UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATES I SISTEMES INFORMÀTICS
MÀSTER EN COMPUTACIÓ

TESI DE MÀSTER

ELASTIC ESTEREL

ESTUDIANT: Marc Galceran Oms
DIRECTORS: Jordi Cortadella i Gérard Berry

DATA: 25 de Juny de 2007

UNIVERSITAT POLITÈCNICA DE CATALUNYA	ABSTRACT OF MASTER'S THESIS

Departament de Llenguatges i Sistemes Informàtics
Master in Computing

Author:	Marc Galceran Oms
Title of thesis:	
Elastic Esterel	
Date:	June 25 2007
Supervisor:	Professor Jordi Cortadella
Supervisor:	Professor Gérard Berry

The aim of this master's thesis is to elasticize Esterel.

Esterel is an imperative hardware description language (HDL) used to describe reactive systems, and oriented to specify control systems. It belongs to the family of synchronous languages, and it allows to describe causality, concurrency and interruptions.

Elastic circuits preserve a protocol that makes it possible for the circuit to be latency-insensitive. Besides, elastic circuits are easy to implement and can be synthesized automatically.

The goal of the thesis is to provide an automatic synthesis method of elastic circuits from Esterel specifications. At the semantic level, it is proven that the generated elastic circuits are functionally equivalent to the conventional circuits generated using Esterel V7 compiler.

Keywords:	Elastic circuits, synchronous programming, Esterel, reactive systems, latency-insensitive design
Language:	English

Acknowledgements

A number of people has helped me making this thesis possible. I would like to thank Prof. Jordi Cortadella for picking my interest in elastic circuits and for his support and guidance throughout this project; Prof. Gerard Berry for allowing me to get into the Esterel compiler source code and for all the interesting discussions about the project; all the people who helped me during my visit at Esterel Technologies in Antibes; and my family and friends that have supported me while I made the project.

Barcelona June 25th 2007

Marc Galceran Oms

Abbreviations and Acronyms

HDL	Hardware Description Language
RTL	Register Transfer Level
SELF	Synchronous ELastic Flow
EB	Elastic Buffer

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Reactive Systems

Reactive systems were introduced in the 80s by Harel and Pnueli in [20]. The computation of such systems is a reaction to the environment's stimuli. Its outputs are produced within a certain time limit from the environment requests; the computation is driven by the environment.

Reactive systems can be defined in contrast to *interactive systems*, in which clients ask for some resources and the machine is the one driving the computation, and in contrast to *transformational systems*, in which inputs are available at the beginning of the computation and the system halts after producing its outputs. In practice, real world complex systems are compositions of the three kinds of systems.

Programming of reactive systems in an essential industrial activity, as most industrial control systems are reactive. Communication protocols, signal processing systems, audio or video protocols, hardware controllers and bus interfaces are some examples of such systems.

Reactive systems are usually **concurrent**. Even though they might be designed using a single component, most complex systems are designed as a set of parallel components communicating with each other in order to perform the computation. Besides, the environment can also be considered an additional concurrent agent.

Behavioral **determinism** is another property of reactive systems. The outputs must be fully determined by its inputs and the moment when these inputs occur. This is a key difference between interactive and reactive systems. Interactive systems lead the computation with no tight time restrictions. Therefore, they can perform non-deterministic choices when deciding the resource distribution or in which order requests are answered.

Safety is critical for many reactive programs. A little bug in an airplane controller could have terrible consequences, including human lives. Therefore, design methods must be accurate and formal methods are often used in order to check the system correctness.

Another property of reactive systems is **timeliness**. The system is a slave of the environment, which cannot stop to wait the program. Time constraints are an important part of the system specification, and they must be satisfied by designs and implementations. That makes performance an important issue, because time reactions must happen exactly in the required time.

Synchronous languages [17] were introduced in the 80s for programming reactive systems. They are both deterministic and concurrent, and that makes them suitable for the implementation of such systems. Furthermore, they can be efficiently implemented, either by software or by hardware, and it is possible to perform correctness proofs on them.

These languages provide ideal primitives that have instantaneous reactions. The order in which events occur is the important issue and physical time becomes just a regular event. Hence, the computation is a set of totally ordered logical instants. At any given instant, any event may or may not occur. Nothing happens outside these logical events. In practice, it is assumed (and checked) that the program reacts fast enough in order to be able to distinguish which events are happening in the same instant and which ones happen either previously or afterwards.

Reactive systems have to handle data and control. Balance between data and control can be very different depending on the system type. On the one hand, some systems may be control intensive and deal with data in a trivial way, and on the other hand other systems may be data intensive and have little control. Specification and programming techniques can vary depending on the amount of control and data. There are synchronous languages, like Lustre [12, 18] and Signal [23, 14, 25] , which are closer to the requirements of data intensive programming, while others, like Esterel [6, 5, 7] , Statecharts [19] or Argos [27, 28], are more suitable for control intensive programming.

1.2 Statement of the Problem

In nanoscale technology, it is difficult to determine latencies during the early phases of design. Transmitting some piece of information between a sender and a receiver may take more than one cycle, and this cannot be known until the placement is done, which is late in the design process. In order to solve this problem, one has to take conservative approaches during the early design phases and go back iterating when it is not possible to reach a placement in which wire latencies are the desired ones.

Latency-insensitive design [11] is a possible solution. Its goal is to assemble components with different latencies in synchronous distributed systems. Carloni et al. [11] propose a protocol for data exchanges which enables any variation (in terms of number of cycles) of the computing time of the components and of the transmission delays between them.

Synchronous elastic networks [22] are inspired by the idea of latency-insensitive design. `SELF` [13], a protocol for elastic circuits, provides a simple and efficient design for latency-insensitive circuits.

The aim of this thesis is to elasticize circuits specified with a synchronous programming language, that is, given a program written in a synchronous programming language that can be compiled into a synthesizable RTL circuit, we want to create a tool that automatically compiles it into a synthesizable RTL elastic circuit. Thus, a high level description of a circuit (a reactive system) can be directly translated into a latency insensitive design with no extra effort by the circuit designer. Besides, it is desirable to keep the circuit as similar as the conventional (non-elastic) one as possible.

Esterel [6, 5, 7] is the synchronous language that will be elasticized. It is an imperative hardware description language (HDL) which is oriented to specify control intensive reactive systems, even though it can handle data as well. Being imperative, Esterel looks closer to classical programming languages than other synchronous languages. That makes it easier to understand for people who are not used to HDLs.

It has been chosen because it is a synchronous language highly related to formal methods. The Esterel compiler generates formally verified code for RTL designs, and Esterel Studio provides tools to verify properties and specifications. Furthermore, Esterel is actually used in industry; it is being commercialized by Esterel Technologies, which has costumers such as Airbus and Texas Instruments.

The target language for the tool will be Verilog [41, 8]. Verilog is a widely

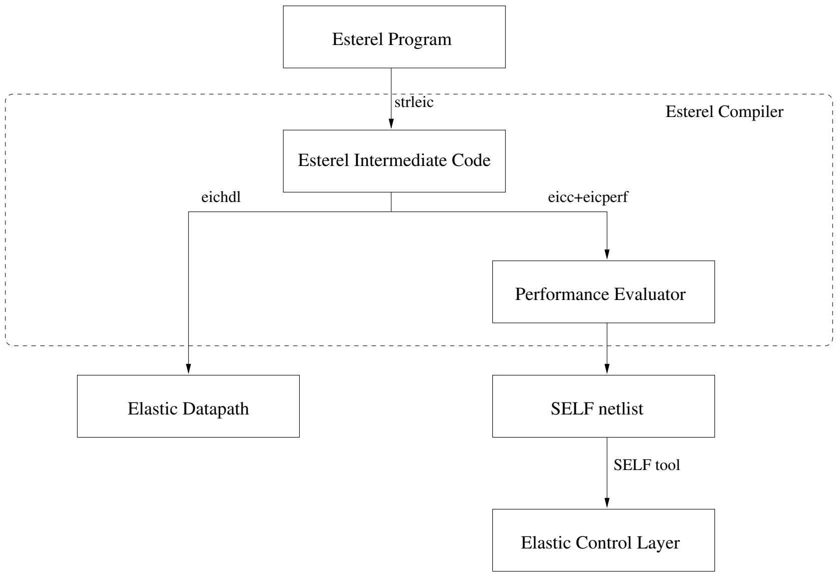


Figure 1.1: Elasticization flow

used HDL. It can be used to design, verify and implement analog, digital and mixed-signal circuits at several levels of abstraction, particularly, at the RTL level, which is the one we are interested in. Furthermore, the Esterel compiler can compile Esterel sources into Verilog RTL, which makes elasticization easier.

Figure 1.1 shows the elasticization procedure embedded into the Esterel v7 compiler procedure. The elasticization procedure has been embedded into the Esterel v7 compiler developed by Esterel Technologies. The Esterel compiler creates an intermediate code from the Esterel program. Afterwards, eicc and eicperf, which are two programs of the Esterel compiler, create a performance evaluator, that can generate a `SELF` netlist. Then, the `SELF` tool creates an elastic control layer written in Verilog from this netlist. On the other hand, eichdl, which is another program of the Esterel compiler, creates the elastic datapath in Verilog. This datapath is very similar to the Verilog circuit the compiler would generate if it was creating a non-elastic design. The control layer and the datapath constitute the elastic design.

1.3 Description of the Chapters

Chapter 1 introduces reactive systems and synchronous languages, and then we state the objective of this thesis: to elasticize automatically an Esterel design. The tool must create an elastic design in Verilog from an Esterel source code.

Chapter 2 presents the Esterel synchronous language. We present the kernel statements of the Pure Esterel language with their intuitive semantics, and then we present some of the main derived statements. Finally, there is an introduction to the Esterel constructive semantics and the Esterel circuit translation.

Chapter 3 presents elastic circuits. Elastic circuits are latency-insensitive, and hence, they can transmit meaningless data some cycles if real data is not prepared. `SELF` is a protocol that implements the theory of elastic machines.

Chapter 4 presents an elasticization procedure. It modifies the Esterel compiler flow in order to elasticize the output Verilog code. There are few changes done in the datapath and they add no overhead in latency, area and power. Then, an addicional elastic control layer implements the elastic part of the circuit.

Chapter 5 presents the methods used to validate that the elasticized designs are equivalent to the original ones. The conventional Verilog code and the elastic Verilog code are simulated in order to make sure that the order of the output data is the same given that the order of the input data is the same.

Finally, Chapter 6 explains the conclusions of this master's thesis and future research topics.

Chapter 2

The Esterel Language

Esterel [6, 5, 7] is a synchronous concurrent programming language used to design reactive systems. It provides high-level control and event manipulation constructs, including most control statements of any conventional imperative language like C. Furthermore, it provides concurrency, preemption, exceptions and a synchronous model of time.

Esterel is scoped for control-intensive reactive behaviours. An Esterel program is a collection of Esterel modules that communicate instantaneously with each other through their interfaces. Moreover, modules can have parallel threads, so parallelism is highly integrated into Esterel.

A program execution is a set of instants (cycles). At every instant, all modules perform an input/output computation. Such computation is called a reaction. At each cycle, the program resumes all running concurrent threads, waits for the inputs, computes the reaction of each thread synchronously and suspends all threads until next cycle. The computation is said to be input driven, because the inputs are the ones that determine the reaction and the status in which threads will stay.

Esterel can be translated efficiently both into hardware and software. The Esterel software translation is based on the mathematical semantics of Esterel and it translates the concurrent reactive program into statically scheduled control flow graphs (see Compiling Esterel [32] for further information about how Esterel is compiled to software). The Esterel hardware translation [4, 2] compiles a program into a synchronous sequential circuit .

Threads communicate with each other using *signals*. Signals are instantaneously broadcasted through all their scope, which can be a thread, a module or a set of modules. The perfect synchrony hypothesis [1] assumes that the program is executed on an infinitely fast machine; thus, signal transmission and reactions are assumed to be instantaneous.

Every cycle, a signal is absent unless someone emits it. The presence of a signal only holds for one cycle, so every cycle it must be emitted again if necessary. Threads can check whether some signal is present. Therefore, every cycle it is decided whether every signal is emitted, and instantaneously afterwards, the reaction is performed. Some constructions using signals can lead to paradoxes, such that *if X is not present, emit X*. Such programs are forbidden because it is not possible to decide whether signals should be emitted. Section 2.3.1 explains which Esterel programs are valid and which ones are not.

This chapter shows some basic Esterel statements and their intuitive semantics to make further examples understandable. In order to have a wider understanding of the Esterel language, see the Esterel Primer [5] and other scientific papers in Esterel Technologies webpage. Pure Esterel, described in section 2.1, allows signals and arrays of signals while Full Esterel, described in section 2.2, adds data handling. Valued signals, when emitted, carry a value which can be used to perform arithmetic operations. Finally, section 2.3 gives an overview on Esterel semantics and its circuit translation.

2.1 Pure Esterel

2.1.1 Modules and Signals

Esterel Modules are the basic programming unit, providing a modular way to write things once. Every module is a behavioral unit that can be instantiated by other modules. Therefore, an Esterel program is a hierarchy of Esterel modules. The top one is the *main module*, which call submodules in an arbitrary way (sequencially, concurrently, ...). Similarly, each submodule may call its own submodules, and so on and so forth.

Every module has a header that specifies its name and its interface, that is, a set of input signals and a set of output signals. A module instantiating another one can see whether its child has emitted any output signals and can emit its child input signals. Figure 2.1 shows a sample module declaration.

Right after the interface declaration, the body of the module specifies how reactions are computed. Every module has a cyclic behavior: at each instant, it reads which inputs have been emitted, it triggers a combinational calculation and determines the outputs and the status for next cycle.

```
module M :
input A, B ;
output O,X ;
    module body
end module
```

Figure 2.1: Sample Module

A *signal* in pure Esterel can be seen as a wire that carries a bit. Signals are the primary objects of Esterel. The value of a signal is its *status*, which can be *absent*, equivalent to 0 or *present*, equivalent to 1.

The status of input signals is given by the environment, while the status of the other signals is absent by default, and it may be set by the program. In each reaction, every signal has a unique status, either absent or present, which only holds for one instant. Broadcasting and testing of signals is instantaneous: when a thread emits (sets its status to present) a signal S, all other threads in the same scope that check the presence of S during the instant will see it is present.

2.1.2 The Pure Esterel Kernel Language

The Pure Esterel Kernel Language, described in [4, 3], is a subset of Pure Esterel from which all other statements can be derived by macro-expansion. Therefore, the Esterel kernel is enough to program any pure Esterel program. The set of statements that form the kernel are the following ones.

Nothing

The *nothing* statement terminates instantaneously. Thus, when a thread reaches this statement, it goes on instantaneously; if there is a statement after this one, it is executed, otherwise, the thread terminates.

Pause

Pause stops the computation one cycle, i.e., it introduces a one-instant delay. The execution of a thread is suspended when it reaches a pause statement, and it is resumed from the next statement in the next instant.

```
module mux :
input S, X, Y ;
output O ;
    if S then
        if X then emit O
    else
        if Y then emit O
    end
end module
```

Figure 2.2: Multiplexer in Esterel

Emit

Executing "**emit** S" sets the status of signal S to present. The status of S is instantaneously broadcasted to all the scope of S and remains valid only during the current instant. Therefore, if any thread checks the status of S during the current instant it will perceive it is present. Emit terminates instantaneously and the next statement is immediately executed.

Present

Executing "**present** S **then** p **else** q **end**" immediately starts p if signal S is emitted in the current instant, and it immediately starts q otherwise. In actual Esterel programming, *present* is obsolete and *if* is used instead. However, when expanding programs to use only kernel statements, present is used. Figure 2.2 shows a module implementing a multiplexer using *if* and *emit* statements.

Local Signal Declaration

The statement "**signal** S **in** p **end**" executes p, and declares a fresh signal S. If there is another signal with the same name, the new signal is the one that will be taken into account. Once p terminates, the status of S is lost.

Suspend

It is useful to freeze a computation during one instant so that it does not react at all, and next instant it can continue from the point it ended during the last instant it reacted. Suspension freezes its body on instants in which its test is true.

When "**suspend** *p* **when** *t*" starts, it immediately starts to compute *p*. On next instants, if the boolean expression *t* is evaluated to true, *p* is not computed. Thus, no signal is emitted and there is no state change. Otherwise, when *t* evaluates to false, *p* is executed from the point it paused on last execution instant. If it terminates, so does the suspend statement.

Sequencing

Operator ';' links statements, so that when the first statement finishes, the second one starts immediately. In "*p*; *q*", *p* is executed to completion, and the instant it terminates, *q* is started. If the first statement exits enclosing traps or preemptions, then the second one is never executed. If both *p* and *q* have no delay, the whole sequence terminates the same instant it started.

Looping

Loops are one of the basic structures of many programming languages, and they allow to write pieces of code once and reuse them. The statement "**loop** *p* **end**" executes *p* and when *p* finishes, it is restarted during the same instant.

The body of a loop cannot terminate instantaneously, i.e., it must have at least a delay of one instant. Otherwise, the computation of one instant could be infinite. In order to get out of a loop statement, it is necessary to put it into a preemption statement.

Parallelism

Concurrency is one of the main characteristics of reactive systems and hence of Esterel. Threads are inherent in synchronous languages; "*p* || *q*" runs in parallel statements *p* and *q*. The || operator forks the computation when it starts and terminates when both branches have terminated. If some signal is emitted by one branch, it is immediately broadcasted to all branches.

```
module ABO :
input A,B ;
output O ;
    { await A || await B };
    emit O
end module
```

Figure 2.3: ABO module

Figure 2.3 shows a module that waits for its two inputs to be emitted, and then emits signal O. A and B can be emitted simultaneously or can be emitted in different instants. The parallel statements waits until both threads have ended, that is, both signals have been emitted. Section 2.1.3 describes the await statement.

Traps

The statement, "**trap** T **in** *p* **end**" defines a trap called T. The statement immediately starts the body *p*. If *p* exits the trap, the trap statement terminates instantaneously, and concurrent processes inside the trap are weakly preempted. If *p* terminates, so does the trap statement. Traps are similar to exceptions in the sense that the internal code can immediately exit a trap executing the exit statement.

Exiting a Trap

Executing "**exit** T" immediately terminates the enclosing trap T. However, if another branch exits during the same instant another trap which encloses T, then the outermost trap is exited.

Figure 2.4 shows code for a controller that helps a patient breathing and at the same time checks the heart rate. *CheckRate* is some piece of code that executes the statement "**exit** tachycardia" if the condition for a tachycardia is true. The system still executes the breathing code during the exiting instant, because it is weakly preempted.

```
trap tachycardia
    loop
        Inhale; Exhale
    end loop
||
    CheckRate
end trap
emit emergency
```

Figure 2.4: Heart rate controller

2.1.3 Derived Statements

All other Pure Esterel statements are syntactic sugar, and they can be described in terms of kernel statements. Some of theses statements are described in this section.

Signals

It is possible to know whether a signal was emitted during the previous instant using the *pre* function. However, pre cannot be nested: it is necessary to define auxiliary signals to know whether a signal was emitted some cycles ago. It is also possible to define *registered signals* using the *reg* keyword. Registered signals transmit their status with a one tick delay. In order to set a registered signal it is necessary to use the *emit next* statement. Figure 2.5 shows different ways to define a multiplexer with one tick delay.

It is also possible to define multiple dimension arrays, which are indexed in a C-like style. At any instant, some parts of the array may be emitted and others may not. It is possible to slice arrays when doing assignments. The assignment operator is used to emit parts of the array : "**emit** M[0][3] <= S **and not** X".

Finally, "**sustain** S" can be used to emit a signal every signal. Sustain is equivalent to "**loop emit** S; **pause end**".

```
module mux :
input S, X, Y ;
output O ;
    if pre(S) then
        if pre(X) then
            emit O
        end
    else
        if pre(Y) then
            emit O
        end
    end
end module
```

```
module mux :
input S, X, Y ;
output reg O ;
    if S then
        if X then
            emit next O
        end
    else
        if Y then
            emit next O
        end
    end
end module
```

Figure 2.5: 1 tick delayed multiplexer

Weak Suspend

Suspension has a *weak suspension* version. In terms of hardware design, weak suspension is a high-level view of clock-gating the internal registers of a circuit, that is, it disables all internal registers and lets the combinational logic work. If its condition is evaluated to true, its body is evaluated and if it exits a trap, so does the whole suspend statement. However, state change or termination are ignored, and it will start from the same state it did on next instant. However, if its condition is evaluated to false, it behaves normally, and the body can terminate or change its state. Figure 2.6 shows a suspend program, a weak suspend program and a trace that points differences between them.

Preemption

Preemption is typically the ability of operating systems to stop an scheduled task to execute a higher priority task. In Esterel, preemption structures control life and death of threads: they can suspend, kill or restart them.

Await simply stops the thread until its guard is true. The format of the guard is a combination of signals and boolean operators, possibly with the number of times some event must happen. The guard is not checked during the starting instant unless the immediate keyword is present, so await delays

```
suspend          1      weak suspend
    pause;       2          pause;
    emit X;      3          emit Y;
    pause        4          pause
when S;          5      when S;
pause;           6      pause;
emit X           7      emit Y
```

Instant	0	1	2	3	4	5	6
S emitted	Y	Y	Y	N	Y	N	Y
Weak suspension							
Starting line	1	2	2	2	4	4	6
Ending line	2	4	4	4	5	6	7
X emitted	N	N	N	Y	N	N	Y
Y emitted	N	Y	Y	Y	N	N	Y

Figure 2.6: Suspend and weak suspend

the computation at least one instant. Figure 2.3 shows a module using the await statement.

"**await** S" is equivalent to "**trap** T **in loop if** S **then exit** T **else pause end end end**". "*p*; **await** S; *q*" executes *p*, waits for someone to emit S and then executes *q*, "*p*; **await** 3 **times** S; *q*" waits until S has been emitted 3 times and "*p*; **await** 3 **times** R or S; *q*" waits until there have been three instants in which either R or S were emitted.

"**abort** *p* **when** *guard*" kills its body when its guard expression is true. Such guard expression is the combination of signals and boolean operators. It is also possible to wait until an event occurs a certain number of times, like "**abort** p **when** 3 tick".

The body of an *abort* statement is always executed during its first instant, even if its guard is true. However, a guard like "**immediate** A" will kill its body if signal A is present during the starting instant. There is a weak version like in suspend where the body is weakly preempted: it is executed a last time during the instant in which the guard is true.

2.2 Valued Esterel

Even though the main focus of Esterel are control intensive programs, it is possible to handle data of arbitrary types. *Valued signals* provide a way to deal with data in Esterel designs.

2.2.1 Valued Signals

Valued signals carry a value of some type in addition to their status. In any instant, a valued signal has exactly one value, which is determined by the reaction through emit statements and the values of other signals. It is possible to combine valued signals through operators which are assumed to be instantaneous like boolean operators on pure signals.

The status of some signal S is denoted simply by its name S, and its value is denoted by the '?' symbol before the signal name ($?S$). When some statement emits a valued signal, it assigns a value to its signal and its status is set to present. For example, "**emit** ?C <= ?B + 3", sets the status of C to present and sets its value to the value of signal B plus three. Signals B and C could be, for example, signals of unsigned type, declared "**signal** {B,C} : **unsigned in** ...".

Operations on pure signals can be applied on valued signals: it is possible to consult the previous value of any valued signal using the *pre* function and to define registered valued signals using the *reg* keyword in the signal declaration "**signal** B : **reg unsigned in** ...".

Hardware translations generate a wire for the signal status and another one for the signal value. However, the status might never be consulted because of design reasons. At the module level, such wire will be swept, but the extra wire will still be created at the module interface level. When this happens, signals should be declared *value-only*, which generates no extra wire in its synthesis, using the *value* keyword. Value-only signals are assumed to be always present, and their status cannot be tested.

Valued signals generate a register by default, i.e., they are persistent. Even if a signal is not being emitted during the current instant, one can consult its value, and the Esterel program will return the last emitted value. If there is not such last emitted value, there is a run-time error unless a initial value has been defined in the source code using the *init* keyword. However, it is possible to define temporary signals (using *temp* keyword), which do not generate any register when synthesizing. Thus, one cannot consult the value of a temporary signal if it is not being emitted during the current instant.

2.2.2 Data Types

Esterel primitive types are bool, unsigned, signed, float, double and string. This section focuses on the most used ones, which are bool, unsigned and signed.

The *bool type* has two possible values, written $true$ and $false$. The status of a signal is considered to be of bool type when used in test expressions.

The *unsigned type* has lots of uses: indexing arrays, addressing memories, counting events, performing arithmetic operations and so on and so forth. For any natural number $N > 0$, the type *unsigned*<N> is a primitive type that contains natural numbers in the range $[0, N[$. It is also possible to define a type *unsigned*<[N]>, which contains natural numbers in the range $[0, 2^N[$. By default, type *unsigned* is *unsigned*<[32]>.

The actual representation of unsigned numbers is abstract, so they are not considered binary-encoded bit-vectors like in most programming languages. It is possible to enforce some encoding (like unsigned binary, one-hot, ...) using different functions.

Unsigned type supports the arithmetic operators used in most programming languages : $+, -, *, mod, /, \ldots$. Notice that integer division can lead to programs that translate into circuits which are not synthesizable. It is also possible to combine different size unsigned numbers by cutting them.

For any natural number $N > 0$, the type *signed*<N> contains all integers in the range $[-M, M[$; *signed*<[N]> is a shorthand for *signed*<2**N> and *signed* stands for *signed*<[32]> by default. Signed type also supports all typical arithmetic operators.

Array types are built from primitive types. There can be arrays of arrays up to any size and dimension, and they are indexed and defined in C-like style. Notice that it is not the same to declare a signal array of unsigned type than to declare an array of unsigned signals. In the first case, declared "**signal** S : **unsigned**[5][5]", S is a single signal containing the whole array, and therefore it has a unique status that indicates whether the signal is emitted. In the second case, declared "**signal** S [5][5] : **unsigned**", S is a matrix of signals, and each one of the 25 signals has an independent status and value. It is also possible to build a mixed signal : "**signal** S [5] : **unsigned** [5]" is an array of five signals, where each one has its own status and its value is an array of five unsigned integers.

Bit-vectors, which are mono-dimensional arrays of booleans, are widely used in hardware. Esterel supports encoding from unsigned and signed to bit-vectors and the other way round. It is also possible to perform comparison, concatenation and other useful operations over bit-vectors.

2.3 Semantics

There are several mathematical semantics for Pure Esterel that have been developed through the time for different reasons and purposes. For example, Esterel denotational semantics [38] precisely formalizes the intuitive temporal concepts, and Esterel behavioral semantics defines executions reaction by reaction using Plotkin's Structural Operational Semantics technique [30]. Esterel denotational semantics and Esterel behavioral semantics are shown to be equivalent by Gonthier [16].

Currently, Esterel constructive semantics [4] is the main semantics. The Esterel v7 compiler is developed directly from this semantics, following the example of Robin Milner's ML language [21]. Other semantics that have been used by Esterel compilers through the time are the ones in [3, 7].

There are several equivalent variants of the Esterel constructive semantics in [4], each one with a different level of abstraction. The Esterel constructive semantics solves the causality problem common to many synchronous formalisms (see Section 2.3.1). The Esterel constructive behavioral semantics is the simplest and more abstract one, and it defines what a program means. Then, the Esterel constructive operational semantics defines a set of microstep rewriting rules. Therefore, it is less abstract and more effective. Finally, the Esterel circuit semantics translates an Esterel program into a constructive boolean circuit. It is the semantics currently used for the Esterel hardware translation.

2.3.1 Correct Programs and Constructiveness

The language intuitive description tells us what is supposed to happen when an Esterel program is executed, but it does not formalize reactions, and hence, the computation is not guaranteed to be unique, not even to actually exist.

An Esterel program is logically correct if it is possible to determine whether every signal is emitted, that is, if there is exactly one status for each signal. The problem is that the status of signals determine which statements are

```
signal S in
    if S else
    emit S end
end
```

(a) Non-reactive Esterel program

```
if S then emit S end
```

(b) Non-deterministic Esterel program

Figure 2.7: Esterel programs that are not logically correct

executed, and the statements that are executed determine the status of the signals. This loop between control and signals can lead to programs that are not correct.

The *logical correctness law* states that a signal S is present in an instant if and only if an "**emit** S" statement is executed in this instant. Given a program, a fixed input event (a status for every input signal) and a global status, (a status for every signal), the flow of control can be uniquely determined and it is possible to know whether any emit is executed in an instant. Then, the global status is *logically coherent* if and only if at least one emit statement is executed for all signals that are assumed to be emitted and no emit statement is executed for the rest of the signals.

For a given program and input event, there may be either none, one or more than one logically coherent global status. If there is at least one logically coherent global status for a given input event, the program is *logically reactive* w.r.t this input event. In the same way, if there is at most one logically coherent global status, the program is *logically deterministic* w.r.t. this input event.

Then, a program is logically correct w.r.t. the input event if it is both logically reactive and deterministic, and it is *logically correct* if it is logically correct w.r.t. all possible input events. Figure 2.7(a) shows a program that is not logically correct because it is not reactive. On the one hand, if S is assumed to be present there is no "**emit** S" in the program flow, and on the other hand, if S is assumed to be absent there is an "**emit** S" in the program flow. Therefore, no global status is coherent. Figure 2.7(b) shows a program that is not logically correct because it is not deterministic: any global status of the program is coherent.

Currently, logical correctness is not used to reject and accept Esterel programs. There are logically correct programs that make no sense in the real world and furthermore, logical correctness is computationally expensive and it is hard to check for it.

Constructiveness was used in Esterel v5 compiler to accept Esterel programs. The idea behind constructiveness is that information must propagate in a cause-effect way through the program. For example, the program "**if** S **then nothing end**; **emit** S" is logically correct, but information does not flow because of cause and effect. The emit statement is executed after the if one but it changes how the if statement is executed; thus, this program is not constructive.

The concept of constructive programs comes from constructive logic [15], where proofs must describe algorithms, and instead of dealing with values like in classical (non-constructive) logic, one deals with fact-propagating proofs. When proving the proposition $\exists n \in \mathbb{N}, P(n)$ one would come across with an algorithm to find such n.

In combinational logic, a possibly cyclic circuit is logically constructive [36, 37] if and only if all wires stabilize to either 0 or 1 within a certain time limit, for any wire and gate propagation delay in the up-bounded inertial delay model [10]. That means that constructive circuits may have combinational cycles, as long as values in the wires and gates remain steady in bounded time. Sequential circuits are constructive if their combinational part is constructive for any input and any reachable state.

An Esterel program is constructive if its translation to a sequential circuit is constructive. As Esterel targets are typically hardware or software controllers, it must be possible to check constructiveness on large programs, with hundreds or even thousands of signals.

In order to check constructiveness, the Esterel v5 compiler translates programs into circuits, and it checks constructiveness in linear-time for each input. If the circuit is acyclic, the program is trivially constructive; otherwise, the problem can be solved using a bdd-based algorithm by Shiple [36], derived from an algorithm by Malik [26].

Currently, the Esterel v7 compiler uses cyclicity in order to decide whether to accept or reject a program. A program is rejected if there are any cyclic instantaneous signal dependencies, i.e., signal S_1 is emitted depending on the status of S_2 during the current instant and signal S_2 is emitted depending on the status of S_1 during the current instant.

This solution is more restrictive than using constructiveness, as all acyclic programs are constructive. However, assuming that programs are acyclic allows to create statically scheduled into fast code and to create faster circuits. The acyclicity tests done while compiling Esterel are described in [32]

nothing	0
pause	1
emit S	$!s$
present S then p else q end	$s?p, q$
suspend p when S	$s \supset q$
$p; q$	$p; q$
loop p end	p^*
$p \mid\mid q$	$p \mid q$
trap T in p end	$\{p\}, \uparrow p$
exit T	k with $k \geq 2$
signal S in p end	$p \backslash s$

Table 2.1: Esterel for semantics

2.3.2 The Constructive Behavioral Semantics of Esterel

The constructive behavioral semantics of Esterel [4] defines executions reaction by reaction. It only accepts constructive programs and it is derived from the logical behavioral semantics of Esterel. Reactions are defined using Plotkin's Structural Operational Semantics technique [30].

When dealing with mathematical semantics, the keywords used in programming languages are too heavy and make rules and proofs difficult to read. Table 2.1 shows the equivalence between the Esterel kernel in keywords and Esterel in mathematical semantics. Nothing, pause and traps are substituted by completion codes, thus, exiting the outermost trap is the statement "2", the second outermost enclosing trap, the statement "3", and so on and so forth.

For example, the multiplexer in Figure 2.2 would be translated into

$$s?(x?!o, 0), (y?!o, 0)$$

and the Esterel program ABO in Figure 2.3 would be translated into

$$\{(a?2, 1)*\}|\{(b?2, 1)*\}; !o$$

In the logical behavioral semantics, and hence in the constructive behavioral semantics, a reaction of a program P is a *behavioral transition* from P to P', where P' is the derivative of P, i.e., the program that will compute the next reaction, and I and O are the input event and the output event

$$P \xrightarrow[\text{O}]{\text{I}} P'$$

Reactions are computed using an auxiliary *statement transition relation* defined by structural induction on the kernel statements. Every transition is made of a statement p, its derivative p', an event E that defines the status of all signals in the scope of p, an event E' that defines the signals that p emits in the current reaction (notice that by coherence $E' \subseteq E$) and a completion code returned by p, which is 0 if the statement terminates, 1 if it pauses, or some $k \geq 2$ if it exits some trap :

$$p \xrightarrow[E]{E', k} p'$$

Then, given a program P of body p and an input event I, the *program transition* of P is defined as follows:

$$P \xrightarrow[O]{I} P' \text{ iff } p \xrightarrow[I \cup O]{O, k} p' \text{ for some } k$$

Transition relations are rules that precisely formalize the intuition behind Esterel kernel statements. Some of these rules are :

$$k \xrightarrow[E]{\emptyset, 0} 0 \qquad \text{(compl)}$$

which is a trivial rule: after execution, a *k statement*, which covers *nothing*, *pause* and *exit*, returns a termination code, and then the program terminates emitting nothing;

$$!s \xrightarrow[E]{\{s^+\}, 0} 0 \qquad \text{(emit)}$$

which adds the signal that is being emitted to the set of emitted signals and terminates with code 0;

$$\frac{p \xrightarrow[E]{E', 0} p' \quad q \xrightarrow[E]{F', l} q'}{p; q \xrightarrow[E]{E' \cup F', l} q'} \qquad \text{(seq2)}$$

which starts statement q immediately after statement p returns completion code 0. This code means that p has terminated, and hence, as p and q are sequenced, q must be started;

$$\frac{p \xrightarrow[E]{E',k} p' \quad q \xrightarrow[E]{F',l} q'}{p \mid q \xrightarrow[E]{E' \cup F', \max(k,l)} p'|q'} \qquad \text{(parallel)}$$

which synchronizes parallel statements through termination codes. If $\max(k, l) > 1$, some branch is exiting an enclosing trap, and therefore the parallel statement exits a trap. If both branches exit a trap, as the outermost trap is the one with higher code (2 is the closest one), the outermost trap will be exited. If $\max(k, l) = 1$, then nobody exited a trap and some branch has paused, hence, there is at least one branch that has not terminated. Thus, the whole parallel statement pauses and will resume execution next instant. Finally, if $\max(k, l) = 0, k = 0$ and $l = 0$, both branches have terminated, and so does the parallel statement.

The constructive behavioral semantics adds constructive restrictions to the rules of the logic behavioral semantics. These restrictions are predicates that state what the program must do and what it cannot do. Such predicates are disjoint and defined in a constructive way, thus, facts about signals and statements are propagated to know what must happen and what cannot happen.

After going through a reaction, the *must* predicate states which signals must be emitted, which traps must be exited, ... and the *cannot* predicate states which signals cannot be emitted, which traps cannot be exited, ... Then, a signal is assumed to be present if and only if it must be emitted, and it is assumed to be absent if and only if it cannot be emitted.

A program is constructive if and only if these predicates can determine presence or absence of all signals.

2.3.3 Translation into Circuits

Esterel programs can be implemented on hardware or software. Software implementation is either by direct translation into another programming language (like C); by simulating a sequential circuit or by using statically scheduled control flow graphs, while hardware implementation translates the program into a sequential circuit.

Implementations build a deterministic finite-state machine that takes care of the control (the Pure Esterel part) and schedules data-handling actions (operations over signal values).

The circuit semantics translates Esterel imperative programs into sequential circuits, refining the constructive behavioral semantics. The translation generates a set of boolean variables, split between input variables, output variables and local variables. At the same time, variables can be split between data ones, that handle values of valued signals, and control ones, that are usually boolean variables, and express a property of the program state such as signal status or halt points, i.e., points where the reaction is paused and resumed on the next reaction.

A first translation can be found in [2], and [4] improves it by being able to deal with all constructive programs. Its basic idea is to associate a subcircuit with each kernel statement, and build the whole circuit by recursive construction. Each reaction corresponds to a clock cycle; at each cycle, the circuit reads the input and computes the outputs and the next state from the inputs and the current state. This is a superficial presentation of the circuit translation, the reincarnation problem (due to loops) makes real translation harder. The problem was first solved in the Esterel constructive semantics book, and a simpler solution was found by Olivier Tardieu [40].

All statements but the pause one generate only combinational logic; the pause statement generates a register that records its state. The generated combinational logic is acyclic unless there are cyclic instantaneous signal dependencies. When cycles arise, the Esterel v7 compiler rejects the program. The Esterel v5 compiler used to check constructiveness and the circuit was built using Shiple's algorithms [36, 26].

Figure 2.8 shows the interface of any block representing a kernel statement. The circuit is build by recursive construction, and hence compound statements like parallel or sequential will have internal blocks with the same interface. The purpose of each pin is the following one:

- GO is an input pin that restarts the statement when it is set during a cycle.
- RES is an input pin that resumes the execution of a statement: a statement that has not terminated its execution during previous cycles and receives a RES continues its reaction from the point it ended on the last execution cycle.
- SUSP is an input pin that suspends the execution of the statement during the current cycle. It cannot be set at the same time that the

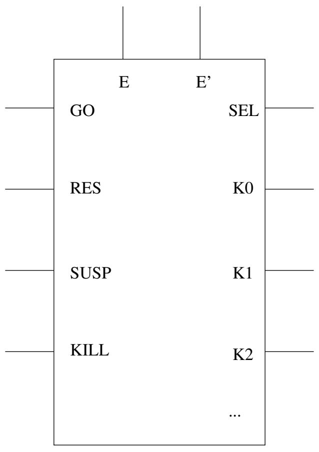


Figure 2.8: Interface of a block representing an statement

RES pin, and when it is set the statement behaves according to the suspend semantics.

- KILL is an input pin that unsets all registers of the statement. Kill is set when a trap must be exited and propagates to all internal subcircuits to unset all pause registers.
- SEL is an output pin that the statement sets when it ends a cycle in a state selected for resumption, i.e., when some internal pause register is set and hence, the reaction has not ended. In compound statements, the SEL pin is propagated from inner statements to outer ones.
- K0, K1, ... are output pins that indicate completion codes. When the statement is started or selected and resumed, the completion code the statement returns is set, otherwise, all of them are unset. There are $n+2$ completion wires, where n is the number of enclosing traps. The K_i wires are one-hot encoded.
- E is a bus indicating the input event: the signals the statement must know somebody has emitted.
- E is an output bus indicating the output event: the input event plus the signals the statement has emitted.

The execution of a statement starts the cycle GO is set. Then, RES is set every cycle unless the execution is suspended. At each execution cycle (when $GO \vee (RES \wedge SEL)$ holds), the circuit propagates its input pins and signals

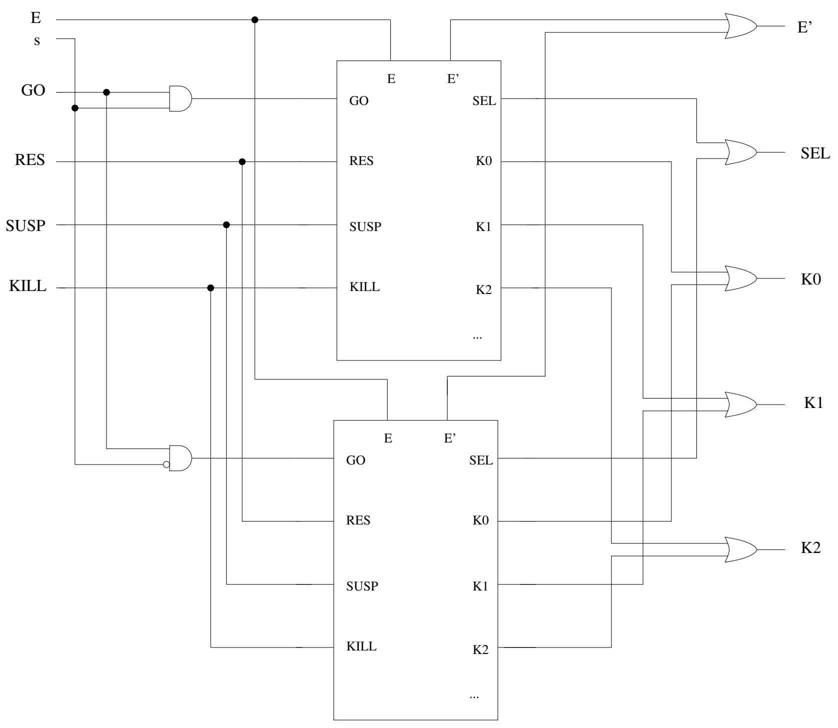


Figure 2.9: Circuit translation of $s?p, q$

and generates the output event and sets the corresponding completion wire. If the statement does not terminate, some pause statement must have been set, and hence the SEL wire is set. The configuration of pause registers form the current state of the reaction. If some upper level statement must preempt it, it sets the KILL pin to reset all pause registers and return them to the initial state. Weak suspension is obtained by setting both RES and SUSP

Every kernel statement connects the pins of the circuit in a different way so that it behaves like the semantics states. Compound blocks connect its interface to the interfaces of inner statements. Figure 2.9 shows the circuit translation of the present statement. See [4] for the circuit translation of each statement and for its correctness proof, which basically shows that the translation follows the circuit semantics, which is shown to be equivalent to higher level semantics.

This direct translation creates rather fast circuits which must be optimized before they can be implemented. Circuit optimization [29] is a wide field which has been broadly studied. There are basically two ways to optimize a circuit : combinational optimization and sequential optimization.

Sequential optimization is difficult because of the state assignment problem:

finding out a way to encode all states of the circuit into registers. Each possible solution changes the equations of the combinational logic which can explode exponentially in size, and that makes the problem hard. In fact, it is NP-complete and no heuristics manage to scale up well.

Esterel encodes the state of the circuit using pause registers, which works fine both in the size of the combinational logic and in the number of registers needed to encode states. Still, it uses some algorithms [34, 35] that reduce the number of registers with no major impact in the encoding, and hence, with no explosion in the logic size.

Combinational optimization consists basically in finding a network of gates and wires with good performance, either in size, speed or some trade-off between them. There are lots of tools and algorithms for combinational logic optimization (see [9, 29]). As the reaction time of Esterel circuits is close to proportional w.r.t. the number of equations, the optimization goal is to find a good network size while avoiding time explosion.

Chapter 3

Elastic Circuits

A conventional synchronous design can be seen as a set of combinational blocks connected to registers. Each block reads its inputs from the environment or from some registers when the cycle begins and writes its results to the environment or to some registers when the cycle ends. Thus, all blocks in the whole design compute data every cycle.

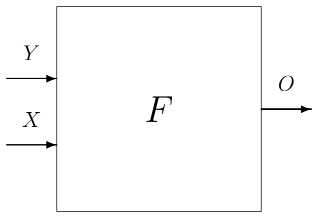
Elastic circuits [13, 22] allow idle cycles, that is, the value of some wire can be meaningless at some cycles and it will not transmit any information. The i-th data item is not transmitted at the i-th cycle necessarily, it can be transmitted afterwards. The transferred data items are called *tokens* and idle cycles contain *bubbles*.

Figure 3.1 shows the difference between a conventional circuit and its elasticization. In Figure 3.1(a) F computes a new value of O from X and Y every cycle. However, in Figure 3.1(b) wires X, Y and O are elastic and therefore they do not carry information (tokens) all the time. For example, wire O carries tokens at cycles $0, 3, 5$ and 7 and bubbles on cycles $1, 2, 4$ and 6.

There must be some protocol to decide whether there is a token or a bubble in a wire. In elastic circuits it is implemented using a pair of control wires (valid and stop) for every wire.

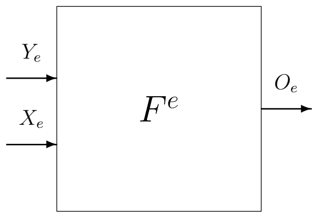
3.1 SELF Protocol

This section presents the `SELF` protocol, using some figures from Cortadella et al. [13] and trying to make it easy to understand. An elastic system is a structure made of elastic modules connected with elastic channels. An elastic



Cycle	0	1	2	3
X	A	B	C	D
Y	D	C	A	C
O	AD	BC	CA	DC

(a) Conventional synchronous trace



Cycle	0	1	2	3	4	5	6	7
X_e	A			B	C		D	
Y_e	D		C			A		C
O_e	AD			BC		CA		DC

(b) Elastic synchronous trace

Figure 3.1: Difference between conventional and elastic traces

module is a set of combinational blocks with latches. It may have fixed or variable delay, even though in this thesis we will always work with one-cycle delay modules. An elastic channel is a set of three wires. The first one carries the data and the other two are the control ones (valid and stop).

Every elastic module can be related to a set of combinational logic and registers in the original design, and channels connecting these modules can be related to wires in the original design.

If we look at a wire in a conventional synchronous design, it will carry meaningful data every cycle. On the other hand, if we look at an elastic channel the picture is not the same. If there is a token at the channel, it means that the sender (the source of the wire) is prepared to send some piece of information. However, it is possible that the receiver (the target of the wire) is not prepared to receive a token. If both are prepared, the valid bit will be set and the stop bit unset, and the token will be propagated. This is called a *transfer cycle*. Otherwise, if the sender is ready but the receiver is not, a *retry cycle* will occur. An *idle cycle* happens when there is a bubble in the wire. Figure 3.2 shows the possible scenarios with their valid and stop values.

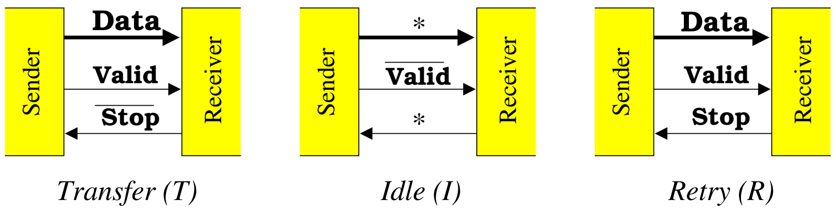


Figure 3.2: The SELF protocol

Cycle	0	1	2	3	4	5	6	7	8	9
data	*	A	B	B	B	C	*	*	D	D
valid	0	1	1	1	1	1	0	0	1	1
stop	0	0	1	1	0	0	0	1	1	0
SELF	I	T	R	R	T	T	I	I	R	T

Table 3.1: SELF trace

Therefore, the behavior in an elastic channel will be the following one. After a certain number of bubbles (which might be zero), some token will arrive at the channel. If the receiver is prepared, the token will be transmitted. Otherwise, SELF requires the sender to be persistent. The token must be kept by the sender until the receiver is prepared and the token can be transmitted through the channel. The behavior can be described by the regular expression $(I^*R^*T)^*$. The persistent behavior is given by the fact that it is impossible to go from a retry cycle to an idle cycle with no transfer cycle in between.

Furthermore, the SELF protocol ensures liveness. Because of persistence, whenever a token arrives at a channel it will stay there until it is possible to transmit it. Unless the receiver is never ready, the token will be transmitted and the communication process will go on. A more formal explanation of liveness properties can be found in Section 3.2.

Table 3.1 shows a trace committing the SELF protocol. In the SELF row, I means idle cycle, R means retry cycle and T means transfer cycle. Therefore, tokens are transmitted in cycles $1, 4, 5, 9$, so the data sequence is A, B, C, D. Data in idle cycles are irrelevant, and that is why it is labelled with a * in cycles $0, 6, 7$. Notice that persistency holds: cycles $3, 4$ and 8 transmit the same data than the previous cycle because it was a retry cycle. Finally, it is possible for the receiver to send a stop signal even if the sender is not transmitting anything, like in cycle 7.

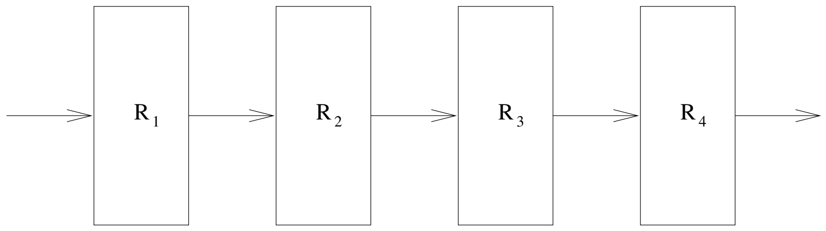


Figure 3.3: Elastic pipeline

3.1.1 Implementation

When implementing `SELF`, there are two important goals. Firstly, it is important that the elastic circuit adds no overhead in area, latency and power to the datapath of the original circuit and secondly, it is important to make the elasticization process possible with no major changes in the original circuit.

The first goal is important because overhead is an important limitation to any feature. Elastic circuits have no latency overhead, and they only have some area and power overhead because a control layer is added to the circuit.

The second goal is important because that makes the elasticization process easy. If it is only necessary to add a control layer and connect it to the original circuit, and this can be made automatic, designers can get an elastic design of their circuits for free, with no extra effort.

Double Storage and Latches

Figure 3.3 shows an elastic pipeline. Every block is an elastic buffer (an elastic fifo with depth one) and they are connected through elastic channels. At every cycle, every block tries to transmit its token to the next block. If at some cycle R_4 receives a stop, it will not be able to transmit its token to the output channel. That means that it has to store its token in order to retry next cycle.

R_3 is unaware of that fact, so it will transmit its token to R_4 that cycle and it will forget about that token. In order to avoid this, R_4 must send a stop to R_3 during the same cycle that it receives the stop bit from the environment. However, that means that this stop signal should also be propagated to R_2 and R_1 during the same cycle as well. That solution creates a long combinational path, which could increase the latency of the whole circuit if the pipeline was long enough. `SELF` would not be scalable and it would have latency overhead. Both effects must be avoided.

A solution to this problem is to give double storage capacity to elastic buffers. Thus, when R_4 receives a stop, it can store both its token and the token that R_3 is sending. Next cycle, R_3 receives a stop bit from R_4, because R_4 cannot store any other tokens. When R_4 manages to transmit its first token, it keeps sending a stop bit to R_3 because R_4 still has another token to send. Finally, while R_4 is sending its second token, it will allow R_3 to transmit again.

There are several possible solutions to implement double storage capacity. `SELF` uses latches to do so. A latch is a sequential device that is transparent during one clock edge and keeps its information during the other edge. During the high edge of the clock, an active high latch is completely transparent; it acts exactly like a wire. However, during the low edge of the clock, the latch remembers the last value it propagated during the high edge. An active low latch works the other way round, during the low edge it works in transparent mode and during the high edge it works in store mode.

Figure 3.4 shows the behavior of an active low latch (its input is signal D and its output is signal mid) and the behavior of an active high latch (its input is signal mid and its output is signal Q). When the clock is 0, mid has exactly the same value as D, otherwise, it remembers the last value it received during the low edge (which is 1 during the first four cycles and 0 during the following three ones). Regarding the other latch, Q has exactly the same value as mid when the clock is on the high edge, otherwise it remembers the last value in the input during the high edge. Notice that an active low latch is an active high latch with a not gate in the clock signal and the other way around.

A flip-flop can be implemented using a pair of latches of different polarity. A rising edge flip-flop is an active low latch (which is the master latch) connected to an active high latch (which is the slave latch, that reacts only to changes in the master latch). Figure 3.5 shows a flip-flop made by a pair of latches and Figure 3.4 shows a trace of such flip-flop. Signal D is the input of the flip-flop, mid is the wire connecting both latches and signal Q is the output of the flip-flop. During the low edge of the clock, D goes through the master latch until the input of the slave latch, which is mid. At the same time, the output of the slave latch is the value the flip-flop is remembering from the previous cycle. When the cycle finishes and the clock rises, the master latch goes into storage mode and its output is the last value of the previous cycle. Meanwhile, the slave latch is transparent, so mid and Q are the same value and the flip-flop updates its output. When the clock goes down, the slave latch stores the value of the master latch, so the output of the flip-flop remains unchanged, and the master latch becomes transparent.

Then, it is possible to transform the registers of a conventional sequential

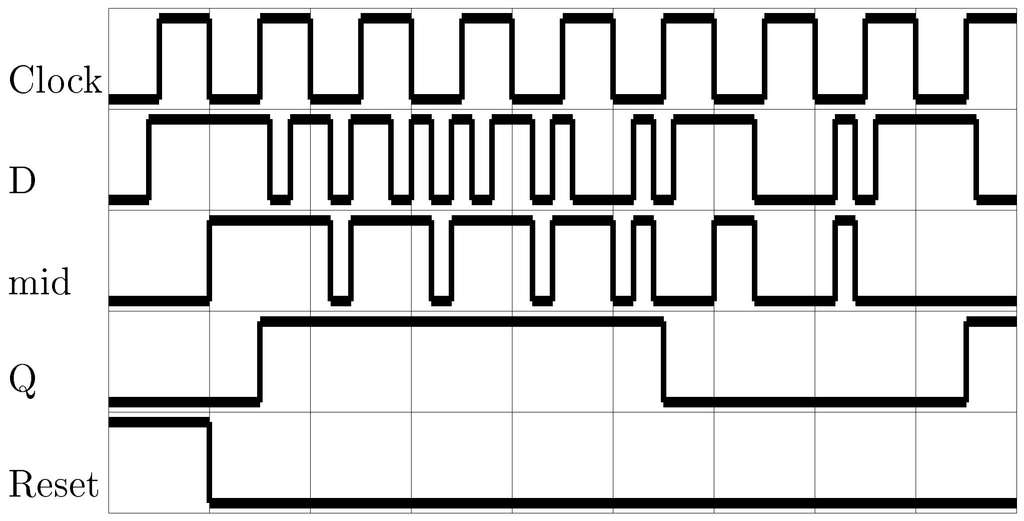
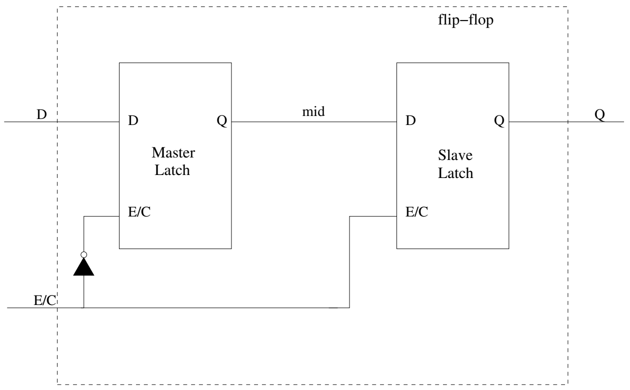


Figure 3.4: Trace showing a flip-flop made by a pair of latches



Latch truth table :

E/C	D	Q
0	X	Q_{prev}
1	0	0
1	1	1

Figure 3.5: Flip-flop made with a pair of latches

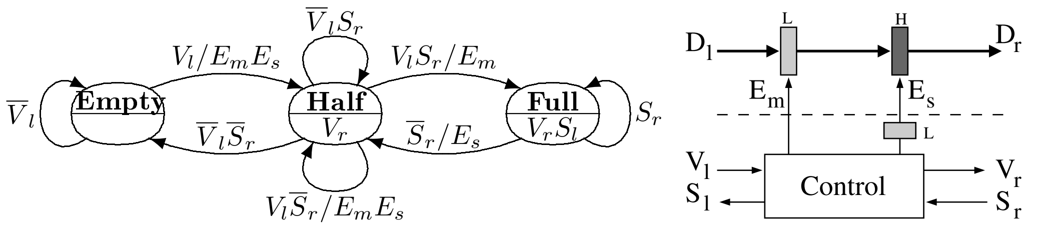


Figure 3.6: FSM of an elastic buffer and how it is connected to the datapath

design into pairs of latches, and by making them independent, using different enable wires, transform a register into a double storage device. Using a pair of latches makes it possible to add double storage capacity to elastic buffers with no additional latency because of the capacity of working in both edges of the clock, as consecutive latches have opposite polarity.

Elastic Buffer Implementation

Once latches provide double storage capacity, it is necessary to have a control layer that makes them independent using enable signals. This control must have no long combinational paths: elastic modules can be chained in pipelines.

An elastic buffer is an elastic fifo with depth one, one single input channel and one single output channel. It is possible to create all control for elastic modules by adding extra logic at the inputs and outputs of an elastic buffer to reduce multiple channels to a single channel. Figure 3.6 shows an FSM specification for an elastic buffer, and how to connect it to a pair of latches of the datapath.

The FSM specification shows three states. The first state, labelled empty, corresponds to the scenario in which bubbles are received. While the valid bit is false all cycles are idle and the elastic buffer can be considered empty. When a token arrives, the elastic buffer will try to transmit this token to the output next cycle. One of the latches contains this token, so the elastic buffer is half full. The output valid bit is set, and a transmit cycle will occur if the stop bit in the output is not set. Then, the elastic buffer will move to empty if no other token arrives or it will continue in half if a new token arrives. On the other hand, if the stop bit is set, the elastic buffer will move to full state, and both latches will be storing a token. The stop from the input channel is set because the elastic buffer cannot store more tokens. It will be full until the stop bit in the output channel is unset and it can go back to half state by transmitting its tokens.

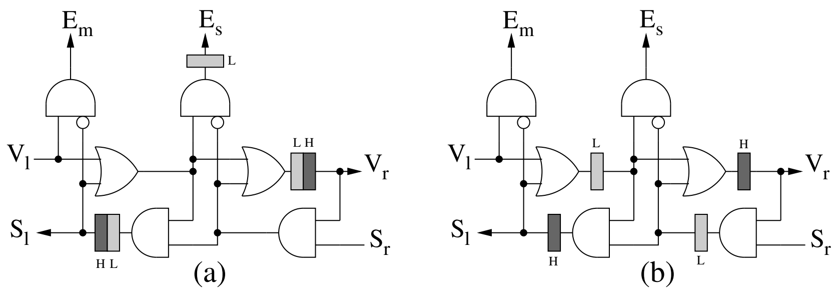


Figure 3.7: Implementations of an elastic buffer control

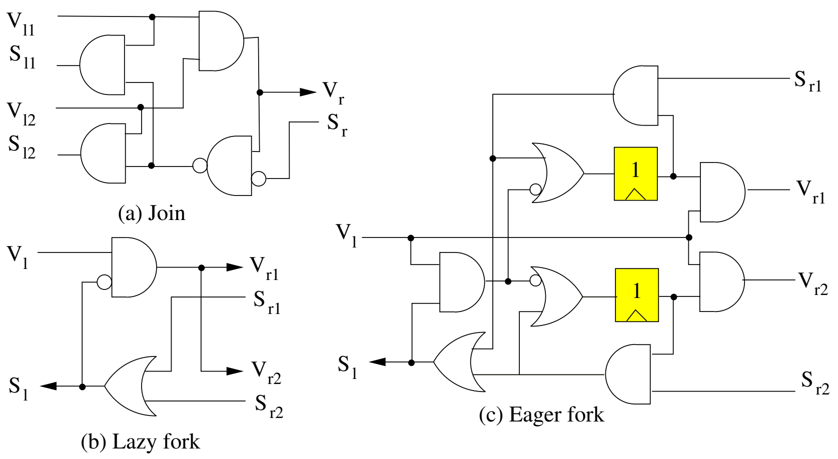


Figure 3.8: Implementation of elastic join and forks

Figure 3.7 shows two possible implementations of the elastic buffer control. Figure 3.7 (b) uses only latches, but Figure 3.7 (a) can be synthesized using either latches or flip-flops, as the pair of latches are connected with no gates in between.

Other Structures

Elastic buffers can be generalized to any number of input and output channels by connecting additional control in the inputs and the outputs. This control must be able to generate one single pair of valid and stop bits from several pairs of them, and to generate several valid and stop bits from a single pair of them.

Figure 3.8(a)shows the control for a join structure. A pair of channels is translated to a single channel. The control transmits a token forward when both channels have a token. That means that the control waits until both senders are ready to transmit. If there is a token in channel one and channel two is idle, a stop bit must be sent through channel one.

Figures 3.8(b) and (c) show the control for two possible implementations of a fork. The lazy fork waits for both receivers before sending a token. If a receiver is ready and the other one is not, the ready receiver has to wait until the other one is prepared. On the other hand, the eager fork allows to send a token to the first receiver that is prepared while waiting for the other. Once the token has been sent to both receivers, the system can go on to the next token. Notice that a join connected to a lazy fork with no elastic buffer in between creates a combinational loop. Eager fork can be use instead to avoid combinational loops.

It is possible to create elastic modules with an arbitrary number of input and output channels by chaining several join or fork structures.

Figure 3.9 shows an elastic design. The elastic control layer is made of a collection of elastic buffers, fork and joins that are parallel to the dependencies in the datapath and it creates the enable wires for the pairs of latches. The internal channels contain a pair of wires (a valid bit and a stop bit) which are explicitly shown in the input and in the output.

In this thesis, logic in the datapath between pairs of latches has one cycle delay, i.e., it is only combinational. However, it is possible to support variable-latency units. A variable latency unit must implement a handshake with signals *go*, *done* and *ack*. When it receives a *go*, it can start computing, and when the output is the desired one it must set the *done* signal. Then, it has to maintain its output until it receives an *ack*. This is the typical handshake for variable-latency units.

In order to support this protocol, it is only necessary to connect the variable latency unit between a pair of elastic buffers. The input valid bit must be connected to *go*. *Done* must be connected to the output valid bit. When the unit is done and the output channel stop bit is unset, it must receive an *ack*, because the token is being transmitted. Finally, the stop bit in the input channel must be set either when the unit is computing or when it is receiving a stop signal from the output channel.

3.2 Elastic Machines

This section presents the results of Synchronous Elastic Networks by Krstic et al.[22], which introduces the theoretical foundations of elastic circuits.

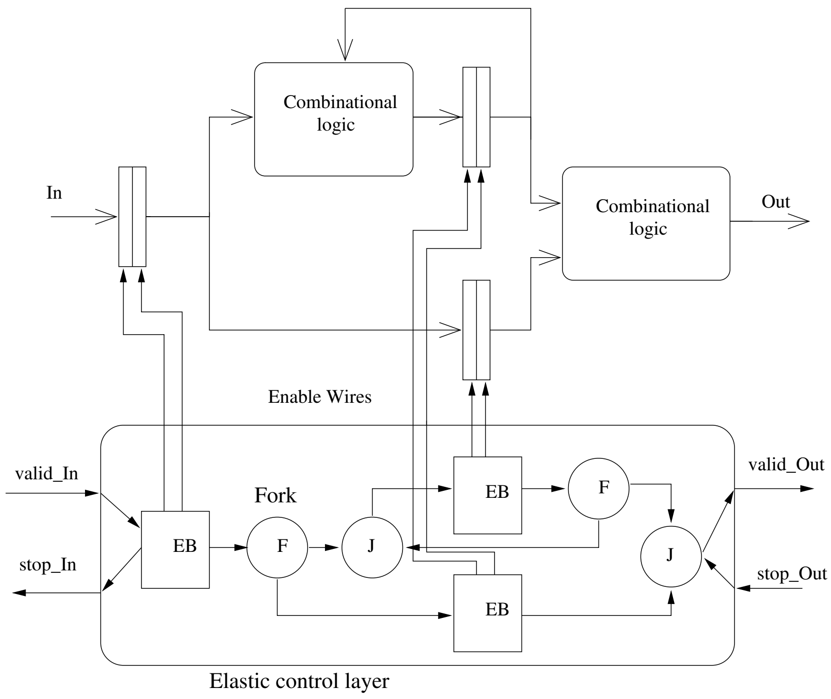


Figure 3.9: Elastic control layer (made of elastic buffers, forks and joins) and its connection to the datapath

Machines are a mathematical abstraction of circuits without combinational cycles, which are not taken into account because nowadays, logic synthesis and timing analysis cannot properly deal with them.

3.2.1 Circuits as Streams

A circuit can be modelled as a set of behaviors, where a behavior assigns to every wire of the circuit a stream, that is, an infinite sequence of values.

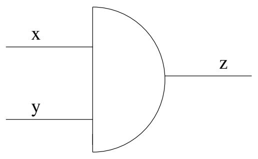
A *stream* over a set A is an infinite sequence of elements of A, that is, a sequence $a[0], a[1], a[2], \ldots a[i] \ldots, \forall i \geq 0, a[i] \in A$. For example, the stream $a[i] = i$ over the set of naturals defines a stream with the naturals in increasing order : $(0, 1, 2, \ldots)$.

Then, A^∞ denotes the set of all possible streams over A, and A^ω the set of all possible sequences over A. Notice that streams are *infinite sequences*, and A^ω also contains finite sequences, so $A^\infty \subset A^\omega$.

It is useful to define the prefix relation between streams. $a \sim_k b$ means that the first k elements of a and b are the same. Notice that $a \sim_{k+1} b \Rightarrow a \sim_k b$. The $\sim_k$ relation can be seen as equivalence relations, and as k is bigger the related elements are more similar to each other. In order to prove that two sequences are the same, it is enough to prove that for all k, $a \sim_k b$.

A circuit can be seen as a set W of *wires*, in which every wire $w \in W$ has a type type(w) . For example, a one-bit wire w_1 will have type$(w_1) = \{0, 1\}$, an eight-bit wire w_2 representing natural numbers will have type$(w_2) = [0, 2^8 - 1]$, and so on so forth. A *behavior* over a set of wires W (a *W-behavior*) is a function σ that associates to every wire $w \in W$ a stream over type(w), that is, $\forall w \in W, \sigma.w \in \text{type}(w)^\infty$. Therefore, a W-behavior shows what value has every wire at any instant, and it is similar to the concept of trace that is usually used in hardware design. Notice that $\sim_k$ can be extended to behaviors. $\sigma \sim_k \tau$ if and only if $\forall w \in W, \sigma.w \sim_k \tau.w$. That is, two behaviors have a common prefix of length k if the streams associated to every wire have a common prefix of length k.

The set of all possible W-behaviors for a give set of wires W is denoted $[\![W]\!]$. A *W-system* is a subset of $[\![W]\!]$, and it characterizes a circuit. It states which W-behaviors are possible for W and which ones are not possible. For example, a circuit made of a pair of independent wires ($W = w_1, w_2$) in which type(w_1) = type$(w_2) = \{0, 1\}$ would be modeled with the W-system $\mathcal{S} = [\![W]\!]$, because the behavior of w_1 is not related to the behavior of w_2 and hence, any sequence of values in the wires is possible. On the other hand,



$W = \{x, y, z\}$
type(x) = type(y) = type$(z) = \{0,1\}$

$$\mathcal{S} = \Big\{(a,b,c) \in \{0,1\}^\infty \times \{0,1\}^\infty \times \{0,1\}^\infty \mid \forall i, c_i = a_i \wedge b_i\Big\}$$

$$\mathcal{S} \models \mathsf{G}\ (z = x \wedge y)$$

Figure 3.10: System $\mathcal{S}$ defining an and gate

if w_1 and w_2 are two connected wires, then there must be the same value in both wires all the time. Thus, the possible W-behaviors are the ones in which $\sigma.w_1 = \sigma.w_2$, so the W-system $\mathcal{S}$ is defined by all pairs of streams $(a, a), a \in \{0,1\}^\infty$. Figure 3.10 shows a W-System modelling an and gate, and two ways to define it using different notations.

Operations on Systems

Hiding and composition are two operations defined to be able to create W-systems from other W-systems. If $W' \subseteq W$, it is possible to define a mapping from a behavior over the set of wires W to a behavior the set of wires W', $\sigma \mapsto \sigma \downarrow W' : [\![W]\!] \rightarrow [\![W']\!]$. $\sigma \downarrow W'$ is the same behavior than σ but defined only over the set of wires W'.

If $\mathcal{S}$ is a W-system and $W' \subseteq W$, *hiding* W' produces a $(W - W')$-system defined by the behaviors that can be mapped from a behavior in $\mathcal{S}$. Formally, $\text{hide}_{W'}(\mathcal{S}) = \{\tau \mid \exists \sigma \in \mathcal{S}, \tau = \sigma \downarrow (W - W')\}$. For example, hiding y in the system of Figure 3.10 produces a system in which z is 0 whenever x is 0, and it can take any value when x is 1.

If $\mathcal{S}_1$ is a W_1-system and $\mathcal{S}_2$ is a W_2-system, the *composition* of $\mathcal{S}_1$ and $\mathcal{S}_2$ is a $(W_1 \cup W_2)$-system defined as : $\mathcal{S}_2 \sqcup \mathcal{S}_2 = \{\sigma \mid \sigma \downarrow W_1 \in S_1 \wedge \sigma \downarrow W_2 \in S_2\}$. In other words, a behavior of a system composed by two systems is made of a behavior of the first system and a behavior of the second system. However, it is not true that all pairs of behaviors $(\sigma_1, \sigma_2), \sigma_1 \in \mathcal{S}_1 \wedge \sigma_2 \in \mathcal{S}_2$ define a behavior of the composed system. Given a wire $w \in W_1 \cap W_2$ and a pair of behaviors $\sigma_1 \in \mathcal{S}_1$, $\sigma_2 \in \mathcal{S}_2$, if $\sigma_1(w) \neq \sigma_2(w)$, the pair σ_1, σ_2 does not describe a behavior of the composed system. In other words, two behaviors of the components must agree about the shared wires in order to constitute a behavior of the composed system.

If the sets of wires are disjoint, all pairs of behaviors form a behavior of the composed system. Given a behavior $\sigma \in (W_1)$-system and a behavior $\tau \in$

(W_2)-system where W_1 and W_2 are disjoint, there exists a unique behavior $\vartheta \in (W_1 \cup W_2)$-system such that $\vartheta \downarrow W_1 = \sigma$ and $\vartheta \downarrow W_2 = \tau$. As ϑ is a composition of σ and τ, we can write that $\vartheta = \sigma * \tau$.

Machines

Given a pair of disjoint sets called I (inputs) and O (outputs), an (I, O)-machine is an (I, O)-system $\mathcal{S}$ that satisfies the following properties :

functional :

$$\forall \sigma \in [\![I]\!], \exists! \tau \in [\![O]\!], \sigma * \tau \in \mathcal{S}$$

causal:

$$\forall \omega, \omega', \forall k \geq 0, \omega \downarrow I \sim_k w' \downarrow I \Rightarrow w \downarrow O \sim_k \omega' \downarrow O$$

In other words, all input behaviors must be possible, and for every behavior in the input there must be a unique behavior in the output. This property is called functional because it implies there is a total function between the inputs and the outputs. The causal property states that the outputs of the machine must be determined by the inputs that have been received so far in the inputs. That is, the first k values in the inputs determine the k-th value in the outputs for any behavior of the machine.

Figure 3.10 shows a system that is a machine. However, the system resulting from hiding one of the inputs is not a machine.

Notice that both functionality and causality imply determinism. For any behavior in the inputs, there is just one possible behavior of the outputs. Formally, $\forall \omega, \omega' \in \mathcal{S}, \omega \downarrow I = \omega' \downarrow I \Rightarrow \omega \downarrow O = \omega' \downarrow O$. Therefore, a machine can be defined as a function from input behaviors to output behaviors $F : [\![I]\!] \rightarrow [\![O]\!]$, such that $F(\sigma) = \tau$ when $\sigma * \tau \in \mathcal{S}$.

3.2.2 Elastic Machines

Every data wire in elastic machines has associated a *valid wire* and a *stop wire.* If I and O are two disjoint sets of wires, let I' be the set $I \cup \{\text{valid}_X \mid X \in I\} \cup \{\text{stop}_X \mid X \in I\}$, and let O' be the set $O \cup \{\text{valid}_X \mid X \in O\} \cup \{\text{stop}_X \mid X \in O\}$. A $[I, O]$-*system* is a (I', O')-machine. The *elastic input channels* are the triples $\langle X, \text{valid}_X, \text{stop}_X \rangle$, for all $X \in I$. Thus, the *elastic output channels* are the triples $\langle X, \text{valid}_X, \text{stop}_X \rangle$, for all $X \in O$.

The type of all valid and stop wires is Boolean. Given a channel X, a *token is transferred during the i-th instant* if $\text{valid}_X \wedge \neg\text{stop}_X$ holds during this instant. Otherwise, there is a *bubble* on channel X. In order to make formulas clearer, we define $\text{transfer}_X \Leftrightarrow \text{valid}_X \wedge \neg\text{stop}_X$.

The *transfer behavior* associated to a behavior σ of an $[I, O]$-system is the set of sequences resulting from extracting bubbles from all streams of σ. In other words, if σ is a behavior of an $[I, O]$-system, for every data wire X, the i-th element of the stream $\sigma.X$ is in $\sigma^T.X$ if $\sigma.\text{transfer}_X[i]$ holds.

An auxiliary counter variable tct_X of integer type is defined to express system properties related to transfers. For every behavior σ, $\sigma.tct_X = (t[0], t[1], \ldots)$, where $t[i]$ is the number of tokens that have been transferred until that state, that is, $t[i] = \#\{j \mid j < i \wedge \sigma.\text{transfer}_X[j]\}$. Let min_tct_S, for any $S \subseteq I \cup O$ be $\min\{tct_X \mid X \in S\}$, the number of transfers in the wire that has had less transfers among those in $\mathcal{S}$.

Persistence

All output wires of an elastic system have a persistent behavior. If an output wire tries to transfer a token and it cannot do so because the stop wire is true, then it must try to transfer the same token on next cycle. Given an $[I, O]$-system $\mathcal{S}$, for every $Y \in O$, the following proposition written in linear temporal logic [31] expresses the *persistence condition*:

$$\mathcal{S} \models \mathsf{G}\ (\text{valid}_Y \wedge \text{stop}_Y \Rightarrow (\text{valid}_Y)^+ \wedge Y^+ = Y) \tag{3.1}$$

Y^+ means the *next-state value* of variable Y. Notice that persistence implies that if there is a token waiting to be transferred and it is always the case that eventually the stop signal of the output is false, a transfer will occur. This is what the *handshake lemma* states :

$$\mathcal{S} \models \mathsf{G}\ \mathsf{F}\ \text{valid}_Y \wedge \mathsf{G}\ \mathsf{F}\ \neg\text{stop}_Y \Rightarrow \mathsf{G}\ \mathsf{F}\ \text{transfer}_Y \tag{3.2}$$

Liveness

Liveness conditions of elastic components express, on the one hand, that tokens that have been received in the inputs must eventually be ready for transmission in the outputs, and on the other hand, that input channels that have received less tokens that other channels must eventually be ready to receive a token. The following expressions state these conditions :

$$\mathcal{S} \models \mathsf{G}\ (min\_tct_O = tct_Y \wedge min\_tct_I > tct_Y \Rightarrow \mathsf{F}\ \text{valid}_Y) \tag{3.3}$$

$$\mathcal{S} \models \mathsf{G}\ (min\_tct_{I \cup O} = tct_X \Rightarrow \mathsf{F}\ \neg \mathrm{stop}_X) \tag{3.4}$$

In practice, elastic components satisfy simpler and stronger liveness properties. For example, Figure 3.11 shows the abstract model for an elastic buffer by Cortadella et al. [13] . Its liveness properties are similar to the ones stated in equations 3.3 and 3.4, but are simpler because there is just one input channel and one output channel in the output.

In this abstract model, an EB is a FIFO implemented as an array of data, with a *read pointer* and a *write pointer*. Whenever a valid data item arrives from the input channel (V_{in} holds), if the buffer is prepared to receive it (S_{in} does not hold), the buffer writes the new data item in the array and updates the write pointer. In the same way, whenever V_{out} holds and the environment is ready to receive new tokens (S_{out} does not hold), the buffer updates the read pointer to the next item to transmit. If there is a token to transmit and the environment is not ready to receive it, the EB must follow the persistence rule and retry until the token has been transmitted.

The '*' symbol stands for do not care value, and therefore its value is decided by implementations. When the FIFO is not empty and it is not full, implementations can decide whether V_{out} must be set, as long as it follows the forward latency rule: if the FIFO is not full, a token must be eventually transmitted. In the same way, the implementation can decide whether to set S_{in} and force the input channel transmitter to wait. However, by the backward latency property, if the output channel is ready to receive a token, the input channel must eventually be ready to receive a token.

Definition of Elastic Machine

Consider a single channel $[I, O]$-system with wire Z satisfying conditions 3.1, 3.3, 3.4. An *elastic consumer* $\mathcal{C}$ for this channel is a system with a single input wire Z and no output wires satisfying:

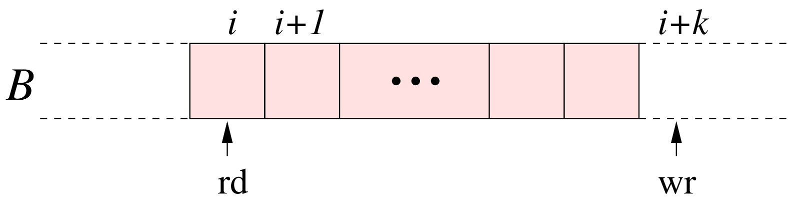
$$\mathcal{C} \models \mathsf{G}\ \mathsf{F}\ \neg \mathrm{stop}_Z \tag{3.5}$$

In a similar way, an *elastic producer* $\mathcal{P}$ for a channel Z is a system with a single output wire Z and no input wires satisfying:

$$\mathcal{P} \models \mathsf{G}\ (\mathrm{valid}_Z \wedge \mathrm{stop}_Z \Rightarrow (\mathrm{valid}_Z)^+) \tag{3.6}$$

$$\mathcal{P} \models \mathsf{G}\ \mathsf{F}\ \mathrm{valid}_Z \tag{3.7}$$

Informally, a consumer of a channel is a system that will be ready to consume any token in the channel. There are no restrictions about when tokens must



State variables:

B : *array* $[0 \dots \infty]$ of *data*; rd, wr : $\mathbb{N}$; retry : $\mathbb{B}$;

Initial state: rd = wr = 0; retry = *false*;

Invariant: wr $\geq$ rd

Combinational behavior:

$$V_{out} = \begin{cases} \textit{true} & \text{if retry} \\ \textit{false} & \text{if rd = wr} \\ * & \text{otherwise} \end{cases} \qquad D_{out} = \begin{cases} \text{B[rd]} & \text{if } V_{out} \\ * & \text{otherwise} \end{cases}$$

$$S_{in} = *$$

Sequential behavior:

$$\text{rd}^+ = \begin{cases} \text{rd}+1 & \text{if } V_{out} \wedge \neg S_{out} \\ \text{rd} & \text{otherwise} \end{cases} \qquad \text{retry}^+ = V_{out} \ \wedge \ S_{out}$$

$$\text{wr}^+ = \begin{cases} \text{wr}+1 & \text{if } V_{in} \wedge \neg S_{in} \\ \text{wr} & \text{otherwise} \end{cases} \qquad \text{B}^+[\text{wr}] = D_{in}$$

Liveness properties (finite unbounded latencies):

Forward latency: $\text{G}(\text{rd} \neq \text{wr} \ \Rightarrow \ \text{F} \ V_{out})$

Backward latency: $\text{G}(\neg S_{out} \ \Rightarrow \ \text{F} \ \neg S_{in})$

Figure 3.11: Abstract model for an EB.

be consumed, but there cannot be a transmission after which the consumer will never be ready to consume another token. A producer of a channel is a system that will be persistent about sending tokens, and furthermore, it will serve an infinite stream of tokens to the channel.

Let $\mathcal{C}_Z$ be the $\{Z, \mathrm{valid}_Z, \mathrm{stop}_Z\}$-system including the most possible number of behaviors and satisfying condition 3.5, and similarly, let $\mathcal{P}_Z$ be the $\{Z, \mathrm{valid}_Z, \mathrm{stop}_Z\}$-system including the most possible number of behaviors and satisfying conditions 3.6, 3.7. An $[I, O]$-*elastic environment* is the system :

$$Env_{I,O} = \bigsqcup_{X \in I} P_X \sqcup \bigsqcup_{Y \in O} C_Y$$

$Env_{I,O}$ is not a machine, because it is not functional. However, it is a system that will produce an infinite stream of tokens for all input channels and will consume all tokens of output channels of an $[I, O]$-system. That is what the *Liveness lemma* states: if $\mathcal{S}$ is an $[I, O]$-system satisfying persistence and liveness conditions, then for every behavior ω of $\mathcal{S} \sqcup Env_{I,O}$, all the component sequences of the transfer behavior ω^T are infinite.

Therefore, it is possible to define an (I, O)-system with the transfer behavior of the elastic system $\mathcal{S}^T = \{\omega^T \mid \omega \in \mathcal{S} \sqcup Env_{I,O}\}$.

Finally, an $[I, O]$-*elastic machine* is an $[I, O]$-system that satisfies properties 3.1, 3.3, 3.4 and its associated system $\mathcal{S}^T$ is deterministic. That is, after removing the bubbles from input and output streams, there is a one to one correspondence between input transfer behaviors and output transfer behaviors.

The main result about elastic machines is that if $\mathcal{S}$ is an $[I, O]$-elastic machine, then $\mathcal{S}^T$ is an (I, O)-machine. Then, we can say that $\mathcal{S}$ is an *elasticization* of $\mathcal{S}^T$ and that $\mathcal{S}^T$ is the *transfer machine* of $\mathcal{S}$.

3.2.3 Networks of Machines

Given a set of machines with disjoint wires, it is interesting to see whether a composition of such machines plus connecting some wires produces a machine. This section presents results that show when a network of machines is a machine, both in the case of non-elastic and elastic machines.

Let $\mathrm{Conn}(u, v)$ be the $\{u, v\}$-system that connects u and v, that is, $\forall \sigma \in \mathrm{Conn}(u, v), \sigma.u = \sigma.v$. Then, if $\mathcal{S}_1, \mathcal{S}_2, \ldots \mathcal{S}_m$ are machines (or elastic machines) with disjoint wire sets, a network of machines is defined as $\mathcal{N} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_m \mid u_1 = v_1, \ldots, u_n = v_n \rangle$, where $u_1, \ldots, u_n$ are n input wires, and

$v_1, \ldots, v_n$ are n output wires, and $\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \mid u_1 = v_1, \ldots, u_n = v_n \rangle$ stands for $\text{hide}_{\{u_1,\ldots,u_n,v_1,\ldots,v_n\}}(\mathcal{S})$, where $\mathcal{S} = \mathcal{S}_1 \sqcup \ldots \sqcup \mathcal{S}_m \sqcup \text{Conn}(u_1, v_1) \sqcup \ldots \sqcup \text{Conn}(u_n, v_n)$.

Combinational Loop Theorem

A pair of wires (u, v) is *sequential* for $\mathcal{S}$, where u is an input wire, v an output wire and $F : [\![I]\!] \rightarrow [\![O]\!]$ the function characterizing $\mathcal{S}$, if for every $\sigma, \sigma' \in [\![I]\!]$ and every $k \geq 0$,

$$\sigma.u \sim_{k-1} \sigma'.u \wedge \forall x \in I - \{u\}, (\sigma.x \sim_k \sigma'.x \Rightarrow F(\sigma).v \sim_k F(\sigma').v$$

This means that the value of v at a given instant does not depend on the value u at this instant, but only on the value of the other inputs and the value of u on previous instants.

A *feedback* system for a system $\mathcal{S}$ is the system $\langle \mathcal{S} \mid u = v \rangle$, where $\mathcal{S}$ is a machine, u is an input and v an output of $\mathcal{S}$. Figure 3.12(a) shows a representation of a feedback system. The *feedback lemma* states that if the pair (u, v) is sequential, then the system $\langle \mathcal{S} \mid u = v \rangle$ is a machine.

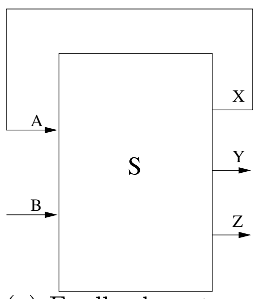
Given an (I, O)-machine $\mathcal{S}$, its *dependency graph* $\Delta(\mathcal{S})$ is the directed graph defined by $\Delta(\mathcal{S}) = G(V, E), V = I \cup O, E = \{(u, v) \mid u \in I \wedge v \in O \wedge (u, v) \text{ is not sequential}\}$. That is, it is a graph whose vertices are wires, and the input wires are connected to those output wires with which they are not sequential. Figure 3.12(b) shows a module and its dependency graph.

For a network $\mathcal{N} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_m \mid u_1 = v_1, \ldots, u_n = v_n \rangle$, the graph $\Delta(\mathcal{N})$ is the direct sum of graphs $\Delta(\mathcal{S}_i)$, in which vertices u_i and v_i are identified, $\forall i \in [1, n]$. Figure 3.13 shows a network of systems and its dependency graph.

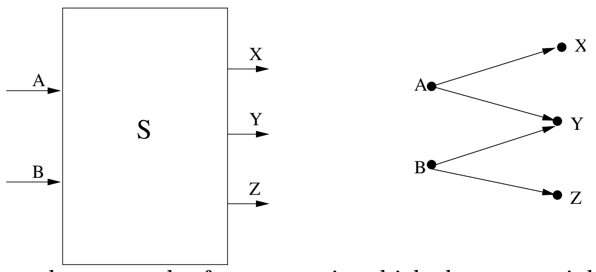
The *Combinational Loop Theorem* states that the network system $\mathcal{N}$ is a machine if the graph $\Delta(\mathcal{N})$ is acyclic. The intuition after this theorem is that a machine must have no combinational loops. The network of Figure 3.13 has not cycles in $\Delta(\mathcal{N})$, therefore, it is a machine.

Networks of Elastic Machines and Buffer Insertion

In the same way than in conventional machines, an elastic feedback is an elastic machine with an output channel connected to an input channel $\langle S \mid P = Q \wedge \text{valid}_P = \text{valid}_Q \wedge \text{stop}_P = \text{stop}_Q \rangle$. Similarly, an input channel P



(a) Feedback system



(b) Dependency graph of a system, in which the sequential pairs are (A,Z) and (B,X)

Figure 3.12: Feedback system and dependency graph

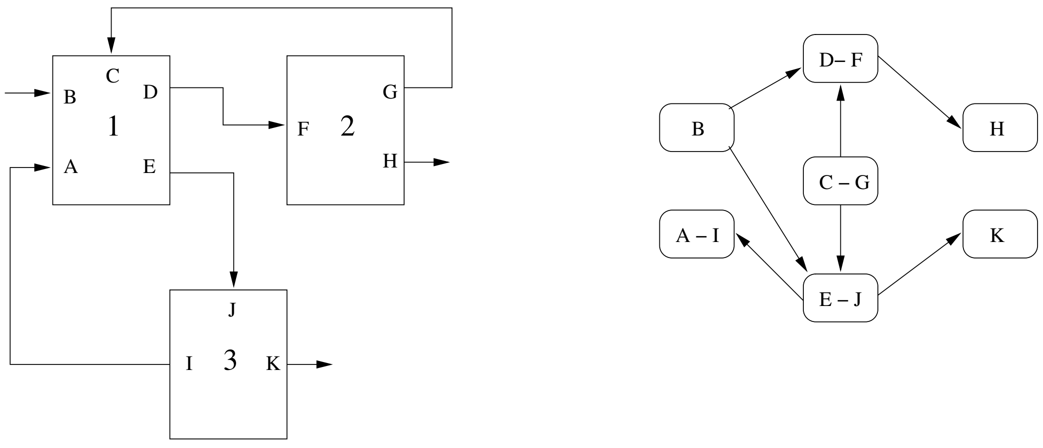


Figure 3.13: Network of machines and its dependency graph $\Delta(\mathcal{N})$. The sequential pairs are (A,D), (A,E) and (F,G)

and an output channel Q are *sequential* if channel Q does not need to wait for the first token of P to transmit its first token.

However, the fact that pairs (P, Q) and (P', Q) are both sequential does not imply that they are simultaneously sequential. Therefore, for every output channel it is necessary to know its *sequentiality information*, that is, for any $Y \in O$, it is needed a set $\delta(Y) \subseteq I$ such that all elements of $\delta(Y)$ are simultaneously sequential with Y. Note that for every $P \in \delta(Y)$, (P, Y) is a sequential pair, but it is not true that these are the only inputs that are sequential with Y. In order to be simultaneously sequential, δ function must commit the following property for any $Y \in O$:

$$\mathcal{S} \models \mathsf{G}\ (min\_tct_{I \cup O} = tct_Y \wedge min\_tct_{I-\delta(Y)} > tct_Y \Rightarrow \mathsf{F}\ \text{valid}_Y) \tag{3.8}$$

Then, given an $[I, O]$-machine $\mathcal{S}$ with a sequentiality interface δ, $\Delta^e(\mathcal{S}, \delta)$ is the directed graph $G = (V, E), V = I \cup O, E = \{(X, Y) \mid X \in I \wedge Y \in O \wedge X \notin \delta(Y)\}$. Then, given an elastic network $\mathcal{N} = \langle\langle \mathcal{S}_1, \dots \mathcal{S}_m \mid X_1 = Y_1, \dots, X_n = Y_n\rangle\rangle$, where there is a sequentiality interface δ_i for every $\mathcal{S}_i$, its elastic dependency graph $\Delta^e(\mathcal{N})$ is the sum of graphs $\Delta(\mathcal{S}_i, \delta_i)$ in which each vertex X_i is identified with vertex Y_i, $\forall i \in [1, n]$.

Theorem 4 of Krstic et al. states that if $\Delta(\mathcal{N})$ and $\Delta^e(\mathcal{N})$ are both acyclic, then the network system $\mathcal{N}$ is an elastic machine and the corresponding non-elastic system $\mathcal{N}^T$ is a machine.

Notice that it is not enough to see that $\Delta^e(\mathcal{N})$ is acyclic, because there might be combinational loops that are not seen in this graph. Cycles in $\Delta^e(\mathcal{N})$ and cycles in $\Delta(\mathcal{N})$ might have no correspondence at all.

An *empty elastic buffer* is an elastic machine $\mathcal{S}$ such that $\mathcal{S}^T = \text{Conn}(X, Y)$ for some X, Y. By the *Buffer Insertion Theorem*, which is consequence of the previous theorem, inserting an empty elastic buffer in a channel of a network $\mathcal{N}$ does not change the functionality of the network and the resulting network will be an elastic machine as well.

3.3 The SELF Tool

The `SELF` tool generates the elastic control logic from an elastic netlist. Roughly, an elastic netlist is a connectivity graph of a circuit with some attributes on its modules and channels. It has been developed by UPC and Intel SCL to simulate, synthesize and verify `SELF` circuits.

3.3.1 SELF netlists

A `SELF` netlist is a graph whose vertices are elastic modules and whose edges are elastic channels. Every module and channel has a set of attributes.

The `SELF` netlist specification is a list of module and channel attributes. A channel attribute specification declares which are the source pin, the target pin and lists the channel attributes. A module attribute specification contains the module name and a list of the module attributes.

`SELF` modules represent computation units or environment modules. Its inputs and outputs are elastic channels. The generated control consists of a set of join structures in the input channels and a set of fork structures in the output channels. In between, there can be a chain of latches (elastic half buffers) before and after the computation unit, which can also have a set of internal latches.

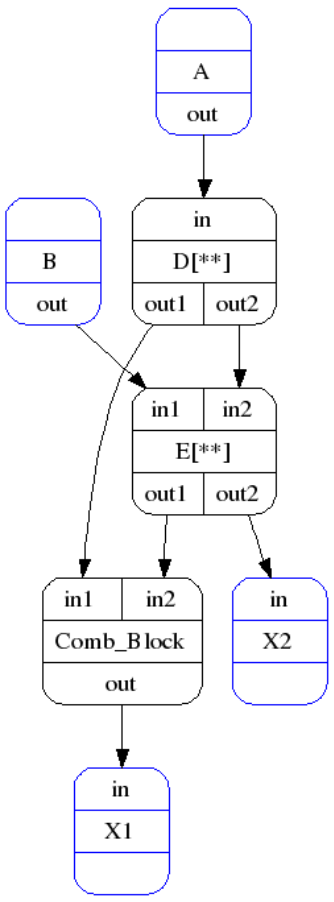
The computation unit can be defined to have fixed or variable delay. Variable delays are specified with a list of delay probabilities. The default delay value is one. Before and after the computation unit there is a chain of latches. Adjacent latches have alternating phases. After an active high latch, there will be an active low latch, and so on so forth.

Figure 3.14 shows a self netlist with its graphical representation. There are four environment modules, A, B, X1 and X2, which have variable delays. Variable delays on environment modules point that they will not be able to send and receive during all cycles. All modules but Comb_Out have two output latches (notated with stars).

In an elastic design, a module waits for all its inputs to be valid and then it computes its outputs and sets them to valid. However, it is possible that depending on some input values, it is not necessary to wait all inputs. The simplest example is a multiplexer, when only one of the data wires is needed if the value of the control wire is known. Even though it is not used in this thesis, `SELF` also provides machinery to control such early evaluations and that can also be specified in `SELF` netlists.

3.3.2 SELF execution

The main use of the `SELF` tool is to parse a `SELF` netlist and generate its elastic control in several HDL languages. This elastic control takes as inputs the valid bit of the inputs and the stop bit of the outputs and generates the stop bit of the inputs, the valid bit of the outputs and the enable signals for



```
// environment
A  {olat=0 env}
B  {olat=0 env}
X1 {olat=0 env}
X2 {olat=0 env}

// module attributes
Comb_Block {olat=0 phase=H}

// channels
A.out          D.in
B.out          E.in1
D.out1         Comb_Block.in1
D.out2         E.in2
E.out1         Comb_Block.in2
Comb_Block.out X1.in
E.out2         X2.in

// inputs and outputs with
// variable delay
A  {delay="1:0.5 2:0.5"}
B  {delay="1:0.5 2:0.5"}
X1 {delay="1:0.5 2:0.5"}
X2 {delay="1:0.5 2:0.5"}
```

Figure 3.14: SELF netlist specification

the latches in the datapath.

The `SELF` tool can also simulate that control using random stimuli or any other stimuli specified by the user, and it can display a picture of the graph in the screen. However, big graphs are usually difficult to understand through a picture.

`SELF` is programmed over the Python interpreter, and it uses `SIS` [33], a system that optimizes sequential digital circuits.

Chapter 4

Elasticization Process

The elasticization process is embedded into the Esterel compiler HDL generator. The Esterel compiler generates an intermediate structure from the Esterel code. Then, HDL code is generated from that intermediate code. The `SELF` option added so that compiler modifies the Verilog generation to make it elastic. Changes in the datapath are minimal, and the main difference is an elastic control layer over the original circuit. Figure 1.1 shows the whole process.

4.1 Datapath

There are few modifications to be made in the datapath. All combinational operations from the Esterel equations remain unchanged. It is only necessary to make one main modification: substituting registers for pairs of latches.

The sequential part from the original HDL generator is not created. Instead, a Verilog module called Double_latch is instantiated for every register that would be created. Figure 4.1 shows the Verilog code for a rising edge Double_latch module. It takes as input the data part of an elastic channel, a pair of enable wires that are connected to latches, an init value and the clock and reset signals. It generates the data part of the output elastic channel.

Figure 4.2 shows an Esterel program that will be used as an example in this chapter. Figure 4.3 shows a circuit implementing this pipeline.

Besides from replacing the sequential part for a set of latch pairs instantiations, it is necessary to add valid and stop bits to the module interface. For example, given the Esterel code in Figure 4.2, the interface changes can be

```
module Double_latch_H (Q, D, En1, En2, Init, Clk, Reset);
  parameter w=1;
  output [w-1:0] Q;
  input  [w-1:0] D;
  input          En1;
  input          En2;
  input  [w-1:0] Init;
  input          Clk;
  input          Reset;
  wire   [w-1:0] mid;
  Latch_H #(w) lh(Q, mid, En2, Init, Clk, Reset);
  Latch_L #(w) ll(mid, D, En1, Init, Clk, Reset);
endmodule //Double_latch_H
```

```
module Latch_H (Q, D, En,
               Init, Clk, Reset);
  parameter w=1;
  output reg [w-1:0] Q;
  input [w-1:0] D;
  input         En;
  input [w-1:0] Init;
  input         Clk;
  input         Reset;
  always @(D, En, Clk, Reset)
    if (Reset) Q <= Init;
    else if (Clk & En) Q <= D;
endmodule // Latch_H
```

```
module Latch_L (Q, D, En,
               Init, Clk, Reset);
  parameter w=1;
  output reg [w-1:0] Q;
  input [w-1:0] D;
  input         En;
  input [w-1:0] Init;
  input         Clk;
  input         Reset;
  always @(D, En, Clk, Reset)
    if (Reset) Q <= Init;
    else if (~Clk & En) Q <= D;
endmodule // Latch_L
```

Figure 4.1: Synthesizable Verilog code for a rising edge register made using a pair of latches

seen in Figure 4.4.

The HDL generator declares a pair of enable wires for most registers. Some registers share the same wires. For example, multidimensional variables are sliced until the bit-vector level, and all slices use the same pair of enable wires.

Esterel can deal with synchronous or asynchronous reset signals. On the one hand, registers are reset on rising edges when using a synchronous reset. On the other hand, registers are reset whenever the reset signal rises when using an asynchronous reset signal. As the Double_latch module use a synchronous reset, the asynchronous reset option is disabled when using elastic Esterel.

The Esterel sequential code for the pipeline in Figure 4.2 is the one in Figure 4.5. Its registers are `V7_Out_2_data_val`, `V7_Reg1_3_data_val`, `V7_Reg2_4_data_val`, `V7_F1_5_data_val`, `Boot_0_0`. Figure 4.6 shows the equivalent Double_latch instantiations of these registers. Notice how the input and output wires and the initial values are equivalent.

```
module my_strl_pipe :
constant n : unsigned <> = 3;
input In1 :  temp unsigned <[n]>;
input In2 :  temp unsigned <[n]>;
output Out : value reg unsigned <[n+3]> init 0;

  signal Reg1 : reg unsigned <[n]> init 0,
         Reg2 : reg unsigned <[n]> init 0,
         F1 : reg unsigned <[n+1]> init 0,
  in
    sustain {
      next ?Reg1 <= ?In1 if In1,
      next ?Reg1 <= 0 if not In1,
      next ?Reg2 <= ?In2 if In2,
      next ?Reg2 <= 0 if not In2,
      next ?F1 <= ?Reg1 + ?Reg2,
      next ?Out <= ?F1 * 4,
    }
  end signal;
end module
```

Figure 4.2: Esterel code for a simple pipeline

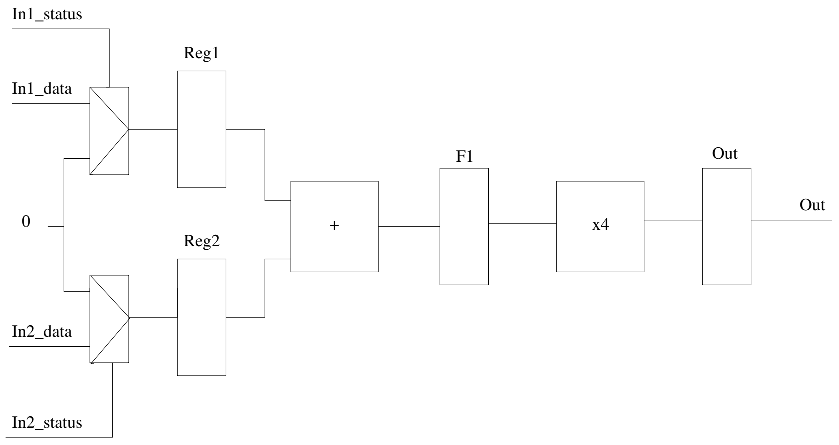


Figure 4.3: Circuit implementation of the sample pipeline

```
module my_strl_pipe (                module my_elastic_pipe (
  clk,                                 clk,
  rst,                                 rst,
  In1,                                 In1, valid_In1, stop_In1,
  In1_data,                            In1_data, valid_In1_data, stop_In1_data,
  In2,                                 In2, valid_In2, stop_In2,
  In2_data,                            In2_data, valid_In2_data, stop_In2_data,
  Out_data);                           Out_data, valid_Out_data, stop_Out_data);
```

Figure 4.4: Difference between conventional and elastic interfaces

```
// = Sequential process =
always @(posedge clk)
begin
  if ( rst == 1'b1 ) begin
    V7_Out_2_data_val <= V7_U2U(0, 6);
    V7_Reg1_3_data_val <= V7_U2U(0, 3);
    V7_Reg2_4_data_val <= V7_U2U(0, 3);
    V7_F1_5_data_val <= V7_U2U(0, 4);
    Boot_0_0 <= 1'b1;
  end
  else begin
    V7_Out_2_data_val <= V7_Out_2_data_next;
    V7_Reg1_3_data_val <= V7_Reg1_3_data_next;
    V7_Reg2_4_data_val <= V7_Reg2_4_data_next;
    V7_F1_5_data_val <= V7_F1_5_data_next;
    Boot_0_0 <= V7_Boot_0_0_last;
  end
end
```

Figure 4.5: Sequential part of the Verilog code for the pipe module

4.2 Control

The main change to the circuit when elasticizing is adding a control layer over it. This layer takes valid and stop bits from the environment and computes the enable wires for the latches in the datapath.

The `SELF` tool (see Section 3.3) is used to generate the elastic control from a `SELF` netlist. Therefore, the goal is to generate a `SELF` netlist, which is a connectivity graph of the circuit, from the Esterel code.

`Eicperf` is an experimental part of the Esterel compiler that generates C++ code out of Esterel programs to make performance evaluation. `Eicperf` generates a signal dependency graph used to get some features of Esterel variables such as combinational depth.

For every signal and variable in Esterel, the data structure knows which inputs and registers it depends on. Furthermore, if the signal is part of an array, it knows its master, which is the signal that refers to the array. For example, signal A[4:0] will create several signals: A, whose master is itself, A[0] whose master is A, A[1], ...

Then, for every master that will become a register, it is necessary to get which masters it is connected from, adding them to a list. Once it is done, every master has its reverse adjacency list (that is, who is connected to him). `Eicperf` can print a `SELF` netlist with that information. Every channel has two pin names and the convention used is : if there is a channel from module A to module B, then the self channel is written this way : "A.B_out B.A_in".

```
// = Sequential Double Latch Instanciation =
Double_latch_H #(6) EL_V7_Out_2_data_val(
  V7_Out_2_data_val,
  V7_Out_2_data_next,
  EL_EN_0_V7_Out_2_data_val,
  EL_EN_1_V7_Out_2_data_val,
  V7_U2U(0, 6),
  clk,
  rst
);

Double_latch_H #(3) EL_V7_Reg1_3_data_val(
  V7_Reg1_3_data_val,
  V7_Reg1_3_data_next,
  EL_EN_0_V7_Reg1_3_data_val,
  EL_EN_1_V7_Reg1_3_data_val,
  V7_U2U(0, 3),
  clk,
  rst
);

Double_latch_H #(3) EL_V7_Reg2_4_data_val(
  V7_Reg2_4_data_val,
  V7_Reg2_4_data_next,
  EL_EN_0_V7_Reg2_4_data_val,
  EL_EN_1_V7_Reg2_4_data_val,
  V7_U2U(0, 3),
  clk,
  rst
);

Double_latch_H #(4) EL_V7_F1_5_data_val(
  V7_F1_5_data_val,
  V7_F1_5_data_next,
  EL_EN_0_V7_F1_5_data_val,
  EL_EN_1_V7_F1_5_data_val,
  V7_U2U(0, 4),
  clk,
  rst
);

Double_latch_H #(1) EL_Boot_0_0(
  Boot_0_0,
  V7_Boot_0_0_last,
  EL_EN_0_Boot_0_0,
  EL_EN_1_Boot_0_0,
  1'b1,
  clk,
  rst
);
```

Figure 4.6: Sequential part of the Verilog code for the elastic pipe module

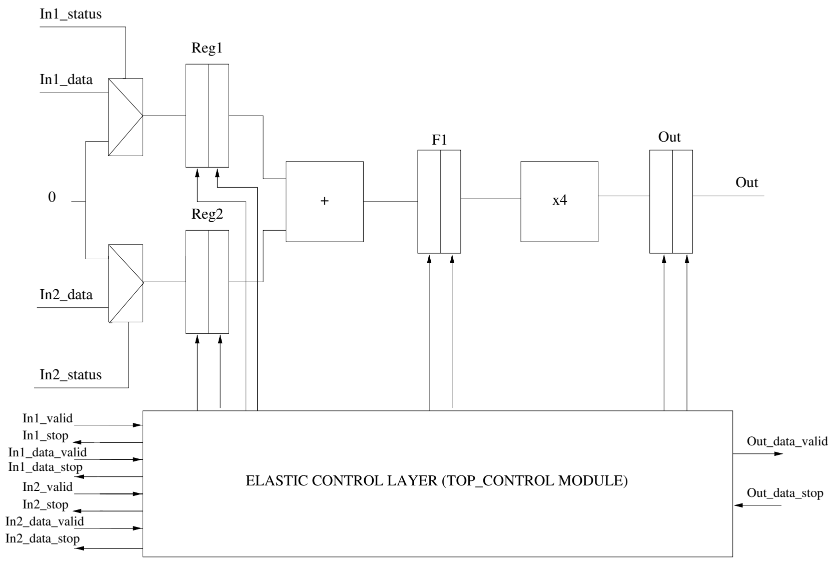


Figure 4.7: Elastic control interface of the pipeline example

Therefore, loops to itself are supported.

Every elastic module is assumed to be a combinational circuit between elastic buffers. Variable latency units are not supported. Thus, elastic modules have two output latches and no input latches. That means that elastic modules have the default attributes.

`SELF` creates a module called `TOP_CONTROL` with the interface of the elastic control layer. When generating the Verilog code in the Esterel compiler, it is necessary to instantiate `TOP_CONTROL` and connect it to the datapath. Figure 4.7 shows how `TOP_CONTROL` is connected to the datapath in the example in Figure 4.2.

`TOP_CONTROL` interface channels are the ones connecting environment modules to regular modules in the `SELF` netlist. For example, take an Esterel module with an environment module *In* connected to registers R_1 and R_2. The generated `TOP_CONTROL` will create valid and stop bits for the channel going from In to R_1 and valid and stop bits for the channel going from In to R_2. In order to avoid this effect, instead of translating inputs and outputs to environment modules, they are translated to an environment module connected to a combinational module that acts as an interface. Then, `TOP_CONTROL` has the desired interface.

4.3 Correctness

It is interesting to determine whether the elasticization procedure is correct, i.e., whether an elastic circuit generated by this procedure is an elasticization of the conventional sequential circuit the Esterel v7 compiler generates.

Esterel translation into sequential boolean circuits is proved to be correct [4]. Therefore, a circuit that has been translated from an Esterel program behaves like the constructive semantics of Esterel states.

Furthermore, all elastic modules in the elasticization process have one-cycle delay, there are no modules with variable-latency. Therefore, provided that the connectivity netlist generated by the Esterel performance evaluator is the one of the circuit, the correctness of the elastic control falls out from the general theory of elastic machines and `SELF` compiler.

Notice that what is elasticized is the whole Verilog design. In order to preserve Esterel semantics, the Esterel compiler adds control over the datapath. In the example used in this chapter, the boot register is a control register. In designs that are control intensive the number of these registers is bigger. If some signals are apparently independent at the datapath level, but there are dependencies between them because of Esterel semantics, these dependencies will be translated into the Esterel intermediate code, in the `SELF` netlist and in the elastic control layer because Esterel control registers will connect them, even if they are not connected in the datapath.

In consequence, dependencies in the Esterel semantics level are also taken into account by the elastic control layer when deciding which registers must be enabled, which channels have valid data and which ones need to be stopped.

Given that the Esterel translation into circuits is correct w.r.t. the Esterel semantics and the change in such circuit is to instantiate pairs of latches instead of flip-flops, which are equivalent; and given that the `SELF` compiler creates an elastic circuit that is correct-by-construction w.r.t. the general theory of elastic circuits, we can conclude that the elastic circuit is an elasticization of the Esterel program. Next chapter will validate this.

Chapter 5

Validation

Once a circuit has been elasticized using the procedure described in Chapter 4, one may want to validate that the elastic circuit is an elasticization of the original circuit. Formal verification is beyond the scope of this work, and it can be future work. Therefore, this chapter explains how it is validated that elasticizated circuits are functionally equivalent to the corresponding conventional circuit.

5.1 Latency Equivalence

For every conventional circuit there is a set of circuits that are elasticizations of it. Every circuit may be different in the granularity of the modules, the protocol it implements, . . . Therefore, there is an equivalence relation between all elasticizations of any circuit and the circuit itself.

Informally, two circuits are latency-equivalent [24] if given the same input behavior, their outputs have the same ordering of events. In terms of elastic circuits, two circuits are latency-equivalent if for any transfer behavior in the input, the transfer behavior in the output is the same.

The same property is defined in [39] as flow-equivalence. Two behaviors are flow-equivalent if and only if they have the same domain and every wire has the same values in the same order.

5.1.1 Latency Equivalence and Elasticization

Given a set of input wires I and a set of output wires O which are disjoint, then an $[I, O]$-elastic machine $\mathcal{M}_e$ with an $[I, O]$-elastic environment $Env_{I,O}$ like the one defined in Section 3.2.2 and a (I, O)-machine $\mathcal{M}$ are latency equivalent if and only if for every behavior $\sigma \in \mathcal{M}$ and every behavior $\tau \in \mathcal{M}_e \sqcup Env_{I,O}$, $\tau^T \downarrow I = \sigma \downarrow I \Rightarrow \tau^T \downarrow O = \sigma \downarrow O$. That is, given a pair of behaviors, if the input behavior of $\mathcal{M}$ is equal to the input transfer behavior of $\mathcal{M}_e$, the output behavior of $\mathcal{M}$ must be equal to the output transfer behavior of $\mathcal{M}_e$.

An elastic machine has a unique transfer machine, and it is interesting to determine whether latency equivalence can decide if a circuit is the transfer machine of an elastic design.

Therefore, we must prove : $\mathcal{M}_e$ and $\mathcal{M}$ are latency equivalent if and only if $\mathcal{M}_e^T = \mathcal{M}$, where $\mathcal{M}_e^T = \{\omega^T \mid \omega \in \mathcal{M}_e \sqcup Env_{I,O}\}$, the transfer machine of $\mathcal{M}_e$.

Proof

$\Rightarrow$

$(\forall \sigma \in \mathcal{M}, \forall \tau \in \mathcal{M}_e \sqcup Env_{I,O},\ \tau^T \downarrow I = \sigma \downarrow I \Rightarrow \tau^T \downarrow O = \sigma \downarrow O) \Rightarrow \mathcal{M}_e^T = \mathcal{M}$

Assuming that $\mathcal{M}_e$ and $\mathcal{M}$ are latency equivalence, we must see that all behaviors of $\mathcal{M}_e^T$ are behaviors of $\mathcal{M}$ and the other way around.

$\mathcal{M}_e^T \subseteq \mathcal{M}$: Let ω be a behavior of $\mathcal{M}_e$, then ω^T must belong to $\mathcal{M}$. $\mathcal{M}$ is a machine, and hence, it is functional. Therefore, for any $\sigma \in [\![I]\!]$, there exists a unique $\tau \in [\![O]\!]$ such that $\sigma * \tau \in \mathcal{M}$. In particular, there is a σ such that $\sigma = \omega^T \downarrow I$ and $\sigma * \tau \in \mathcal{M}$. As $\mathcal{M}_e$ and $\mathcal{M}$ are latency equivalent, $\sigma = \omega^T \downarrow I$ implies that $\tau = \omega^T \downarrow O$. If $\sigma = \omega^T \downarrow I$, $\tau = \omega^T \downarrow O$ and the set of wires is $I \cup O$, then $\omega^T = \sigma * \tau$, therefore, as $\sigma * \tau \in \mathcal{M}$, $\omega^T \in \mathcal{M}$.

$\mathcal{M} \subseteq \mathcal{M}_e^T$: Let ϑ be a behavior of $\mathcal{M}$, then ϑ must belong to $\mathcal{M}_e^T$. Given that $\mathcal{M}_e^T = \{\omega^T \mid \omega \in \mathcal{M}_e \sqcup Env_{I,O}\}$, $\mathcal{M}_e$ is an $[I, O]$-elastic machine and $Env_{I,O}$ is the largest (in behavior inclusion) elastic environment, $\mathcal{M}_e^T$ is a machine by Theorem 2 of Krstic et al. [22]. Then, $\mathcal{M}_e^T$ is functional: for any $\sigma \in [\![I]\!]$, there exists a unique $\tau \in [\![O]\!]$ such that $\sigma * \tau \in \mathcal{M}_e^T$ and there is some $\omega \in \mathcal{M}_e$ such that $\omega^T = \sigma * \tau$. Therefore, there is a σ such that $\sigma = \vartheta \downarrow I$ and $\sigma * \tau \in \mathcal{M}_e^T$, and some $\omega \in \mathcal{M}_e$ such that $\sigma = \omega^T \downarrow I$ and $\tau = \omega^T \downarrow O$. As $\mathcal{M}_e$ and $\mathcal{M}$ are latency equivalent, $\omega^T \downarrow I = \vartheta \downarrow I$ implies that $\omega^T \downarrow O = \vartheta \downarrow O$. If

$\sigma = \vartheta \downarrow I$, $\tau = \vartheta \downarrow O$ and the set of wires is $I \cup O$, then $\vartheta = \sigma * \tau$, and finally, as $\sigma * \tau \in \mathcal{M}_e^T$, $\vartheta \in \mathcal{M}_e^T$.

$\Leftarrow$

$\mathcal{M}_e^T = \mathcal{M} \Rightarrow (\forall \sigma \in \mathcal{M}, \forall \tau \in \mathcal{M}_e \sqcup Env_{I,O},\ \tau^T \downarrow I = \sigma \downarrow I \Rightarrow$
$\tau^T \downarrow O = \sigma \downarrow O)$

As both $\mathcal{M}_e^T$ and $\mathcal{M}$ are machines, they are both deterministic. Determinism is $\forall \omega \in \mathcal{M}, \forall \omega' \in \mathcal{M}, \omega \downarrow I = \omega' \downarrow I \Rightarrow \omega \downarrow O = \omega' \downarrow O$. As it is assumed that $\mathcal{M}_e^T = \mathcal{M}$, we can substitute $\mathcal{M}_e^T$ for $\mathcal{M}$: $\forall \omega \in \mathcal{M}, \forall \omega' \in \mathcal{M}_e^T,\ \omega \downarrow I = \omega' \downarrow I \Rightarrow \omega \downarrow O = \omega' \downarrow O)$. By $\mathcal{M}_e^T$ definition, there exists one behavior of the elastic machine whose transfer behavior is ω' for any ω' in the transfer machine, that is, $\forall \omega' \in \mathcal{M}_e^T, \exists \sigma \in \mathcal{M}_e \sqcup Env_{I,O}, \sigma^T = \omega'$. Then, as all behaviors in $\mathcal{M}_e^T$ are related to at least one behavior in $\mathcal{M}_e$, the previous formula can be rewritten as $\forall \omega \in \mathcal{M}, \forall \sigma \in \mathcal{M}_e \sqcup Env_{I,O},\ \sigma^T \downarrow I = \omega \downarrow I \Rightarrow \sigma^T \downarrow O = \omega \downarrow O$, which is the definition of latency equivalence.

5.1.2 Implementation

In order to check equivalence for a given transfer input behavior σ, it is necessary to simulate the conventional circuit that Esterel generates using σ, simulate the elastic circuit using an elastic behavior ω such that $\omega^T = \sigma$, and compare the output behavior of the conventional circuit to the output transfer behavior of the elastic one.

Every input channel is connected to an elastic producer. In the initial cycle, the producer can decide whether to put a bubble or a token in the channel. In case it is a token, the data wire transmits the first data item of the transfer input behavior that is being simulated. Next cycle, the producer decides again whether to put a bubble or a token, unless there was a token and it could not be transmitted. In this case, the producer is persistent and tries to transmit the same token until it is possible to do so. Each input producer is independent from all other producers.

Similarly, each output of the elastic circuit is connected to an elastic consumer. Each cycle, every elastic consumer decides whether to set its stop bit to force the circuit to transmit the corresponding output again next cycle.

Each simulation has three parameters. The first one is the number of tokens to be transmitted: the number of cycles that the conventional circuit is simulated is this number of tokens, as there is a new token every cycle, and the elastic circuit is simulated until all output channels have received that

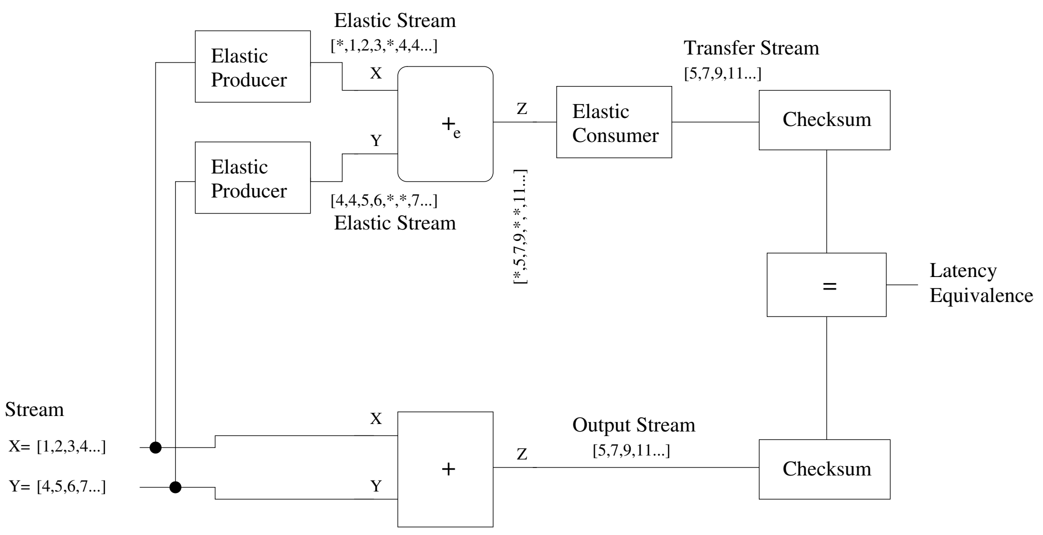


Figure 5.1: Latency equivalency checking

number of tokens. The second one is the valid probability: every time a producer has to decide whether to transmit a new token or insert a bubble, this probability is used to make this decision. Finally, the third one is the stop probability: every time a consumer has to decide whether to be ready to receive a new token, it uses this parameter to do so. The valid probability must be greater than 0, and the stop probability must be less than 1 in order to ensure termination of the simulation. Otherwise, some producer could decide to transmit no tokens, or a consumer could decide to be never ready.

Once one has simulated several times a circuit, every time with a different input and with different valid and stop probabilities, and the outputs are latency equivalent in all of them, we can assume that the elastic circuit and the conventional circuit are behaviorally equivalent.

Figure 5.1 shows how latency equivalence of a circuit with two inputs and one output is checked. Firstly, the simulation produces a random finite sequence for each input. Then, the conventional circuit is simulated using the generated input sequence, and the outputs it generates each cycle are used to create a checksum (it would be too memory expensive to store all outputs of all cycles). Then, the same input sequences are connected to the elastic producers, that send them to the elastic circuit with some inserted bubbles. The elastic consumer in the output sets the stop bit at random cycles, and the received tokens are used to compute a checksum. Finally, both checksums are compared to check latency equivalence.

5.1.3 Tests

The elasticization procedure has been executed on four different designs in order to test latency equivalence between the conventional Verilog designs created by the Esterel compiler and the elastic Verilog designs.

Each design is simulated 125 times with different valid and stop probabilities in the elastic consumers and producers, and each simulation transmits 5000 tokens. Therefore, the simulation does not end until all output channels have received 5000 tokens. Latency equivalence has been achieved in all simulations, validating the elasticization procedure presented in this work.

The Esterel code of these designs can be found in the appendix. Table 5.1 shows some features of the designs : the number of inputs and outputs, whether it deals with data (otherwise it is a pure Esterel program), and the number of elastic modules and channels in the elastic design.

The first design is a simple pipeline. There are no preemptions or other control structures, and hence, there is no need for synchronization registers.

The second design has two parallel statements of the form "**await** input; **emit** output". Being independent, there should be no synchronization between them. However, they are enclosed in a loop, and therefore, by the Esterel semantics, a loop execution cannot start until the previous one ended. The Esterel compiler creates a register called "PauseReg_20_0" that synchronizes both inputs. This register is taken into account when elasticizing, and therefore, the input channels are synchronized so that one channel cannot send two tokens if there is not a token of the other channel in between.

The third design is from the Esterel v7 primer [5] and simulates a traffic light controller in the intersection of two roads. The controller receives requests and decides which road traffic light must be green. It is a pure Esterel design, and it is the one that creates more elastic modules and channels because it is control intensive.

Finally, the forth design computes the greatest common divisor of the inputs. It is also from the Esterel v7 primer [5]. It uses both data and control structures and it iteratively substracts the smaller number to the larger one until the gcd is found.

Design	**inputs**	**outputs**	**data**	**elastic modules**	**elastic channels**
pipeline	2	1	Y	16	21
parallel emits	2	2	Y	24	64
traffic lights	3	6	N	32	237
gcd	3	1	Y	18	58

Table 5.1: Simulated designs

5.2 Granularity

Elaticization can be applied to different levels of granularity in terms of block size and in terms of number of channels.

On the one hand, one can arbitrarily choose the size of the elasticizated blocks. It is possible to consider the whole design as a block; it is possible to consider functional units as elastic modules with some fixed or variable delay (as long as all of them follow the `SELF` protocol for variable delay units) and synchronize them using elastic circuits; etcetera. The finest grain solution is the one in which every combinational logic block in an elastic module. This is the solution used in this thesis.

On the other hand, one can choose the number of input and output channels in terms of number of valid and stop pairs. If some inputs are known to be always either both valid or none of them valid, one may make them share valid and stop bits in order to improve control size and performance. This can be detected and implemented either manually or automatically.

Both approaches are design decisions that have an impact into performance and circuit size. This section uses a sample circuit to show how the performance changes when merging channels so that different inputs share the same valid and stop bits.

5.2.1 Approach

Performance evaluation is an important issue in system design. A possible way to measure the performance of an elastic channel is the transfer rate, i.e., the number of transfers per cycle. Optimally, a new token is transferred every cycle, and the transfer rate is 1. The transfer rate of a channel that is an output of an elastic module depends on the transfer rates of the module inputs and the number of cycles in which the output receivers are ready for the transmission. Furthermore, the elastic join structures that synchronize

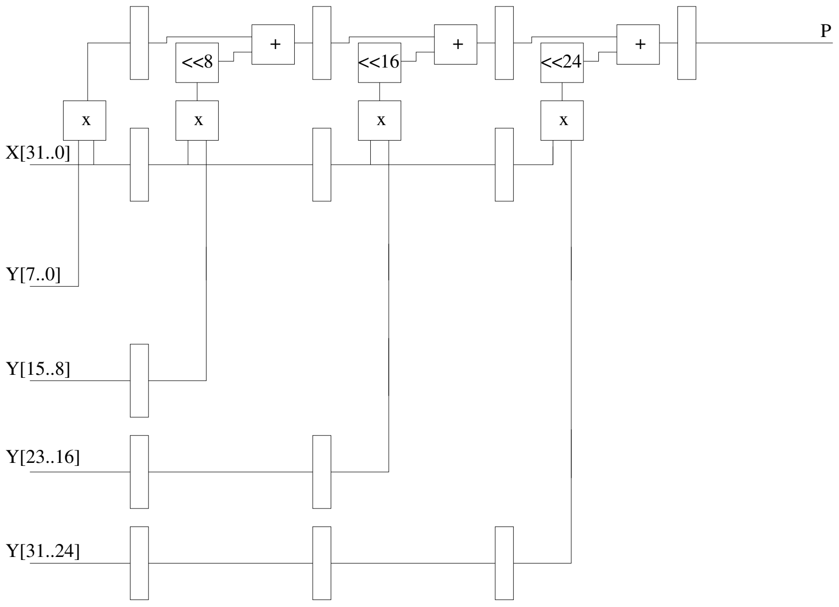


Figure 5.2: Multiplication circuit

input channels in the elastic modules will decrease the performance because of this synchronization.

In order to evaluate the impact of granularity, the circuit in Figure 5.2 is used. It multiplies two unsigned integers of 32 bits and the result is a unsigned integer of 64 bits. The second operand is divided into four numbers of 8 bits each. Thus, the multiplication is done in 4 pipelined stages. Each part of the second operand has a different pipeline depth, so that a different number of tokens can be stored in each one.

The circuit has been generated using an Esterel program, and then manually edited in order to achieve three different implementations. Every implementation merges input channels from the previous one in the `SELF` netlist, and then plugs each enable wire to the registers it must be connected. In the first one, there are five channels, one per input. In the second one, all bytes of the second operand share the same valid and stop bits. In the last one, there is one input channel for all data.

As there is just one output, the difference in performance between implementations is due to input synchronization. The more input channels, the less performance there should be.

The simulation of each design is done like validation was. Therefore, it is checked whether implementations are latency equivalent to the conventional

Esterel design and, at the same time, the simulator computes the number of tokens per cycle in the output. There are two different values that can be tuned. The first one is the rate of tokens in the inputs. That is, the probability of producing a token every time the environment can decide whether to produce a token or a bubble in some input channel. The second one is the probability of setting the stop bit of the output.

5.2.2 Results

Tables 5.2, 5.3 and 5.4 show the results of the executions. Every implementation has been tested with 25 different configurations of valid and stop rates. Every configuration has been simulated 5 times, each time transmitting 5000 tokens. Thus, the number of simulated cycles is more than 5000, depending on the performance of the circuit.

Each row in the tables shows the transfer rate (number of tokens transferred per cycle) for a given valid frequency in the inputs. For example, the first row shows the transfer rate in the output when elastic producers inserts new tokens each cycle with a probability of 20 %. Notice that each input is independent from the other ones. The fact that the probability is 20 % does not necessarily mean that there are tokens in exactly a 20 % of the cycles. However, the number of tokens should be around 20 %.

Each column in the tables shows the transfer rate for a given stop frequency of the output. For example, the first column shows the transfer rate in a simulation where the stop bit of the output is never set, the second column shows the transfer rate in a simulation where the stop bit is set in a cycle with a 20 % probability. Thus, every position of the table shows the average transfer rate for a given configuration.

When the stop rate of the output is 0, the transfer rate is very close to the valid rate of the inputs, as the bubbles of the outputs are only due to synchronization of the inputs. Therefore, in Table 5.4, where there is just one input channel, the transfer rate in the first column is nearly the valid rate. The little difference between the expected value and the real value is because the random insertion of bubbles in the inputs does not guarantee that the percentage of tokens is exactly its probability.

Figure 5.3(a) shows the transfer rate of all simulations in function of the valid rate of the inputs, and Figure 5.3(b) shows the transfer rate of all simulations in function of the stop rate of the output. Clearly, the valid rate and the transfer rate are proportional, and the stop rate and the transfer rate are inverse proportional.

	Valid Rate at Inputs					
Stop Rate at Outputs		**0.0**	**0.2**	**0.4**	**0.6**	**0.8**
	0.2	0.13	0.13	0.13	0.13	0.12
	0.4	0.27	0.27	0.27	0.26	0.20
	0.6	0.42	0.42	0.42	0.38	0.23
	0.8	0.61	0.60	0.56	0.44	0.25
	1.0	1.00	0.87	0.71	0.52	0.29

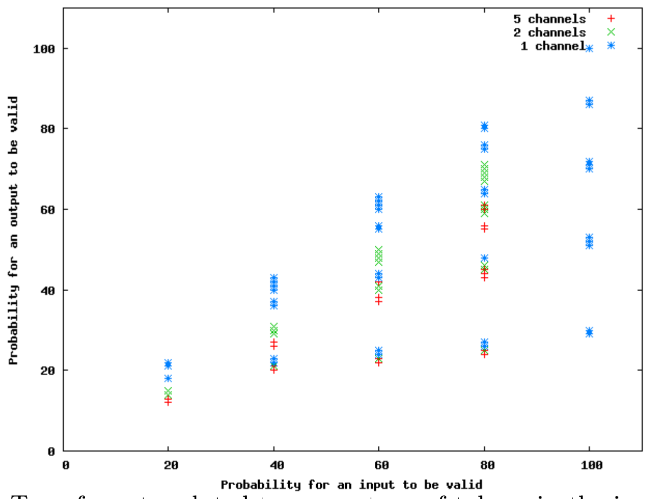
Table 5.2: Transfer rate using 5 input channels

	Valid Rate at Inputs					
Stop Rate at Outputs		**0.0**	**0.2**	**0.4**	**0.6**	**0.8**
	0.2	0.15	0.15	0.15	0.15	0.14
	0.4	0.31	0.31	0.31	0.30	0.21
	0.6	0.49	0.48	0.47	0.40	0.23
	0.8	0.70	0.67	0.60	0.46	0.25
	1.0	1.00	0.87	0.71	0.52	0.29

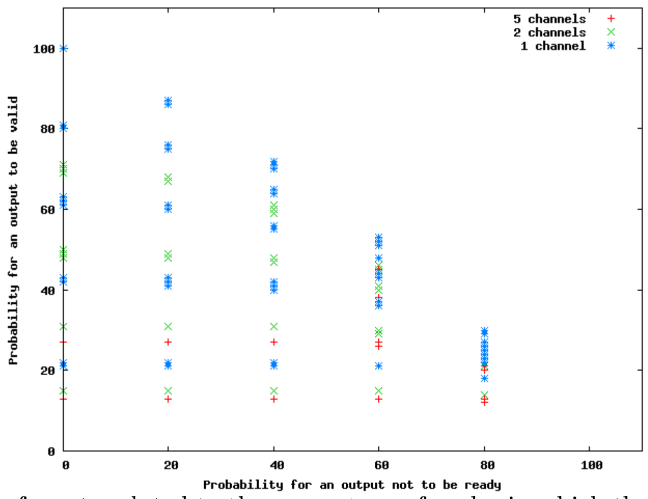
Table 5.3: Transfer rate using 2 input channels

	Valid Rate at Inputs					
Stop Rate at Outputs		**0.0**	**0.2**	**0.4**	**0.6**	**0.8**
	0.2	0.19	0.20	0.20	0.20	0.18
	0.4	0.41	0.39	0.39	0.36	0.22
	0.6	0.62	0.60	0.55	0.44	0.24
	0.8	0.81	0.75	0.64	0.48	0.27
	1.0	1.00	0.87	0.71	0.52	0.29

Table 5.4: Transfer rate using 1 input channel



(a) Transfer rate related to percentage of tokens in the inputs



(b) Transfer rate related to the percentage of cycles in which the receiver is not ready

Figure 5.3: Transfer Rate figures

5.2.3 Conclusion

Both the tables and the figures show that the less channels, the more performance. When a module has two input channels, it needs to synchronize them: if there is a token in one channel and a bubble in the other one, the channel that is trying to transmit the token must wait until the other one is ready. During this time, the outputs of such module will carry bubbles, and therefore the transfer rate of the outputs will always be less than the transfer rate of the inputs.

However, when there is either a lot of congestion or very few congestion, the difference between implementations is not that big. When the output channel has the stop bit set most of the cycles, channels are waiting to transmit a token most of the time, and there are few cycles in which it is necessary to insert bubbles in the output to synchronize the inputs.

Similarly, when there are tokens most of the cycles, all inputs are always ready to transmit, and the elastic modules do not have to stop channels in order to synchronize them. Therefore, the difference between implementations is mostly seen when there is average congestion.

As a conclusion, whenever it is possible to merge channels by allowing them to share the valid and stop bits, it should be done to increase the performance. This is just a test case over a single circuit. Further research can be done in order to confirm this hypothesis.

Chapter 6

Conclusions

6.1 Results

The validation of latency equivalence shows that generated elastic designs preserve the functionality of the original Esterel code. The automatic elasticization of Esterel designs allows to create a latency-insensitive design from a high level language with no extra effort. Thus, the amount of time needed to create a synchronous circuit and a synchronous elastic circuit which is functionally equivalent is the same.

Furthermore, `SELF` adds little overhead, as changes in the datapath are slight and it is only necessary to add a control layer. The elasticization creates synthesizable circuits as long as the original Esterel design is synthesizable. These facts make elastic circuits an attractive option for microarchitectural designs.

As a result of this work, future versions of the Esterel v7 compiler developed by Esterel Technologies will be able to generate elastic circuits out of Esterel programs. The author has embedded into the Esterel compiler source code this functionality.

6.2 Future work

The elasticization procedure is highly embedded into the Esterel compilation procedure. A possible direction for future work can be making elasticization independent of the hardware description language that is used.

Furthermore, this master's thesis validates latency equivalence of the elasticized circuits, but does not check that the procedure works using formal methods such as verification. This can also be done as future work.

Finally, further research can be done in correct-by-construction microarchitectural transformations. The main idea here is to apply changes in elastic designs, such as retiming, merging elastic channels, and so on and so forth in order to improve the performance of the circuit.

Bibliography

[1] Benveniste, A., and Berry, G. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE 79* (1991), 1270–1282.

[2] Berry, G. Esterel on hardware. *Mechanized reasoning and hardware design* (1992), 87–104.

[3] Berry, G. Preemption and concurrency. *Proc. FSTTCS. Lecture Notes in Computer Science 761, Springer Verlag 93* (1993), 72–93.

[4] Berry, G. The Constructive Semantics of Pure Esterel. *Draft book, available at http://www.esterel-technologies.com/technology/scientific-papers/* (1999).

[5] Berry, G. The Esterel Primer. *Available at http://www.esterel-technologies.com/technology/scientific-papers/* (2000).

[6] Berry, G. The Foundations of Esterel. *Proof, language, and interaction: essays in honour of Robin Milner. MIT Press* (2000), 425–454.

[7] Berry, G., and Gonthier, G. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming 19*, 2 (1992), 87–152.

[8] Bhasker, J. *A Verilog HDL Primer.* Star Galaxy, 1997.

[9] Brayton, R., Hachtel, G., and Sangiovanni-Vincentelli, A. Multilevel logic synthesis. *Proceedings of the IEEE 78*, 2 (1990), 264–300.

[10] Brzozowski, J., and Seger, C. *Asynchronous Circuits.* Springer-Verlag, 1995.

[11] Carloni, L., McMillan, K., and Sangiovanni-Vincentelli, A. Theory of latency-insensitive design. *IEEE Transactions on CAD 20*, 9 (Sept 2001).

[12] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. LUSTRE: A declarative language for programming synchronous systems. *14th Symposium on Principles of Programming Languages (POPL 87), Munich* (1987).

[13] Cortadella, J., Kishinevsky, M., and Grundmann, B. Synthesis of synchronous elastic architectures. In *Proc. ACM/IEEE Design Automation Conference* (July 2006), pp. 657–662.

[14] Gautier, T., Le Guernic, P., and Besnard, L. *SIGNAL: A declarative language for synchronous programming of real-time systems.* Springer-Verlag London, UK, 1987.

[15] Girard, J. Y., Lafont, Y., and Taylor, P. *Proofs and types.* Cambridge University Press New York, 1989.

[16] Gonthier, G. Semantique et modeles d'execution des langages reactifs synchrones; applicationa Esterel. *These d'informatique, Universite d'Orsay* (1988).

[17] Halbwachs, N. *Synchronous programming of reactive systems.* Springer, 1998.

[18] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE 79*, 9 (1991), 1305–1320.

[19] Harel, D., et al. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (1987), 231–274.

[20] Harel, D., and Pnueli, A. On the development of reactive systems. *Logics and models of concurrent systems* (1985), 477–498.

[21] Harper, R., Milner, R., and Tofte, M. *The Definition of Standard ML.* MIT Press, 1991.

[22] Krstić, S., Cortadella, J., Kishinevsky, M., and O'Leary, J. Synchronous elastic networks. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)* (Nov. 2006).

[23] Le Guernic, P., Benveniste, A., Bournai, P., and Gautier, T. Signal – A data flow-oriented language for signal processing. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on 34*, 2 (1986), 362–374.

[24] Le Guernic, P., Talpin, J., and Le Lann, J. Polychrony for system design. *Journal for Circuits, Systems and Computers 12*, 3 (2003), 261–304.

[25] LeGuernic, P., Gautier, T., Le Borgne, M., and Le Maire, C. Programming real-time applications with SIGNAL. *Proceedings of the IEEE 79*, 9 (1991), 1321–1336.

[26] Malik, S. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 7 (1994), 950–956.

[27] Maraninchi, F. *Argos: un langage graphique pour la conception, la description et la validation des systemes reactifs.* PhD thesis, Universite Joseph Fourier, Grenoble I, Janvier 1990.

[28] Maraninchi, F. Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra. *Proceedings of the international workshop on Automatic verification methods for finite state systems* (1990), 38–53.

[29] Micheli, G. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[30] Plotkin, G. A structural approach to operational semantics. Report DAIMI FN-19. *Computer Science Department, Aarhus University* (1981).

[31] Pnueli, A. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science* (1977), pp. 46–57.

[32] Potop-Butucaru, D., Edwards, S., and Berry, G. *Compiling ESTEREL.* Springer, 2007.

[33] Sentovich, E., Singh, K., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R., and Sangiovanni-Vincentelli, A. SIS: A system for sequential circuit synthesis. Tech. rep., University of California, Berkeley, 1992.

[34] Sentovich, E., Toma, H., and Berry, G. Latch optimization in circuits generated from high-level descriptions. *Proc. International Conf. on Computer-Aided Design (ICCAD)* (1996).

[35] Sentovich, E., Toma, H., and Berry, G. Efficient latch optimization using exclusive sets. *Proceedings of the 34th annual conference on Design automation conference* (1997), 8–11.

[36] Shiple, T., Berry, G., and Touati, H. Constructive analysis of cyclic circuits. *Proc. International Design and Test Conference ITDC* (1996).

[37] Shiple, T., Singhal, V., Berry, G., Brayton, R., and Sangiovanni-Vincentelli, A. Analysis of combinational cycles. Tech. rep., Technical Report UCB/ERL M96, 1996.

[38] Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press Cambridge, MA, USA, 1977.

[39] Suhaib, S., Berner, D., Mathaikutty, D., Talpin, J., and Shukla, S. Presentation and formal verification of a family of protocols for latency insensitive design. Tech. rep., Technical report, Virginia Tech, 2005.

[40] Tardieu, O., and De Simone, R. Loops in Esterel. *Transactions on Embedded Computing Systems 4*, 4 (2005), 708–750.

[41] Thomas, D., and Moorby, P. *The Verilog (r) Hardware Description Language.* Kluwer Academic Publishers, 2002.

APPENDIX

.1 Simple Pipeline

```
main module my_strl_pipe:
constant n : unsigned <> = 3;
input In1 : temp unsigned <[n]>;
input In2 : temp unsigned <[n]>;
output Out : value reg unsigned <[n+3]> init 0;
signal Reg1 : reg unsigned <[n]> init 0,
         Reg2 : reg unsigned <[n]> init 0,
         F1 : reg unsigned <[n+1]> init 0,
         F2 : value reg unsigned <[n+3]> init 0
in
      sustain {
         next ?Reg1 <= ?In1 if In1,
         next ?Reg1 <= 0 if not In1 and In2,
         next ?Reg2 <= ?In2 if In2,
         next ?Reg2 <= 0 if not In2 and In1,
         next ?F1 <= ?Reg1 + ?Reg2,
         next ?F2 <= ?F1 * 3,
         next ?Out <= ?F2
      }
end signal
end module
```

.2 Parallel Emit

```
main module parallel_emit:
constant n : unsigned <> = 3;
input In1 : unsigned <[n]> init 0;
input In2 : unsigned <[n]> init 0;
output Out1 : unsigned <[n]> init 0;
output Out2 : unsigned <[n]> init 0;
loop
 {
     await immediate In1; emit ?Out1 <= ?In1;
  ||
     await immediate In2; emit ?Out2 <= ?In2;
 };
 pause
end loop
end module
```

.3 Traffic Lights

```
module toggle:
input i;
output o;
sustain o <= i xor pre(o)
end module

main module traffic_lights:
input second;
input north_road_req, west_road_req;
output north_red, north_yellow, north_green;
output west_red, west_yellow, west_green;
signal change_open_road, north_road_open in
    run toggle [change_open_road/i, north_road_open/o]
  ||
    loop
        abort
            await 30 second
          ||
            if north_road_open then
               await west_road_req
            else
                await north_road_req
            end if
        when 60 second;
        emit change_open_road
    end loop
  ||
    loop
        abort
            if north_road_open then
                sustain {north_red, west_yellow}
            else
                sustain {west_red, north_yellow}
            end if
        when 2 second;
        abort
```

```
               sustain {west_red, north_red}
            when 1 second;
            if north_road_open then
               sustain {north_green, west_red}
            else
               sustain {west_green, north_red}
            end if
        each change_open_road
    end signal
    end module
```

.4 Greatest Common Divisor

```
main module gcd:
input {a,b} : value unsigned;
input restart;
output d : unsigned;
loop
  signal r0 : value reg unsigned init ?a,
         r1 : value reg unsigned init ?b
  in
   weak abort
    always
        var {l,s} : temp unsigned in //larger, smaller
           if ?r0 >= ?r1 then
              l := ?r0;
              s := ?r1;
           else
              l := ?r1;
              s := ?r0;
           end if;
           emit {
              next ?r0 <= l-s,
              next ?r1 <= s,
              ?d <= next(?r1) if next(?r0) = 0
           }
        end var
    end always
   when d
  end signal
each restart
end module
```