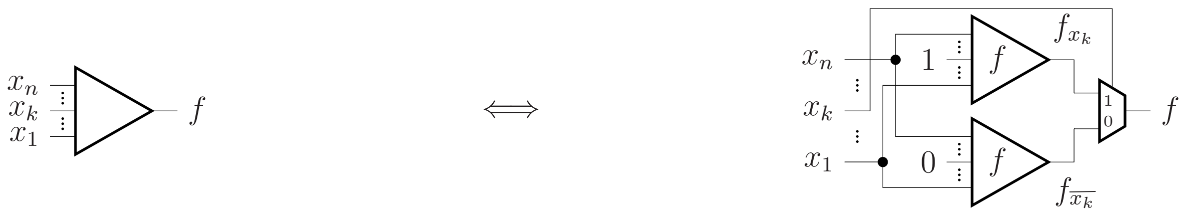


Figure 4.4: Shannon decomposition of f with respect to x_k

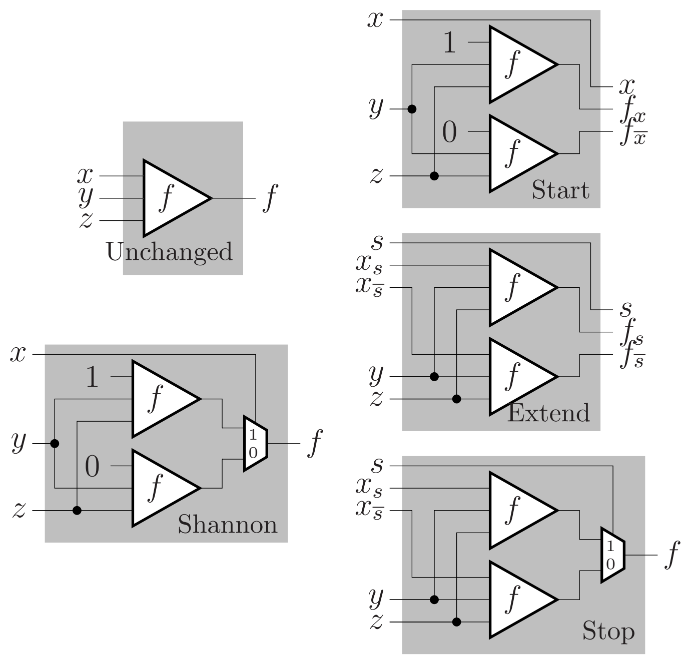
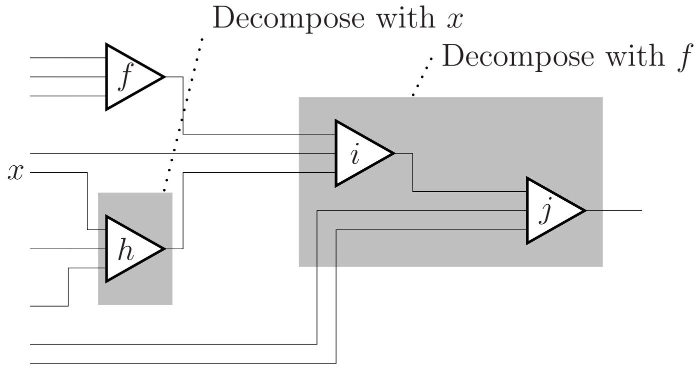
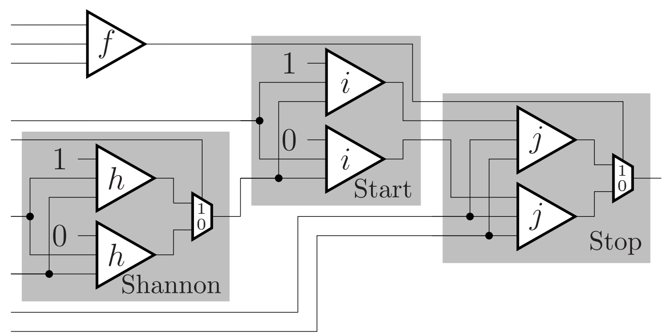


Figure 4.5: The basic Shannon "cell library"

Since the circuit must compute both $f_{\overline{x_k}}$ and f_{x_k}, the area typically increases. Intuitively, this is speculation: I start computing f before knowing x_k, thus both two possible scenarios are considered; the correct one is selected after when x_k becomes available.



(a) Initial circuit



(b) After Shannon transformation

Figure 4.6: Shannon decomposition through node replacement.

4.2.3 Shannon decomposition as a kind of technology mapping

A key to my technique is the ability to consider many different resynthesis options simultaneously. My approach resembles technology mapping in that I consider replacing each node in a network with one taken from a cell library. Figure 4.5 is the beginnings of my library for characterizing multiple Shannon decompositions (my full algorithm considers a larger library that builds on this one—see Section 4.2.5). *Unchanged* leaves the node unchanged and *Shannon* bypasses one of the inputs using Shannon decomposition. Both of these are local changes; the *Start* variant begins a Shannon decomposition that can either be extended by *Extend* or terminated by *Stop*.

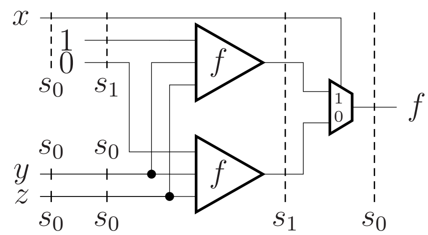


Figure 4.7: The Shannon transform as redundant encoding.

While the *Unchanged* and *Shannon* cells can be used in arbitrary places, the three-wire output of a *Start* cell can only feed the three-wire input of an *Extend* or *Stop* cell. Furthermore, to minimize node duplication, I only allow a single Shannon-encoded input per node, so at most one input to an *Extend* node may be *Start* or *Extend*.

Figure 4.6 illustrates how this works. Figure 4.6a shows the two Shannon transforms I wish to perform, one involving a single node, the other involving two. I replace node h with *Shannon*, node i with *Start*, and node j with *Stop*. Figure 4.6b shows the resulting circuit, which embodies the transformation I wanted.

4.2.4 Redundant encodings and arrival times

Using Shannon decomposition to improve circuit performance is a particular case of the more general idea of using redundant encodings to reduce circuit delay. The main challenge in producing a fast circuit is producing the information later gates need as early as possible. There is not much flexibility when a single bit travels down a single wire, but using multiple wires to transmit a single bit might allow the sender to transmit partial information earlier. This can enhance performance if the downstream computation can be restructured so it performs most of the computation using only partial data and then quickly calculating the final result using the remaining late-arriving data.

Figure 4.7 illustrates this idea on the Shannon transformation of a single node. On the left, inputs x, y, and z are each conveyed on a single wire, a trivial encoding we label s_0. Just after that, however, I choose to reencode x using the three-wire encoding s_1, in which one wire selects which of the other two wires actually carries the value. This is the basic Shannon-inspired encoding. On the left side of Figure 4.7 this "encoding" step amounts to adding two constant-value wires and interpreting the original wire (x) as a select. However, once I feed this encoding into the two copies of f, the result is more complicated. Again, I am using the s_1 encoding with x as the select wire, but instead of two constant values, the two other wires carry f_x and $f_{\overline{x}}$. On the right of Figure 4.7, I use a two-input multiplexer to decode the s_1 encoding into the single-wire s_0 encoding.

Using such a redundant encoding speeds-up the circuit if the x signal arrives much later than y or z. When I re-encode a signal, the arrival times of its various components can differ, which is the source of the speed-up. For example, at the s_1 encoding of x, the x wire arrives later while the two constant values arrive instantly.

A central trick in the first phase of our algorithm is the observation that only arrival times matter when considering which cell to use for a particular node. In particular, the detailed topology of the circuit, such as whether a Shannon decomposition had been used, is irrelevant when considering how best to resynthesize a node. Only arrival times and the encoding of each arriving signal matter.

4.2.5 My family of Shannon encodings

While a general theory of re-encoding signals for circuit resynthesis could be (and probably should be) developed, in this chapter I restrict my focus to a set of encodings derived from Shannon decompositions that aim to limit area overhead. In particular, evaluating a function with a single encoded input only ever requires us to make two copies of the original function.

The basic cell library of Figure 4.5 works well for most circuits, but some transformations demand the richer library I describe in this section. For example, the larger family is required to convert a ripple-carry adder into a carry-select adder.

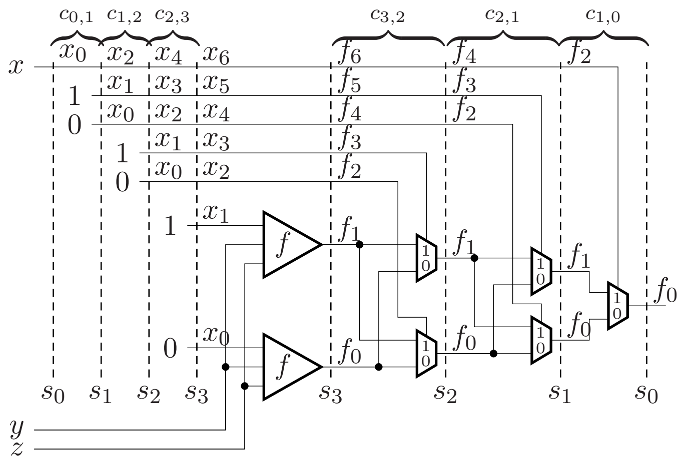


Figure 4.8: Encoding x, evaluating $f(x, y, z)$, and decoding the output for the s_0, s_1, s_2, and s_3 codes.

Technically, I define an encoding as a function $e : \mathbb{B}^k \to \mathbb{B}$ that maps information encoded on k wires to a single bit. My family of Shannon-inspired encodings $S = \{s_0, s_1, \ldots\}$ are defined recursively:

$$s_i = \begin{cases} x_0 \mapsto x_0 & \text{if } i = 0, \\ x_0, \ldots, x_{2i} \mapsto & \\ \quad s_{i-1}(\overline{x_2}x_0 + x_2x_1, \overline{x_3}x_0 + x_3x_1, x_4, \ldots, x_{2i}) & \text{otherwise.} \end{cases}$$

The first few such encodings are listed below.

$$\begin{aligned} s_0 &= \quad x_0 \mapsto x_0 \\ s_1 &= \quad x_0, x_1, x_2 \mapsto \overline{x_2}x_0 + x_2x_1 \\ s_2 &= \quad x_0, x_1, x_2, x_3, x_4 \mapsto \overline{x_4}(\overline{x_2}x_0 + x_2x_1) + x_4(\overline{x_3}x_0 + x_3x_1) \end{aligned}$$

Figure 4.8 shows a circuit that takes three inputs, x, y, and z, encodes x to the s_1, s_2, and s_3 codes before evaluating $f(x, y, z)$ with the s_3-encoded x, then decodes the result back to the single-wire s_0 encoding. While this circuit as a whole is worse than the simple Shannon decomposition of Figure 4.7, subsets of this circuit provide a way to move between encodings.

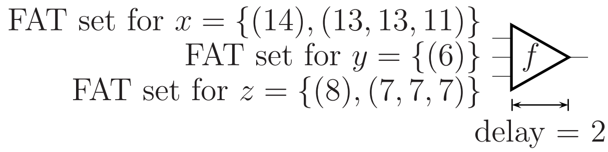
I think of the layers of Figure 4.8 as "Shannon codecs"—small circuits that transform an encoding s_a to s_b. I write $c_{a,b}$ for such a codec circuit. For $a < b$, $c_{a,b}$ just adds pairs of wires connected to constant 0's and 1's. For $a > b$, $c_{a,b}$ consists of some multiplexers.

I chose my family of Shannon encodings so that evaluating a functional block with a single input with a higher-order encoding only ever requires two copies of the functional block. Figure 4.8 shows this for the s_3 case; other cases follow the same pattern.

In general, I only allow a single encoded input to a cell (all others are assumed to be single wires, i.e., s_0-encoded). To evaluate a function f, I make two copies of it, feed the non-encoded signals to each copy, feed x_0 and x_1 from the encoded input signal to the two copies, which compute f_0 and f_1 of the encoded output signal, and pass the rest of the x_i from the encoded signal directly through to form f_2, etc. The *Extend* cell in Figure 4.5 is a case of this pattern for encoding s_1.

I consider one important modification of this basic idea: the introduction of "codec" stages at the inputs and outputs of a cell. Following the rules suggested in Figure 4.8, I could place arbitrary sequences of encoders and decoders on the inputs and output of a functional block, but to limit the search space, I only consider cells that place one encoder on a single input and one or more decoders on the output. For instance, the *Stop* cell in Figure 4.5 is derived from the *Extend* cell by appending a decoder to its output. Similarly, the *Start* cell consists in an encoder plus the *Extend* cell. Finally, the *Shannon* cell adds both an encoder and a decoder to the *Extend* cell.

Neither the number of inputs of a cell nor the index of the encoded input matter to the computation of arrival times since I assume each node has the same delay from each input to the output. Therefore, I identify a cell with a triplet of encodings: $\langle s_i, s_f, s_o \rangle$ where s_f denotes the encoding of the encoded input fed to the combinational logic, s_i denotes the encoding of the input at the interface of the cell (which



	Unchanged	**Shannon**	**Start**	**Stop**	**Extend**
	$\langle s_0, s_0, s_0 \rangle$	$\langle s_0, s_1, s_0 \rangle$	$\langle s_0, s_1, s_1 \rangle$	$\langle s_1, s_1, s_0 \rangle$	$\langle s_1, s_1, s_1 \rangle$
x	(16)	(15)	(10,10,14)	(16)	(15,15,11)
y	(16)	(17)	(16,16,6)		
z	(16)	(17)	(16,16,8)	(17)	(16,16,7)

Figure 4.9: Computing feasible arrival times for a node f

may differ from s_f because of an encoder: $i = f$ or $i = f - 1$), s_o denotes the encoding of the output of the cell (which may differ from s_f because of decoders: $0 \leq o \leq f$).

The columns of the table in Figure 4.9 correspond to the triplets for each of the five cells in the basic Shannon cell library (Figure 4.5).

My algorithm considers many additional cell types, which arise from higher degree encodings. In theory, there is an infinite number of such cells, but in practice, only a few more are interesting. The circuit in Figure 4.8 illustrates some higher-degree encodings, corresponding to $\langle s_0, s_3, s_0 \rangle$. My algorithm would never generate this particular cell in its entirety because if s_f is s_3, then s_i may be either s_3 or s_2, not s_0. It could, however, generate the cell with an s_2-encoded input, i.e., $\langle s_2, s_3, s_0 \rangle$.

4.2.6 Sets of feasible arrival times

The first phase of my algorithm (Figure 4.3) attempts to compute a set of feasible arrival times that will guide me in resynthesizing the initial circuit into a form that can be retimed to give us the target clock period c.

In classical static timing analysis, the arrival time at the output of a gate is a single number: the maximum of the arrival times at its input plus the intrinsic delay of the gate (see equation (4.1)). In my setting, however, I represent the arrival time of a signal with a set of tuples, each representing the arrival of the signal under a particular encoding. Considering sets of arrival times allows me to simultaneously consider different circuit variants.

My arrival-time tuples contain one real number for each wire in the encoding. Since no two encodings use the same number of wires, the arity of the tuple effectively defines the encoding. For example, an arrival time 3-tuple such as $(2, 2, 3)$ is always associated with encoding s_1 since it is the only one comprised of three wires; the 5-tuple (3,3,5,5,6) corresponds to s_2, etc.

The example in Figure 4.9 illustrates how I compute the set of feasible arrival times (FATs) at a node f through brute-force enumeration. The code for this appears in lines 17–24 of Figure 4.3.

The FAT sets shown in Figure 4.9 indicate that input x can arrive as early as 14 as an unencoded (s_0) signal, or at times 13, 13, and 11 for the three wires in an s_1-encoded signal. Similarly, y can arrive as early as 6 and z may arrive as early as 8 in s_0 form or $(7, 7, 7)$ in s_1 form.

Considering only the cell library of Figure 4.5, the table in Figure 4.9 shows how I enumerate the possible ways f can be implemented. The rows of this table correspond to the iteration over the fanins of the node—line 17 in Figure 4.3. The columns are produced from the iterations over input encoding from the fanin (line 19) and output encoding (line 21).

The *Unchanged* case is considered first for each input. Since every input has the s_0 encoding for this case, I obtain the same arrival time for each input. Here, this is $14 + 2 = 16$, which is the earliest arrival time of x, the latest arriving s_0 signal, plus 2, the delay of f.

The *Shannon* case is considered next for each input. This produces an s_0-encoded output, so the resulting arrival times are singletons. For example, if y is the s_1-encoded input, the longest path through the cell starts at x (time 14), passes through

f (time 16), and goes through the output mux (time 17; I assume muxes have unit delay). This gives the (17) entry in the *Shannon* column in the y row.

Next, the algorithm considers a *Start* cell: one that starts a Shannon decomposition at each of the three inputs. For example, if I start a Shannon decomposition with s_0 input x, the two s_0 inputs for y and z are passed to copies of f (this is the structure of a *Start* cell) and the s_0 x input is passed to the three-wire output (*Start* cells produce an s_1-encoded output). The outputs of the two copies of f become the x_0 and x_1 outputs of the cell and arrive at time $8+2=10$ because z arrives at time 8 and f has a delay of 2. The s_0 x input is passed directly to the output, which I assume is instantaneous, so it arrives at the output at time 14. Together, these produce the arrival time tuple $(10, 10, 14)$, the entry in the x row in the *Start* column.

The *Stop* and *Extend* cases require one of their inputs to be s_1-encoded. Since no such encoding for y is available (i.e., there is no triplet in its arrival time set), I do not consider using y as this input, hence the *Stop* and *Extend* columns are empty in the y row.

In Figure 4.3, the AT function (called in lines 22–24) is used to compute the arrival time for each of these cases. In general, the arrival time $\mathrm{AT}(\langle s_i, s_f, s_o\rangle, t, T, d(n))$ of a variant of node n with shape $\langle s_i, s_f, s_o\rangle$ depends on the arrival time t of the encoded input, the set of arrival times T of the other inputs, and the delay $d(n)$ of the combinational logic. It is computed using regular static timing analysis, i.e., equation (4.1) for each of the components of the cell. I do not present pseudocode for the AT function since it is straightforward yet fuzzy.

Even this simple example produced over thirteen arrival time tuples from five (Figure 4.9 does not list the higher-order encodings my algorithm also considers); such an increase is typical. Fortunately, most are not interesting as they obviously lead to slower circuits. Since in this phase I am only interested in the fastest circuit, I can discard most of the results of this exhaustive analysis to greatly speed the analysis of later nodes—I discuss this below.

4.2.7 Pruning feasible arrival times

As Figure 4.9 showed, a node can usually be resynthesized in many ways. Fortunately, most variants are not interesting because they are slower than others. In this section, I describe my policy for discarding implementations that are never faster than others since in the first phase I am only interested in whether there is a circuit that will run with period c or faster. In the second phase, I will use slower cells off the critical path to save area — see Section 4.3.

If p and q are two arrival times at the output of a node, then I write $p \preceq q$ if an implementation of the circuit where the arrival time of the node is q cannot be faster (i.e., admit a smaller clock period) than an implementation where the arrival time of the node is p. Consequently, if I find cell implementations that produce p and q, I can safely ignore the implementation that produces q without fear of erroneously concluding that a particular period is unattainable. My FAT set pruning operation removes all such dominated arrival times from the set of arrival times produced as described above. In Figure 4.3, this pruning is performed on lines 25–26.

Practically, I implement a conservative version of the $\preceq$ relation described above because the precise condition is actually a global property. If a node is off the critical path, for example, it is probably the case that more arrival times could be pruned than my implementation admits, but practically I find my pruning works quite well in practice.

For s_0-encoded signals, the ordering is simple: the faster-arriving signal is superior.

For two arrival times for signals encoded in the same way, the ordering is piecewise: if every component is faster, then the arrival time is superior, otherwise the two arrival times are incomparable because a later Shannon decomposition might be able to take advantage of the differential in ways I cannot predict locally.

For arrival times corresponding to different encodings, the argument is a little more subtle. Consider Figure 4.8. In general, only the first two wires in an encoded signal are ever fed directly to functional blocks (e.g., x_0 and x_1 in the s_3 encoding

$$(10, 10, 14)$$

$$(15, 15, 11)$$

$$(15) \preceq (16) \preceq (17)$$

$$(16, 16, 6) \preceq (16, 16, 7) \preceq (16, 16, 8)$$

Pruned FAT set: $\{(15), (10, 10, 14)\}$

Figure 4.10: Pruning the feasible arrival times from Figure 4.9.

in Figure 4.8 and the others are fed to a collection of multiplexers. The wires in higher-level encodings must eventually meet more multiplexers than those in lower-level encodings, so a lower-level encoding whose elements are strictly better than the first elements in a higher-level encoding is always going to be better.

Concisely, my choice of $\preceq$ (our pruning rule) is, for two arrival times $p = (p_0, p_1, ..., p_n)$ and $q = (q_0, q_1, ..., q_m)$ for potentially different encodings,

$$p \preceq q \text{ iff } n \leq m \text{ and } p_0 \leq q_0, p_1 \leq q_1, \ldots, p_n \leq q_n.$$

Figure 4.10 illustrates how pruning reduces the size of the FAT set computed in Figure 4.9. The singleton (15) dominates most of the other arrival times (some of which appear more than once in Figure 4.10—remember that I ultimately operate on FAT sets, not the table of Figure 4.9), but $(10, 10, 14)$ is not comparable with (15) (for a singleton to dominate a triplet, the first value in the triplet must be greater or equal).

That the eleven arrival times computed in Figure 4.9 (only eight are distinct) boil down to only two interesting ones (Figure 4.10) is typical. Across all the circuits I have analyzed, I find pruned FAT sets seldom contain more than four elements. This is a key reason my algorithm is efficient: although it is considering many circuit structures at once, it only focuses on the fastest ones.

4.2.8 Retiming

At this point, I have described my technique for restructuring circuits based on Shannon decomposition: I consider re-implementing isolated nodes with variants taken from a virtual library (Figure 4.5 shows a subset) and discussed how I can represent these variants as feasible arrival time (FAT) sets. I want now to choose variants so as to improve the effects of a later retiming step.

Retiming [LS91] follows from noting that moving registers across combinational nodes preserves the circuit functionality. Retiming tries to move registers to reduce long (critical) combinational paths at the expense of short (non-critical) ones. However, it cannot decrease the total delay along a cycle.

Let $\mathrm{ret}(S)$ be the minimum period achievable through retiming. If $d_{\mathcal{C}}$ and $r_{\mathcal{C}}$ are the combinational delay and the number of registers of the cycle $\mathcal{C}$ in S then $\mathrm{ret}(S) \geq \; d_{\mathcal{C}}/r_{\mathcal{C}}$. Similarly, if $\mathcal{P}$ is a path from spi to spo having $r_{\mathcal{P}}$ registers and of combinational delay $d_{\mathcal{P}}$ then $ret(S) \geq d_{\mathcal{P}}/(r_{\mathcal{P}}+1)$. Thus, $\mathrm{ret}(S) \geq \mathrm{lb}(S)$ where

$$\mathrm{lb}(S) = \max\left(\max_{\mathcal{C}\in \mathrm{cycles}(S)} \frac{d_{\mathcal{C}}}{r_{\mathcal{C}}}, \max_{\mathcal{P}\in \mathrm{paths}(S,\mathrm{spi},\mathrm{spo})} \frac{d_{\mathcal{P}}}{r_{\mathcal{P}}+1} \right) \tag{4.2}$$

is known as the fundamental limit of retiming.

Classical retiming may not achieve $\mathrm{lb}(S)$. To achieve it in general, I must allow registers to be inserted at precise points inside the nodes. I will assume this is possible (which it is, for example, in FPGAs [TSSV92]), so $\mathrm{ret}(S) = \mathrm{lb}(S)$ holds. I shall thus focus on transforming S to minimize $\mathrm{lb}(S)$.

4.2.9 Using Bellman-Ford to compute arrival times

Computing $\mathrm{lb}(S)$ by enumerating cycles and applying (4.2) is not practical because the number of cycles may be exponential; instead I use the Bellman-Ford single-source shortest path algorithm[1], where the source node is spi.

[1] Bellman-Ford reports if a graph has any negative cycles or not. Only if all cycles are positive, it can compute the shortest paths; it runs in $O(VE)$. Technically, I change signs, so I detect positive

To apply Bellman-Ford, which has no notion of registers, I treat registers as nodes with negative delay: $\forall r \in R, d(r) = -c$, where c is the desired period. Only registers are assigned these artificial delays; the other nodes, i.e., the ones containing normal combinational logic, keep their positive delays $d(n)$ as defined before in Section 4.2.1. Pan [Pan99] also uses this technique.

Now, the total length of a path $\mathcal{P} \in \text{paths}(S)$ becomes $\sum_{n \in \mathcal{P}} d(n) = \sum_{n \in \mathcal{P} \setminus R} d(n) + \sum_{n \in \mathcal{P} \cap R} d(n) = d_{\mathcal{P}} - c \cdot r_{\mathcal{P}}$, where the first term is the delay of the combinational logic, and the second corresponds to the registers.

For any path $\mathcal{P} \in \text{paths}(S, \text{spi}, \text{spo})$, we have

$$c \geq d_{\mathcal{P}}/(r_{\mathcal{P}} + 1) \Leftrightarrow d_{\mathcal{P}} - c \cdot r_{\mathcal{P}} \leq c \Leftrightarrow \sum_{n \in \mathcal{P}} d(n) \leq c.$$

Moreover, any cycle $\mathcal{C} \in \text{cycles}(S)$ is a closed path, so

$$c \geq d_{\mathcal{C}}/r_{\mathcal{C}} \Leftrightarrow d_{\mathcal{C}} - c \cdot r_{\mathcal{C}} \leq 0 \Leftrightarrow \sum_{n \in \mathcal{C}} d(n) \leq 0.$$

Equation (4.2) becomes

$$c \geq \text{lb}(S) \Leftrightarrow \begin{cases} \forall \mathcal{C} \in \text{cycles}(S) & \sum_{n \in \mathcal{C}} d(n) \leq 0 \\ \forall \mathcal{P} \in \text{paths}(S, \text{spi}, \text{spo}) & \sum_{n \in \mathcal{P}} d(n) \leq c \end{cases}.$$

That is, the period $c \geq \text{lb}(S)$ iff no cycle is positive, and the longest path from spi to spo is at most c. The first condition is verified if the Bellman-Ford algorithm converges in a bounded number of iterations. If so, it also gives us $\text{at}(n)$—the longest path between spi and any node n. I verify the second condition by checking if $\text{at}(\text{spo}) \leq c$.

Therefore, $\text{lb}(S)$ can be approximated by binary search on the period c.

To consider the combined effect of our restructuring and retiming, I use a variant of the Bellman-Ford algorithm that uses the FAT computation plus pruning operation

cycles instead of negative and compute the longest path instead of the shortest if all cycles are negative. Note that this is not solving the longest simple-path problem, which allows positive cycles and is known to be NP-complete.

as its central relaxation step (Figure 4.3). The main loop (lines 7–13 in Figure 4.3) terminates when the relaxation has converged to a solution or it has become fairly obvious that no solution will be found. This latter case is actually a heuristic, which I discuss in the next section.

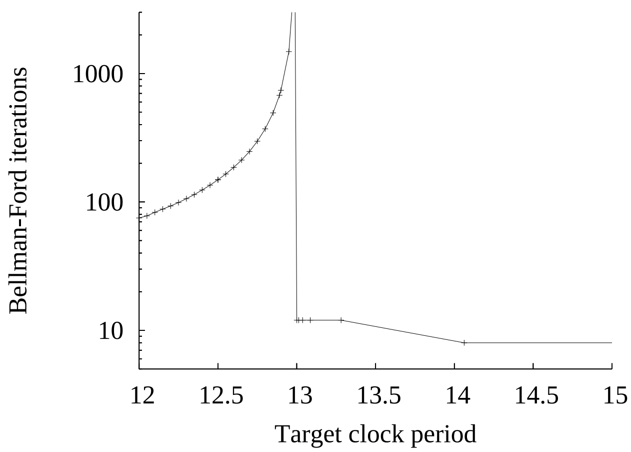
If Bellman-Ford converges, i.e., reaches a fixed-point such that $\mathrm{at(spo)} \leq c$, then there exists an equivalent circuit for which $\mathrm{lb}(S) \leq c$, so, after retiming, c is feasible. To prove this claim, I simply build a circuit using the cell variants I considered during the relaxation procedure. However, such a brute-force construction produces overly large circuits, so instead I use a more careful construction that limits Shannon-induced duplication to critical paths only, the subject of Section 4.3.

I illustrate my brute-force construction on the sample in Figure 4.12. The original circuit appears at the top of the figure. Using a desired period $c = 3$, after a number of steps, depicted below, my augmented Bellman-Ford algorithm converges; this implies a fixed point solution, depicted on the last row, i.e., a FAT set for each node, that is stable under the pruned FAT set computation. Hence, I claim that period $c = 3$ is feasible. In the sample, for simplicity, each gate has delay 2, and the multiplexers to be inserted have delay 1.

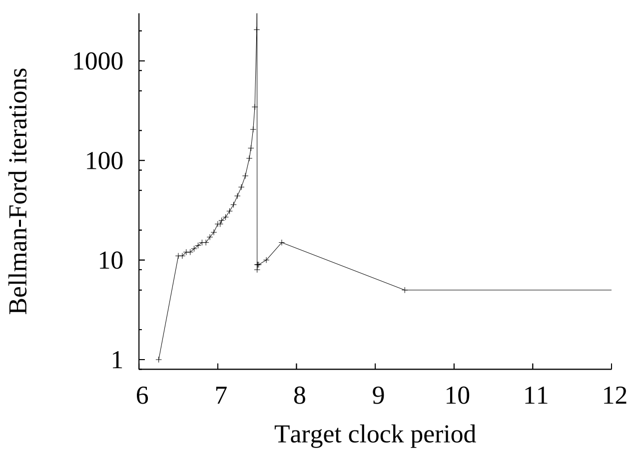
For each node, I build an implementation corresponding to each element of its FAT set; I am free to choose any cell from Figure 4.5 and use any FAT elements at each input, as described in Section 4.2.6.

For example, for node h at the bottom of Figure 4.12, I consider two implementations. These are *Start* and *Shannon* (Figure 4.5), both with g's output as the select. These give arrival times of $(4, 4, 8)$ and (9).

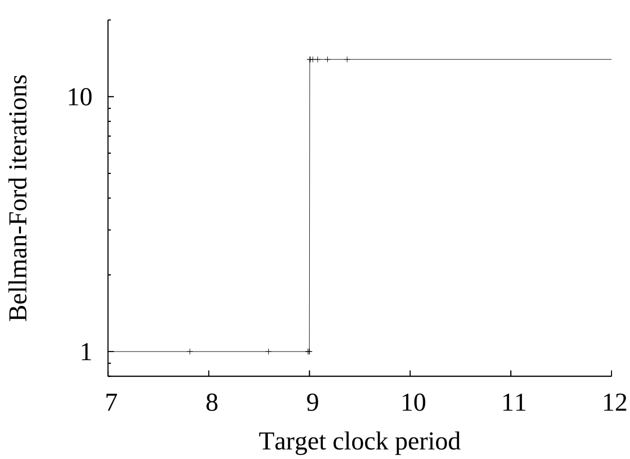
The reconstruction procedure will succeed for each node as a consequence of how I computed the pruned FAT sets during the Bellman-Ford relaxation. If Bellman-Ford converges, the resulting network will have $\mathrm{lb}(S) \leq c$, so I will have a solution after retiming.



(a) s38417 circuit sample



(b) s9234 circuit sample



(c) s13207 circuit sample

Figure 4.11: Iterations near the lowest c

4.2.10 Termination of Bellman-Ford and *max-iterations*

My modified Bellman-Ford algorithm has three termination conditions: two that are exact and one that is a heuristic. The most straightforward is the check for a fixed-point on line 12. This is the only case in which I conclude the relaxation has converged, and is usually reached quickly in practice.

The second termination condition is due to the topology of the circuit graphs. If no fixed point exists, the s_0-encoded arrival time in fat(spo) will eventually become greater than c. This is because any positive cycle (the absence of a fixed point means such a cycle exists) must pass through spo, and along any positive cycle the arrival times must keep increasing during the relaxation procedure. So checking the s_0-encoded signal (there is always exactly one) suffices. This is the check in line 10.

The third termination condition—the hard iteration bound of *max-iterations* on line 7—is a heuristic. I employ such a bound because convergence usually happens quickly while detecting non-convergence can be very slow. It is always safe to terminate the loop earlier, i.e., assume that the rest of iterations, if continued, would have never converged: there is no inconsistency risk, but I may get a sub-optimal solution.

I expect convergence to be fast because of the behavior of the usual Bellman-Ford algorithm. When it converges, it does so in at most n iterations, where n is the number of vertices in the graph[2] because the smallest positive cycle has a length of at most n. However, provided the vertices with positive weights are visited in a topological order (w.r.t. the DAG obtained from the graph by removing the registers), the convergence is in practice much faster. In my adapted Bellman-Ford algorithm, I use such an ordering for the inner loop at lines 8–9. Therefore, I expect fast convergence when the period c is feasible. In particular, the speed of convergence, while depending on

[2]Unfortunately, we do not have a similar result for our variant because of the complex behavior of the FAT set generation and pruning. However, even if I knew a closed-form bound, I would still prefer to use my heuristic early-termination condition because it produces very good results much faster.

the circuit topology, should be essentially independent of the distance between the feasible clock period c tried and the lowest clock period achievable by my technique. However, when c is unfeasible and gets closer to the lowest achievable period, the number of iterations before overflow may increase arbitrarily.

The graphs in Figure 4.11 provide empirical evidence for this. There, I plot iteration counts for three large circuit samples (with *max-iterations* arbitrarily set to 2000; I observe similar behavior across all circuits). The scale is logarithmic. For the first two examples, I do observe that divergence is indeed costly to detect when c gets close to the limit. In the third case, probably because of a tight cycle close to one circuit output, I have no such curve. However, what matters is that I observe that convergence is always very fast. Therefore, it makes sense to choose *max-iterations* to be a small quantity. I choose *max-iterations*=200 and get good results for all the benchmarks, that is to say the algorithm never fails to obtain stable FAT sets because of an early timeout. As a result, if the minimum feasible c is computed by binary search, the result is very close to the true value.

4.3 The Second Phase: Resynthesizing the Circuit

The first phase of the algorithm focused exclusively on performance: it considers only the fastest possible circuit it can find and ignores the others for efficiency. This makes sense since it is trying to establish the existence of a fast-enough circuit, but I would really like to find a fast-enough circuit that is as small as possible. The goal of the second phase of the algorithm, described in this section, is to select a minimal-area circuit that still meets the given timing constraint. I do this heuristically.

The circuit produced by this part of my algorithm is often worse than the original. The circuit in Figure 4.13a happens to have the same minimum period (9) as the original circuit in Figure 4.12. However, $\mathrm{lb}(S)$ as defined by equation (4.2) is 3 and thus after retiming (Figure 4.13b) the minimum period drops to our target of 3.

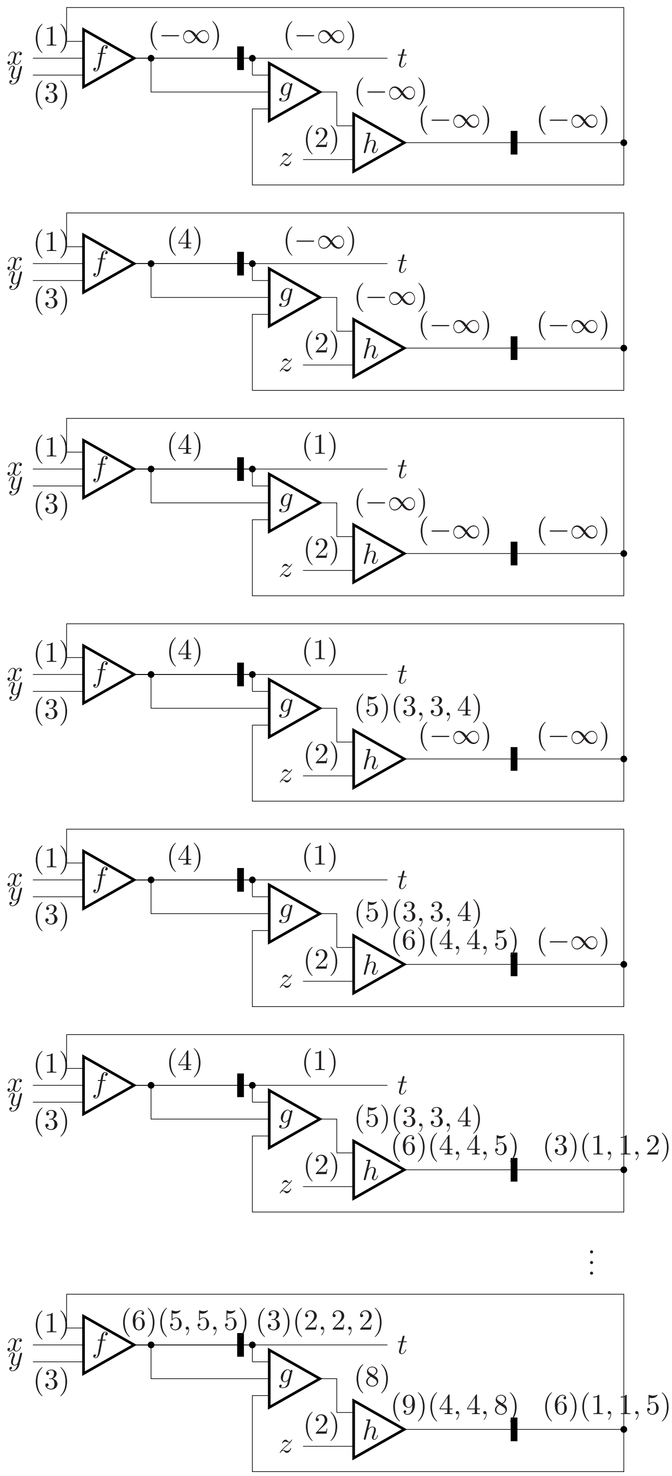
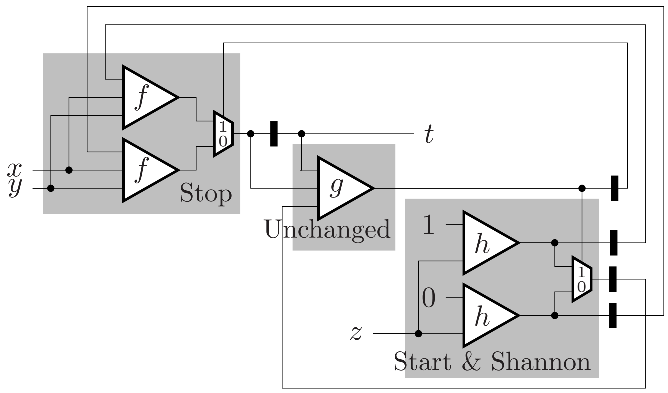
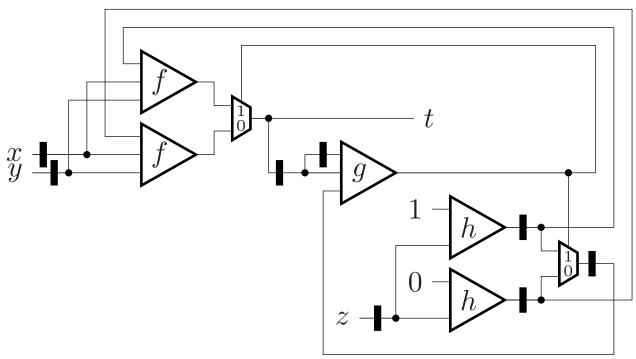


Figure 4.12: Computing feasible arrival times for a circuit with desired period $c = 3$. The topmost circuit is the initial condition; the bottom is the fixed point. From a total of seven relaxation steps, only the first are shown.



(a) After Shannon decomposition

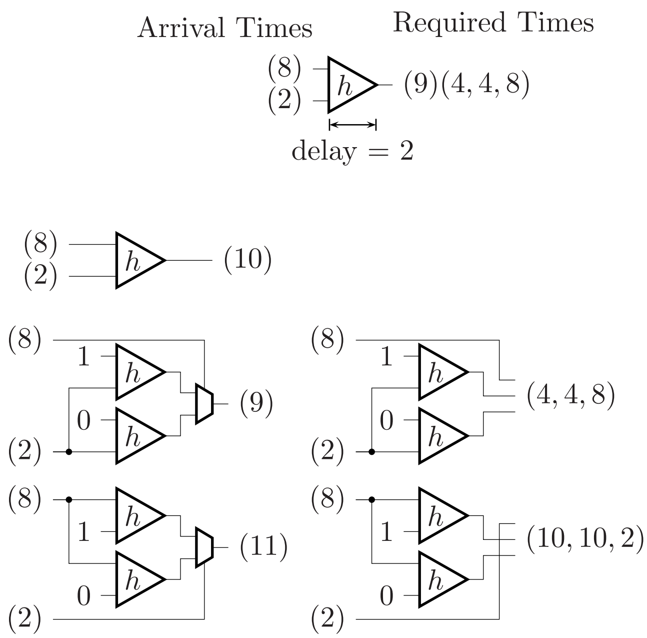


(b) Final retimed circuit

Figure 4.13: (a) The circuit from Figure 4.12 restructured according to the FAT sets. Its period (9) happens to be the same as the original; in general it is different (and not necessarily better). (b) After retiming, the circuit runs with period 3 as desired.

To minimize area under a timing constraint—the main goal of the second phase—I use a well-known scheme from technology mapping: nodes that are not along critical paths are implemented using smaller, slower cells in such a way that the overall minimum clock period is the same.

My construction rebuilds the circuit starting at the primary outputs and register inputs and working backwards to the primary inputs and register outputs. I insist the circuit is free of combinational loops so this amounts to a reverse topological order in the DAG obtained from the graph after removing the registers.



Feasibility Table

(9)	(4, 4, 8)
0	0
1	1
0	0

Figure 4.14: Selecting cells for node h in Figure 4.12.

For each gate, I consider the feasible arrival times of its fanins computed in phase one and the feasible required times of its fanouts, which I compute as part of the reconstruction procedure, as described in Section 4.3.3. Figure 4.14 illustrates this procedure for the h node from Figure 4.12. The feasible required times for primary outputs and registers are exactly the feasible arrival times for these nodes computed in the first phase.

For each gate, I construct a cell (occasionally more than one) that is just fast enough to meet the deadline (i.e., compute the outputs to meet the required times given the arrival times) without being larger than necessary.

A feasible required time (FRT) set for a node is much like a FAT set: it consists of tuples of numbers that describe when each wire in an encoded signal is needed. At each node, I consider different cells for the node that produce the desired encoding (i.e., the arity of the FRT tuple). Since the FAT sets were produced by considering all possible cells at each node, I know that some cell will achieve the desired arrival time. To save area, I select the smallest such cell.

If I am lucky, an *Unchanged* cell suffices, meaning the functional block does not need to be duplicated and there are no additional multiplexers. In Figure 4.13a, node g appears as *Unchanged*.

For node f, the *Stop* cell was selected. Note that the signal from h to f uses an s_1 encoding.

For h, I actually selected two cells: *Start* and *Shannon*. Fortunately, they share logic—the duplicated h logic block. The s_1-encoded output of *Start* goes to f; the s_0 output of *Shannon* is connected to g.

Note that when a register appears on an encoded signal, it is simply replicated on each of its wires. In Figure 4.13a, for example, the register on the output of h became four: one on the output of the mux, an s_0 encoded signal, and three for the s_1 encoded signal: the two outputs from the two h blocks and the selector signal (the output of g). While this may seem excessive, the subsequent retiming step may remove many of these registers.

4.3.1 Cell families

That I need multiple cells for a particular node to meet timing happens often enough that I developed a heuristic to reduce the area in such a case. It follows from a simple observation: many of the cells in Figure 4.5 are structurally similar. In fact, some are subsets of others and share the two copies of the same functional block, so if I happen to be able to use a cell and its subset to implement a particular node, it requires less area than if I use two different cells.

I call cells that only differ by the addition or deletion of multiplexers on the output members of a *family*. By this definition, each cell is a member of exactly one family. In Figure 4.5, there are three such families: *Unchanged* is a family by itself, *Shannon* and *Start* is another family (they both encode one of their inputs), and *Extend* and *Stop* are a third (each takes a single s_1-encoded input). Families with cells with higher order encodings are larger.

To save area, I try to use only cells taken from the same family since each cell in a family can also be used to implement others in the family without making additional duplicates of the node's function.

4.3.2 Cell selection

Working backward from the outputs and register inputs, I repeat the enumeration step from the first phase (e.g., Figure 4.9) at each node to generate the set of all cells that are functionally correct for the node (i.e., have the appropriate input and output encodings). From this set, I try to select a set of cells that are both small and meet the timing constraint at the node. Figure 4.14 illustrates this process for the h node in Figure 4.12.

I consider the cells generated by the enumeration step one family at a time in order of increasing area. Practically, I build a feasibility table whose rows represent cell families and whose columns represent required times. Such a table appears on the right side of Figure 4.14. My goal with this table is to select a minimum number of rows with small area to cover all the required times for the node. I consider rows instead of cells because implementing multiple cells from the same family is only as costly as implementing the largest cell in the family.

An entry in the table is 1 if some cell in the row's family satisfies the required time of the column. To evaluate this, I construct each possible cell for the node (such as those on the left side of Figure 4.14) and calculate the arrival times of the outputs from the arrival times of the inputs. An arrival time satisfies a required time if it corresponds to the same encoding and the arrival time components are less than or equal to the required times. Note that these criteria are simpler than the $\preceq$ relationship used in the pruning.

I select several rows of minimum area that cover all columns (i.e., a collection of cell families that contain cells that are fast enough to meet the timing constraint). As mentioned, there is usually a row that covers all columns, so I usually pick that.

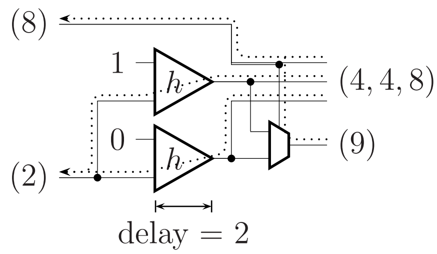


Figure 4.15: Propagating required times from outputs to inputs. Only the critical paths (dotted lines) impose constraints.

Otherwise, I continue to add rows until all columns are covered. Figure 4.14 is typical: the second row covers all columns so I select it.

Selecting a cell in this way is a greedy algorithm that does not consider any later ramifications of each choice. I have not investigated more clever heuristics for this problem, although I imagine some developed for performance-oriented technology mapping would be applicable.

4.3.3 Propagating required times

Once I have selected a set of families that cover all of the required time constraints (i.e., solved the feasibility problem), I construct all cells in these families (sharing logic within each family) and connect them to the appropriate fanouts. In Figure 4.14, I choose the second row and build both cells in that family. Figure 4.15 shows this: a single pair of h nodes are built, but both an s_1-encoded signal is produced (that with required time $(4, 4, 8)$) and an s_0 signal (required time (9)).

At this point, I now have a circuit that includes cells implementing the node I am considering. Because I built them earlier, I know the required times at the inputs to each the fanouts, so I work backward from these times to calculate the required times at the inputs to my newly-created cells. This is simple static timing analysis on a real circuit fragment (i.e., I now only have wires and gates, not cleverly-encoded signals).

Figure 4.15 illustrates how I propagate required times at the outputs of a cell back to the inputs. In this case it is easy because I do not need to consider encoded signals specially. This is standard static timing analysis. For example, the $(4, 4, 8)$ constraint on one of the outputs means that the inputs to each copy of h must be available at time 2 (h has a delay of 2).

Note that in general required times may differ from arrival times because of slack in the circuit. We do not see that in this example since it is small and h is on the critical path, but it does happen quite frequently in larger examples. Again, I take advantage of this to reduce area.

I may now have several required times for each input of a family of cells but the encoding must be the same for a particular input because they are all in the same family. In this case, I simply compute the overall required time of each input by taking the pairwise minimum and place it on the corresponding fanin.

If two or more cell families are built, several required times with different encodings will be placed on the fanins. In this case, the fanin nodes will see the current node as two or more distinct fanouts instead of one. Such complex situations rarely occur in practice.

4.4 Experiments

I implemented my algorithm in C++ using the SIS libraries [SSM+92] to handle BLIF files. My testing platform is a 2.5 GHz, 512 MB Pentium 4 running Fedora Core 3 Linux.

4.4.1 Combinational circuits

While my algorithm is most useful on sequential circuits, it can also be applied to purely combinational circuits. However, classical combinational performance optimization techniques such as the *speedup* function in SIS outperform my technique

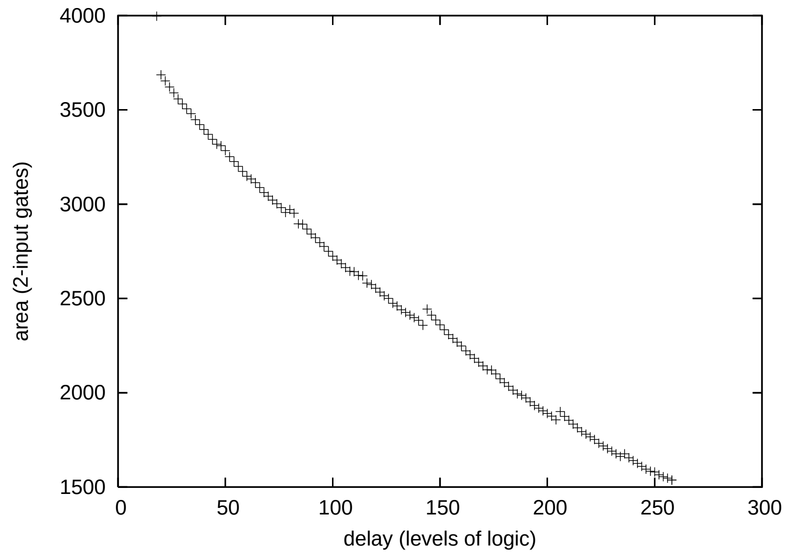


Figure 4.16: The performance/area tradeoff obtained by my algorithm on a 128-bit ripple carry adder

because they consider more possible transforms than combinations of Shannon decompositions. In particular, the best ones perform Boolean manipulations on the nodes' internal function. My algorithm treats nodes as black boxes, which both greatly reduces the space of optimizations I consider and greatly speeds my algorithm.

My algorithm does perform well on certain combinational circuits. Figure 4.16 shows how it is able to trade area for reduced delay with a 128-bit ripple-carry adder. For this example, I varied the requested c and generated a wide spectrum of adders, ranging from the original ripple-carry at the lower right to a full carry-select adder at the upper left. My algorithm makes evaluating such a tradeoff practical: it only took 22s to generate all 122 variants.

4.4.2 Sequential benchmarks

I ran my algorithm on mid- and large-sized ISCAS89 sequential benchmarks and targeted an FPGA-ike, three-input lookup-table architecture. Hence, I report delay as levels of logic and area as the number of lookup tables.

Table 4.1: Experiments on ISCAS89 sequential benchmarks

	Reference		**Retimed**		**Ours**		**Time**	**Speed**	**Area**
	period	**area**	**period**	**area**	**period**	**area**	**(s)**	**up**	**penalty**
s382	7	103	7	103	5	115		40%	11%
s386	7	85	7	85	7	85			
s400	8	103	7	103	6	111		16%	7%
s420	8	70	8	70	7	71		14%	1%
s444	8	101	7	101	6	106		16%	5%
s510	8	184	8	184	8	184	0.5		
s526	5	113	5	113	5	113			
s641	11	115	11	115	9	122	1.1	22%	6%
s713	11	118	11	118	10	121	0.9	10%	3%
s820	7	206	7	206	7	206	0.5		
s832	7	217	7	217	7	217			
s838	10	154	10	154	8	162	2.6	25%	5%
s1196	9	365	9	365	9	365	0.6		
s1238	9	338	9	338	9	338			
s1423	24	408	21	408	13	460	3.8	61%	12%
s1488	6	453	6	453	6	453	0.7		
s1494	6	456	6	456	6	456	0.8		
s5378†	8	819	7	819	7	819	1.4		
s9234	11	662	8	656	8	684	6.7		
s13207	14	1382	11	1356	9	1416	18.0	22%	4%
s15850†	20	2419	14	2413	10	2530	4.7	40%	5%
s35932†	4	3890	4	3890	3	4978	3.9	33%	28%
s38417	14	7706	14	7652	13	7871	113	7%	3%
s38584†	14	7254	13	7220	11	7296	56	18%	1%

† These circuits could not be processed by SIS's *full_simplify* algorithm—part of *script.rugged* —within 8 hours. I used *simplify* instead, therefore the starting points are poorer compared to the other samples.

Following Saldanha et al. [SHM+94], I run *script.rugged* and perform a speed-oriented decomposition *decomp -g; eliminate -1; sweep; speed_up -i* on each sample. I then reduce the network's depth while keeping its nodes three-feasible with *reduce_depth -f 3* [Tou91]. I report the results of this classical FPGA delay-oriented flow in the Reference column of Table 4.1.

Starting from these optimized circuits, I compare running retiming alone (*retime -n -i*, modified to use the unit delay model) with running my algorithm followed by retiming. The Retimed and Ours columns list the period and area results. My running time, listed in the Time column, includes finding the minimum period by binary search, so this actually includes multiple runs of my algorithm.

I verified the sequential equivalence of the input and output of my algorithm using VIS [BHSV+96]; my reported times do not include this since the verification is not a part of the algorithm.

Although my algorithm does nothing to half of the examples, it provides a significant speed-up for the other half at the expense of an average 5% area increase. The algorithm is very fast, especially when no improvement can be made. Its runtime appears linear in the circuit size. Its memory requirements are low, e.g., 70MB for the largest example s38417. My technique therefore appears to scale well.

That my algorithm is not able to improve certain circuits is not surprising. This algorithm is fairly specialized (compared to say, retiming) and only attacks time-critical feedback loops that have small cuts (the loops can be broken by cutting only a few wires). Circuits on which I show no improvement may have wide feedback loops (e.g. the PLA-style next state logic of a minimum-length encoded FSM, or a multi-bit "arithmetic" feedback), or may be completely dominated by feed-forward logic (e.g. simple pipelined data-paths).

4.4.3 Packet processing pipelines

I also ran my algorithm on small packet processing pipelines that I synthesized using the method described in Chapter 2. These pipelines contain several packet editors and the interconnecting FIFOs.

I did not include any auxiliary elements because they usually contain specialized components such as internal memory blocks or DDR controllers accessing off-chip memory and optimizing them is outside the scope of the presented algorithm. I simply consider the signals to and from these elements as primary inputs and outputs. But I did instantiate the FIFOs between the packet editors and these auxiliary elements.

I only use two-position FIFOs because it is realistic to implement them using only gates and flip-flops. In a real FPGA design, any larger FIFO would use a specialized (hand-optimized) memory element that my algorithm would not be able to optimize.

I first synthesize benchmarks using the Xilinx *xst* tool. I then map them and extract a structural Verilog model using the usual tools from the Xilinx *ise* flow (e.g., *ngdbuild*, *map*, *netgen*). I ran the synthesis and mapping steps with an aggressive optimization setting that included retiming; most optimizations attempt to minimize the clock period.

Next, I transform the Verilog netlist into a BLIF network using a custom script and decompose it into two-input gates by using the *speed_up -i* command from the SIS package. This performs a decomposition optimized for time.

I took the packet editors from real packet processing pipelines. The first pipelines (1–8) are trivial: they consist of only one module. The other examples (9–12) are pipelines with two modules. I have also run experiments on longer pipelines but the results are not interesting because the performance is usually limited by the most complex module in the pipeline. This trend becomes visible for the last examples.

These experiments are somewhat unrealistic because they do not consider higher-level primitives such as adders. In my flow, I reduce such primitives to small gates; a commercial flow would preserve such primitives and map them to dedicated resources

Table 4.2: Experiments on packet processing pipelines

		Reference		**Retimed**	**Ours**		**Speed**	**Area**
		period	**area**	**period**	**period**	**area**	**up**	**penalty**
1	AddressLearn	5	3125	5	4	3160	25%	1%
2	AddressLookup	7	2851	6	4	2864	50%	0%
3	UpstrVLANProc	18	4998	10	7	5070	42%	1%
4	DwstrVLANProc	20	4655	7	7	4877		
5	PortIDEdit	5	2738	5	5	2780		
6	Drop	7	4870	5	4	4874	25%	0%
7	VLANedit	12	6896	12	10	6925	20%	0%
8	VLANfilter	7	2618	5	4	2669	25%	2%
	modules 1 and 2	6	4703	5	4	4995	25%	6%
	modules 3 and 4	21	8599	10	7	8719	42%	1%
	modules 5 and 6	7	6262	5	4	6281	25%	0%
	modules 7 and 8	12	8447	12	10	8512	20%	1%

in the FPGA. Adapting my algorithm to handle such "black-box" primitives would be fussy, but not technically difficult. Thus, the initial timing estimates may not accurately reflect the actual timing after decomposition.

Nevertheless, my results in Table 4.2 show the same trend I observed for the ISCAS89 benchnmarks (Table 4.1), showing that both my proposed algorithm and the benchmarking setup are consistent.

4.5 Conclusions

I presented an algorithm that systematically explores combinations of Shannon decompositions while taking into account a later retiming step. The result is a procedure for resynthesizing and retiming a circuit under a timing constraint that can produce faster circuits than Shannon decomposition and retiming run in isolation. My decompositions are a form of speculation that duplicates logic in general, but I deliberately restrict each node to be duplicated no more than once, bounding the area increase and also simplifying the optimization procedure.

My algorithm runs in three phases: it first attempts to find a collection of feasible arrival times that suggests a circuit exists with the requested clock period. If successful, it then resynthesizes the circuit according to these arrival times and heuristically limits duplication of nodes off the critical path to reduce the area penalty. Finally, the resynthesized circuit is retimed to produce a circuit that meets the initial timing constraint (a minimum clock period).

Experimental results show my algorithm can significantly improve the speed of certain circuits with only a slight increase in area. Its running times are small, suggesting it can scale well to large circuits.

Chapter 5

Final Conclusions

5.1 Claim

I present an integrated, coherent methodology for designing packet processing pipelines. It considers the entire design flow, starting from high-level specifications, followed by automatic synthesis, worst-case performance analysis and some RTL-specific optimization.

I had to address two principal issues, which come in natural contradiction. On one hand, one desires abstract specifications, short development time, and design maintainability. On the other hand, high-speed network devices require very efficient implementations, which are hard to generate by a generic high-level synthesis flow. I consider my proposed approach not a compromise between these two demands but a way to address both with minimal drawbacks.

I claim that the work in this thesis can dramatically improve the design process of these pipelines, while the resulting performance matches the expectations of a hand-crafted design.

My claim is supported by experiment. Two commercial packet processing pipelines, i.e., the ingress and egress pipelines of a ADSL-like network switch, were completely synthesized using my proposed technique. Their performance is the same that one expects from a manually designed RTL circuit, while the productivity is greatly improved over the classic RTL design process.

For a detailed summary of the current thesis, I send the reader to Section 1.8. The specific conclusions of the three main sections, i.e., specification and synthesis, worst-case performance analysis and FIFO sizing, and RTL sequential optimization can be found in Section 2.8, Section 3.9 and Section 4.5.

The rest of this chapter describes some general conclusions. I consider many of them relevant outside the scope of this thesis. Finally, I address some open issues and I speculate about ways to surmounting them.

5.2 PEG — the human factor

I proposed a new Domain Specific Language, PEG, to serve as the entry point of my design flow (Section 2.4). PEG provides a high-level, abstract way to specify a packet editing module. This shortens design time and makes maintenance much easier.

I have argued in Section 2.4 that the challenge for finding a good specification language is twofold: is has to offer an abstract view, familiar to the typical designer, and it has to allow an efficient automatic synthesis procedure. I have also mentioned that generic techniques address the first issue but not the second.

Experimental results suggest that PEG addresses the second problem very well. However, specifying a packet editing module in PEG proved to be hard, even if PEG offers the right abstraction model for this class of designs: I was the only one able to write a complex PEG specification.

Therefore, I reconsidered PEG's place within the design flow. PEG became an intermediate format: the designer specifies the modules in a textual language that I trivially translate into PEG. Once I implemented this translation everything went as smoothly as expected in the first place and I was able to report the first solid experimental results of my synthesis flow.

This language is developed by a different team. Its syntax borrows many constructs from C, VHDL, functional languages, etc. Not exactly my choice, especially because in many cases these constructs have a different semantics. Moreover, there are a lot of counter-intuitive restrictions, such as limiting the number of conditional inserts to one (truly, more may be present but at most one executed).

The reason for these peculiarities is the computational model used by its authors, which is different from PEG. I intended to develop my own textual language, to expose the user to the full power of my PEG computational model but I had the good inspiration to abandon that before I began. The reason is twofold.

First, several designs have been already written in the "old" language, and the developers in my team simply refused to write the same thing twice. Additionally, they were reluctant to invest time in a immature technology, since at that point my synthesis algorithm was in its early development stages.

Second, describing the computation as a graph (i.e., an unordered set of rules defining the operations and the data dependency) looked awkward compared to the familiar C-like constructs.

I conclude that right abstraction and good performance are not sufficient. The human factor has a decisive word to say about the success or failure of a Domain Specific Language. Yes, I knew that from the beginning. Still, I had to learn it again.

5.3 Synthesis — taking advantage of others

I presented an efficient algorithm for synthesizing the above abstract specification into VHDL (Section 2.5). This VHDL code has to be further synthesized and optimized by traditional EDA techniques. Practically, this is done using a state-of-the-art commercial tool such as Synplify-Pro.

The high-level synthesis tool can not and is not supposed to perform low-level optimizations such as Boolean logic minimization. This is the task of the abovementioned tool. Moreover, the tool does it much better than an ad hoc technique coded inside the main synthesis algorithm.

However, there are optimizations that take full advantage of the structure of the high-level specification. They can not be done by the low-level tool. Truly, the key for obtaining an efficient result consists of applying these optimizations as aggressively as possible. This is the first, natural challenge.

The second challenge is hidden: keeping the high-level synthesis algorithm as simple as possible, or equivalently, taking full advantage of the low-level synthesis tool. Identifying what optimization belongs to each step is not a simple task. I identify two reasons.

First, some optimizations are wrongly included in the high-level step because they are considered necessary for triggering other high-level optimizations. I give a real example: constant propagation for arithmetic operations. Indeed, the result of adding two constants has arrival time zero, regardless of the adder delay: the scheduler can take advantage of this property. However, implementing constant propagation for all supported PEG (i.e., VHDL) operators is tedious and error-prone. The correct solution is to flag the constant result as constant, without computing its value, and propagate these flags. Now, the scheduler has the desired information while the real constant propagation is deferred to the low-synthesis tool.

Second, one may not know what optimizations are supported by the low-level synthesis tool. More commonly, one has a vague idea but not a clear image of their limitations and performance. The solution is to invest time in studying the tool: I systematically synthesized my examples with Synplify-Pro and inspected not only the synthesis reports but the schematics of the mapped design. Following the critical paths in a heavily optimized retimed design requires a lot of perseverance. But it paid off.

The drawback: it is difficult to answer questions like "your results are very good, couldn't you improve them by adding at least some trivial optimizations?"

5.4 Analysis — choosing what to measure

I dedicated the entirety of Chapter 3 to computing the worst-case performance of packet processing pipelines. The sequential behavior of such systems is complex and their analysis is equally hard.

The analysis algorithm is only the tip of the iceberg. The difficult part is to define exactly what to measure and to make sure there is a practical correspondence between the reported numbers and what one expects as result. For example, my whole analysis focuses on computing the worst-case throughput. What is throughput and what is worst-case?

If some pipeline modules insert or remove data, the amount of data entering and exiting the pipeline differs. Hence I define two throughputs, corresponding to the input and output rates. The step from this intuition to the mathematical definitions in Section 3.3.1 exhibits a two-fold challenge: It is not clear in the first place that these limits exist (I proved it later). Next, do these limits define exactly what one expects? (I argue they do: they are relevant for *ingress* and *egress* pipelines, respectively.)

Defining worst-case proves more difficult. Apparently, the word defines itself: the worst-case throughput considers any possible packet stream (this has an rigorous formulation, see Section 3.6). I was very happy with this definition since I gave a strong argument against analyzing streams of random packets. However, this definition is not exactly what the design engineer expects. I treat this open issue in Section 5.6.2.

5.5 Logic synthesis — beyond BLIF

I presented in Chapter 4 a low-level RTL sequential optimization technique, which alleviates the performance of cycle dominated sequential networks such as those automatically generated from PEG specifications.

I started the work in the SIS environment. This is a very practical academic approach since it provides all the basic EDA algorithms and is open source. The basic file format is BLIF, which is very close to SIS's internal representation: the only primitives are (simple) boolean logic nodes and registers that have no explicit reset signal and are assumed by most algorithms to share the same clock signal, etc.

Truly, most academic EDA algorithms can be practically implemented and verified within this framework. The problem appears when the algorithm has to be tested on real designs; that was the case with my packet processing pipelines, which were synthesized by a commercial FPGA oriented flow. Converting such a design to BLIF is a task one would like to avoid in the first place. The process itself is a collection of hacks such as decomposing the FPGA primitives into registers and boolean functions, verifying that all registers belong to the same clock domain, replacing the register's reset net with initial register values, etc.

Consequently, I tested my algorithm on the ISCAS89 benchmarks during the development stage and only later I used it on real packet processing pipelines.

The only viable alternative to SIS/BLIF I am aware of is OpenAccess. I have avoided it since it has the reputation of a very complicated programming environment, which requires a lot of time to become familiar with. If I started the same project now I would definitely explore this alternative.

5.6 Open issues

5.6.1 Synthesis - Design space exploration

I have mentioned that design space exploration offers a promising potential for improving the quality of the design in terms of area, performance, or both. I identify two directions to explore.

First, there is not a unique way to split the overall computation of a packet processing pipeline into several stages. I have assumed this split is the responsibility of the pipeline architect, but the process can be automated. This dimension is explored, to my knowledge, by researchers working on a commercial product.

Second, I have claimed in Section 2.5.9 that my synthesis method based on a trivial scheduling with no resource sharing gives a very reasonable performance trade-off. However, this is true for one module taken in isolation, i.e., that has an ideal source and sink. A different cost function has to be used if a module is not throughput-critical, i.e., area becomes more important than performance. To address this problem, one may propose an adaptation of existing scheduling algorithms that also consider area.

5.6.2 Analysis — Performance under-estimation

I have mentioned that the worst-case throughput computed by the proposed algorithm is an under-estimation. This has two causes.

First, at module level, I assumed each FSM is analyzed abstracting its data-path (Section 3.3.4); however, while this abstraction gives good results in most cases, one can easily imagine scenarios when this approximation differs significantly from the real worst-case. The problem becomes even more complicated when several connected modules are analyzed together.

Second, some extreme cases considered by the worst-case analysis never happen or are irrelevant in practice. For example, PPPOE discovery packets are very rarely found in a real flow: if it happens that a sequence of PPPOE discovery packets excites the worst-case pipeline behavior, then the analysis result is irrelevant.

Therefore, the computed performance guarantee is pessimistic. The natural solution to address this problem is to classify the packets into several classes and use this information in the analysis process. This is a compromise between the present approach, i.e., no data path, and the ideal approach, i.e., when the full data-path considered.

5.6.3 Low level optimization — Experiments

I tested the Shannon-retiming algorithm on both standard ISCAS89 benchmarks and on some sequential networks generated by my processing pipeline synthesis technique. I found the results are consistent (see the end of Chapter 4).

However, a careful performance evaluation would require the full synthesis of the entire design. This follows the observation that placing and routing may dramatically influence the design metrics, both in terms of area and performance. Moreover, for exact results, the rest of the components (i.e., those outside the packet processing pipelines) also must be instantiated and synthesized because they can interact with our modules directly (i.e., by wires) or indirectly (by influencing the placement and routing of the entire design).

However, this would entail integrating my algorithm in a commercial synthesis design flow, a task I consider more suitable for an industrial enterprise than for academic research.

Bibliography

[BHLT90] C. Leonard Berman, David J. Hathaway, Andrea S. LaPaugh, and Louise H. Trevillyan. Efficient techniques for timing correction. In *Proceedings of the International Symposium on Circuits and Systems*, pages 415–419, 1990.

[BHSV+96] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[BJRKK04] Gordon Brebner, Phil James-Roxby, Eric Keller, and Chidamber Kulkarni. Hyper-Programmable Architecture for Adaptable Networked Systems. In *Proceedings of the 15th International Conference on Application-Specific Architectures and Processors*, 2004.

[BM91] Steven M. Burns and Alain J. Martin. Performance analysis and optimization of asynchronous circuits. In *Proceedings of the University of California/Santa Cruz Conference on Advanced Research in VLSI*, pages 71–86. MIT Press, 1991.

[CM05] Mario R. Casu and Luca Macchiarulo. Throughput-driven floorplanning with wire pipelining. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(5):663–675, May 2005.

[CMSV01] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, 20(9):1059–1076, September 2001.

[Das04] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, October 2004.

[DBSV85] Giovanni De Micheli, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-4(3):269–285, July 1985.

[De 94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[HE96] Soha Hassoun and Carl Ebeling. Architectural retiming: pipelining latency-constrained circuits. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 708–713, New York, NY, USA, 1996. ACM Press.

[IEE03] Virtual Bridged Local Area Networks. IEEE Std. 802.1Q, May 2003.

[Kar78] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[KBS04] Chidamber Kulkarni, Gordon Brebner, and Graham Schelle. Mapping a domain specific language to a platform FPGA. In *Proceedings of the 41th Design Automation Conference*, pages 924–927, San Diego, California, 2004.

[KMC+00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[LK03] Ruibing Lu and Cheng-Kok Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 227–231, 2003.

[LK06] Ruibing Lu and Cheng-Kok Koh. Performance analysis of latency-insensitive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):469–483, 2006.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[LS91] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[LT01] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[LWGH97] Eric Lehman, Yosinori Watanabe, Joel Grodstein, and Heather Harkness. Logic decomposition during technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):813–834, August 1997.

[MB04] Praveen K. Murthy and Shuvra S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems*, 9(2):212–237, April 2004.

[MBSVS91] Patrick C. McGeer, Robert K. Brayton, Alberto L. Sangiovanni-Vincentelli, and Sartaj K. Sahni. Performance enhancement through the generalized bypass transform. In *International Workshop on Logic Synthesis*, Tahoe City, California, 1991.

[MR01] Maria-Cristina Marinescu and Martin Rinard. High-level specification and efficient implementation of pipelined circuits. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 655–661, New York, NY, USA, 2001. ACM Press.

[MSBSV91] Sharad Malik, Ellen M. Sentovich, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):74–84, January 1991.

[OP92] Sean O'Malley and Larry L. Peterson. Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, 1992.

[Pan99] Peichen Pan. Performance-driven integration of retiming and resynthesis. In *Proceedings of the 36th Design Automation Conference*, pages 243–246, New Orleans, Louisiana, June 1999.

[RVC01] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol Label Switching Architecture. RFC 3031, IETF, January 2001.

[SE07] Cristian Soviani and Stephen A. Edwards. FIFO sizing for high-performance pipelines. In *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, San Diego, California, June 2007.

[SG05] Graham Schelle and Dirk Grunwald. CUSP: a modular framework for high speed network applications on FPGAs. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 246–257, Monterey, California, 2005.

[SHE06] Cristian Soviani, Ilija Hadžić, and Stephen A. Edwards. Synthesis of high-performance packet processing pipelines. In *Proceedings of the 43rd Design Automation Conference*, pages 679–682, San Francisco, California, July 2006.

[SHM+94] Alexander Saldanha, Heather Harkness, Patrick C. McGeer, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Performance optimization using exact sensitization. In *Proceedings of the 31st Design Automation Conference*, pages 425–429, San Diego, California, June 1994.

[Sin92] Kanwar Jit Singh. *Performance optimization of digital circuits*. PhD thesis, Berkeley, CA, USA, 1992.

[SSM+92] Ellen M. Sentovich, Kanwar Jit Singh, Cho Moon, Hamid Savoj, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 328–333, Cambridge, Massachusetts, October 1992.

[STE06] Cristian Soviani, Olivier Tardieu, and Stephen A. Edwards. Optimizing sequential cycles through Shannon decomposition and retiming. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1085–1090, Munich, Germany, March 2006.

[STE07] Cristian Soviani, Olivier Tardieu, and Stephen A. Edwards. Optimizing sequential cycles through Shannon decomposition and retiming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(3):456–467, March 2007.

[SWBSV88] Kanwar Jit Singh, Albert R. Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 282–285, 1988.

[Tou91] Hervé J. Touati. Delay optimization of combinational logic circuits by clustering and partial collapsing. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 188–191, November 1991.

[TSSV92] Hervé J. Touati, N. Shenoy, and A. Sangiovanni-Vincentelli. Retiming for table-lookup field programmable gate arrays. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 8994, 1992.