The Kiel Esterel Processor: A Multi-Threaded Reactive Processor

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr. -Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Xin Li

Kiel
2007

1. Gutachter	Reinhard von Hanxleden
2. Gutachter	Michael Mendler
Datum der mündlichen Prüfung	23. Juli 2007

Acknowledgements

This thesis would not have been possible without the support of many people. My deepest gratitude is to my supervisor, Professor Reinhard von Hanxleden, for his precious suggestions, friendly advice, and giving me the freedom to pursue a research topic. Most of all, for his patience with my mistakes and extreme generosity with his time. His scientific vision, high standards, detailed comments and discussions, and solid insight into embedded system design have given me guidance to follow throughout my research efforts. I am also grateful to him for having provided me chances to attend many conferences, which have expanded my horizon and inspired me to conduct this research work.

Furthermore, my special thanks go to Prof. Michael Mendler at the Universität Bamberg, Prof. Stephen A. Edwards at Columbia University, Dr. Stavros Tripakis and Dr. Claudio Pinello at Cadence Berkeley Labs, and Prof. Alberto Sangiovanni-Vincentelli at the University of California Berkeley for their inspiring talks, suggestions and advice which were very helpful for my work.

Claus Traulsen and Malte Tiedje read a draft version of this thesis and gave me lots of helpful comments, I really appreciate their help. I thank also Marian Andreas Boldt for his discussions and collaboration. In addition to this, I am grateful for all the discussions with Steffen Prochnow, Jan Lukoschus, and Sascha Gädtke. I further thank my colleagues Gesa Walsdorf, Isabella Cembrowski, Maren Lutz, Hauke Fuhrmann, and Tim Grebien for their kind help during my study in the Real-time and Embedded Systems group.

I acknowledge the DAAD and DFG for giving me the financial support for this work.

Last, but not least, I would never have finished this work without the moral support and encouragement of my parents (Zhenhua Li and Sujiao Wu), my wife (Di Zang) and my daughter (Luopian Li) to whom I dedicated this dissertation.

Abstract

Many embedded systems belong to the class of *reactive systems*, which continuously react to inputs from the environment by generating corresponding outputs. The programming of reactive systems typically requires the use of non-standard control flow constructs, such as concurrency or exception handling. Most programming languages do not support these constructs at all, or their use induces non-deterministic program behavior. To address these difficulties, the synchronous language Esterel has been developed to express reactive control flow patterns in a concise manner, with a clear semantics that imposes deterministic program behavior under all circumstances.

There are different options to synthesize an Esterel program into a concrete system, *e. g.*, software, hardware, and HW/SW co-design implementations. However, these classical synthesis approaches suffer from the limitations of traditional processors, with their instruction set architectures geared towards the sequential von-Neumann execution model, or they are very inflexible if HW synthesis is involved.

Recently, another alternative for synthesizing Esterel has emerged, the *reactive processing* approach. Here the Esterel program is running on a processor that has been developed specifically for reactive systems. However, the main challenge when designing a reactive architecture is the handling of control.

This thesis presents the Kiel Esterel Processor (KEP). In the KEP, the multi-threaded reactive architecture is responsible for managing the control flow of all threads. The KEP Instruction Set Architecture is *complete* in that it allows a direct mapping of all Esterel statements onto KEP assembler. It supports Esterel's concurrency operator || in a very precise, direct and efficient way. It also supports full Esterel preemptions, *i. e.*, the delayed and immediate strong/weak abortion and suspension. All other Esterel kernel statements, *e. g.*, the Esterel exception, delay, and signal emission, etc., are also implemented directly and semantically accurate by the KEP.

As the experimental comparison with a 32-bit commercial RISC processor indicates, the KEP has advantages in terms of memory use, execution speed, and energy consumption. Another advantage is the predictability of its timing behavior at the program level.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

1.1 Introduction

In the past decades, computer systems have rapidly surrounded us. Nowaday, if we take a close look at applications, we will be surprised at how many computer systems can be found in our daily life, *e. g.*, mp3 players, the computer controlled microwave ovens, washing machines, engine controllers and ABS for automobiles. Unlike a general-purpose computer, such as a computer on our desktop, these *embedded systems* are special-purpose systems in which computers are completely encapsulated by the devices they control. In general, an embedded system is pre-defined for very specific requirements.

Applications can be divided into one of three categories [18], *i. e.*, the *transformational systems*, which compute output values from inputs values and then stop, *e. g.*, batch processing, simulations, compilers; the *interactive systems*, which constantly interact with their environment in such a way that the computers can be viewed as the masters of the interaction, *e. g.*, databases, operating systems; or the *reactive systems*, which continuously react to stimuli coming from their environment by sending back to stimuli, *e. g.*, engine controllers, traffic control, microcontrollers, etc. In general, most of embedded systems are reactive systems.

Reactive systems are purely input-driven and they must react at a pace that is dictated by the environment. Any automatic control system can be classified as a reactive system, *e. g.*, nuclear plant controllers, airplane flight systems, etc. The essential characteristics of reactive systems can be summarized as following [49]:

Criticality They are highly critical, just like the systems they control are critical. Failure of these systems could cause catastrophic consequences for human life.

Parallelism At least the parallelism between the system and its environment must be taken into account during the specification. Moreover, it is very often convenient for the designer to conceive the system as a set of parallel components, cooperating in order to achieve the desired behavior.

Determinism A reactive system determines a sequence of output signals from a sequence of input signals in a unique way. This determinism makes their design, analysis, and debugging much easier. Thus it must be preserved by the implementation.

For programming a reactive system, a traditional programming language, *e. g.*, C or Java, can be employed. However, this has several shortcomings. First, classical asynchronous languages lack high-level parallel programming primitives, and asynchronous parallelism can cause an unwanted non-determinism. The correctness of a model that is directly described by common program language would be hard to certify, which is an essential requirement of a safety-critical system. Second, the classical programming languages lack statements for modelling reactive control structures. Furthermore, those languages are relatively low level, hence, the developers have to not only focus on specifying the functions of a module, but also have to implement those functions.

In the 1980s, the synchronous languages were introduced, which can cope with the above mentioned drawbacks of traditional programming language. Although several languages, *e. g.*, Esterel [23], Lustre [65], and Signal [63], were presented for different purposes, they still have some common features [11, 10]. First, *concurrency*—all of these languages support functional concurrency and include that express concurrency in a user-friendly manner [55]. Second, *simplicity*—those languages have a simple formal model to make formal reasoning tractable, especially for clearly describing formal model as simple as possible [77]. Finally, *synchronicity*—they are based on the synchrony hypothesis, which states in essence that a system responds in zero time to environmental requests [23]. Gradually, the synchronous languages have attracted some leading companies which develop automatic control software for critical applications, such as Schneider, Dassault, Aerospatiale, Snecma, Cadence, Texas Instruments, and Thomson [10, 49]. The key advantage of synchronous languages is that the synchronous approach has a rigorous mathematical semantics which allows the programmers to develop critical software faster and better.

Our work focuses on the Esterel language, which currently appears to be the best-known synchronous language in industry and academia [10]. In the industry area, Esterel is used in applications such as developing DSP chips for mobile phones [30, 3], designing and verify DVD chips, and programming the flight control software of Rafale fighters [22, 49]. Although its strength has been proven, it still contains some weaknesses, for example, the comparatively inefficient implementation of Esterel modules in general [102].

1.2 Motivation

Esterel is an imperative language dedicated to the programming of reactive applications [21, 19]. In contrast to many high-level languages, Esterel is both parallel and deterministic. The language is based on a formal mathematical semantics.

In Esterel, *signal*s are used to communicate internally and with the environment. The execution of an Esterel program is divided into logical *instants*, or *ticks*. The synchrony hypothesis of Esterel implies that the outputs generated from given inputs occur at the same logical instant [23]. Signals are *present* or *absent* throughout an instant, indicating the occurrence of certain events, and they may also carry a value.

The Esterel *parallelism* is expressed by the concurrency operator ("||"). It groups statements in parallel, which also can be regarded as threads. When several threads are active concurrently, they may communicate back and forth instantaneously, that is, within the same logical tick. The communication between concurrent modules depends on local signals.

The Esterel *preemption* includes weak and strong abortion, suspension [18], and exception handling. The strong abortion kills its body immediately; the weak abortion first terminates its current reaction. On the contrary, the suspension can temporarily halt its body when the trigger occurs. The exception handling mechanism defines an exit point for a trap body. If the body exits the trap, the trap statement immediately terminates and weakly aborts the trap body.

Neither traditional processors nor classical programming languages have similar structures or statements (instructions) to handle corresponding Esterel statements efficiently. Hence, the implementation of the Esterel semantics on commercial off-the-shelf (COTS) processors is problematic since it must be simulated. Therefore, an Esterel-based design proves its efficiency on model description and validation, but can hardly enhance the implementation performance or reduce resource usage. Although researchers have studied different compiling techniques for synthesizing Esterel programs to efficient intermediate languages, the final execution code is still fairly large and has long execution times.

Since traditional processors have difficulties to handle Esterel programs efficiently, it is natural to raise the question whether a special processor can be developed for handling Esterel structure directly? In other words, whether an Application Specific Instruction-set Processor (ASIP) can be used for targeting Esterel programs? Furthermore, what benefits can result from this alternative strategy, especially compared with previous Esterel implementations?

1.3 Research Contribution

The focus of this dissertation is the development of a reactive architecture. The project was driven by the desire to achieve predictable, competitive execution speeds at minimal resource usage, in terms of processor size and power usage as well as instruction and data memory. This work has resulted in the *Kiel Esterel Processor* (*KEP*). It is a custom multi-threaded reactive processor, to our knowledge it is the first of this kind.

This thesis presents the architecture of the KEP. Notable features of the KEP include the following:

1. The KEP is the first reactive processor which employs a multi-threaded architecture for directly handling concurrency. This strategy uses resources efficiently and easily scales up to very high degrees of concurrency.

2. The KEP contains a full-custom *reactive core*, whose instruction set and data path have been tailored exclusively for the processing of Esterel code. Hence, all types of Esterel preemptions, delays, and exceptions, can be handled by KEP very efficiently.

3. The KEP also includes an interface block for handling Esterel input, output and local types of pure and valued signals. Furthermore, testing the presence and values of signals across logical instants (corresponding to Esterel's pre operator) are also directly supported.

4. Throughout the development of the KEP, scalability has been considered, hence the allowed number of signals, the maximum thread number, the nesting depth of preemption primitives, and other design parameters are fully configurable.

5. Unlike other reactive processing approaches, the KEP Instruction Set Architecture (ISA) is *complete* in that it allows a direct mapping of all Esterel statements onto KEP assembler. All the Esterel kernel statements, including delay, preemption, concurrency and exception handling, are implemented directly and semantically accurately by the KEP, and they can be freely combined and nested as defined by the Esterel semantics. However, it can also make unrefined processing approaches fairly costly. The KEP ISA therefore not only supports common Esterel statements directly, but also takes into consideration the statement context. Providing such a *refined* ISA further minimizes hardware usage while preserving the generality of the language.

Advantages of the KEP compared with traditional processors include:

Performance As the instruction set and data path have been developed specifically for Esterel execution, the Esterel module can be executed fairly fast on KEP. This benefits two key aspects of system performance, *i. e.*, the *Worst Case Reactive Time* (WCRT) and *Average Case Reactive Time* (ACRT).

Memory Because most typical Esterel statements can be expressed directly with just a single KEP instruction, an Esterel program executed on the KEP has very low instruction and data memory usage.

Power Usage For controller programming, the main goal of Esterel, the control signals tend to be more often absent than present [19]. Due to the architecture of the KEP, very few instruction cycles are needed for executing a *blank event*, which corresponds to the condition of all signals being absent.

Logic Area The KEP offers a novel light-weight thread model, *i. e.*, the multi-threaded architecture, to implement Esterel concurrency efficiently. This characteristic significantly reduces its logic resource usage for implementing a practical (industry scale) Esterel module.

Predictability The KEP is not designed to optimize (average) performance for general purpose computations, and hence does not have a hierarchy of caches, pipelines, branch predictors, etc. This leads to a simpler design and execution behavior and further implies that control-flow is preserved while compiling Esterel into machine code, and that the execution platform has a very predictable timing behavior.

In summary, the KEP is an efficient reactive processor for handing practical Esterel modules, and appears to be very competitive with other implementations.

This dissertation focuses on the KEP hardware, *i. e.*, its architecture and execution model. However, since starting the KEP project, a set of closely related activities have started, in particular considering its compiler [85, 26], timing analysis [86, 28, 27], and HW/SW co-design [58, 57]. This dissertation will summarize these activities as is appropriate to help the understanding of the KEP itself.

1.4 Thesis Organization

This dissertation is composed of five main sections. The next chapter provides an overview of the Esterel language and existing Esterel implementation methods. It also reviews some previous reactive processor approaches. In Chapter 3, we give an overview of the KEP instruction set architecture (ISA) design. Chapter 4 provides the detailed descriptions of the core contributions of this research work, *i. e.*, the multi-threaded reactive processor architecture model. It is followed by a presentation of the KEP evaluation platform, which includes the KEP compiler, the evaluation hardware platform and the evaluation software. Experimental results are presented in Chapter 5, and are also compared with results of its competitors and other implementations. We finally conclude with a summary of this work, and propose considerations for future work in Chapter 6.

Chapter 2

Background and Related Work

Most reactive applications can be divided to the data handling part and control handling [19] part. Unlike the data handling, which continuously produces output values from input values, the control handling produces discrete output signals from input ones. For example, transportation systems, robots, communication protocols, peripheral drivers, and human-machine interface fall into this category. The Esterel language was developed to design control-dominated reactive programs as an imperative concurrent language.

There exist two major versions of Esterel language, *i. e.*, Esterel V5 [19], which most academical tools use; and the newer Esterel V7 [116], enhances some complex control description statements and new powerful hardware datapath expressions. It is currently evolving into an IEEE standard. The KEP targets Esterel V5 language.

In this thesis, we generally refer to V5 unless indicated otherwise.

2.1 Implementation Technologies

To build a real system for an Esterel specification, several implementation methods have been introduced, which can be distinguished by what they generate:

- **Hardware Synthesis**
  Hardware implementations [14, 117, 47, 111, 110, 32], where an Esterel program is synthesized into a hardware circuit presentation (e. g. VHDL or Verilog HDL), lead to small footprints (low memory requirements) and cheap implementations. However, hardware implementations are not flexible, meaning that even a tiny modification of the program will require a re-synthesis. Furthermore, for an industry scale Esterel module, which may include some data path handling, its resourc usage may increase rapidly. In short, it can be used for implementing pure Esterel modules, *e. g.*, a RAM controller, but is not suitable for realizing common large scale Esterel applications [38].

- **Software Synthesis**
  In a software implementation [23, 44, 46, 39, 50], an Esterel program is first synthesized into sequential, lower level language codes (*e. g.*, C or JAVA), and then compiled to codes which can be executed at a target COTS processor. See also Section 2.2. In contrast to the hardware synthesis, it is a very flexible solution, and has low costs for the data path and arithmetic operations. However, classical processor architectures cannot handle reactive control constructs, such as abortions, directly, and cannot concurrently observe multiple signals. Therefore, handling these control constructs correctly, including priority resolution, turns out to be fairly expensive on classical software implementations. Moreover, the footprint (memory requirement) can be too large for low-cost microcontrollers.

- **Hardware/Software Co-design**
  A co-design implementation partitions a model into hardware and software components [7]. This implementation suits small control-dominated embedded systems, and tries to achieve a good balance of flexibility, performance and cost. It is composed of a few Application Specific Integrated Circuits (ASIC) combined with software procedures on general-purpose processors [37, 6]. The SW/HW interface is synthesized for internal communication [107, 93]. As a result, it combines the advantages of hardware and software implementation methods, but also inherits some shortages [24, 79]. The co-design approach has been explored for example by the POLIS project [6, 99].

- **Reactive Processors**
  The reactive processor implementation aims to combine the advantages of custom hardware and traditional software. See also Section 2.3. It implements an Esterel program on a reactive processor whose instruction set has been tailored to Esterel. In other words, it can be viewed as the ASIP (Application Specific Instruction Processor) implementation [31, 75]. However, depending on whether a traditional processor core is used as a part of reactive processor, this approach can be further distinguished to two variants:

  Patched Reactive Processor implementations combine a COTS processor core with an external hardware block, which implements additional Esterel-style instructions.

  Custom Reactive Processor implementations consist of a full-custom reactive core, whose instruction set and data path have been tailored exclusively for the processing of Esterel code.

Driven by the limitations of traditional processors, the *reactive processing* approach tries to achieve a more efficient execution of reactive programs by providing an ISA that is a better match for reactive programming. The architectures proposed so far specifically support Esterel programming; however, they should be an attractive alternative to traditional processor architectures for reactive programming in general.

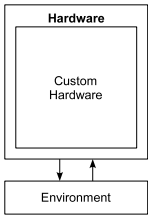
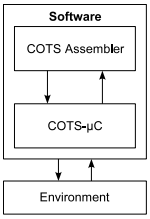
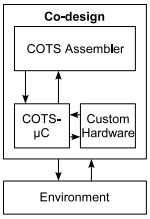
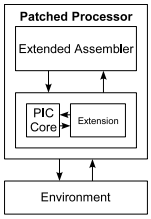
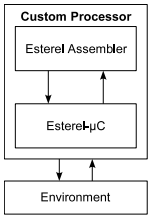
		Hardware	Software	Co-design	Patched Processor	Custom Processor
Architecture						
Speed		++	−	+	+	+
Selected References		Berry [14], Edwards [47]	Berry *et al.* [23], Edwards [44]	Baleani *et al.* [7]	Roop *et al.* [105]	Li *et al.* [90]
Flexibility		− −	++	−	+/−	+
Esterel Compliance		++	++	+/−	−	++
Cost	Logic Area	++/−	+	+	− −	+/−
	Memory	++	− −	−	+	+
	Power Usage	++	−	−	− −	+
Appl. Design Cycle		− −	++	+/−	++	++

Table 2.1: Comparison of implementation alternatives.
++ represents best; − − means worst, *e. g.*, Cost ++ means very low production costs.

Of course, each implementation has its advantages and drawbacks. Table 2.1 provides a high-level comparison of these implementation alternatives.

2.2 Compilation Approaches

In general, an Esterel program is first validated via a simulation-based tool set, and then compiled to an intermediate language, *e. g.*, C or VHDL. Different technologies are used to compile the Esterel language. In the past, various techniques have been developed to synthesize Esterel into software; see [48, 10] for an overview, which also places Esterel code synthesis into the general context of compiling concurrent languages. The KEP compiler belongs to the family of simulation-based approaches, which try to emulate the control logic of the original Esterel program directly, and generally achieve compact and yet fairly efficient code. These approaches first translate an Esterel program into some specific graph formalism that represents computations and dependencies, and then generate code that schedules computations accordingly.

A nice historical overview of the original Esterel compilers can be found on the web [54]. The Esterel V1 and V2 compilers built automata for Esterel programs. Later the V3

compiler accelerated the automata-building process by simulating the intermediate code (IC) format – a concurrent control-flow graph hanging from a reconstruction tree handles Esterel's concurrency and preemption statements. However, the problem is the state explosion. As a result, although these compilers can produce very fast code, they can hardly be scaled to an industry program size [10].

To address this shortcoming, one can employ circuits to represent the intermediate code, since circuits are roughly linear while automata are exponential. The later Esterel compilers V4 and V5 adopt this idea. For example, the Esterel compiler V5, which is one of the most used Esterel compilers, translates the IC into a combinational logic network in a very direct way, and then uses a simple topological-sort-based scheduling technique to translate the logic network into sequential code [48].

However, the logic netlist representation of the V5 compiler still has some shortcomings. For example, in the circuit, every part is assumed to be always active. However, the software code which is generated via the circuit approach wastes time on evaluating idle portions of the program – a simulation of evaluating each gate in the network in every clock cycle. Obviously, the compilation technology of V5 could result in slow code [10].

The EC/Synopsys compiler first constructs a *concurrent control flow graph* (CCFG), which it then sequentializes [46]. Threads are statically interleaved according to signal dependencies, with the potential drawback of superfluous context switches; furthermore, code sections may be duplicated if they are reachable from different control points ("surface"/"depth" replication [17]).

The SAXO-RT compiler [39] divides the Esterel program into basic blocks, which schedule each other within the current and subsequent logical tick. An advantage relative to the Synopsis compiler is that it does not perform unnecessary context switches and largely avoids code duplications. However, the scheduler it employs has an overhead proportional to the total number of basic blocks present in the program. The grc2c compiler [100] is based on the *graph code* (GRC) format, which preserves the state-structure of the given program and uses static analysis techniques to determine redundancies in the activation patterns. A variant of the GRC has also been used in the *Columbia Esterel Compiler* (CEC) [50, 51], which again follows SAXO-RT's approach of dividing the Esterel program into atomically executed basic blocks. However, their scheduler does not traverse a score board that keeps track of all basic blocks, but instead uses a compact encoding based on linked lists, which has an overhead proportional to just the number of blocks actually executed.

On the other hand, in the POLIS project, an Esterel program will be translated to the Co-design Finite State Machines (CFSMs). The CFSM sub-network can be directly mapped into the abstract hardware description format BLIF. For the software, a Control-Data Flow Graph (CDFG) called S-GRAPH specifies the transition function of a single CFSM [6].

In summary, there is currently not a single Esterel compiler that produces the best code on all benchmarks, and there is certainly still room for improvements. For exam-

```
% Esterel
module REINC:
input S;
output O1,O2;
loop
   signal S in
      present S then
         emit O1;
      else
         emit O2;
      end;
      pause;
      emit S
   end signal
end loop
```

(a)

```
% Esterel
module REINC:
input S;
output O1,O2;
loop
   signal S in
      present S then
         emit O1;
      else
         emit O2;
      end;
      pause;
      emit S;
   end signal;
   % replicate
   signal S in
      present S then
         emit O1;
      else
         emit O2;
      end;
      pause;
      emit S;
   end signal;
end loop
```

(b)

Figure 2.1: The schizophrenia problem (a) and the typical solution (b).

ple, the simulation-based approaches presented so far restrict themselves to interleaved single-pass thread execution, which in the case of repeated computations ("schizophrenia" [17]) requires code replications; it should be possible to avoid this with a more flexible scheduling mechanism.

The schizophrenia problems are not only a problem of Esterel, but occur in all synchronous languages that provide local declarations as micro steps [106]. Schizophrenic programs are those compound statements which can be run and exited or terminated and reentered inside the same reaction [114]. The problem is caused because the local variables and signals involved may have two distinct occurrences in the same reaction with different values or statuses.

For example, consider the situation of a local declaration nested inside of a loop body, which is shown in Figure 2.1(a). The signal S is defined inside the loop. Each iteration refers to a fresh signal S. In a given instant two instances of S cohabit: the first being emitted at the end of the loop, and the second being tested as the loop is reentered. The problem is that these micro steps all belong to the same macro step and therefore all data values coexist at the same point of time.

The traditional solution to this problem is unrolling of loops and renaming local declarations in the different loop bodies, as shown in Figure 2.1(b). Obviously, this method will generate unnecessarily large code. Some other solutions are also discussed by Tardieu *et al.* [115] and by Schneider *et al.* [106]. In short, they all lose efficiency as the cost of curing the schizophrenia problem. However, this schizophrenia program will not bring any problem in the KEP implementation because the micro-step of the KEP execution will exactly follow the original definition of the Esterel program, see also Section 3.2.4.

2.3 Handling Esterel via Reactive Processors

Up to now, there are only limited and fairly recent investigations for reactive processors. Since all the introduced reactive processors are handling Esterel programs, they can also be called *Esterel Processor*s. An introduction of different reactive processor implementations is presented in [119].

The first Esterel processor, called REFLIX [105], was presented by Salcic, Roop *et al.* in 2002. In this approach, a traditional soft microcontroller core (FLIX) is combined with a custom hardware block that extends the instruction set of the traditional microcontroller by certain new, Esterel-like instructions. Although its supported Esterel-style statements (instructions) were very limited, it performed better than its competitors, *i. e.*, the FLIX and other microcontrollers. Another Esterel processor, which is similar to the REFLIX, was also presented by Chow *et al.* in 2004 [38]. In this work, the FLIX processor is replaced by the PIC processor [95], which is more popular in the industry control domain. However, both of them have to depart from the original Esterel semantics to adhere the control path of the traditional processor. For example, considering nested traps, the control path there delicately depends on address ranges and parallel relations of the traps—there is no corresponding control path for handling this situation directly in the traditional processors. This is further explained in Section 4.2.2. In 2005, Z. Salcic *et al.* presented the REMIC as a custom processor [104]. Since the essence of the REMIC is similar to its ancestors, it still inherits some of their limitations, *e. g.*, the weak Esterel semantic compatibility. Both the RePIC and the REMIC were extended to multi-processing architectures, *e. g.*, the EMPEROR [41], to handle Esterel concurrency [103, 104].

In 2004, we have presented the first prototype of the KEP [89]. The architecture described in that paper is now referred to as the "KEP1". It represented to our knowledge the first custom-designed reactive processor, and the first reactive processor that correctly handled weak and strong abortion. However, it did not provide full concurrency, and logic and arithmetic expression were also not supported.

In the following year, the KEP2 improved over the KEP1 in that it includes an interface block that supports the PRE-operator, and can handle further Esterel-constructs such as variables and local signals [86, 88]. Furthermore, it contains an ALU and supports some classical logic and arithmetic expression. The KEP2 also includes a Tick Manager,

which can provide a constant logical tick length and detects timing overruns. We have also presented [86] an approach to analyze the Worst Case Reaction Time (WCRT) of the KEP2.

An important improvement of the next generation of the KEP is the implementation of concurrency [90] in 2006. The KEP3 was the first truly concurrent KEP. It implements Esterel's concurrency operator via multi-threading, which scales well to high degrees of concurrency with minimal resource overhead. Half a year later, the KEP3a and its compiler were presented [85, 84]. The KEP3a improves over the KEP3 in that it supports exception handling and provides context-dependent preemption handling instructions. The compiler employs a priority assignment approach that makes use of a novel concurrent control flow graph and has a complexity that in practice tends to be linear in the size of the program. Unlike earlier Esterel compilation schemes, this approach avoids unnecessary context switches by considering each thread's actual execution state at run time. Furthermore, it avoids code replication present in other approaches.

The latest version of the KEP is the KEP4, which is presented in this thesis. It enriches its control path for handling some delicate and complex mixed Esterel control structures, and supports more options for generating various configured processor series. It is the most powerful, flexible, and stable version of the KEP so far.

The KEP has also been employed as a platform for HW/SW co-design. Some extended works enrich the research of the multi-threaded reactive processor. Gädtke *et al.* [57, 58] presented an approach to accelerate reactive processing via an external logic block that handles complex signal expressions. An Esterel program is synthesized into a software component, running on the Kiel Esterel Processor, and a hardware component, consisting of simple combinational logic. The transformation process involves a two-step procedure, which first partitions the program at the source level and subsequently performs the synthesis. An intermediate logic minimization, at the source code level, facilitates the synthesis of compact logic blocks.

Another important part of the KEP project is the development of the KEP compiler [26, 85], including WCRT analysis [27, 28]. The analysis of the WCRT is influenced by the KEP in two ways: the exact number of instructions for each statement and the way parallelism is handled. The analysis is performed on a graph representation, the Concurrent KEP Assembler Graph (CKAG). In a first step we compute whether concurrent threads terminate instantaneously, thereafter it is able to compute for each statement how many instruction are maximally executable from it in one logical tick. The maximal value over all nodes gives us the WCRT of the program.

According to the classification method in Section 2.1, which depends on whether the processor description strategy is based on an existing COTS processor or not, the whole KEP series and the REMIC falls in the *Custom Processor* [78] implementation approach. Other Esterel processors, *i. e.*, REFLIX and RePIC, belong to the *Patched Processor* [80] implementation approach. Another classification method is based on the concurrency handling methods of these Esterel processors. Figure 2.2 compares the *multi-processing*

and the *multi-threaded* architectures. Of course it is always possible to translate an Esterel program into an equivalent program that has a flattened state space, *i. e.*, sequentialization. The REFLIX, the RePIC, and the KEP2 and its predecessor can be classified to this type. However, to handle concurrency directly, in the *multi-processing* implementation approach, every Esterel thread is mapped onto an independent processor to be executed, and a thread control unit handles the synchronization and communication between processors, as is done by the EMPEROR [124]. It allows the distributed execution of Esterel programs and also handles Esterel's concurrency operator. The EMPEROR uses a cyclic executive to implement concurrency, and allows the arbitrary mapping of threads onto processing nodes. This approach has the potential to speed up execution relative to single-processor implementations. However, their execution model potentially requires to replicate parts of the control logic at each processor. The most efficient concurrency implementation approach is *multi-threaded*, which employs multi-threading to implement concurrency. In this way, a single KEP core is extended to handle concurrency by an interleaved control flow. Each Esterel thread has an independent program counter and threads are scheduled according to their activation status and a dynamical changed priority. The KEP3 and later versions of the KEP fall into this category.

Although the multi-processing strategy seems a straightforward solution for the implementation of the Esterel language, it cannot avoid two essential limitations. First, to handle Esterel preemption and statements, it can neither support necessary Esterel-style instructions, nor follow the original Esterel semantics directly. For example, again consider the nested trap illustrated in Section 4.2.2. Since the Esterel threads are executed on different processors, when both of them throw exceptions, some complex mechanisms should be built for handling these exceptions correctly—this also degrades efficiency. Second, to handle the Esterel concurrency, such an architecture can hardly support the arbitrary nesting of concurrency and preemption. Furthermore, for a real embedded system, in particular if one wants to scale up to high degrees of concurrency and preemption nests, the multi-processor approach is relatively hardware-intensive.

Since multi-processing and multi-threaded reactive processors employ different strategies to handle Esterel concurrency, their compilers, which synthesize an Esterel program to the target reactive processor codes, also use different approaches to implement the communication between threads. For example, for multi-processing, the EMPEROR Esterel Compiler 2 (EEC2) [124] is based on a variant of the GRC, and appears to be competitive even for sequential executions on a traditional processor. However, their synchronization mechanism, which is based on a three-valued signal logic, does not seem to be able to take compile-time scheduling knowledge into account, but replaces it by repeating cycles through all threads until all signal values have been determined. Hence the compiler needs to generate sync instructions, to ensure that signals are not tested before they are emitted [41]. On the other hand, the multi-threaded implementation approach implements interleaving by inserting priority setting code at the context switch point. See also Section 3.2.1.

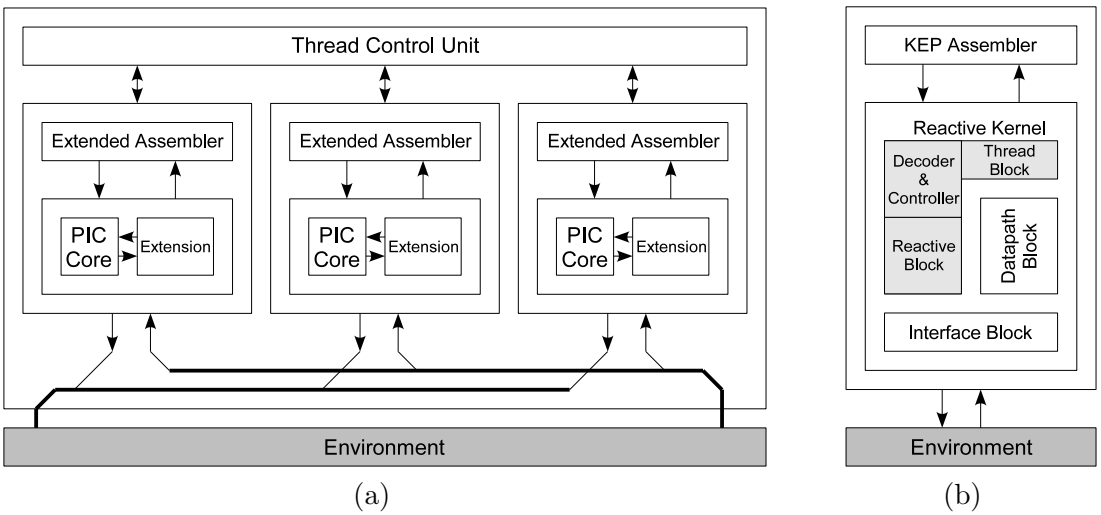


Figure 2.2: The architecture overview of multi-processing (a) and multi-threaded (b).

To give a summary review of previous Esterel processors other than the KEP series, we note that they have three significant limitations. First, the function of their Esterel style instructions is incomplete and cannot implement reactive control flow according to the original Esterel semantics. Second, it is not obvious how this design, *i. e.*, multi-processing for handling Esterel concurrency, would support the arbitrary nesting of concurrency and preemption. Finally, for a real embedded system, in particular if one wants to scale up to high degrees of concurrency and preemption nests, the multi-processor approach leads to relatively hardware-intensive implementations and large executables.

Chapter 3

The KEP Instruction Set Architecture

The development of the KEP was driven by the desire to achieve competitive execution speeds at minimal resource usage, considering processor size and power usage as well as instruction and data memory. A key to achieve this goal is the instruction set architecture (ISA) of the KEP, which allows the mapping of Esterel programs into compact machine codes while still keeping the processor light-weight. Notable features of the KEP ISA include the following:

- Unlike earlier reactive processing approaches, the KEP ISA is *complete* in that it allows a direct mapping of all the Esterel statements onto KEP assembler. All the Esterel kernel statements, including delay, preemption, concurrency and exception handling, are implemented directly and semantically accurate by the KEP, and they can be freely combined and nested as defined by the Esterel semantics. Valued signals and local variables are also supported.

- The KEP ISA is *efficient* in that most of the commonly used Esterel statements can be expressed directly with just a single KEP instruction.

- A characteristic of the Esterel language is that it provides a set of powerful control flow operators, which can be combined with each other in an arbitrary fashion. This makes the language concise and facilitates formal analysis; however, it can also make unrefined processing approaches fairly costly. The KEP ISA therefore not only supports common Esterel statements directly, but also takes into consideration the statement context. In particular, it provides preemption instructions that map onto different types of hardware units depending on whether preemptions are nested or not and whether they include single threads or multiple threads. Providing such a *refined* ISA further minimizes hardware usage while preserving the generality of the language.

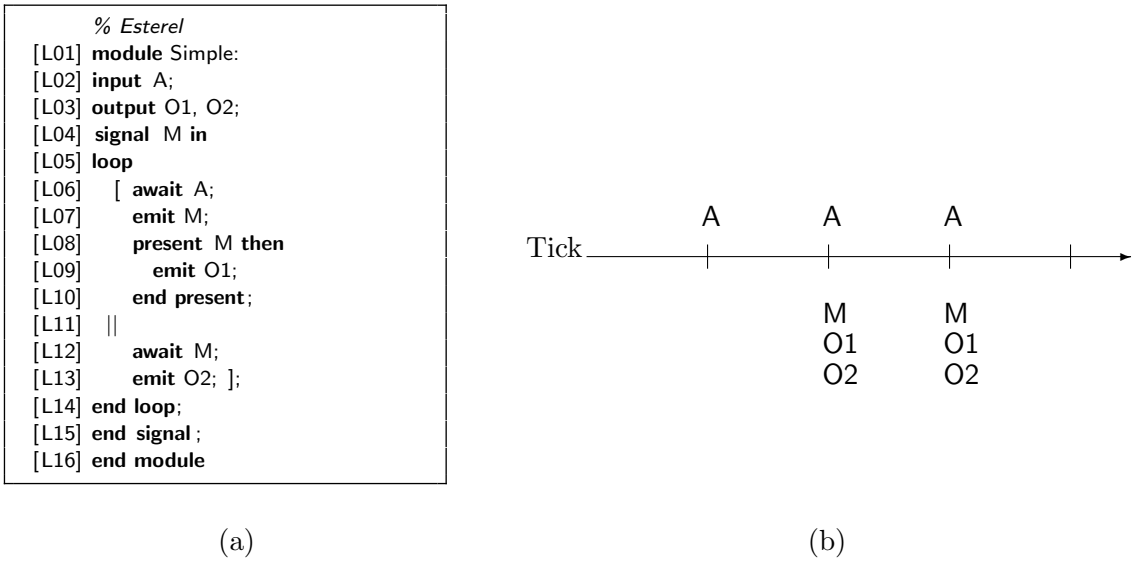


Figure 3.1: Simple: a module illustrating the structure of the Esterel program (a), and a possible execution trace (b).

This chapter presents the KEP instruction set architecture. In Section 3.1 we introduce some chosen Esterel statements/structures. The following Section 3.2 describes the design of the KEP's Esterel-type instructions, and classical processor instructions are presented in Section 3.3. Section 3.4 illustrates how to translate an Esterel program to the KEP assembler. Finally, the method of the instruction encoding is sketched in Section 3.5.

3.1 The Esterel Language

Esterel is an imperative synchronous language for the development of complex reactive systems. The *module* is the basic Esterel program unit. It is composed of the module name, the description of input/output signal interfaces, and an executable body. To write an Esterel program, a designated main module is necessary, and a collection of modules can also be included and referred to in the main module.

Figure 3.1(a) shows a simple example. In this case, the name of the module is defined as *Simple* by module Simple:$_{L01}$[1], and the end module$_{L16}$ indicates the end of the module.

As mentioned before, the *signal* is a fundamental concept of Esterel. The *interface signals* declared in the module interface are used to communicate with the environment. For example, in Figure 3.1, line 2 defines an input signal A, and line 3 defines two output

[1]To aid readability, we here use the convention of subscripting instructions with the line number where they occur.

signals O1 and O2. The other signal type is *local signals*. It aims to handle the internal communication of the Esterel program. The signal M_{L04} and end $signal_{L15}$ define the scope of local signal M.

Furthermore, there is a special pure signal tick, which represents the activation clock of the reactive program. Its status is present in each instant. This signal is declared implicitly and cannot be redeclared.

Signals have a presence status, *i. e.*, *present* or *absent*. At each tick, a signal is either present (*emitted*) or absent (not emitted). By default, signals are absent except for the tick signal. The statuses of signals are instantaneously broadcast throughout the program, which implies that all statements see each of them in a consistent way. In Esterel, the broadcasting of signals is used for processing communication. It is easy to understand that the input broadcasting is implicit because concurrent statements evolve in lock step in the same input environment. Furthermore, the signal broadcasting allows that the presence status of a signal can be tested by multiple signal receivers. For example, in the second tick, the signal M is emitted by emit M_{L07}, and is tested by present M $then_{L08}$ and triggers await M_{L12} simultaneously. Besides, from within the programs, the input and output signals can be equally tested for presence or absence, as well as the local signals used within the program.

An Esterel module reacts to an input event by generating an output event. The reaction is conceptually considered as instantaneous, and is called an instant or *tick*, see also Section 4.5. It reacts to the input event sequences (input histories) repeatedly and generates output histories. An Esterel module is deterministic, *i. e.*, the module always produces the same sequence of outputs when it is given the same sequence of inputs.

Figure 3.1(b) shows a possible execution trace of this module, with input signals shown above the time line and local and output signals below the time line. Note that the reaction is assumed to take no time because of the synchrony hypothesis. From the user's point of view, there is no need to worry about the internal reaction time, the output is considered as being generated at the same time as the input event occurrs—without logical delay.

An Esterel statement starts in some instant, remains active for a while, and may terminate in the current or some later instant. A statement is *instantaneous* if it terminates in the same instant it starts in; *e. g.*, in the second tick of the Simple module, the emit M_{L07} emits signal M, and present M $then_{L08}$ tests this signal immediately. Both of them are instantaneous statements. On the contrary, a non-instantaneous statement may delay some instants after it starts. For example, the delay statement await A_{L06} always takes time. It starts at the first instant, stays active, and terminates at the second instant even if A occurs at the first instant. Hence, it lasts one A.

In summary, the timing semantics of the Esterel statements relies on four structural notions. The *starting* instant of an Esterel statement is determined by the context of this statement in a program. The internal execution environment of this statement determines its *termination*. Since Esterel has block exits, the execution of a statement

can also be determined to exit a *trap* before this statement terminates. Finally, a statement can be *aborted* (killed) by some other part of the program.

The *timing* characteristic of the Esterel statements, *i. e.*, at which "instant" they are performed, is one of the key properties of Esterel statements. Furthermore, the preemption and concurrency statements provide that Esterel has more powerful control flow than traditional programming languages. Hence, to design a processor targeting the direct execution of Esterel, it is necessary to study the semantics of the Esterel statements first.

3.1.1 Esterel Statements

There are dozens of Esterel statements. Hence, it seems difficult to decide which statements should be chosen as the instructions to be implemented by the processor directly. In fact, the choice of the instruction set is also a common issue in the ASIP design field [1, 62, 75].

Fortunately, there is a very small number of *kernel statements* in Esterel and a comparatively large number of *derived statements*. The derived statements can always be replaced by equivalent constructions that involve only kernel statements; the derived statements are merely syntactic sugar, *i. e.*, convenient shorthands for the programmer. In fact, the accepted set of Esterel kernel statements has evolved over time. For example, the halt statement, which performs no action and never terminates nor exits traps [23], used to be regarded as a kernel statement [13, 11], but now is considered as the combination of the loop and the pause. We here adopt the definition of which statements are kernel statements from the v5 standard [19, 17].

The following description briefly introduces the semantics of the Esterel kernel statements. For a more detailed discussion, refer to [23, 16, 19].

- ||
  "||" is the parallel statement. It forks control into concurrently executed *threads*. The two threads of a parallel immediately start when the parallel statement starts. The parallel statement will stay active whenever one of its threads remains active, unless a branch exits a trap. If both of its threads are terminated, the parallel statement will terminate instantaneously.

  If a parallel's threads terminate in different instants, the parallel will wait for the last one to terminate. Furthermore, parallel threads may simultaneously exit traps. If one thread exits the trap T or both threads exit the same trap T, of course the parallel will exit T. However, if threads try to exit different traps T1 and T2 in the same instant, the parallel will exit the outermost of these traps, the other one will be discarded.

- suspend ... when *S*
  The suspend statement provides a *suspension* mechanism. It freezes the state of a

body for the instant when the trigger event occurs. In this statement, the sensitive signal *S* is also called the *guard*. In the initial instant, the body of the suspension is started; and then the guard controls execution of the body in each instant. If the body of the suspension terminates or exits a trap, so does the suspend statement; if it pauses, then the suspend statement also pauses. Note the default format of this statement implies a delay. Hence, only if the body of suspension does not terminate in the first instant then the sensitive signal *S* will be tested for presence as long as the body of suspension remains active.

From the second tick, if signal *S* is present, the suspension body will not be executed in the instant and it is kept frozen for the next instant. In this case, the body of the suspension is suspended for this instant. On the other hand, if signal *S* is absent, then the body receives the control for this instant, *i. e.*, the body is activated for the instant.

Incidentally, the suspension statement was not a kernel statement before.

- trap *T* in ... exit *T* ... end trap
  The trap statement expressed as trap *T* in ... end trap, and the exit statement expressed as exit *T*, are a statement pair which provides the exception mechanism. The trap statement defines a scoped exit point *T* for its body; and the exit statement exits from the trap.

  When the trap statement starts, it immediately starts its body and behaves as its body until termination or exit. The termination of its body terminates the trap statement itself. If the body exits the trap *T*, then the trap statement immediately terminates, and its body will be weakly aborted. Furthermore, when traps are nested, the outer one takes priority.

- pause
  The pause statement pauses for one instant. It pauses when started, and then it terminates in the next instant. The semantics of pause exactly equals await tick, which means waiting for the next tick to arrive. The pause statements act as state variables. Incidentally, the halt statement, which was considered as a kernel statements, pauses forever and never terminates.

- signal *S* in ... end
  This statement is a local signal declaration statement. Unlike the interface declaration, which just can be placed in the beginning of the Esterel module, the local signal declaration is an executable statement, and can be placed wherever a statement can.

  When started, this statement immediately starts its body with a fresh local signal *S* overriding any such signal that might already exist; and then it behaves as its body until termination or exit. The status of the local signal *S* is not exported.

- emit *S*
  The emit statement provides signal emission. An emit *S* statement instantaneously

broadcasts the signal *S*, *i. e.*, sets its status to present and terminates instantaneously. The emission of signal *S* is just valid for the current instant, *i. e.*, although the status of a signal S is present in current instant, it will be absent again in the next instant (unless it is emitted again).

- present *S* then ... else ... end
  Corresponding to the signal emission statement, which sets the presence of a signal, the present statement tests the signal presence.

  When the signal test statement starts, it checks a signal expression and performs a conditional branch. Each of the then and else branches can be omitted, but at least one of them must be specified. The present statement terminates when the body of the corresponding branch terminates.

- nothing
  This is a dummy statement, which performs no action and terminates instantaneously.

- loop ... end loop
  This statement provides an infinite loop. When started, it immediately starts its body. Whenever its body terminates, it is immediately restarted; and if its body exits a trap, so does the whole loop. Note that the body of a loop cannot terminate instantaneously from its starting, *i. e.*, any execution path of its body must include either a pause or an exit statement. A loop statement never terminates. Hence, the solution of escaping from the loop is to enclose it within a trap, and to execute an exit statement to exit the trap.

- ;
  The ";" separates two statements to a sequence. The first statement of a sequence starts when the sequence starts. When the first statement terminates, control is passed instantaneously to the second one, which determines the behavior of the sequence from then on. When the first statement exits a trap, the second statement will never start.

  A sequence of emit S1; emit S2 emits two signals simultaneously and terminates instantaneously. However, it does not mean that the instantaneous statements can be placed arbitrarily in a sequence. For example, testing a signal at first and then emitting it is forbidden because such a sequence breaks constructiveness [18]. On the contrary, testing and emitting a signal in two parallel threads is allowed.

To map the Esterel statements to an Esterel processor in a direct way, it would suffice implement all of the kernel statements, although some Esterel expressions, *e. g.*, valued signals, cannot be described by the Esterel kernel statement [19]—some basic data handling statement are required. In principle, any set of Esterel statements from which the remaining statements can be constructed is considered as a valid set of kernel statements. This process, *i. e.*, expanding derived statements into equivalent, more

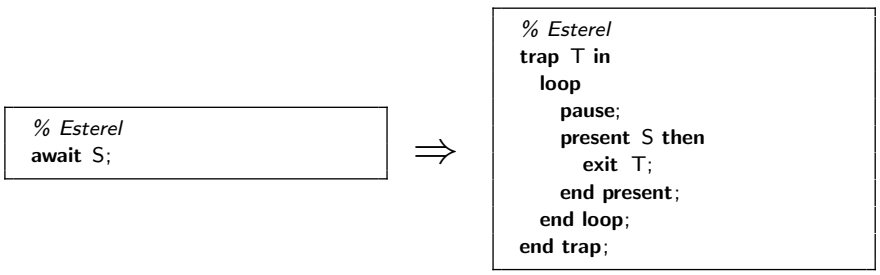


Figure 3.2: The equivalent expression of the await statement by kernel statements.

primitive statements, is called *dismantling*. However, it is questionable whether it really is a proper strategy to just implement the kernel statements and then dismantle all of other statements into the kernel statement expression.

Let us review the alternative expression of the pause statement, *i. e.*, await tick. It implies the statement waits for a tick presence event. Hence, the basic expression of waiting for a signal *S* is await *S*, which is one of the most frequently used Esterel control statements. Figure 3.2 lists a possible dismantling expression of this statement.

Obviously, it is inefficient to represent all Esterel functions and structures by kernel statements. Employing derived statements could express a reactive module in a more flexible and simplified way. If we put all Esterel control statements together, and further consider what gives Esterel an advantage over traditional programming languages for modelling reactive systems, we can identify groups of Esterel constructs and statements that are not supported by traditional programming languages, *i. e.*, signal emission, delay, concurrency, preemption and exception [19].

- *Signal Emission*: *e. g.*, emit, sustain, etc.
  As mentioned above, a signal is absent unless it is emitted. However, the previous presence of a signal in the last tick is recorded. In Esterel, the *pre(S)* directly expresses the status of S at the previous instant.

  Note that there is another signal type, *i. e.*, the *valued signal*. In addition to its presence status, a valued signal carries a value, which can be of arbitrary type. For a valued signal, the emit *S(e)* statement evaluates the data expression *e*, emits *S* with that value, and terminates instantaneously. Unlike its presence status, the valued signal's value, which is expressed as *?S*, is persistent. The value is kept until the valued signal is emitted in a forthcoming tick. The expression *pre(?S)* yields the value of *S* at the previous instant. An initial value can be given to a signal, and it is initial for both *?S* and *pre(?S)*.

  Furthermore, the sustain statement provides continuous emission of a signal. Figure 3.3 shows its equivalent expression by kernel statements.

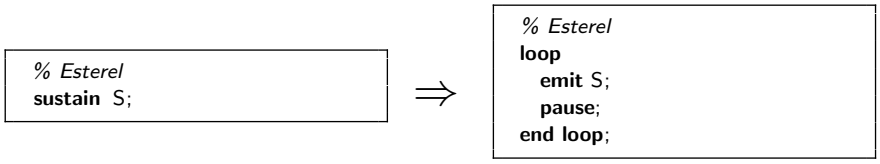
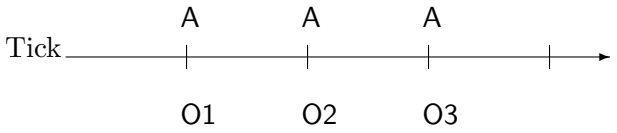


Figure 3.3: The equivalent expression of the sustain statement by kernel statements.

```
% Esterel
module Delay:
input A;
output O1, O2, O3;
 [ await immediate A;
   emit O1;
||
   await A;
   emit O2;
||
   await 2 A;
   emit O3; ]
end module
```

(a)



(b)

Figure 3.4: Delay: an Esterel module illustrating the difference of Esterel delay expressions (a), and a possible execution trace (b).

- *Delay*: *e. g.*, await, pause, etc.
  The await statement is the simplest temporal statement. In its basic form, *i. e.*, await S, it waits for a delay. In total, there are three forms of delay expressions:

  - ***standard delays*** start in current instant, and then wait until a delay elapses in some further instant. Standard delays never elapse instantaneously.
  - ***immediate delays*** terminate instantaneously if the signal expression is true in the starting instant.
  - ***count delays*** are similar to the standard delays, but wait for a specified delay count.

  Figure 3.4 illustrates differences of those delay expressions of the await statement. The await immediate A is an immediate delays statement. The await A provides a standard delay for signal A. And the await 2 A specifies the count number for the delay as 2.

- *Concurrency*: *i. e.*, ||
  In Esterel, a thread forks on a || parallel statement, and terminates when all its branches have terminated. The signals emitted by any of its branches or by the rest of the program are instantaneously broadcast to all branches in each instant.

  When several threads are active concurrently, they may communicate back and forth instantaneously, that is, within the same logical tick. Therefore, implementing a concurrent Esterel program onto a sequential processor efficiently and correctly becomes a challenge.

- *Preemption*: *e. g.*, [weak] abort, suspend, etc.
  Various types of preemption are one of the key features for making Esterel's control flow primitives richer than that of traditional, sequential programming languages [15]. The Esterel preemption includes abortion and suspension [18]:

  – Abortion
    An abortion statement kills its *abort body* upon a specific *trigger signal*. In *strong abortion*, expressed by abort, the body does not receive control at the instant when the trigger occurs. For *weak abortion*, performed by weak abort, the body receives control for a last time at the abortion time. The abortion statements support all kind of delay expressions. For example, an abort ... when 2 S statement kills its body when the signal S occurs twice after the tick of the statement started.

  – Suspension
    The suspension performed by suspend freezes the state of a body for the instant when the trigger event occurs. Unlike the abortion statements, the suspend statement does not support count delays, and there is also no weak suspension[2].

  The following three illustrations describe the subtle difference between abort and weak abort.

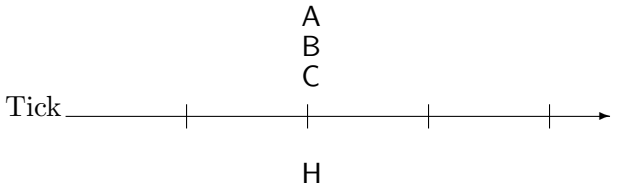
  Figure 3.5(a) shows the Esterel module Abort1 as an example of a nested strong abort. After starting, the module watches signals A and B as abortion trigger signals and stays to wait for signal C at line 3. Those two abortions are nested, and the outer abortion, triggered by B, has higher priority. In case signals A, B, and C occur simultaneously, after the initial instance (as the abort is not immediate), signal B takes priority over A. That means the module will kill the bodies of abortion A and abortion B immediately, executes emit H_{L11} and then $halt_{L12}$. This is illustrated by the trace in Figure 3.5(b).

  Figure 3.6(a) shows the module Abort2, which illustrates the response trace of a mixed nested strong/weak abortion. Similar to Abort1, the module watches signals A and B as abortion trigger signals and stays on the delay statement waiting for

[2]The Esterel V7 adds *weak suspend* as a new control statement. However, it is not an issue in this thesis.

```
      % Esterel
      module Abort1:
      input A, B, C, D;
      output E, F, G, H;
[L01] abort
[L02]   abort
[L03]     await C;
[L04]     emit E;
[L05]     await D;
[L06]     emit F;
[L07]   when A;
[L08]   emit G;
[L09]   await D;
[L10] when B;
[L11] emit H;
[L12] halt
      end module
```

(a)



(b)

Figure 3.5: Abort1: an Esterel module illustrating the strong abort nest (a), and a possible execution trace (b).

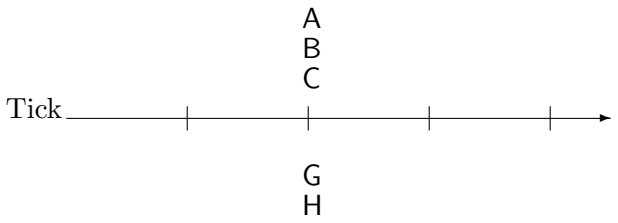
signal C. When A, B, and C occur simultaneously, the signal A again triggers a strong abortion, so the body of abortion A is killed immediately. However, the body of abortion B will not be killed immediately. That means emit G_{L08} will be executed. The following statement await D_{L09} will result in pause, so it has no further effect in this instance. The body of abortion B is killed and then emit H_{L11} and $halt_{L12}$ will be executed. The corresponding trace is shown in Figure 3.6(b).

Figure 3.7(a) shows the module Abort3, which illustrates the response trace of a nested weak abortion. In this case, when A, B, and C occur simultaneously, the body of abortion A receives the control for a last time. That means await C_{L03} is terminated. Then the following concurrent statement emit E_{L04} will be executed. The await D_{L05} statement is a sequential statement which takes multiple cycles (equal to two sequential statements: pause; await immediate D;), so the body of abortion A is killed, emit G_{L08} will be executed. For similar reasons, emit H_{L11} and $halt_{L12}$ will be executed in this tick. The resulting trace is shown in Figure 3.7(b).

- *Exception*: *i. e.*, trap and exit
  As mentioned before, an exception is *declared* with a trap scope, and is *thrown* with an exit statement. An exit T statement causes control flow to move to the end of the scope of the corresponding trap T declaration. Compared with control structures of traditional programming languages, this is similar to a goto statement. However, there are complications when traps are nested or when the trap scope includes concurrent threads. The following rules apply: *if one thread raises an exception and the corresponding trap scope includes concurrent threads, then the concurrent*

```
        % Esterel
        module Abort2:
        input A, B, C, D;
        output E, F, G, H;
[L01]   weak abort
[L02]     abort
[L03]       await C;
[L04]       emit E;
[L05]       await D;
[L06]       emit F;
[L07]     when A;
[L08]     emit G;
[L09]     await D;
[L10]   when B;
[L11]   emit H;
[L12]   halt
        end module
```

(a)



(b)

Figure 3.6: Abort2: an Esterel module illustrating the mixed abort/weak abort nest (a), and a possible execution trace (b).

threads are weakly aborted; if concurrent threads execute multiple exit *instructions in the same tick, the outermost trap takes priority.*

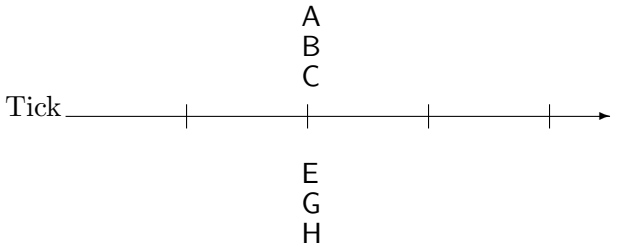
All these constructs are orthogonal. This means that they can be freely mixed at any nesting depth without restriction. Many languages limit concurrency to the top level, therefore loosing orthogonality. In fact, an Esterel program consists of a collection of nested, concurrently running threads described using a traditional imperative syntax, and arbitrary preemption structures also can nest or be nested by those threads [15]. The following section gives an Esterel example to illustrate the intricacies of the reactive control flow constructs.

3.1.2 An Example Program

Let us consider the EXAMPLE Esterel module in Figure 3.8(a). In this module, two concurrent threads are enclosed in an every block, which restarts its body whenever the input signal S is present (except for the initial tick, when S is ignored, as the every is not "immediate"). At the beginning of the body of the every block, it declares two trap scopes, *i. e.*, the T1 trap and the T2 trap. The T1 trap is the outer one. Then two threads are forked. The first thread initially waits for the input signal I, and then builds two nested preemption blocks. The outer one is a weak abortion, which can be triggered by the local signal A at the first tick (immediate). The inner one employs the H signal as the guard of a suspension. Its body is a sustain R_{L08} statement which emits the local

```
      % Esterel
      module Abort3:
      input A, B, C, D;
      output E, F, G, H;
[L01] weak abort
[L02]   weak abort
[L03]     await C;
[L04]     emit E;
[L05]     await D;
[L06]     emit F;
[L07]   when A;
[L08]   emit G;
[L09]   await D;
[L10] when B;
[L11] emit H;
[L12] halt
      end module
```

(a)



(b)

Figure 3.7: Abort3: an Esterel module illustrating the weak abort nest (a), and a possible execution trace (b).

signal R continuously. Following the body of the weak abortion A, the output signal O1 will be emitted, and then the control exits the T1 trap. The second thread initially idles for two ticks, and then emits A if R is present, and exits the T2 trap.

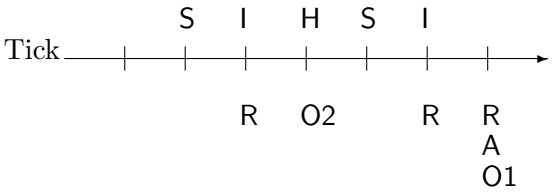
A possible execution trace is shown in Figure 3.8(b), with input signals shown above the time line and local and output signals below the time line. All signals are absent at the initial tick; at the second tick, the presence of signal S triggers the start of the every body, and then the initial thread creates two sub threads, *i. e.*, the thread 1 and the thread 2; afterward the control stays at the delay statements of either of sub threads, *i. e.*, the await $\mathsf{I}_{\mathsf{L05}}$ of the thread 1, and the await 2 $\mathsf{tick}_{\mathsf{L14}}$ of the thread 2.

At the third tick, the presence of signal I terminates the await $\mathsf{I}_{\mathsf{L05}}$ statement. The control goes through, and enters the scope of the preemption nest, emits the inner signal R, and then stays there. Thread 2 pauses for one tick. However, the count delay statement await 2 $\mathsf{tick}_{\mathsf{L14}}$ does not terminate.

At the fourth tick, since the signal H is present, the body of the suspension is frozen. Hence, the inner signal R will not be emitted. On the other hand, the thread 2 terminates the await 2 $\mathsf{tick}_{\mathsf{L14}}$ statement. It tests the status of the signal R, and jumps to the end of the present scope because the signal R is absent. The execution of exit $\mathsf{T2}_{\mathsf{L18}}$ causes an exception of the T2 trap. The thread 2 is terminated immediately, and thread 1 is also weakly aborted. Hence, the emit O2 statement, which is located after the end of the T2 trap scope, is executed. Now the control stays at the end of the every block, and will be halted until the presence of input signal S in some later instant restarts the body of the every block.

```
          % Esterel
          module EXAMPLE:
          input S, I, H;
          output O1, O2;
 [L01,T0] signal A,R in
 [L02,T0] every S do
 [L03,T0]   trap T1 in
 [L04,T0]     trap T2 in
 [L05,T1]       [ await I;
 [L06,T1]         weak abort
 [L07,T1]           suspend
 [L08,T1]             sustain R;
 [L09,T1]           when H;
 [L10,T1]         when immediate A;
 [L11,T1]         emit O1;
 [L12,T1]         exit T1;
 [L13]          ||
 [L14,T2]         await 2 tick;
 [L15,T2]         present R then
 [L16,T2]           emit A;
 [L17,T2]         end present;
 [L18,T2]         exit T2; ];
 [L19,T0]     end trap;
 [L20,T0]     emit O2;
 [L21,T0]   end trap;
 [L22,T0] end every;
 [L23,T0] end signal
          end module
```

(a)



(b)

Figure 3.8: EXAMPLE: an Esterel module illustrating Esterel parallel, preemption, and exception statements (a), and a possible execution trace (b). The KEP assembler includes labels (in brackets) that list the line number ("Lxx") and thread id ("Tx").

The signal S is present in the fifth tick, and in the sixth tick the signal I is present. The behavior of the program in those two ticks is similar to what happened in the second and third ticks.

In the seventh tick, none of the input signals are present. Hence, the signal R is emitted by the sustain R_{L08} statement of the thread 1. Note that in the same tick, the thread 2 tests the signal R via the present R_{L15} statement, so the second thread enters the "then" branch and executes emit A_{16} to set the inner signal A as present, and then throws an exception of the T2 trap by the execution of exit T2. Note that the weak abortion A in the thread 1 is triggered by the presence of signal A, hence the body of the abortion will be weakly killed. The control jumps to the emit $O1_{L11}$ and executes it, and then exits the T1 trap. Hence, the parallel threads throw two exceptions at the same time. Since the T1 trap and the T2 trap are nested and the T1 trap is the outer one, it takes priority. Therefore, the control jumps to the end of the scope of the T1 trap, and then

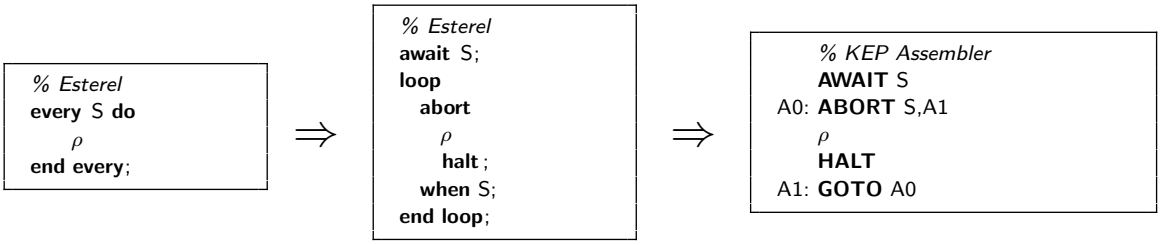


Figure 3.9: Translation rules of the every statement.

halts until the presence of input signal S in some later instant restarts the body of the every block.

3.2 Design of the Esterel-type Instructions

To design the instruction set architecture for directly describing the high level language Esterel, one of the challenges is how to describe the control structures in a natural way, and to allow arbitrary nesting. Of course, implementing all Esterel control statements directly might be possible, but would not be practical when considering silicon real estate, cost, and complexity. Hence, the implementation of those instructions should achieve competitive execution speeds at minimal resource usage, considering processor size and power usage as well as instruction and data memory. The key to achieve this goal is the instruction set architecture (ISA) of the KEP, which allows the mapping of Esterel programs into compact machine code while still keeping the processor light-weight.

To achieve the above target, we use two simple and effective design strategies. First, all Esterel kernel statements should be implemented. It provides the possibility of handling all Esterel control structures correctly and efficiently. Second, some other frequently used non-kernel Esterel statements/structures are also directly supported. The choice of these Esterel statements is mainly based on our experiences, which is a common method in the ASIP instruction set design strategy.

Of course, the second strategy also implies some other Esterel statements, which are infrequently used, or are difficult/inefficient to be implemented, will be handled by standard Esterel syntax translation. For example, the every statement which was used in Section 3.1.2, can be easily dismantled. Figure 3.9 shows the translations. The final KEP expression costs four instructions, which is still acceptable.

Recall the classification of the Esterel statements described in Section 3.1.1. The following discusses how to handle each class of statements, *i. e.*, handling concurrency, preemption, exception, emission, and delay.

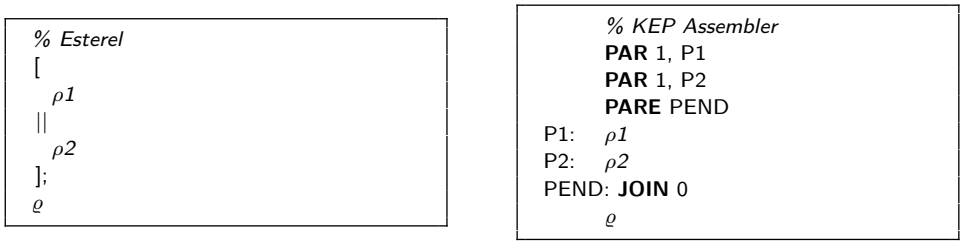


Figure 3.10: The KEP instructions for handling concurrency.

3.2.1 Handling Concurrency

One of the key issues of Esterel-type instruction design is how to implement Esterel's concurrency operator ("||"). A concurrent Esterel statement with n concurrent threads extended by the ||-operator is translated into KEP assembler as follows. First, threads are *forked*, meaning n new threads will be created, in addition to the previously existing thread(s). The fork is performed by a series of instructions that consist of n PAR instructions and one PARE instruction, which together initialize the Thread Block. Each PAR instruction creates one thread, by assigning a *start address*, *i. e.*, initializing an address register that is associated with this thread in the Thread Block, and a non-negative priority. The main thread that starts the program is implicitly assigned priority 0, which is the lowest possible priority. For example, in the KEP code shown in Figure 3.10(b), the PAR 1, P1 instruction creates a thread with priority 1 that starts at label P1. The *end address* of each thread is the address of the instruction which is immediately after the last instruction of this thread. This address is stored by the Thread Block as the end address of this thread. The end address is either given by the start address specified in a subsequent PAR instruction, or, if there is no more thread to be created, it is specified in a PARE instruction. In Figure 3.10(b), the fork instructions create two threads, the first with address ranging from P1 to (but excluding) P2, and the second ranging from P2 to (but excluding) PEND. Furthermore, the Thread Block still separately keeps track of the incoming thread that executes the fork instructions; for this thread, its program count is set to the end address of the finally created thread. The fork instructions are followed by KEP instructions for each of the created threads, in the specified address ranges. The code block for the last thread is followed by a JOIN instruction, which waits for the termination of all forked threads and concludes the concurrent statement.

The KEP employs the multi-threaded architecture to handle Esterel's concurrency (see Section 4.2.1). Therefore, the KEP can deal with concurrency by an *interleaved control flow*. This is different from statically scheduled interleaving, which could for example be implemented with gotos. The threads are assigned dynamic priorities, and the controller of the processor runs the individual threads accordingly.

To implement concurrency, a hurdle is how to interleave thread execution to allow the communication among threads within a logical tick. As already illustrated in the example shown in Figure 3.8, a thread may be executed partially, then control may jump to another thread, and later return to the first thread, all within the same tick. To handle this, the KEP employs a *multi-threaded architecture*, where each thread has an independent program counter (PC) and threads are scheduled according to their activation status and a dynamically changing priority, see Section 4.2.1.

The priority of a thread is assigned when the thread is created. Furthermore, it can be changed subsequently by executing a priority setting instruction (PRIO). To obtain a more general understanding of how the priority mechanism influences the order of execution, recall that at the start of each instant, all unterminated threads are activated, and are subsequently scheduled according to their priorities. Furthermore, each thread is assigned a priority upon its creation. Once a thread is created, its priority remains the same, unless it changes its own priority with a PRIO instruction, in which case it keeps the new priority until it executes another PRIO instruction, and so on. Neither the scheduler nor other threads can change its priority. Note that a PRIO instruction is considered instantaneous; the only non-instantaneous instructions, which delimit the logical ticks, are the PAUSE instruction and derived instructions, such as AWAIT and SUSTAIN. This mechanism has a couple of implications:

- At the start of a tick, a thread is resumed with the priority corresponding to the last PRIO instruction it executed during the preceding ticks, or with the priority with which it has been created if it has not executed any PRIO instructions. In particular, if we must set the priority of a thread to ensure that at the beginning of a tick the thread is resumed with a certain priority, it is not sufficient to execute a PRIO instruction at the beginning of that tick; instead, that PRIO instruction must have already been executed in the preceding tick.

- A thread is executed only if no other active thread has a higher priority. Once a thread is executed, it continues to execute until a non-instantaneous statement is reached, or until its priority is lower than that of another active thread. While a thread is executed, it is not possible for other inactive threads to become active; furthermore, while a thread is executed, it is impossible for other threads to change their priority. Hence, the only way for a thread's priority to become lower than that of other active threads is to execute a PRIO instruction to lower its priority by itself.

3.2.2 Handling Preemption

As mentioned in Section 3.1.1, an Esterel preemption structure defines some key parameters, which include its scope, sensitive signal, preemption type and delay expression. The design of KEP preemption instructions targets to express these parameters in an efficient format.

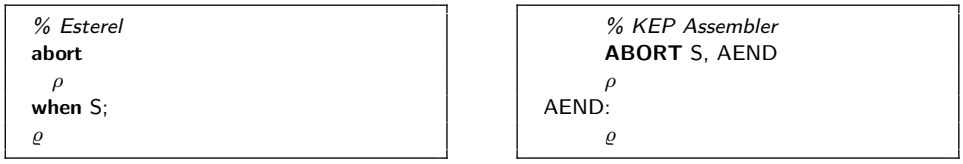


Figure 3.11: The KEP instruction for handling preemption.

Figure 3.11 shows an example to illustrate the way of handling preemption. The "abort ... when A" statement of the Esterel version has been turned into an "ABORT A, AEND" KEP assembler instruction, which states that the following statements until the label AEND constitute an abort body that should be strongly aborted when the signal A has occurred once. Hence, the scope of the abortion is defined as from where the ABORT instruction is located to the AEND. Furthermore, the delay count is assigned as the default 1, see also Section 3.3.

As discussed further in Section 4.2.2, the Reactive Block contains a configurable number of Watcher modules, which are responsible for implementing the preemption operations. According to the Esterel semantics, a preemption (abortion or suspension) is *enabled* when control is in its body, and *disabled* when control is outside of its body. When a preemption is *enabled*, the corresponding trigger signal is watched and the module can react to the presence of it (is *active*). Otherwise, the signal will not cause preemption. We call this scheme *Inside/Outside Preemption Range Watching* (*IOPRW*) [89, 88, 86]. Considering the nesting structure of preemption, we can conclude that the outer abortion has a wider address range which covers the inner one. Hence, the Reactive Block handles the hierarchy of preemption nesting by this feature.

However, if we further study the structure of an Esterel program, we will that find the hierarchy of the preemptions tends to be parallel or sequential rather than nested. For example, most of the preemptions just act on a small range, and do not nest other preemptions. Hence, if these kinds of preemption are loaded into a watcher, it is unnecessary to test whether another preemption is located inside of its scope or not. In other words, these preemptions are mutually exclusive. Hence, designing some trimmed-down watcher versions to fit the common structure of the Esterel program can save hardware resources. See also Section 4.2.2.

To configure different versions of watchers, several instructions are introduced. The thread-abortion instructions (*i. e.*, TABORT, TABORTI, TWABORT, and TWABORT) can replace corresponding abortions which neither include concurrent threads nor other preemptions. An intermediate variant are the local-abortion instructions (*i. e.*, LABORT, LABORTI, LWABORT, and LWABORT). They can replace corresponding abortions which may include concurrent threads and also preemptions handled by thread-abortion instructions, but cannot include another local-abortion instruction.

```
% Esterel
module RUNNER:
every MORNING do
  trap HEART_ATTACK in
    loop
      abort
        sustain WALK
      when 10 METER;
      abort
        [ every STEP do
            emit JUMP
          end every
        ||
          loop
            await SECOND do
              exit HEART_ATTACK
            end
          each HEART_BEAT
      when 5 SECOND;
      sustain RUN
    each LAP
  handle HEART_ATTACK do
    emit GO_TO_HOSPITAL
  end trap
end every
end module
```

(a)

```
        % KEP Assembler
        % module RUNNER
        EMIT _TICKLEN,#20
        AWAIT MORNING
A2:     ABORT MORNING,A3
A4:     ABORT LAP,A5
        LOAD _COUNT,#10
        TABORT METER,A6
        SUSTAIN WALK
A6:     LOAD _COUNT,#5
        LABORT SECOND,A8
        PAR 1,A9,1
        PAR 1,A10,2
        PARE A11,1
A9:     AWAIT STEP
A14:    TABORT STEP,A15
        EMIT JUMP
        HALT
A15:    GOTO A14
A10: A17:
        TABORT HEART_BEAT,A18
        AWAIT SECOND
        EXIT HEART_ATTACK
A18:    GOTO A17
A11:    JOIN 1
A8:     SUSTAIN RUN
A5:     GOTO A4
HEART_ATTACK:
        EMIT GO_TO_HOSPITAL
        HALT
A3:     GOTO A2
```

(b)

Figure 3.12: Translating the Esterel RUNNER module (a) to the KEP assembler program (b) with refined instructions employed.

Figure 3.12 indicates how the RUNNER, a well-known Esterel example [19], is translated to the KEP assembler program, and then optimized by the defined instructions. Section 4.2.2 describes the architecture of various watcher types in detail, and Section 5.3 further illustrates the benefit of this strategy.

Finally, the KEP provides preemption instructions that map onto different types of hardware units depending on whether preemptions are nested or not and whether they include single threads or multiple threads. This feature is fit for the common structure of the Esterel program, and further minimizes hardware usage while preserving the generality of the language.

```
% Esterel
trap T in
 ρ1
  exit T;
 ρ2
end trap;
ϱ
```

```
        % KEP Assembler
TSTART:
        ρ1
        EXIT TEND, TSTART
        ρ2
TEND:
        ϱ
```

Figure 3.13: KEP instruction for handling exception.

3.2.3 Handling Exceptions

To implement Esterel's exception handling mechanism (trap/exit), the EXIT instruction is introduced. It associates the trap scope with the corresponding exit. The *startAddr* indicates the beginning of the trap body (entry point), and the *endAddr* defines the end of the trap scope (exit point).

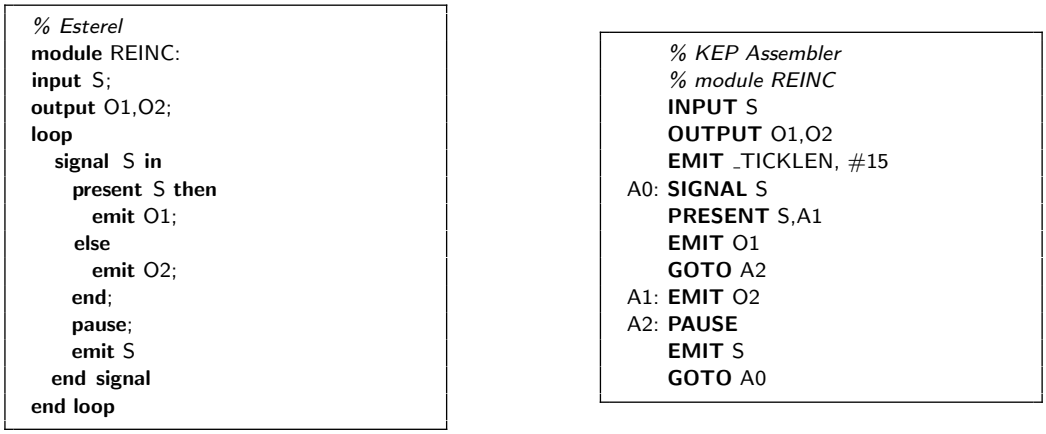
Figure 3.13 shows a simple case. Two address labels mark the trap scope, and then they are added into the corresponding exit statement. When the EXIT instruction is executed, those parameters, which include the whole address information of the trap scope, will be totally transferred to the Exception Element in the Reactive Block.

3.2.4 Handling Signal and Schizophrenia

Handling emission and testing of a pure signal is fairly straightforward. The basic principle of the KEP signal handling is setting the status of a signal as "present" if an emission instruction is executed, and the statuses of all signals will be cleared as "absent" when a new tick starts.

To enhance the ability of the emission instruction of the KEP, we extend some additional parameters to support valued signals directly. For example, an Esterel emit A(5) statement could be expressed as EMIT A,#5, which implies the immediate data 5 will be put into a data register of the signal A.

Another issue of the Esterel signal handling is how to deal with local signals, which are declared in Esterel with signal declarations. We introduce the SIGNAL instruction to implement a signal scope and to correctly handle reincarnation. When executing a SIGNAL *S* statement, the previous and current status of *S* are both cleared, thus a fresh signal is effectively introduced. Obviously, this straightforward method follows the original semantic definition of Esterel and can deal with schizophrenic programs efficiently and correctly—if an Esterel statement must be executed for multiple times within a tick, the KEP simply does so.



```
% Esterel
module REINC:
input S;
output O1,O2;
loop
  signal S in
    present S then
      emit O1;
    else
      emit O2;
    end;
    pause;
    emit S
  end signal
end loop
```

(a)

```
    % KEP Assembler
    % module REINC
    INPUT S
    OUTPUT O1,O2
    EMIT _TICKLEN, #15
A0: SIGNAL S
    PRESENT S,A1
    EMIT O1
    GOTO A2
A1: EMIT O2
A2: PAUSE
    EMIT S
    GOTO A0
```

(b)

Figure 3.14: REINC: Translation of the Esterel signal declaration (a) into to the KEP SIGNAL instruction (b).

To illustrate this, consider again the Esterel module REINC in Figure 3.14(a) (cited from Figure 2.1(a)) along with the corresponding KEP assembler. In the first instant, the local signal S is declared and initialized. Therefore, its status is absent. The else branch of the present statement is taken and O2 is emitted. In the second instant, S is emitted. The loop body terminates and then it is restarted. The local signal declaration is immediately re-initialized, and the fresh incarnation is absent. The present statement tests the fresh incarnation and only O2 is emitted.

3.2.5 Handling Delays

It is easy to introduce the basic delay statement: an Esterel await A statement could be expressed as AWAIT A instruction to let the control pause in current tick, and then waits for signal A in the following ticks.

A further question is how to handle the Esterel await case statement, which awaits several delays simultaneously. Of course it is possible to dismantle this statement into its kernel statement representation, however, this would be fairly inefficient. On the other hand, we can introduce some instructions, which are similar to the PAR/PARE instructions, to describe the scope of the case branches. However, those two structures are different. The PAR/PARE instructions create sub threads of the current thread. Hence, the control of the thread is also 'branching'. Therefore, the end address of a branched thread is necessary for controlling the status of the sub thread, *e. g.*, handling an exception. On the other hand, for the await case statement, the control simply *jumps*

```
% Esterel
await
  case A do
    ρ1
  case immediate B do
    ρ2
  case C do
    ρ3
end await;
ϱ
```

```
        % KEP Assembler
        CAWAITS
CWS:    CAWAIT A, ASTART
        CAWAITI B, BSTART
        CAWAIT C, CSTART
        CAWAITE CWS
ASTART: ρ1
        GOTO END
BSTART: ρ2
        GOTO END
CSTART: ρ3
        GOTO END
END:    ϱ
```

Figure 3.15: The KEP instruction for handling multiple signal awaiting.

to the entry point of the corresponding branch, and all of the other branches will be abandoned. Once the body of a branch terminates, the control will be directed to the end of the whole block.

Therefore, the KEP CAWAIT instructions define the start address of each branch block to guide the control entering into a branch body; and the GOTO instructions located at the end of each case branch will direct the control to the end of the await case block. Furthermore, the GOTO instruction located at the end of the last case branch block can be left out. The behavior of the KEP program exactly matches the original semantics.

The Esterel program shown in Figure 3.15 immediately starts *ρ2* if the signal B occurs at start time. Otherwise, the first delay to elapse determines the subsequent behavior: *ρ1* is started if signal A elapses first, *ρ2* is started if signal B occurs first, and so on. If several delays elapse at the same time, the first one in the list takes priority, and others will not respond. For example, if A and C occur at the same time, the await statement starts *ρ1*, and terminates the await case statement when *ρ1* terminates.

3.2.6 Summary of Esterel-type Instructions

The KEP assembler language (*KASM*) contains 20 Esterel-type instructions for representing 15 Esterel statements directly.

In the KEP assembler, a signal expression (*Sexp*) can be one of the following:

- *S*: signal status (present/absent)
  For example, a KEP EMIT *S* instruction equals an Esterel emit *S* statement.
- PRE(*S*): previous status of signal
  See also Section 3.1.1. For example, a PRESENT PRE(S) THEN instruction equals an Esterel present pre(S) then statement.

Mnemonic, Operands	**Esterel Syntax**	**Notes**
PAR *Prio, startAddr* [, *ID*] PARE *endAddr* JOIN *Prio*	[*p* || *q*]	Fork and join, see also Section 4.2.1. An optional *ID* explicitly specifies the ID of the created thread.
PRIO *Prio* [,*ID*]		Set the priority of the current thread.
[L|T][W]ABORT [*n*,] *S*, *endAddr*	[weak] abort ... when [*n*] *S*	The prefix [L|T] denotes the type of watcher to use, see also Section 4.2.2. **L** : Local Watcher **T** : Thread Watcher ***none*** : general Watcher
[L|T][W]ABORTI *Sexp*, *endAddr*	[weak] abort ... when immediate *Sexp*	
SUSPEND[I] *Sexp*, *endAddr*	**suspend** ... **when** [immediate] *Sexp*	
EXIT *endAddr*,*startAddr*	**trap** *T* **in** **exit** *T* **end trap**	Exit from a trap, the *startAddr* and *endAddr* specifies trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing ||.
PAUSE	**pause**	Wait for a signal. AWAIT TICK is equivalent to PAUSE.
AWAIT [*n*,] *Sexp*	await [*n*] *Sexp*	
AWAIT[I] *Sexp*	await [immediate] *Sexp*	
CAWAITS CAWAIT[I] S, *addr* CAWAITE addr	await case [immediate] *Sexp* do end	Wait for several signals in parallel.
SIGNAL[V] *S*	**signal** *S*[:integer] **in** ... **end**	Initialize a local signal *S*.
EMIT *S* [, {#*data*|*reg*}]	**emit** *S* [(*val*)]	Emit (valued) signal *S*.
SUSTAIN *S* [, {#*data*|*reg*}]	sustain *S* [(*val*)]	Sustain (valued) signal *S*.
PRESENT *S*, *elseAddr*	**present** *S* **then** ... **end**	Jump to *elseAddr* if *S* is absent.
NOTHING	**nothing**	Do nothing.
HALT	halt	Halt the program.
GOTO *addr*	**loop** ... **end loop**	Jump to *addr*.
CALL *addr* RETURN	call *P*	call a procedure, and return from the procedure

Table 3.1: Overview of the KEP Esterel-type instruction set architecture. Esterel kernel statements are shown in **bold**.

- TICK: always present.

Furthermore, a numeral *n* can be one of the following:

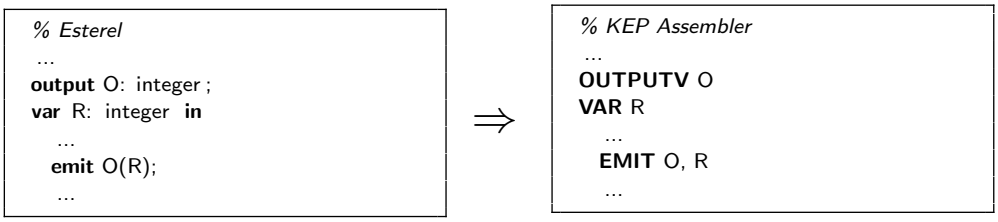


Figure 3.16: Translating the Esterel variable declaration to the KEP instructions.

- *#data*: immediate data
  Due to the limitation of the instruction width, the range of the immediate data is 16 bits. The 32-bit data must be loaded into a register or written to the value of a signal first.

- *reg*: register contents
  It corresponds to the *variable* of Esterel. Figure 3.16 shows an example to illustrate the relation of the KEP *reg* and the Esterel variable.

- ?*S*: value of a signal
  For example, an Esterel R := ?S statement equals a KEP LOAD R, ?S instruction.

- PRE(?*S*): previous value of a signal
  For example, an Esterel R := pre(?S) statement equals a KEP LOAD R, PRE(?S) instruction.

Table 3.1 illustrates the relationship between Esterel statements and the KEP Esterel-type instructions.

3.3 Further Instructions

Except for the above mentioned Esterel-type instructions, the KEP employs some other non-traditional instructions to implement an Esterel module completely. Table 3.2 gives an overview of these special instructions.

- INPUT[V]|OUTPUT[V] *S*/SETV *S*,*reg*
  The INPUT[V]/OUTPUT[V] instructions define the interface of the Esterel module, and the optional V indicates that this is a valued signal.

  To initialize the value of a valued signal, the SETV *S*, *#data*|*reg* instruction is introduced. It is similar to the EMIT *S*, *#data*|*reg* instruction, but it will not affect the presence status of the signal.

Mnemonic, Operands	**Esterel Syntax**	**Notes**
INPUT[V] *S*	input *S* [:integer;]	input definition
OUTPUT[V] *S*	output *S* [:integer;]	output definition
SETV *S*, *#data|reg*		Set the initial value for a valued signal *S*.
EMIT _TICKLEN, *#data|reg*		Set the tick length.
LOAD _COUNT, *n*		Loading data for Count delays.
LOAD _UINT32REG, *#data32*		Loading a 32-bit immediate data to an intermediate register.
CLRC/SETC		Clear/set carry bit
LOAD *reg*, *n*	*reg* := *n*	Load/store register
{SR[C]|SL[C]|NOTR} *reg*		Shift (with carry)/negate
{ADD[C]|SUB[C]|MUL} *reg*, *n*	+, -, *	Add, subtract (with carry), multiply
{ANDR|ORR|NOTR|XORR} *reg*, *n*	and, or, not, xor	Logical operations
CMP[S] *reg*, *n* JW *cond*, *addr*	>, <, <=, >=, <>, =	Compare (with signed), branch conditionally.

Table 3.2: Overview of the KEP non-Esterel-type instruction set architecture.

Figure 3.17 shows an example of translating the interface definition of an Esterel module to the KEP assembler.

- LOAD _COUNT, *n*
  In the KEP assembler syntax listed in Table 3.1, the count delays are expressed directly. For example, the Esterel await 2 S statement can be replaced by the KEP AWAIT 2 S instruction. However, on the machine code level, this instruction will be divided to two instructions. First the LOAD _COUNT, #2 instruction loads the immediate data 2, and then follows the AWAIT S instruction.

  The LOAD _COUNT,*n* is the *counter specified* instruction, which loads a value to a counter register for the following instruction (see also Section 4.4). If there is no counter specified instruction before the preemption or the await instruction, the corresponding counter value will be initialized as 1. Hence, an Esterel statement which contains the count delays expression will be represented by two KEP instructions.

- EMIT _TICKLEN, *#data|reg*
  The KEP has a special valued signal _TICKLEN, which can be set to a certain value by the assembler program to define an upper bound on the number of instructions that may be executed within a logical tick. When the program executes an "EMIT _TICKLEN, *val*" instruction, generally at the beginning of a module, this initializes the Tick Manager (see Section 4.5). When more than *val* instructions have been executed since the last tick delimiting instruction (pause or await), the Tick Manager

```
% Esterel
input A;
output B:=5:integer;
...
signal C:=3:integer in
  ...
end signal;
...
```

```
% KEP Assembler
INPUT A
OUTPUTV B
SETV B, #5
...
SIGNALV C
SETV C, #3
...
```

Figure 3.17: Translating the Esterel interface declaration to the KEP instructions.

considers this a *tick length timing violation*. In this case, the current tick length will be extended automatically until the tick is finished. Furthermore, the timing violation is signaled to the environment via the TickWarn pin.

- LOAD _UINT32REG, *#data32*
  To load an immediate data which exceeds 16-bit (65535), an intermediate register is used. When the program executes a "LOAD _UINT32REG, *#data32*" instruction, the immediate data is loaded to the _UINT32REG register. Then the content of this register can be transferred to other registers, or directly participate in calculations involving other registers.

The KEP ISA also includes some classical arithmetic, logical, and conditional branch operation instructions. Table 3.2 lists these instructions briefly. See also Appendix A for more details.

Note that those classical operations are also optimized for the Esterel language. They provide high efficiency for the signal and data handling. For example, the *?S* expression, which refers to the value of a signal or previous value of a signal, can be used in the arithmetic operations directly. This simplifies the assembler program, and further reduces memory usage and speeds up the execution.

Figure 3.18(a) shows the Esterel COUNT module [19]. It counts the number of occurrences of the input I seen so far, and broadcasts it as the value of a COUNT signal at each new I.

To illustrate the compactness of the KEP assembler, consider the Esterel module COUNT, shown in Figure 3.18(a). This module references the previous value of a valued signal. To generate the assembler, the translation rules shown in Figure 3.9 are employed.

```
% Esterel
module COUNT:
input I;
output COUNT := 0 : integer;
every I do
  emit COUNT(pre(?COUNT) + 1)
end every
end module
```

(a)

```
        % KEP Assembler
        % module COUNT:
        INPUT I
        OUTPUTV O
        VAR REG0
        EMIT _TICKLEN, #10
        SETV O, #0
        AWAIT I
A0:     ABORT I, A1
        LOAD REG0, #1
        ADD REG0, PRE(?COUNT)
        EMIT COUNT, REG0
        HALT
A1:     GOTO A0
```

(b)

Figure 3.18: Translating the Esterel COUNT module (a) to the KEP assembler program (b).

3.4 From Esterel to KEP Assembler

A central problem for compiling Esterel onto the KEP is the need to manage thread priorities during their creation and their further execution. In the KEP setting, this is not merely a question of efficiency, but a question of correct execution.

The technical details of the KEP compiler are described in the thesis of Marian Boldt [26], the following provides a brief overview.

3.4.1 Code Generation for the KEP – The Compiler's Perspective

Boldt has implemented a compiler for the KEP based on the CEC infrastructure [43, 26]. The priority assigned during the creation of a thread and by a particular PRIO instruction is fixed. Due to the non-linear control flow, it is still possible that a given statement may be executed with varying priorities; in principle, the architecture would therefore allow a fully dynamic scheduling. However, we here assume that the given Esterel program can be executed with a statically determined schedule, which requires that there are no cyclic signal dependencies. This is a common restriction, imposed for example by the Esterel v7 [53] and the CEC [43] compilers. Note that there are also Esterel programs that are causally correct (*constructive* [20]), yet cannot be executed with a static schedule and hence cannot be directly translated into KEP assembler using the approach presented here. However, these programs can be transformed into equivalent, acyclic Esterel programs [92], which can then be translated into KEP assembler. Hence, the actual run-time schedule of a concurrent program running on KEP is *static* in the

sense that if two statements that depend on each other, such as the emission of a certain signal and a test for the presence of that signal, are executed in the same logical tick, they are always executed in the same order relative to each other, and the priority of each statement is known in advance. However, the run-time schedule is *dynamic* in the sense that due to the non-linear control flow and the independent advancement of each program counter, it in general cannot be determined in advance which code fragments are executed at each tick. This means that the thread interleaving cannot be implemented with simple jump instructions; a run-time scheduling mechanism is needed that manages the interleaving according to the priority and actual program counter of each active thread.

Regarding a general, systematic scheme for assigning thread priorities, we should first convince ourselves that it is always possible to come up with a set of priorities, which preserves the semantics of the original program. To that end, recall the restriction we imposed on the Esterel program regarding the dependency structure: we forbid cyclic dependencies, there must always exist a static ordering (schedule) of statements, which can be obtained for example from the Event Graph [39] of the program or from the Concurrent Control Flow Graph [46].

The next question then is whether our priority control mechanism always allows to enforce such a schedule. To answer this question, we first consider the start of a logical tick: as observed above, in this case all threads are activated, and if we want to enforce an order in which each thread starts to execute, we can do so by placing a PRIO instruction immediately before the delayed instruction that denotes the start of the tick. In some cases, this may require expanding the delayed statement until the PAUSE statement becomes exposed, as has been the case in the EXAMPLE module, where we had to dismantle the original SUSTAIN R statement.

Next, we consider the situation where the schedule demands a context switch after a tick has been started. As observed above, a thread can yield to another thread by lowering its priority below the priority of that other thread. As the program cannot have cyclic dependencies (scheduling constraints) and schedules are static and finite, we can thus assign priorities to each code segment, where each statement that must precede another instruction must have a higher priority than that other instruction. Then each thread may continuously lower its priority while executing a tick, always yielding to the next thread on the schedule; if this next thread is not active, it follows the next thread, and so on, until all threads have finished their tick. Only towards the end of a tick a thread might have to raise its priority again—not to influence the scheduling of the current tick, but to get the proper thread ordering at the beginning of the next tick.

Regarding the potential increase in code size due to additional PRIO instruction, a conservative estimate—that disregards the ordering from the priorities assigned during thread creation—is that for each dependency in the program, there may be up to two PRIO instructions needed to enforce it. In practice, the fraction of PRIO instructions of the overall KEP instructions tends to be below 1% (see also Chapter 5, Tables 5.4 and 5.8).

3.4.2 EXAMPLE: Code Translation

As an intermediate representation for the compilation of Esterel to KEP assembler, including the thread priority assignment, we use a directed graph structure called Concurrent KEP Assembler Graph (CKAG). Figure 3.19 illustrates the CKAG of the Esterel EXAMPLE module.

In this section, we translate the Esterel EXAMPLE module to the KEP assembler by the above mentioned translation rules, and then we describe the execution flow of the KEP program to explain how it follows the behavior of the original Esterel program.

Up to now, there is still a large room for the optimization of the KEP compiler. Hence, for illustrating the ability of the KEP architecture, which is the main purpose of this thesis, we translate the Esterel program to the KEP assembler manually. Figure 3.20(a) illustrates the Esterel EXAMPLE module, and the corresponding KEP assembler program is shown on Figure 3.20(c). Similar as the Esterel module, a KEP program always starts at the input/output definition. In Figure 3.20(c), line 1 to line 3 defines input signals: S, I; output signals: O; and two inner signals A and R.

The following EMIT _TICKLEN, #20 instruction assigns the tick length of this program as fixed 20 instruction cycles. Here it is an appropriate value for _TICKLEN. Boldt *et al.* have also developed an analysis procedure that automatically performs this *Worst Case Reaction Time* (WCRT) analysis [86, 28]. This analysis has been integrated into a compiler that translates Esterel to KEP assembler and automatically sets _TICKLEN.

For the program body, the generation of the KEP assembler for the EXAMPLE module is in general straightforward; as mentioned before, most common Esterel statements can almost literally be translated into corresponding KEP instructions, and there are direct equivalence rules for the remaining statements. Here, two dismantling rules, *i. e.*, the every translation rule in Figure 3.9 and the sustain translation rule in Figure 3.3, are employed. Note that the KEP assembler includes the SUSTAIN instruction to emit a signal in each instant. However, here we focus on how to implement interleaving correctly. The micro-step of the execution of the SUSTAIN first emits the signal, and then turns the currently executed thread inactive—before the preemption trigger signal is detected. Therefore, the SUSTAIN R has to be divided to its equivalent Esterel kernel statement expression, *i. e.*, the emit R; pause; statements which locate in a loop. Such that priority setting instructions can be inserted to trigger a context switch.

Now the only non-trivial aspect of code generation is the assignment of priorities when executing threads concurrently. To understand how these priorities are assigned, we consider the thread scheduling constraints that must be obeyed to run the EXAMPLE module faithful to Esterel's semantics. As we already noted when first considering this example in Section 3.1.2, the two threads enclosed in the every block can communicate back and forth via the R and A signals, within a logical tick. In this case, there is a *dependency* between the statements that (may) emit these signals (the *dependency sources*) and the statements where these signals are tested (the *dependency sinks*) [56, 33, 125].

First, let us consider the dependency involving R. It is clear which instruction is the dependency source: the EMIT R_{L13} instruction. It is also obvious that the PRESENT R_{L22} instruction is the dependency sink. This allows to formulate the first dependency present in the EXAMPLE module: whenever the EMIT R_{L13} and the PRESENT R_{L22} instructions are executed within the same logical tick, the execution of the EMIT must precede the execution of the PRESENT.

As for the dependency involving A, its dependency source is the EMIT A_{L23} instruction, which appears in the second thread. However, it is less obvious which is the dependency sink, which we have defined as the "statements where these signals are tested." The first thread reacts to A when it has entered the weak abort block, in that case A triggers the abort. Hence, whenever we execute a statement in that block, which in the KEP assembler is the instruction between the $WABORTI_{L11}$ and the label $A2_{L18}$, we should also watch for the presence of A. However, closer inspection yields that as this is a weak abort, it suffices to check at the end of each logical tick whether the block is aborted, that is, whenever we execute a non-instantaneous instruction. In this case, the only non-instantaneous instruction in the abort block is the $PAUSE_{L16}$.

In the EXAMPLE, the first thread is started with priority 3, and the second thread is started with priority 2. If none of the threads would execute any PRIO instruction, the first thread would always be executed before the second thread. This would be sufficient to ensure meeting the first dependency; however, the second dependency would be violated. To remedy this, the first thread must yield to the second thread before it executes the $PAUSE_{L16}$ instruction, to give the second thread a chance to test for the presence of R and to emit an A—and it must regain control again before that $PAUSE_{L16}$ instruction, to perform the abortion in case A is present. This is achieved by the PRIO 1_{L14} and the PRIO 3_{L15} instructions; it is also easy to see that the first dependency is still met.

3.5 Encoding KEP Instructions

As a trade-off between resource usage, speed, and performance, the KEP employs a 36-bit wide instruction word and a 32-bit data bus. The performance of the KEP is predictable. All instructions can be executed in a single instruction cycle. A separate 16-bit wide inner data bus gives a range of up to 65535 for directly expressing signal counts, which indicate how often a signal must occur before for example an abort is triggered. This should suffice for most reasonable requirements.

To restrict the length of a word to 36 bits, some bits are used to present various fields in different instructions. In general, the instruction encoding method depends on the form of the KEP assembler.

The parameters of the KEP instructions include the following:

- Operation code (*Op*)
  The KEP uses 5 bits for its major operation code field. Depending on instruction types, additional 6 bits may be used as the extended field of the operation code, *i. e.*, *ExtOp*.

  However, there are two exceptions for the *Op* code. The *Op* could just use 4 bits for two instruction forms. One is the instruction that handles 32-bit data. To encode the 32-bit immediate data, the width of the *Op* code is reduced to 4 bits. A similar encoding method is also used for the instruction that handles two address code.

- Program address code (*Addr*)
  The program address code takes 16 bits. Hence, the maximum address range of the current KEP is 65536 instruction words, and the word width is 36 bits. For an executable KEP program, its address begins from 0, where generally the EMIT _TICKLEN, *#data* instruction is located, which defines the tick length of this program.

- Signal code (*Signal*)
  Although the signal code uses 9 bits, it just presents 256 signals because the lowest bit is used as the pre extended bit of the original signal. For example, assuming the code of signal A is 000000110b, then the code of pre(A) is 000000111b. In this way, all of the signals' current or previous states are mapped to a set of signals and can be accessed directly. A similar method is used for the values of signals.

  From the view of the programmer, the maximum utilizable signal number is 255. The code 000000000b is used for the special pure signal TICK. Furthermore, the valued signal _TICKLEN uses the same code. Note that since the TICK can only be tested, and _TICKLEN can only be emitted, they will not with conflict each other.

- Register code (*Reg*)
  The register code has 10 bits. Similar to the signal code, the code 0000000000b is reserved for the counter register, which is used in the counter specified instruction, *i. e.*, the LOAD _COUNT, *val*. Furthermore, the 0000000001b is also used by the _UINT32REG register, which is used in the 32-bit immediate data specified instruction, *i. e.*, the LOAD _UINT32REG,*#data32*. Hence, up to 1022 registers can be used.

- Thread priority value code (*Prio*)
  The priority value of a thread ranges from 0 to 127, *i. e.*, it takes 7 bits. Value 0 is always assigned to the thread 0 (the initial thread). For any other thread, the priority value could be from 1 to 127.

- Watcher identification code (*WatcherID*)
  The width of the watcher identification code is 6 bits, *i. e.*, there are up to 64 watchers. Note that the configuration instructions for different watcher types use different *Op* codes. Hence, the watcher ids of different watcher types are independent, *i. e.*, the 6-bit *WatcherID* can express 64 Watchers and 64 Local Watchers

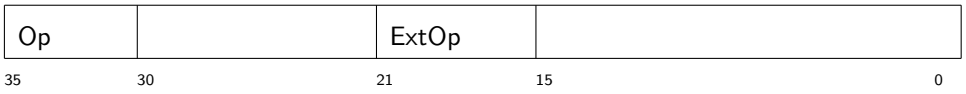
at once. Incidentally, the Thread Watcher is operated by the current executing thread, it is unnecessary to encode them.

- Thread identification code (*ThreadID*)
  The thread identification code allows instructions to create or modify the priority value of a specific thread. It is 7 bits wide, hence, the KEP can handle 128 threads directly. The code of the initial thread is 0000000b.

- Immediate data code (*#data*, *#data32*) code
  Most of the KEP data operations can directly handle 16-bit data, *i. e.*, immediate data. However, to deal with data that are larger than 65535, 32-bit data should be loaded to the _UINT32REG register, and then be transferred to any register, or be calculated with other data, content of a register, and so on.

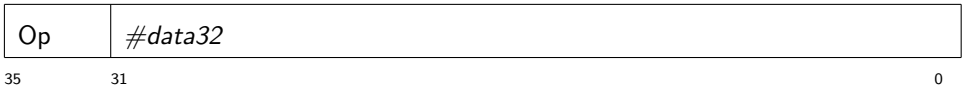
The various combinations provide a very flexible instruction format of the aforementioned codes. The KEP instruction encoding is illustrated below, see Appendix A for more information.

- OP



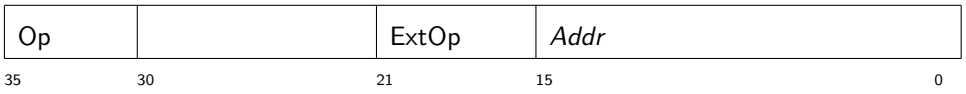
  e. g., NOTHING, HALT

- OP *#data32*



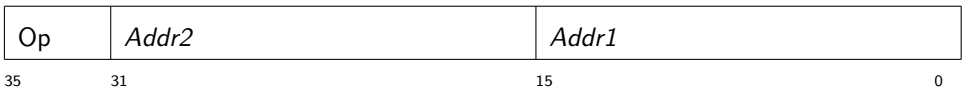
  e. g., DEF32 *#data32*

- OP *Addr*



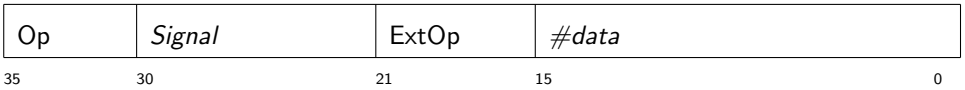
  e. g., GOTO *Addr*

- OP *Addr1*, *Addr2*



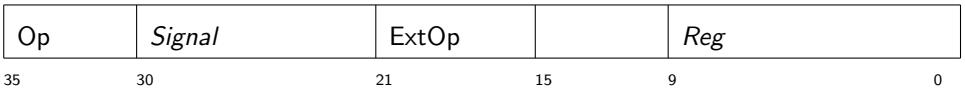
  EXIT *EndAddr*, *StartAddr*

- OP *Signal*, *#data*



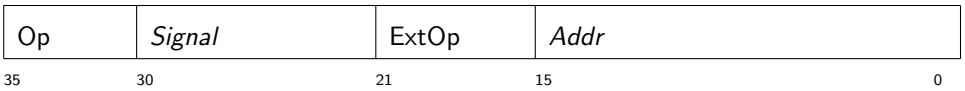
e. g., EMIT *S*, *#data*

- OP *Signal*, *Reg*



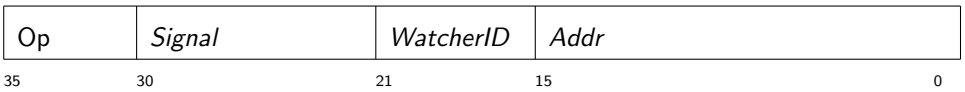
e. g., EMIT *S*, *#Reg*

- OP *Signal*, *Addr*



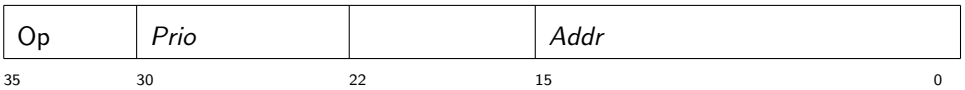
e. g., CAWAIT *S*, *Addr*

- OP *Signal*, *Addr*, *WatcherID*



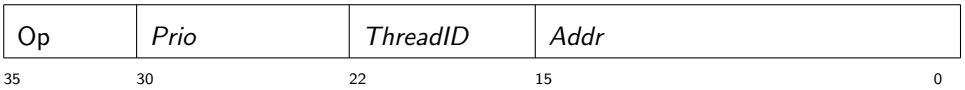
e. g., ABORT *S*, *Addr*, *[WatcherID]*

- OP *Prio*, *Addr*



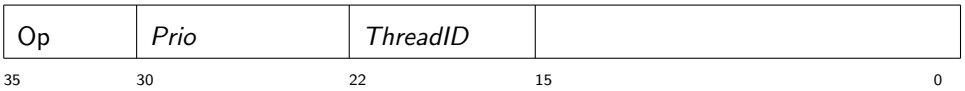
e. g., PARE *endAddr* [,*Prio*]

- OP *Prio*, *Addr*, *ThreadID*



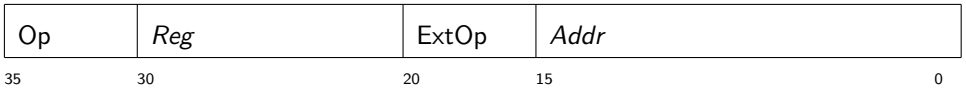
e. g., PAR *Prio*, *startAddr*, *[ThreadID]*

- OP *Prio*, *ThreadID*



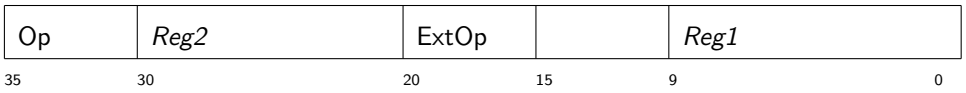
e. g., PRIO *Prio* [,*startAddr*]

- OP *Reg*, *#data*



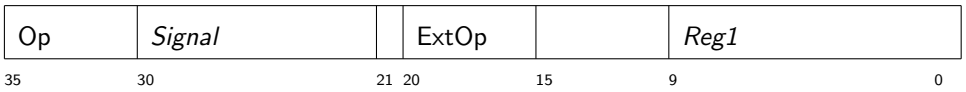
e. g., ADD *Reg*, *#data*

- OP *Reg1*, *Reg2*



e. g., SUB *Reg1*, *Reg2*

- OP *Reg*, *Signal*



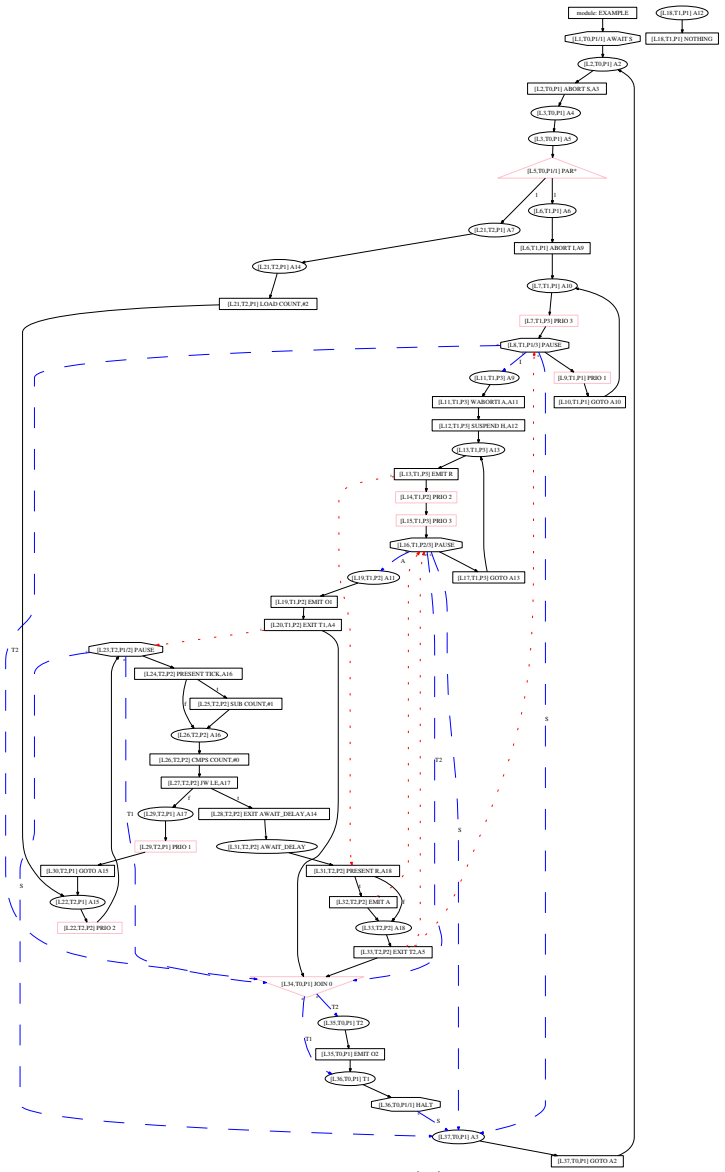
e. g., MUL *Reg1*, *?S*

3.6 Summary

The design of the Instruction Set Architecture (ISA) of the KEP reflects a tradeoff between rich functionality, powerful control flow, and system cost. The most remarkable property of the KEP ISA is that it is the first instruction set which directly supports all of the Esterel kernel statements. Hence, all of the Esterel control statements can be expressed in a very efficient way. Some other frequently used non-kernel Esterel statements/structures are also directly supported. Besides, the provision of context-dependent preemption handling instructions allows efficient resource usage without limiting generality. Furthermore, the design also considers the Esterel optimized data path, which maps Esterel data handling to traditional arithmetic functions. Finally, the KEP instructions employ various address range definition strategies to define the Esterel control structures. These strategies make the KEP instructions support arbitrary control structures nests, which is also a characteristic of the Esterel language.

```
% Esterel
module EXAMPLE:
input S, I, H;
output O1, O2;
signal A,R in
every S do
  trap T1 in                    % Thr0
    trap T2 in                  % Thr0
      [ await I;                % Thr1
        weak abort              % Thr1
          suspend               % Thr1
            sustain R;          % Thr1
        when H;                 % Thr1
        when immediate A;       % Thr1
        emit O1;                % Thr1
        exit T1;                % Thr1
      ||
        await 2 tick;           % Thr2
        present R then          % Thr2
          emit A;               % Thr2
        end present;            % Thr2
        exit T2; ];             % Thr2
    end trap;                   % Thr0
    emit O2;                    % Thr0
  end trap;                     % Thr0
end every;                      % Thr0
end signal;
end module
```

(a)



(b)

Figure 3.19: The EXAMPLE: (a) Esterel; (b)Concurrent KEP Assembler Graph (CKAG), where rectangles are transient nodes, octagons are delay nodes, and triangles are fork/join nodes.

```
% Esterel
module EXAMPLE:
input S, I, H;
output O1, O2;
signal A,R in
every S do
  trap T1 in                    % Thr0
    trap T2 in                  % Thr0
      [ await I;                % Thr1
        weak abort              % Thr1
          suspend               % Thr1
            sustain R;          % Thr1
        when H;                 % Thr1
      when immediate A;         % Thr1
      emit O1;                  % Thr1
      exit T1;                  % Thr1
    ||
      await 2 tick;             % Thr2
      present R then            % Thr2
        emit A;                 % Thr2
      end present;              % Thr2
      exit T2; ];               % Thr2
    end trap;                   % Thr0
    emit O2;                    % Thr0
  end trap;                     % Thr0
end every;                      % Thr0
end signal;
end module
```

(a)

```
                    % KEP Assembler
                    % (by compiler)
                    % module EXAMPLE:
[L01,T0]            AWAIT S
[L02,T0] A2:        ABORT S,A3
[L03,T0] A4:A5:     PAR 1,A6,1
[L04,T0]            PAR 1,A7,2
[L05,T0]            PARE A8,1
[L06,T1] A6:        ABORT I,A9
[L07,T1] A10:       PRIO 3
[L08,T1]            PAUSE
[L09,T1]            PRIO 1
[L10,T1]            GOTO A10
[L11,T1] A9:        WABORTI A,A11
[L12,T1]            SUSPEND H,A12
[L13,T1] A13:       EMIT R
[L14,T1]            PRIO 2
[L15,T1]            PRIO 3
[L16,T1]            PAUSE
[L17,T1]            GOTO A13
[L18,T1] A12:       NOTHING
[L19,T1] A11:       EMIT O1
[L20,T1]            EXIT T1,A4
[L21,T2] A7:A14:    LOAD COUNT,#2
[L22,T2] A15:       PRIO 2
[L23,T2]            PAUSE
[L24,T2]            PRESENT TICK,A16
[L25,T2]            SUB COUNT,#1
[L26,T2] A16:       CMPS COUNT,#0
[L27,T2]            JW LE,A17
[L28,T2]            EXIT AWAIT_DELAY,A14
[L29,T2] A17:       PRIO 1
[L30,T2]            GOTO A15
[L31,T2] AWAIT_DELAY:
                    PRESENT R,A18
[L32,T2]            EMIT A
[L33,T2] A18:       EXIT T2,A5
[L34,T0] A8:        JOIN 0
[L35,T0] T2:        EMIT O2
[L36,T0] T1:        HALT
[L37,T0] A3:        GOTO A2
```

(b)

```
                    % KEP Assembler
                    % (manually)
                    % module EXAMPLE:
[L01]               INPUT S, I, H
[L02]               OUTPUT O1, O2
[L03]               SIGNAL A, R
[L04]               EMIT _TICKLEN, #20
[L05,T0]            AWAIT S
[L06,T0] A0:        ABORT S, A1
[L07,T0] T1S: T2S:  PAR 3, P1
[L08,T0]            PAR 2, P2
[L09,T0]            PARE P3
[L10,T1] P1:        AWAIT I
[L11,T1]            WABORTI A, A2
[L12,T1]            SUSPEND H, A4
[L13,T1] A3:        EMIT R
[L14,T1]            PRIO 1
[L15,T1]            PRIO 3
[L16,T1]            PAUSE
[L17,T1]            GOTO A3
[L18,T1] A4: A2:    EMIT O1
[L19,T1]            EXIT T1E,T1S
[L20,T2] P2:        LOAD _COUNT, #2
[L21,T2]            AWAIT TICK
[L22,T2]            PRESENT R, A5
[L23,T2]            EMIT A
[L24,T2] A5:        EXIT T2E,T2S
[L25,T0] P3:        JOIN 0
[L26,T0] T2E:       EMIT O2
[L27,T0] T1E:       HALT
[L28,T0] A1:        GOTO A0
```

(c)

Figure 3.20: Translating the Esterel EXAMPLE module (a) to the KEP assembler program (b) by KEP compiler, or manually (c).

Chapter 4

The KEP Architecture

Chapter 3 has presented the KEP instruction set architecture. Obviously, such an ISA facilitates the handling of Esterel statements/structures. However, it raises the following question: how can the KEP implement the behavior of these instructions? And how does the KEP deal with them in a way that exactly follows the original Esterel semantics?

In this chapter, we present the KEP's multi-threaded reactive architecture. First, an overview of the KEP architecture is given in Section 4.1. The Reactive Core of the KEP, which deals with all of the KEP's Esterel-type instructions, is introduced in Section 4.2. The Esterel signal, which is used to communicate with the environment and within the program, is handled by the Interface Block of the KEP, see Section 4.3. Section 4.4 illustrates the Data Handling block, and the Tick Manager of the KEP is discussed in Section 4.5. Finally, an example presents the cooperation of the KEP blocks in Section 4.6.

4.1 The KEP Architecture Overview

The top-level I/O signals of the KEP are illustrated in Figure 4.1. The Sinout signals indicate the presence of input/output signals, and the SDir is used by the environment to select the read or write signal status. To access the value of a valued signal, the environment controls the SDataID and the SDataWR pin, then the content can be read/written through the SData bus. In the KEP, every signal can be a valued signal.

ROMData and ROMAddr are data and address buses for the instruction memory. An external clock must be connected to the OscClk pin; we use T_{osc} to denote the rate of that clock. The InstrClk indicates the instruction clock; each instruction cycle lasts three OscClk cycles. The Tick pin indicates the logical tick of Esterel, and the TickWarn pin is set high when a timing violation is detected, see also Section 4.5.

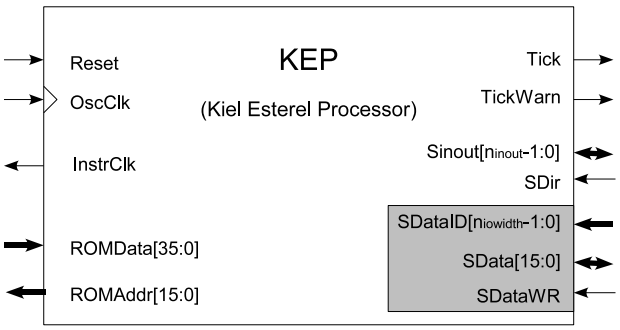


Figure 4.1: The interface connections of the KEP.

To reset the KEP, the Reset pin should be connected to high ('1') for at least one oscillator clock cycle. During this period, the Reset Event is automatically generated, and all outputs and inner signals will be set to '0' (absent).

The architecture of the KEP is inspired by the three layers that constitute a reactive program [23], *i. e.*:

1. An *interface* with the environment, which handles input reception and output production.

2. A *reactive kernel*, which decides what computations and what outputs must be generated when it reacts to inputs.

3. A *data handling* layer, which performs classical computations.

We use the above-mentioned layers as KEP's architectural levels. In the KEP, a Reactive Core accomplishes the tasks of the reactive kernel. The functions of the interface layer are mapped to the Interface Block. And the classical computations are performed by the Data Handling Block, which also employs optimized data path for the efficient execution of the Esterel program. All of these blocks are semi-custom (scalable).

Figure 4.2 shows an overview of the KEP architecture. Light gray blocks indicate the reactive core of the KEP; the interface block is marked as dark gray; and the traditional ALU (Arithmetic Logical Unit) part of a processor is compose by translucent blocks. Besides, there are some other special peripheral elements. For example, the Tick Manager, described further in Section 4.5, provides the fixed length for a logical tick and also detects tick length timing violations.

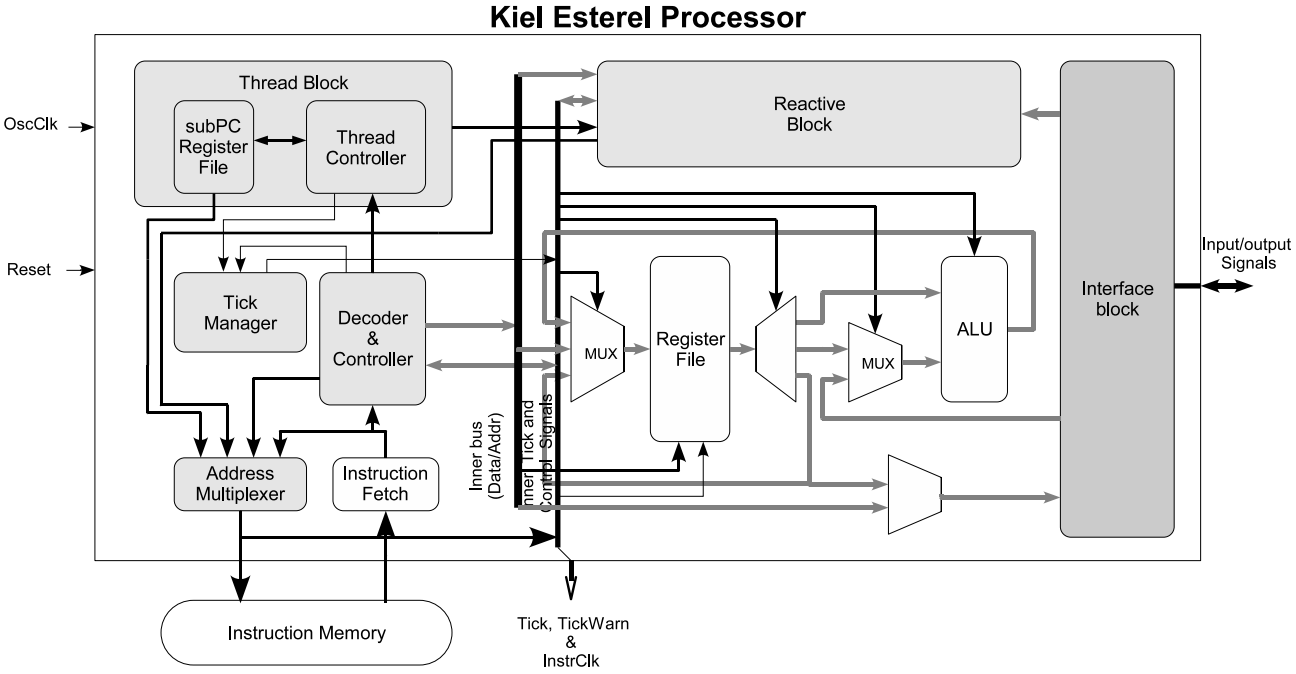


Figure 4.2: Overview of the architecture of the KEP.

4.2 The Reactive Core

The Reactive Core is the core of the reactive processor. The implementation of Esterel's reactive statements relies on the cooperation of KEP's Reactive Block, Thread Block, and Decoder & Controller, which together form the Reactive Core.

The main difficulty of the Reactive Core design is how to implement the Esterel preemption and concurrency. Our strategy is based on the following Esterel essentials:

- Regarding the *preemption* handling mechanism, a preemption block must respond to its trigger signal while the program counter (PC) is within its body. The KEP provides a configurable module, which watches a certain address range and outputs its control signals. The watched address range corresponds to a preemption body. Concurrent threads in a preemption (or nested preemption) body can be handled properly; since executions of these threads are sequential, each thread can be controlled by the preemption once the thread is fetched.

- In Esterel, threads fork on a || parallel statement, and terminate when all their branches have terminated. Therefore, Esterel threads can be simulated by multi-threaded structures. The fork point corresponds to the thread configuration instructions, and the thread waits for the termination of all branched threads at the join point. The program counter is watched to determine the termination of

branched threads. An advantage of this approach compared to the sequential processor implementation is that the dynamic scheduling strategy avoids the recording and testing of thread states, which is required for multi-way branch in the sequential processor implementation. The context switch is provided by the processor hardware, which eliminates overhead and effectively results in a zero-cycle context switch [42].

4.2.1 The Thread Block

The KEP employs a *multi-threaded architecture*, where each thread has an independent program counter (PC) and threads are scheduled according to their activation status and a dynamically changing priority. Hence, an Esterel thread can be mapped to a KEP thread directly.

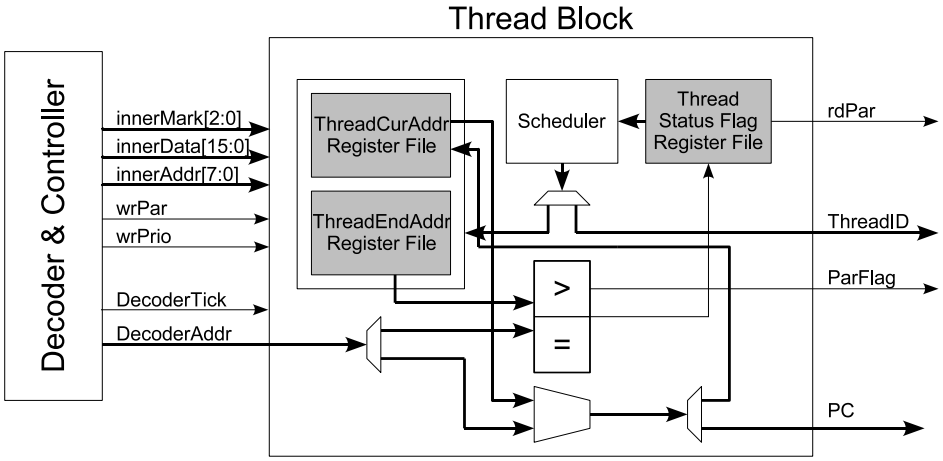


Figure 4.3: Architecture of the Thread Block.

Figure 4.3 shows the architecture of the Thread Block, which is responsible for managing the threads. For each instruction cycle, it decides which thread to execute next, based on the current status of each thread. As mentioned before, context switches do not cost any extra clock cycles, and the lean design of the Thread Block still permits a comparatively high instruction frequency.

A KEP thread goes through three phases, *i. e.*, creation, execution, and termination, which are controlled by the Thread Block.

- Thread Creation
  A complete thread creation requires on two PAR instructions, or a PAR and a PARE instruction. When a PAR instruction creates a thread, the start address

parameter of this thread will be stored as the ThreadCurAddr. Note that the start address of the next concurrent thread is located behind this thread, so the ThreadCurAddr of the next thread is also stored as the ThreadEndAddr of this thread. Furthermore, the current $ThreadID$ is also restored to identify the parent of a thread. Therewith, the PARE instruction ends the configuration of the last concurrent thread. The value of the parent thread address register points to the end of the last incoming thread, where the JOIN instruction is located, which waits for all its branch threads to terminate.

In addition to the ThreadCurAddr and ThreadEndAddr, the Thread Block uses two status flags to keep track of each thread's status. The ThreadEnable flag indicates whether the thread is still running (*enabled*) or already terminated (*disabled*), and the ThreadActive flag indicates whether the thread should still be scheduled within the current logical tick (is *active*) or not (*inactive*). After a thread is created, those two flags are both set to '1', which means the thread is ready to be scheduled. However, the Scheduler will not become active until all of the thread configurations are finished. After the PARE instruction is executed, the activated threads can be invoked by the priority-based preemptive scheduling mechanism.

Figure 4.4 shows the procedure for creating a KEP thread.

- Thread execution
  The execution status of a thread is illustrated in Figure 4.5, using the SyncChart formalism [2]. At the beginning of each instruction cycle, the Scheduler inspects all active threads. If there are multiple active threads, the Scheduler executes the thread with the highest priority; if several active threads have the highest priority, the Scheduler executes the thread that has the highest *id*. To reduce its path length, the Scheduler employs a tree structure [113, 59]. If a thread is scheduled, its $ThreadCurAddr$ parameter, *i. e.*, the value of its sub program counter, will be mapped as the value of the program counter of the processor. The Thread Block is responsible for managing threads, as illustrated in Figure 4.6.

  Once a non-instantaneous instruction is executed, such as an AWAIT or a HALT, the ThreadActive flag will be set to '0', meaning that this thread will not be scheduled any more in the current tick. If all threads are inactive, the current tick is finished. At the start of the next tick, the ThreadActive flags of all enabled threads will again be set to '1'. Figure 4.7 indicates above mentioned processes.

- Thread termination
  A thread termination could be caused by three reasons.

  1. The thread finishes all statements in its body, in which case the expected fetch address will equal the ThreadEndAddr associated with that thread (normal termination).
  2. The expected fetch address is greater than the ThreadEndAddr, for example in such cases when the thread is aborted by an enclosing abortion or exceptions

```
% Definition of Parameters
% Instr
  % Currently executed instruction.
% Instr.Op
  % Operation code of the current instruction.
% Instr.Addr
  % Address code of the current instruction.
% Instr.Prio
  % Priority code of the current instruction.
% Instr.ThreadID
  % Thread ID number code of the current instruction.
% LastInstr
  % The previous executed instruction.
% ThreadScheduler.Enable
  % The Scheduler of the Thread Block will
  % be disabled when the status of this parameter is false.
% ThreadConfigureBegin
  % It is true when the current executed instruction
  % is the first PAR instruction of a PAR/PARE group.
```

```
% For the Thread Block
if Instr.Op = PAR then
  % Whenever the PAR instruction is executed
  ThreadScheduler.Enable = false
    % Forbidding the scheduling during the creation of threads.
  if LastOp = PAR then
    % If the current executed PAR instruction is not the first
    % PAR instruction of a PAR/PARE group.
    Thread_{LastInstr.ThreadID}.ThreadEndAddr = Instr.Addr
      % The end address of the last branch thread is configured as
      % the start address of the current created branch thread
  end if
  Thread_{Instr.ThreadID}.ThreadCurAddr = Instr.Addr
    % The Program Counter (PC) of the current created branch thread is configured
    % as the Instr.Addr, which points to the beginning of this branch thread.
  Thread_{Instr.ThreadID}.Priority = Instr.Prio
    % Configuring the priority value of this branch thread.
  Thread_{Instr.ThreadID}.ParentThread = ThreadID
    % Currently executed thread is the parent thread of
    % the currently created branch thread.
  Thread_{Instr.ThreadID}.Enable = true
  Thread_{Instr.ThreadID}.Active = true
    % The statuses of a created thread are set as enabled and active.
  LastInstr = Instr
    % Recording the corresponding information for the next PAR/PARE instruction.
elsif Instr.Op = PARE then
  % The end of the PAR/PARE group.
  Thread_{LastInstr.ThreadID}.ThreadEndAddr = Instr.CurAddr
  Thread_{ThreadID}.CurAddr = Instr.Addr
    % Setting the Program Counter (PC) of the current executed thread
    % to point to the join point of these incoming sub threads.
  if Instr.Prio > 0 then
    % The priority of the current executed thread will be configured
    % when the PARE instruction carries a non-zero priority value; or else,
    % it will be kept.
    Thread_{ThreadID}.Priority = Instr.Prio
  end if
  ThreadScheduler.Enable = true
    % Enabling the scheduling when the creation of branch threads is finished.
end if
```

Figure 4.4: Algorithm for creating KEP threads.

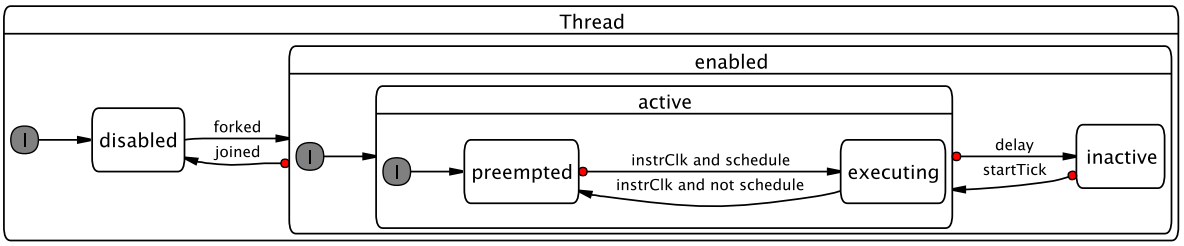


Figure 4.5: Execution status of a single thread.

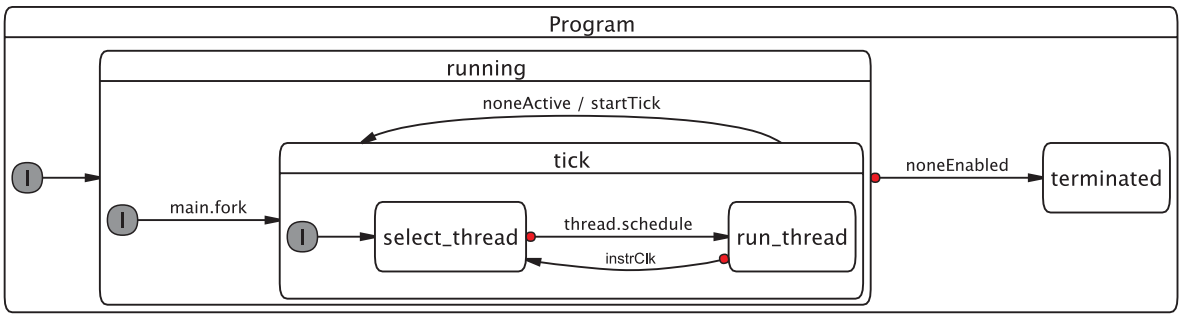


Figure 4.6: The status of the whole program, as managed by the Thread Block.

```
% Definition of Parameters
% Tick.Start
  % The moment when the status of the Tick signal
  % changes from inactive to active.
% Tick.Finish
  % It denotes whether all of threads are inactive
  % or not. Note due to the fix tick length feature
  % of the KEP, the Tick signal may not be
  % set as inactive immediately when this event
  % happens, because the KEP may idle for some
  % instruction cycles to meet the _TICKLEN
  % constrain. See also Section 4.5.
% PC
  % Program Counter. The PC value also equals
  % the value of ROMAddr.
% ThreadID
  % The ID of the current executed thread. It
  % is generated by the Scheduler.
% N_Thread
  % The total thread number of this processor.
```

```
% For the Thread Block
if Tick.Start = true then
  % All enabled threads will be set as active
  % when a new tick starts.
  forall n ∈ N_Thread do
    Thread_n.Active = Thread_n.Enable
  end
end if

if ThreadScheduler.Enable = true
  % Mapping the sub PC of the selected executed
  % thread as the PC of the processor.
  PC = Thread_ThreadID.ThreadCurAddr
end if

Tick.Finish = true
forall n ∈ N_Thread do
  if Thread_n.Active = true
    % The Tick.Finish is false when any
    % thread is set as active.
    Tick.Finish = false
  end if
end
```

Figure 4.7: Algorithm for running threads.

(sudden termination). In this scenario, the TerminationType parameter of this thread will be set to *sudden* to indicate that the thread is terminated by an abortion/exception.

3. A thread could be terminated by its parent thread at the join point. This mechanism is employed for handling Esterel exceptions efficiently, see also Section 4.2.2.

An exception to this mechanism is that the program invokes a function, *i. e.*, executes a CALL instruction and does not return. Since the program of the function could be located at any place of the memory (outside of the address range of the current thread), any expected address is allowed and will not cause the termination of the thread.

At the join point, the parent thread will evaluate the status of all its child threads. If any child thread is enabled, the parent will still wait unless an exception terminates the child threads. Else, the control will go through if all its sub threads are terminated normally, or respond an abortion/exception if any of its sub threads are preempted.

Figures 4.8 and 4.9 illustrate the procedure of managing thread status of the KEP.

The Thread Block and other blocks tightly interact with each other through several control signals to ensure the proper handling of arbitrary preemption and concurrency control flow. See the following Section 4.2.2.

4.2.2 The Reactive Block

The Reactive Block contains a Preemption Element, which has a configurable number of Watcher modules that are responsible for implementing the preemption operations. The Reactive Block also contains the AWAIT Element, which implements signal awaiting, and the PRESENT Element, which tests signal presence.

Handling Signal Tests

The implementation of the PRESENT statement depends on the PRESENT Element shown in Figure 4.10. The basic form of the Esterel PRESENT statement checks for one signal expression and performs binary branching. We map the signal code (*SignalCode*)[1] of the instruction to the selected-signal-coder port of the PRESENT Element. The status of the selected signal is immediately presented on a signal which is named rdPRESENT.

Considering the process of instruction execution, the selected signal's code is directly mapped to the selected-signal-code port when the KEP fetches a PRESENT instruction.

[1] To avoid confusion, from now on the code of a parameter will be named as *parameterCode* explicitly. For example, the Signal encode, which is named as *Signal* in Section 3.5, will be called as *SignalCode*.

```
% Definition of Parameters
% DecoderAddr
  % The desired program address generated by the Decoder & Controller. It is
  % transferred to the Thread Block for further being judged to achieve the PC.
% TickFlag
  % It comes from the Decoder & Controller, denoting that the current executed
  % instruction causes a delay when it equals false.
% CallFlag
  % It denotes whether the KEP is executing a procedure or not. The status
  % of the current executed thread will be kept when this signal is true.
% StackAddr
  % It stores the address where the KEP calls the procedure.
% ChildThreadInactive, ChildThreadDisable
  % These two signals indicate the status of all child threads of the current
  % executed thread. At the join point, if all child threads are inactive or
  % disable, the ChildThreadInactive or the ChildThreadDisable will be set
  % as true. They are utilized by the Decoder & Controller.
% ChildThreadSuddenTermination
  % Similar as above, if any child thread is terminated by an abotion/exception,
  % this signal will be set as true.
```

```
% For the Thread Block
% Handling the statuses of threads
if CallFlag = false then
  % When the KEP is not executing a procedure
  if DecoderAddr > Thread_ThreadID.EndAddr then
    % The target address exceeds the address range of the current executed
    % thread, which implies an abortion or an exception is active,
    Thread_ThreadID.Enable = false
    Thread_ThreadID.Active = false
    % The thread is terminated immediately.
    Thread_ThreadID.TerminationType = sudden
    % Denoting the thread is terminated by an abortion or an exception.
  elsif DecoderAddr = Thread_ThreadID.EndAddr then
    % The target address equals the address range of the current executed
    % thread, which implies the execution of this thread is finished.
    Thread_ThreadID.Enable = false
    Thread_ThreadID.Active = false
    % The thread is terminated immediately.
    Thread_ThreadID.TerminationType = normal
    % Denoting the thread is terminated normally.
  elsif TickFlag = false then
    % A delay instruction (e. g., AWAIT) is executed
    Thread_ThreadID.Active = false
    % The thread is set as inactive in this tick.
  else
    Thread_ThreadID.ThreadCurAddr = DecoderAddr
    % The execution of this thread will be continue.
  end if
else
  % When the KEP is executing a procedure
  Thread_ThreadID.ThreadCurAddr = DecoderAddr
end if
```

Figure 4.8: Algorithm for managing thread status (1).

```
% For the Decoder & Controller
if Instr.Op = CALL then
  CallFlag = true
  StackAddr = PC + 1
  % Setting the CallFlag when the KEP
  % calls a function, and pushing the
  % address of the next instruction
  % for future returning.
  DecoderAddr = Instr.Addr
  % Going to the beginning of
  % the procedure program.
end if
if Instr.Op = RETURN then
  CallFlag = false
  % Clearing the CallFlag when
  % the procedure program returns.
  DecoderAddr = StackAddr
  % Returning where the main program
  % calls the procedure.
end if
```

```
% For the Thread Block
% Scanning all incoming threads at the join point
ChildThreadDisable = true
ChildThreadInactive = true
SuddenTermination = false
forall n ∈ N_thread do
  if (Thread_n.ParentThread = ThreadID)
    % This thread is a child thread of the current
    % executed thread.
    if (Thread_n.Enable = true) then
      ChildThreadDisable = false
      % Setting ChildThreadDisable as false if
      % any of its branch threads does not terminate.
    end if
    if (Thread_n.Active = true) then
      ChildThreadInactive = false
      % Setting ChildThreadInactive as false if
      % any of its branch threads does not inactive.
    end if
    if Thread_n.TerminationType = sudden then
      ChildThreadSuddenTermination = true
      % Setting SuddenTermination as true if
      % any of its branch threads was terminated by
      % an abortion/exception.
    end if
end
```

Figure 4.9: Algorithm for managing thread status (2).

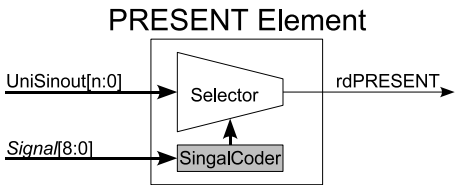


Figure 4.10: Architecture of the Present Element.

When the Decoder & Controller executes the instruction, it just needs to test the rdPRESENT signal to decide whether to branch or not. This feature is also used to implement some other instructions, *i. e.*, AWAITI *S* (await immediate *S*). See also Section 4.2.2.

Figure 4.11 shows the behavior of the PRESENT Element.

Handling Delay

The AWAIT Cell is the basic component for handling delay instructions. Except for the creation/termination of an Esterel thread, every Esterel thread will start its execution from a delay instruction and finally be blocked by the same or another delay instruction.

```
% Definition of Parameters
% UniSignal
  % It is the Unicode Signal. In the KEP, all
  % the input/output and inner signals are
  % recoded into a Unicode Signals bus, which
  % can be accessed by inner blocks and
  % elements directly.
% Instr.SignalCode
  % The signal code of any corresponding
  % instruction. e. g., the PRESENT S,
  % AWAIT S, etc.
```

```
% For the Present Element
% Mapping the status of the selected signal
% to the rdPRESENT port.
rdPRESENT = UniSignal(Instr.SignalCode)
```

```
% For the Decoder & Controller
if Instr.Op = PRESENT then
  % Executing the PRESENT instruction
  if rdPRESENT = true then
    % When the signal to be tested is present
    DecoderAddr = PC + 1
    % Going through
  else
    % When the signal to be tested is absent
    DecoderAddr = Instr.Addr
    % Going to the address of the else branch.
  end if
end if
TickFlag = true
% The PRESENT instruction will not
% cause a delay
```

Figure 4.11: Algorithm for handling signal test.

Hence, one AWAIT Cell per thread suffices. The AWAIT Cell can be indexed by the ID of the currently executed thread.

When the Reactive Core executes an AWAIT or PAUSE instruction for the first time, it configures the AWAIT Cell of the AWAIT Element via inner buses. At the end of the current tick, the Decoder & Controller waits for the terminating signal from the AWAIT Element. If the Reactive Core responds to an active abortion before the current AWAIT instruction terminates, the AWAIT instruction will be cancelled.

However, how does the Reactive Core 'know' whether the currently fetched delay instruction is executed to configure the AWAIT Element or test a sensitive signal to terminate? Considering the KEP ISA, a thread will be blocked by a delay statement, and may terminate in later instants. Hence, the address of the delay statement can be used to decide the execution method of the delay statement.

When the Reactive Core executes an AWAIT or PAUSE instruction, it compares the current address and the stored address of the delay statement of the current thread. The difference of these values implies how the current execution should configure the AWAIT Element. So the information of this delay instruction, *e. g.*, its address and the counter value, will be written to the corresponding AWAIT Cell. On the contrary, if these values are the same, the execution will test the signal and then decide whether the delay should be terminated or not.

The AWAIT Cell contains two parameter registers. One register stores the counter value, and the other stores the address of the last executed delay instruction. Note that the sensitive signal code is contained in the code of the instruction, which can be decoded when the instruction is fetched. Hence, it is unnecessary to recode it. If the watched signal is '1', the counter value will be decremented. When the counter value equals zero, the AWAIT Element sets the rdAWAIT signal to '1' to indicate the termination of AWAIT.

When a delay instruction terminates, the Reactive Core will initialize the corresponding AWAIT Cell, *i. e.*, write the address register of the current AWAIT Cell as "0", for the execution of the next delay instruction. The behavior is similar when the delay instruction is cancelled.

```
% Esterel
loop
  trap T in
    [ pause;
      exit T;
    ||
      await 2 tick;
      emit O;
    ];
  end trap;
end loop;
```

(a)

```
                    % KEP Assembler
[L00,T0]            EMIT _TICKLEN,#10
[L01,T0] A0: TS:    PAR 1, P1
[L02,T0]            PAR 1, P2
[L03,T0]            PARE P3
[L04,T0] P1:        PAUSE
[L05,T0]            EXIT TE, TS
[L06,T0] P2:        LOAD _COUNT, #2
[L07,T0]            AWAIT TICK
[L08,T0]            EMIT O
[L09,T0] P3:        JOIN
[L10,T0] TE:        GOTO A0
```

(b)

Figure 4.12: Executing an AWAIT instruction twice in a tick.

There are some further conditions, as illustrated by the example in Figure 4.12. In this module, thread 2 configures its AWAIT Cell via AWAIT TICK_{L07} at the first tick. In the second tick, thread 1 throws an exception. At the join point, thread 1 and 2 are both terminated. The loop causes control to re-enter both threads again. Note that the address of the delay statement, which is stored in the AWAIT Cell of thread 2, is not initialized. The Reactive Core could now wrongly assume that the signal be tested. However, this is not the case, as the await 2 tick statement is just re-entered, and it is not an immediate AWAIT. Obviously, a similar problem could also happen for example when a processor is reset during its execution and then enters the program again.

To avoid these problems, the KEP initializes the corresponding AWAIT Cell when a thread is created or the processor is reset. Figure 4.13 shows the process of the Decoder & Controller which handles the AWAIT instruction. Figure 4.14 illustrates the mechanism of the AWAIT Element. The architecture of the AWAIT Element is shown in Figure 4.15. In a previous version of the KEP, the Tick signal is sampled for deciding how to handle a fetched AWAIT instruction [88], *i. e.*, to configure the AWAIT Cell or count down its counter. In the current KEP version, this is unnecessary because the choice is made according to the address information of the AWAIT instruction.

The implementation of parallel await statements, *i. e.*, the Esterel await case statements, does not depend on some special components (as had been the case in the previous KEP versions [88, 89] and other reactive processors [38]). Instead, the Decoder & Controller can handle this structure in an efficient way.

```
% Definition of Parameters
% wrAWAIT, rdAWAIT
  % The AWAIT Element uses rdAWAIT signal to denote the termination of
  % a delay instruction. And the Decoder & Controller employs the wrAWAIT
  % signal to tell the AWAIT Element to configure the AWAIT Element.
% PreemptionElement.Type, PreemptionElement.Addr
  % The triggered preemption status of the Preemption Element. It can be "strong"
  % (strong abortion), "weak" (weak abortion), "suspension" (suspension),
  % or "none" (no triggered preemption). The target address of the triggered
  % abortion is expressed by the PreemptionElement.Addr.
```

```
% For the Decoder & Controller
...
if rdAWAIT = true then
  % The current executed delay instruction is terminated.
  DecoderAddr = PC + 1
  % The control goes through.
  TickFlag = true
elsif (Instr.Op = AWAITI) or (Instr.Op = AWAIT) or (Instr.Op = HALT) then
  % Executing a delay instruction
  if Instr.Op = AWAITI and rdPRESENT = true then
    % Executing an immediate delay and the signal is present.
    DecoderAddr = PC + 1
    % The delay instruction should be terminated immediately.
    TickFlag = true
  else
    if PreemptionElement.Type = weak then
      % The control responds to the triggered weak abortion.
      DecoderAddr = PreemptionElement.Addr
      % Going to the target address of the active abortion.
      TickFlag = true
    else
      DecoderAddr = PC
      % The controller is blocked by the delay instruction.
      if Instr.Op != HALT then
        % Configuring the corresponding AWAIT Cell
        rdAWAIT = true
      end if
      TickFlag = false
      % Causing the delay (inactive) of the current execution thread.
    end if
  end if
end if
```

Figure 4.13: Algorithm for handling the delay instructions (Decoder & Controller).

As described in Section 3.2.5, for the KEP, the await case statements are expressed as a group of CAWAIT instructions, which are started by the CAWAITS and ended by the CAWAITE. When the group of CAWAIT instructions starts, the CAWAITS instruction sets a CAWAITStartFlag flag to true. The Decoder & Controller will go through the following CAWAIT statements. If it encounters a CAWAITI instruction and the sensitive signal is present, it will jump to the corresponding case block. Or else, at the end of the tick, it will halt at the CAWAITE instruction, and in the next tick, the control

```
% Definition of Parameters
% Processor.Reset
  % The reset event of the processor.
% AWAITCell.Addr
  % The address of the last executed
  % delay instruction.
% AWAITCell.Count
  % The count value of the current executed
  % delay instruction.
% CountSpec
  % The count value which is transferred
  % from the Data Handling Block.
% wrPAR
  % The signal which denotes whether a PAR
  % instruction is executed or not.
% ParInstr.ThreadID
  % The ID of the thread which is created
  % via the PAR instruction.
```

```
% For the AWAIT Element
if Processor.Reset = true then
  % Whenever the processor is reset.
  forall n ∈ AWAITCell_{N_Thread} do
    AWAITCell_n.Addr = 0
    % Initializing all of AWAIT Cells
  end
end if

if wrPAR = true then
  % Whenever a thread is created.
    AWAITCell_{ParInstr.ThreadID}.Addr = 0
    % Initializing the corresponding AWAIT Cell
end if

if wrAWAIT = true then
  if PC != AWAITCell_{ThreadID}.Addr then
    % Executing a delay instruction: configuring
    AWAITCell_{ThreadID}.Addr = PC
    AWAITCell_{ThreadID}.Count = CountSpec
    % Configuring the corresponding AWAIT Cell
  end if
end if
```

```
% For the AWAIT Element
if PC = AWAITCell_{ThreadID}.Addr then
  % Executing a delay instruction: counting down
  if (PreemptionElement.Type = weak) \...
  or (PreemptionElement.Type = None) then
    % An active weak abortion or no preemption
    if rdPRESENT = true then
      % The sensitive signal is present
      if AWAITCell_{ThreadID}.Count = 1 then
        % The counter value should be zero now.
        rdAWAIT = true
        % The delay is terminated.
        AWAITCell_{ThreadID}.Addr = 0
        % Initializing the AWAIT Cell
      else
        % The counter value should not be zero.
        AWAITCell_{ThreadID}.Count = \...
        AWAITCell_{ThreadID}.Count − 1
        % Counting down
        rdAWAIT = false
        % The delay is not terminated.
      end if
    end if
  elsif PreemptionElement.Type = \...
  strong then
    % The active strong abortion kills the delay
    % instruction immediately
    AWAITCell_{ThreadID}.Addr = 0
    % Initializing the AWAIT Cell
    rdAWAIT = false
    % The delay is not terminated.
  else
    % The active suspension keeps the state
    % of the program. The Await Cell
    % does nothing.
    rdAWAIT = false
    % The delay is not terminated.
  end if
end if
```

Figure 4.14: Algorithm for handling the delay instructions (AWAIT Element).

will start from the CAWAITE instruction, and move back to the first CAWAIT/CAWAITI instruction of this block. Then, it will check the present status of sensitive signal of all CAWAIT/CAWAITI instructions, and respond to the first triggered one. Hence, due to the handling of CAWAIT instructions enclosed by CAWAITS/CAWAITE instructions, the interleaving between different CAWAIT instruction groups is not allowed. Note that this does not restrict which programs we accept—as shown in Figure 4.16, concurrent await case statements can be eliminated in a semantics-preserving way. Figure 4.17 illustrates the behavior of the CAWAIT instructions.

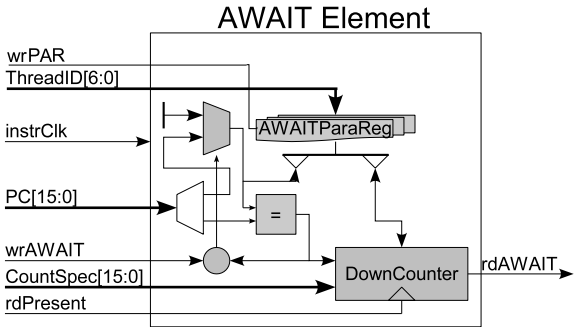


Figure 4.15: Architecture of the AWAIT Element.

```
% Esterel
await
  case A do
    ρ1
  case immediate B do
    ρ2
  case C do
    ρ3
end await;
||
await
  case E do
    ρ4
  case F do
    ρ5
end await;
```

(a)

```
% Esterel
await
  case A do
    ρ1
  case immediate B do
    ρ2
  case C do
    ρ3
end await;
||
trap T in
  loop
    pause;
    present E then
      ρ4
      exit T;
    else
      present F then
        ρ5
        exit T;
      end;
    end;
  end loop;
end trap;
```

(b)

Figure 4.16: Translation of concurrent Esterel await case statements (a) into to an equivalent Esterel program without concurrent await case statements (b).

Handling Preemption

One of the main challenges of designing the Reactive Block is how to deal with the Esterel preemption. In the KEP, the Reactive Block has a Preemption Element, which contains

```
% For the Decoder & Controller
if Instr.Op = CAWAITS then
  % Entering the CAWAIT instruction group
  CAWAITStartFlag = true
  % Setting the CAWAITStartFlag to
  % denote this tick is the entering instant
  % for the CAWAIT instruction group.
  CAWAITFlag = true
end if

if Instr.Op = CAWAITE then
  if CAWAITFlag = true then
    % Execution reaches the end
    DecodeAddr = DecodeAddr
    % Staying at the CAWAITE instruction.
    TickFlag = false
    % The current executed thread should be
    % set as inactive.
    CAWAITFlag = false
    CAWAITStartFlag = false
    % Clearing corresponding status flags
  else
    % Implying the control starts from CAWAITE
    % in a later instant (not the entering tick).
    DecodeAddr = Instr.Addr
    % Going to the first CAWAIT instruction of
    % this CAWAIT instruction group.
    CAWAITFlag = true
    % Setting the corresponding status flag
    TickFlag = true
  end if
end if
```

```
% For the Decoder & Controller
if Instr.Op = CAWAIT or \...
Instr.Op = CAWAITI then
  if rdPRESENT = true then
  % The sensitive signal is present.
    if Instr.Op = CAWAITI then
      % Immediate delay should be responded to
      DecodeAddr = Instr.Addr
      % Going to the corresponding
      % CAWAIT branch
      CAWAITStartFlag = false
      CAWAITFlag = false
    % Clearing corresponding status flags
      TickFlag = true
    else
      if CAWAITStartFlag = false then
        % It is not the entering tick.
        DecodeAddr = Instr.Addr
        % Going to the corresponding
        % CAWAIT branch
        TickFlag = true
      else
        % Standard delay should not be
        % responded to in the entering tick.
        DecodeAddr = DecodeAddr + 1
        % Going through
        TickFlag = true
      end if
    end if
  else
    % The sensitive signal is absent.
    DecodeAddr = DecodeAddr + 1
    % Going through
    TickFlag = true
  end if
end if
```

Figure 4.17: Algorithm for handling the parallel await.

a configurable number of Watcher modules that are responsible for implementing the preemption operations.

According to the Esterel semantics, a preemption (abortion or suspension) is *enabled* when control is in its body, and *disabled* when control is outside of its body. When a preemption is *enabled*, the corresponding trigger signal is watched and the module can react to the presence of it (is *active*). Otherwise, the signal does not cause preemption. As mentioned in Section 3.2.2, we call this scheme *Inside/Outside Preemption Range Watching* (*IOPRW*).

A Watcher, shown in Figure 4.18, contains two functions to implement the IOPRW, *i. e.*, the *Enable Watcher* and the *Trigger Watcher*.

Enable Watcher watches the program counter (PC) and compares it with the corresponding preemption's start and end addresses. Based on that, it decides whether this preemption should be in the *enabled state* or in the *disabled state*. If the watched signal is present on the Tick rising edge and the Watcher is in the enabled state, the Watcher triggers a corresponding action, unless it is overridden by another Watcher with higher priority, *e. g.*, an enclosing nesting activates a suspension and freezes the state of its body. Once the Watcher changes its state from disabled to enabled, which means that the PC re-enters the watching range, the SignalCount will be reloaded into the counter.

Trigger Watcher depends on the configuration of the Watcher. For an abortion, it watches the trigger signal; if the signal occurs, the Watcher goes into the *triggered state* and counts down the signal count; then, depending on whether the trigger signal count specified by the abortion statement has already been reached or not, the Watcher decides whether it should go into the *terminated state*, which would kill the abort body, or not. For a suspension, the Watcher watches the trigger signal and decides whether the suspension body ought to go into the *suspended state* or not. Once an abortion is terminated or a suspension is activated, a TW event will be emitted.

Every Watcher contains 5 reconfigurable parameters, *i. e.*, StartAddr, EndAddr, SignalCode, SignalCount, and PreemptionFlag. The Watcher is configured by the corresponding preemption instruction and then it runs autonomously. StartAddr and EndAddr assign the watching range of the Watcher. SignalCoder indicates which signal ought to be watched. If the watched signal is valid and Watcher is in the enabled state, the value of SignalCount is decreased. The Watcher emits a TW event to the environment when the counter value equals zero. The PreemptionFlag, which indicates the preemption type and EndAddr registers can be accessed by other KEP components.

When a preemption instruction is executed, the EndAddr, SignalCode, and PreemptionFlag parameters are given by the preemption instruction code directly, and the current PC will be stored as the StartAddr. The SignalCount value is gotten from the CountSpec port. Furthermore, the Watcher identification (*WatcherID*) specifies which Watcher should be configured for the current preemption instruction.

For the immediate strong preemption, the Decoder & Controller will check the status of the sensitive signal before writing parameters to the corresponding watcher. If the signal is present, the control will respond to the preemption immediately. For example, an immediate strong abortion will cause the control jump to the end of abortion scope directly, and the watcher will not be configured.

Once an indexed Watcher receives an active wrPreemption signal, it will decode the parameters it needs from the current instruction code and some other signal ports.

Each Watcher contains a Downable flag to indicate whether its SignalCount can count down in this tick or not. For the non-immediate preemption (standard or count delays), this flag will be cleared to avoid the immediate triggering of the Watcher. For

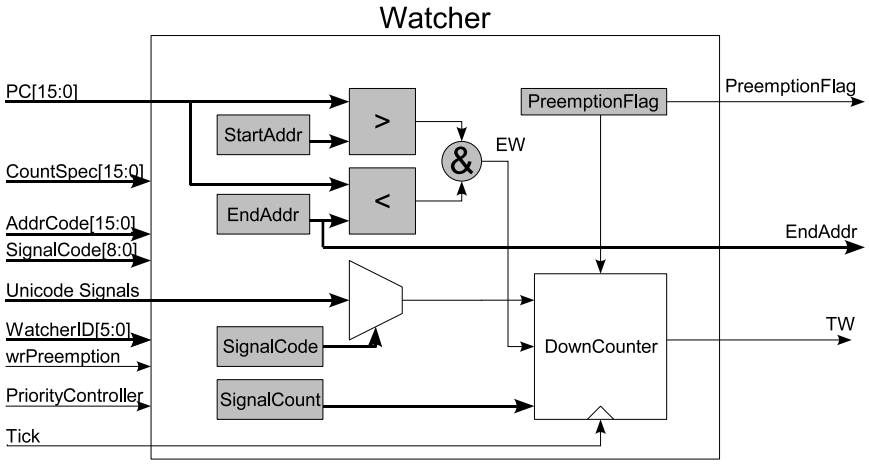


Figure 4.18: Architecture of the Watcher.

the immediate preemption, this flag is set to allow the Watcher to receive control. At the beginning of the next tick, the Downable flags of all Watchers will be set. Once the SignalCount of a Watcher is decremented, this flag will be cleared again to avoid that multiple signal emissions would be considered as multiple signal presences. Figure 4.19 shows the procedure for configuring a Watcher. The $PreemptionInstr.Configure$ parameter denotes what kinds of watcher should be configured. The value of it can be $Watcher$, $LWatcher$, and $TWatcher$.

Due to the independence of all Watchers, multiple Watchers in a Preemption Element can run in parallel. Each Watcher works independently, according to the configured parameters. The distributed Watchers structure makes the architecture of ABORT Element clear, concise and scalable.

Figure 4.20 shows the architecture of a Reactive Block including three Watcher modules. The Priority Controller checks the TW and PreemptionFlag signals, and uses certain rules to judge which Watcher's output signals should be mapped to the Reactive Core via MUXs that are controlled by the Priority Controller.

Considering the priority of a nested preemption, if a strong abortion is triggered, its body will be killed immediately, *i. e.*, the program segment which is located inside of the Watcher will be not executed. The control jumps to the end of the abortion scope. If a suspension is triggered, its body will be frozen immediately, in other words, its inner program segment will also not be executed. This feature results from the chain structure of the Watchers. The Watcher which is located at the higher level corresponds to the outer preemption. If it is triggered, it will disable all lower ones via the PriorityControl signal. Or else, if a weak abortion is triggered, lower Watchers can still be triggered.

Hence, for a strong abortion nest, the Priority Controller will check the statuses of the Watchers from outer to inner. Once there is a triggered Watcher, the Priority Controller

```
% Definition of Parameters
% PreemptionInstr.Type
  % Refers to parameters of PreemptionElement.Type.
  % The PreemptionInstr.Type can be "strong", "weak", or "suspension".
% PreemptionInstr.ImmediateFlag
  % It indicates whether this instruction expresses an immediate preemption or not.
% wrPreemption
  % This signal denotes whether a watcher should be configured or not.
```

```
% For the Decoder & Controller
if Instr.Op = PreemptionInstr then
  % Executing a preemption instruction
  if (PreemptionInstr.ImmediateFlag = true) \...
  and (rdPRESENT = true) and PreemptionInstr.Type != weak then
    if PreemptionInstr.Type = strong then
      % For a triggered immediate strong abortion
      DecodeAddr = Instr.Addr
      % The control jumps to the end of abortion scope directly
      wrPreemption = false
      TickFlag = true
    elsif PreemptionInstr.Type = suspend
      % For a triggered immediate suspension
      DecodeAddr = PC
      % The control stops at the current address
      wrPreemption = false
      TickFlag = false
      % Setting the current executed thread as inactive.
    end if
  else
      % Configuring the watcher and going through
      DecodeAddr = PC + 1
      wrPreemption = true
      TickFlag = true
  end if
end if
```

```
% For the Preemption Element
if (wrPreemption = true and PreemptionInstr.Configure = Watcher) then
  % Writing parameters to the Watcher_n
  Watcher_{Instr.WatcherID}.StartAddr = PC
  Watcher_{Instr.WatcherID}.EndAddr = Instr.Addr
  % Configuring watching address range of the Watcher
  Watcher_{Instr.WatcherID}.SignalCount = CountSpec
  Watcher_{Instr.WatcherID}.SignalCode = Instr.SignalCode
  % Configuring the value of the count delay and the sensitive signal.
  Watcher_{Instr.WatcherID}.PreemptionFlag = PreemptionInstr.Type
  % Configuring the preemption type
  Watcher_{Instr.WatcherID}.Downable = PreemptionInstr.ImmediateFlag
  % Denoting whether the count could be counted down immediately or not
  % (for immediate weak abortion)
end if
```

Figure 4.19: Algorithm for configuring watchers.

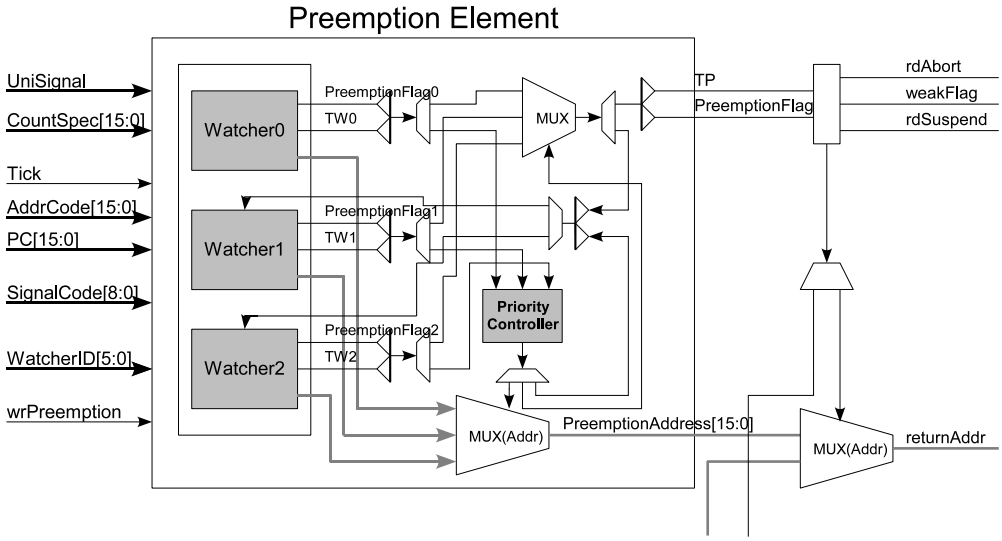


Figure 4.20: Architecture of a Reactive Block with three Watchers.

will respond to it immediately and ignore remaining Watchers. For the weak abortion nest, the Priority Controller will check triggered Watchers from innermost to outermost, and respond to the first triggered weak abortion.

Note that a suspension could also be nested in a weak abortion. The Priority Controller follows the order of handling weak abortion nests. If both of them are triggered, the first triggered weak abortion will be responded to. If there is no triggered weak abortion, the Priority Controller will map the triggered suspension as the preemption type of the Preemption Element.

To illustrate its operation, consider the Esterel module NESTED in Figure 4.21(a), which is an example of a nested preemption. Figure 4.21(b) shows the corresponding KEP assembler program of the NESTED module, and a possible execution trace is given in Figure 4.22.

After starting, this Esterel module watches C and A as the abortion trigger signals, and B as the suspension trigger signal. The execution stays on the delay statement, which is located inside the preemptions, to wait for signal D. Those three preemptive statements constitute a mixed preemption nest, and the priority of the outer preemptive statement is higher than that of the inner one.

When the KEP executes NESTED, first the watchers $\mathsf{Watcher}_0$, $\mathsf{Watcher}_1$ and $\mathsf{Watcher}_2$ are configured via three preemption instructions in line L01 – L03. The PC stays at AWAIT $\mathsf{D}_{\mathsf{L04}}$ until any of the signals A, B, C, or D occur. Since this address is within each of the watcher's watching range, all of the watchers are enabled now.

```
% Esterel
module NESTED:
input A,B,C,D;
output E,F,G,H;
weak abort
  suspend
    abort
      await D;
      emit E;
    when C;
    emit F;
    await D;
    emit E;
  when B;
  await D;
  emit G;
when A;
emit H;
halt;
end module
```

(a)

```
                % KEP Assembler
                % module NESTED
                INPUT A,B,C,D
                OUTPUT E,F,G,H
[L00,T0]        EMIT _TICKLEN, \#20
[L01,T0]        WABORT A,A2
[L02,T0]        SUSPEND B,A1
[L03,T0]        ABORT C,A0
[L04,T0]        AWAIT D
[L05,T0]        EMIT E
[L06,T0] A0:    EMIT F
[L07,T0]        AWAIT D
[L08,T0]        EMIT E
[L09,T0] A1:    AWAIT D
[L10,T0]        EMIT G
[L11,T0] A2:    EMIT H
[L12,T0]        HALT
```

(b)

Figure 4.21: NESTED: the Esterel module illustrating the preemption statements (a), the KEP assembler program (b).

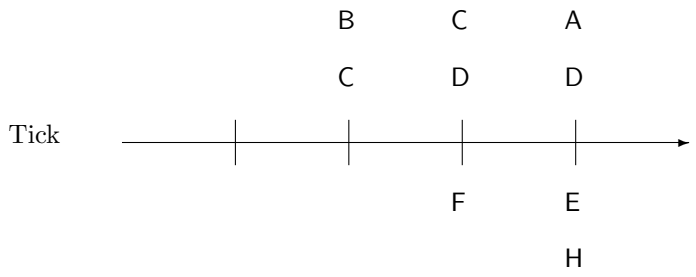


Figure 4.22: A possible execution trace of the NESTED module.

If B and C occur simultaneously, TW1 and TW2 are set at the same time. The PreemptionFlag1 indicates a suspension, and PreemptionFlag2 indicates strong abortion. The Priority Controller processes TW events based on rules about priorities and preemption types of Watchers according to the Esterel semantics. In this case, the suspension triggered by B has higher priority. The Priority Controller maps Watcher_1's outputs to the Reactive Block's output, so the Preemption Element's TP, which means Trigger Preemption, is '1' for denoting an active preemption, and the PreemptionFlag indicates a suspension. The generated control signals will be broadcast to all of the lower priority

watchers and other relevant elements. Watcher_2 receives the output of the Priority Controller, and since the current active preemption is a suspension, Watcher_2 will keep its state, which means that the counter of this watcher will not be decremented. A decoder analyzes the TP and PreemptionFlag signals, and decodes them to three signals of the Reactive Block, *i. e.*, rdAbort, weakFlag and rdSuspend, which denote the current active preemption for the Decoder & Controller. The Decoder & Controller checks the active event type of the Preemption Element (which is composed of the rdAbort, the weakFlag, and the rdSuspend signals), the status of the rdAWAIT, and so on, simultaneously. Since the active preemption is a suspension, the KEP keeps its state until the current tick is finished.

Assuming now that the signals C and D occur simultaneously in the next instant, Watcher2 takes priority. The outputs of Watcher2 are mapped to that of the Preemption Element, so the Reactive Block's rdAbort is '1' to denote that there is an active abortion. The returnAddr points the next instruction address behind the body of abortion C, *i. e.*, L06, and the weakFlag is '0' to indicate a strong abortion type. Since the sensitive signal of the AWAIT D_{L04} statement is present, the AWAIT Element sets rdAWAIT to '1' to denote that the AWAIT instruction is terminated. The Decoder & Controller checks the rdAbort, weakFlag and rdAWAIT signals and responds to the strong abortion. The returnAddr is mapped to the PC via the Address Multiplexer. The KEP jumps out of the abortion D's body and executes "EMIT F_{L06}" and "AWAIT D_{L07}". Now the Watcher_2 is disabled because the PC is out of its watching range, but the Watcher_0 and Watcher_1 are still enabled.

When signals A and D occur simultaneously in the following instant, weak abortion A takes priority. The Priority Controller maps Watcher_0's outputs to that of the Preemption Element, so the Reactive Block's rdAbort is set to '1' for denoting an active abortion, the returnAddress points to L11—the next instruction address behind the body of abortion A, and the weakFlag is '1' to indicate a weak abortion type. At the same time, the AWAIT Element sets rdAWAIT to '1' to denote that the AWAIT is terminated. Since the active abortion is a weak abortion, the KEP will respond to the terminated AWAIT D_{L07} instruction first. The EMIT E_{L08} is executed and then the AWAIT D_{L09} is fetched. Since it is a non-instantaneous statement, the Reactive Core will ignore it—*i. e.*, not configure the AWAIT Element—and instead respond to the weak abortion. The returnAddress is mapped to the PC, and then control jumps to A2_{L11}. The KEP executes EMIT H_{L11} and HALT_{L12}.

The above mentioned process illustrates the mechanism of the KEP Preemption Element. In short, the Watcher is the basic cell which is designed as an all-purpose preemption handling component. A Watcher chain permits arbitrary nesting of preemptions, and also the combination with the concurrency operator. However, it is questionable whether the unrestricted use of such an elaborate architecture is the most effective for a real Esterel application, which commonly does not contain deep preemption nests but instead typically uses parallel or sequential preemptions. Hence, in practice this architecture

often turns out to be more general than necessary, and hence wasteful of hardware resources.

Therefore, two subclasses of the Watcher have been proposed to refine the Preemption Element. The innermost preemption handling component is the Thread Watcher (TWatcher), which belongs to a thread privately, and is used to handle a preemption which neither nests concurrent threads nor nests other preemptions. It is the least powerful, but also the cheapest variant. An intermediate variant is the Local Watcher (LWatcher). It offers a group of parallel preemption handling components, which can cross threads, nest Thread Watchers, but cannot nest each other or nest the Watcher.

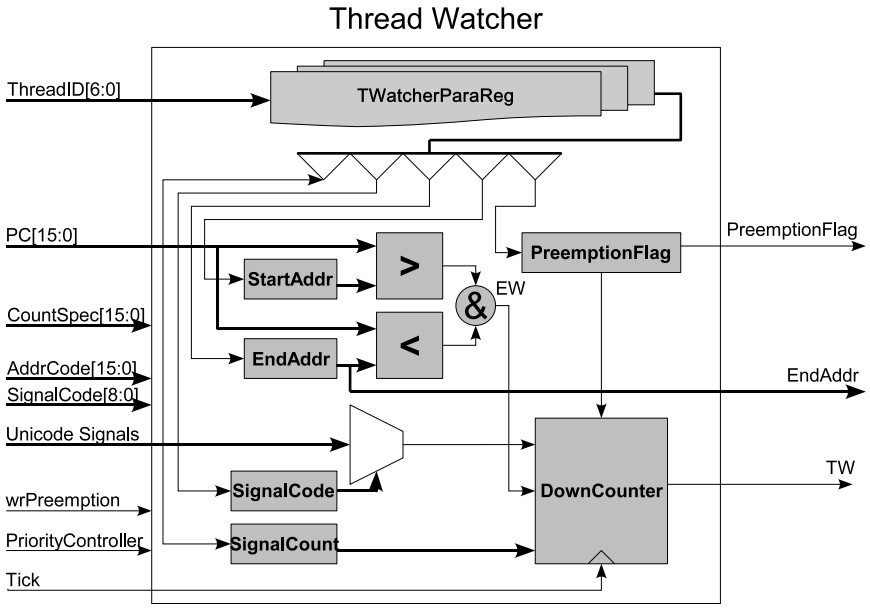


Figure 4.23: Architecture of the Thread Watcher (TWatcher).

The architecture of the Thread Watcher is illustrated in Figure 4.23. It is similar to a Watcher, however, it employs a register file to record parameters of the preemptions. The register file is indexed by the $ThreadID$ signal, which comes from the Thread Block and identifies the currently executing thread. When the instruction for configuring the TWatcher is executed, the Reactive Core selects the corresponding register via the $ThreadID$ and writes parameters into it. Note that only one thread is executed at a time in a multi-threaded processor. Hence, when this thread is executed again, the TWatcher will work as a single Watcher.

The principle of the Thread Watcher is also used for the Local Watcher (LWatcher). However, the index method is different. To the LWatcher, the index value is generated by the comparison of the current PC and configured start address of the LWatchers. Then the end address of the indexed LWatcher will be compared with the PC. If the

PC is located in the range of the indexed LWatcher, the LWatcher will be enabled. The LWatcher index strategy costs n address comparators for n LWatchers. For comparison, the Watcher needs $2n$ address comparators for n Watchers.

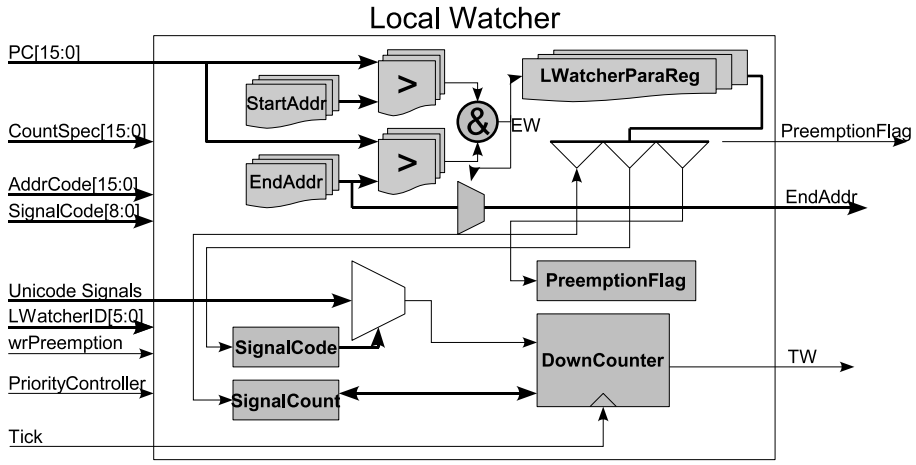


Figure 4.24: Architecture of the Local Watcher (LWatcher).

Due to the LWatchers characteristics, only *one* (or *none*) LWatcher can be selected (enabled) at once. Hence, the LWatcher can also be regarded as a single Watcher. Figure 4.24 shows the architecture of the Local Watcher. The process of indexing the LWatcher and LWatcher can be found in Figure 4.25.

As mentioned in Section 3.2.2, the KEP employs different instructions to configure various watchers, see also the Watcher chain in Figure 4.20. If the lower two Watchers are replaced by the LWatchers and TWatchers, the depth of the preemption nest is still three levels, and the process of handling preemption is similar. However, the capability of the Preemption Element for handling parallel preemptions is obviously enhanced by slightly increasing resource usage. To make a tradeoff between the Preemption Element's resources usage and its capabilities, the abilities of handling suspension of the LWatcher and the TWatcher are reduced.

Figure 4.26 illustrates the process of triggering Watchers, and the triggering process of the LWatcher and the TWatcher is shown in Figure 4.27.

As mentioned before, the Watcher Priority Controller handles all triggered watchers, and decides to which watcher the parameters should be mapped as the output of the Preemption Element. This mechanism is provided by the "if-elsif" statements in VHDL [83]. Figure 4.28 illustrates the model of Watcher Priority Controller, which handles two Watchers, the Local Watcher, and the Thread Watcher.

```
% For the Preemption Element
% Indexing the Local Watcher
% (by the comparison between the PC and the address range of Local Watchers)
for n = N_LWatcher −1 to 0 do
  if (PC > LWatcher_n.StartAddr) then
    LWatcherID = n
    % Getting the ID of the LWatcher which could be enabled
    % (note address ranges of Local Watchers are exclusive)
  end if
end

if PC < LWatcher_LWatcherID.EndAddr then
  LWatcher.EW = true
  % Ensuring this LWatcher is enabled.
else
  % The PC is out of the watching address range of this LWatcher
  LWatcher.EW = false
end if

% Mapping parameters of the enabled LWatcher for the process
% of triggering Local Watcher
if LWatcher.EW = true
  LWatcher.SignalCount = LWatcher_LWatcherID.SignalCount
  LWatcher.SignalCode = LWatcher_LWatcherID.SignalCode
  LWatcher.PreemptionFlag = LWatcher_LWatcherID.PreemptionFlag
end if
```

```
% For the Preemption Element
% Indexing the Thread Watcher
% (by the ThreadID)
TWatcher.StartAddr = TWatcher_ThreadID.StartAddr
TWatcher.EndAddr = TWatcher_ThreadID.EndAddr
TWatcher.SignalCount = TWatcher_ThreadID.SignalCount
TWatcher.SignalCode = TWatcher_ThreadID.SignalCode
TWatcher.PreemptionFlag = TWatcher_ThreadID.PreemptionFlag
```

Figure 4.25: Algorithm for indexing the LWatcher and the TWatcher.

Handling Exceptions

In the KEP, a trap is initialized when its exception is thrown, *i. e.*, the execution of the EXIT makes a pair of startAddr/endAddr record the address of this trap scope. Besides, the *ParentThread* parameter of the current thread, which comes from the Thread Block, is also stored. Once another EXIT instruction is executed, its parameters are also recorded. An exitFlag will be set to active ('1') when any exception is active. Once the control of the processor (PC) reaches the end of the trap scope, the parameters of this trap will be cleared.

If the scope of the trap T crosses several threads and one of them executes an EXIT instruction, a sequence of processes will ensure the correctness of the KEP exception implementation. First, the thread which executes this instruction will be set to the disabled status, because it tries to access an address which is out of the thread range.

```
% For the Preemption Element
% Judging the Enabled status (EW) of Watchers
forall n ∈ N_Watcher do
  if (PC > Watcher_n.StartAddr and PC < Watcher_n.EndAddr) then
    % The PC is located inside of the watching range
    Watcher_n.EW = true
    % The Enabled status (EW) of this Watcher is true
  else
    Watcher_n.EW = false
  end if
end

% Judging the Triggered status (TW) of Watchers
PriorityController = false
SuspensionFlag = false
% Initializing statuses of the Preemption Element
for n = 0 to N_Watcher −1 do
  % From the outer Watcher to inner one step by step
  Watcher_n.TW = false
  % Initializing the trigger status of the Watcher
  if Watcher_n.EW = true then
    % When the Watcher is enabled
    if PriorityController = false then
      % There is no outer triggered strong preemption.
      if UniSignal(Watcher_n.SignalCode) = true then
          % The sensitive signal is present
          if Watcher_n.Downable = true then
            % The count value of this Watcher could be counted down
            if Watcher_n.PreemptionFlag = Suspend then
              % No count delay for suspension (Esterel semantic)
              Watcher_n.TW = true
              % Watcher Triggered
              SuspensionFlag = true
              % Denoting a triggered suspension for lower priority Watchers
            else
              Watcher_n.SignalCount = Watcher_n.SignalCount − 1
              % Counting down the count value of this Watcher
              Watcher_n.Downable = false
              % This Watcher cannot be counted down again in this tick.
            end if
          end if
          if Watcher_n.SignalCount = 0 then
            % Watcher Triggered
            Watcher_n.TW = true
            % Denoting a triggered preemption
          end if
          if Watcher_n.TW = true then
            if Watcher_n.Type = strong or Watcher_n.Type = suspension then
              PriorityController = true
              % Lower priority Watchers are controlled by the strong preemption.
            end if
          end if
        end if
      end if
    end if
  end if
end
```

Figure 4.26: Algorithm for triggering Watchers.

```
% For the Preemption Element
if (PC > TWatcher.StartAddr and PC < TWatcher.EndAddr) then
  TWatcher.EW = true
  % The PC is located inside of the watching range
  else
  TWatcher.EW = false
end if

% Judging the Triggered status (TW) of the Local Watcher
if PriorityController = false and LWatcher.EW = true then
  % There is no higher priority triggered strong preemption
  % and the LWatcher is enabled.
  LWatcher.TW = false
  % Initializing the trigger status of the LWatcher
  if UniSignal(LWatcher.SignalCode) = true then
    % The sensitive signal is present
    if LWatcher_LWatcherID.Downable = true then
      % The count value of this LWatcher could be counted down
      LWatcher.SignalCount = LWatcher.SignalCount − 1
      LWatcher_LWatcherID.SignalCount = LWatcher.SignalCount
      % Counting down, and writing back to the corresponding LWatcher.
      LWatcher_LWatcherID.Downable = false
    end if
    if LWatcher.SignalCount = 0 then
      % LWatcher Triggered
      LWatcher.TW = true
      % Denoting a triggered preemption
      if LWatcher.Type = strong then
        PriorityController = true
        % Lower priority watcher (TWatcher) is controlled by the strong preemption.
      end if
    end if
  end if
end if

% Judging the Triggered status (TW) of the Thread Watcher
if PriorityController = false and TWatcher.EW = true then
  % There is no higher priority triggered strong preemption
  % and the TWatcher is enabled.
  TWatcher.TW = false
  % Initializing the trigger status of the LWatcher
  if UniSignal(TWatcher.SignalCode) = true then
    % The sensitive signal is present
    if TWatcher_ThreadID.Downable = true then
      % The count value of this TWatcher could be counted down
      TWatcher.SignalCount = TWatcher.SignalCount − 1
      TWatcher_ThreadID.SignalCount = TWatcher.SignalCount
      % Counting down, and writing back to the corresponding TWatcher.
      TWatcher_ThreadID.Downable = false
    end if
    if TWatcher.SignalCount = 0 then
      % TWatcher Triggered
      TWatcher.TW = true
      % Denoting a triggered preemption
    end if
  end if
end if
```

Figure 4.27: Algorithm for triggering the LWatcher and the TWatcher.

```
% For the Preemption Element
% Judging the final result of the Preemption Element
if Watcher_0.TW = true and Watcher_0.PreemptionFlag = strong then
  % The strong abortion takes priority
  PreemptionElement.Addr = Watcher_0.EndAddr
  PreemptionElement.Type = strong
elsif Watcher_1.TW = true and Watcher_1.PreemptionFlag = strong then
  % The strong abortion takes priority
  PreemptionElement.Addr = Watcher_1.EndAddr
  PreemptionElement.Type = strong
elsif LWatcher.TW = true and LWatcher.PreemptionFlag = strong then
  % The strong abortion takes priority
  PreemptionElement.Addr = LWatcher.EndAddr
  PreemptionElement.Type = strong
elsif TWatcher.TW = true and TWatcher.PreemptionFlag = strong then
  % The strong abortion takes priority
  PreemptionElement.Addr = TWatcher.EndAddr
  PreemptionElement.Type = strong
elsif TWatcher.TW = true and TWatcher.PreemptionFlag = weak then
  % The inner weak abortion should first be responded to
  PreemptionElement.Addr = TWatcher.EndAddr
  PreemptionElement.Type = weak
elsif LWatcher.TW = true and LWatcher.PreemptionFlag = weak then
  % The inner weak abortion should first be responded to
  PreemptionElement.Addr = LWatcher.EndAddr
  PreemptionElement.Type = weak
elsif Watcher_1.TW = true and LWatcher.PreemptionFlag = weak then
  % The inner weak abortion should first be responded to
  PreemptionElement.Addr = Watcher_1.EndAddr
  PreemptionElement.Type = weak
elsif Watcher_0.TW = true and LWatcher.PreemptionFlag = weak then
  if SuspensionFlag = true then
    % Note for a "weak abortion–suspension" nest, if both of them are triggered,
    % the body of the suspension will be freezed. Then the control will jump
    % to the end of the weak abortion scope, i. e., responds to the weak
    % abortion like a strong abortion.
    PreemptionElement.Type = suspension
  else
    % The weak abortion should be responded to
    PreemptionElement.Type = weak
  end if
  PreemptionElement.Addr = Watcher_0.EndAddr
else
  % Excepting for all above mentioned conditions
  if SuspensionFlag = true then
    PreemptionElement.Type = suspension
    % The suspension should be responded to.
  else
    PreemptionElement.Type = none
    % No any active preemption.
  end if
  PreemptionElement.Addr = 0
end if
```

Figure 4.28: Algorithm for handling all watchers.

However, other concurrent threads will receive the control one last time and then also be disabled.

There is another possible state, *e. g.*, some concurrent threads were executed before executing the EXIT in this tick and were set to the inactive status. Hence, they cannot directly respond to the exception because they will not awaken in the current instant. To handle this problem, the Thread Manager will judge the status of the exitFlag at the join point. If there is an active exception, all concurrent threads which wait at this join point will be disabled, and then control will not execute the following instruction but respond to the exception.

As for trap nests, the question is how to distinguish which one ought to take priority. A simple idea is that the outer trap, which has the larger address range, will override the inner one. Unfortunately, this strategy is too simple to satisfy all cases.

Two examples are shown in Figure 4.29. The Trap1 and Trap2 modules are similar. The initial thread defines trap TP1 and then branches thread 1 and thread 2 as two parallels. Thread 2 defines trap TP2 when it starts, and then forks thread 3 and thread 4.

For either module, thread 3 will exit TP2 trap in the second tick, just after it terminates the pause statement and emits the S2 signal. The difference of those two modules is that thread 1 of Trap2 module executes an exit $\mathsf{TP1}_{\mathsf{L04}}$ statement to exit trap TP1, just after it emits S1 signal; however, this exception is thrown by thread 4 in the Trap1 module.

Obviously, trap TP1 and trap TP2 are nested, and either of the two modules will throw those two exception in the second tick. However, due to the difference of two modules, their behaviors are different.

For the Trap1 module, after the initial tick, the control of its threads stays on three $\mathsf{pause}_{\mathsf{L02,L05,L08}}$ statements. In the second tick, thread 1 terminates the $\mathsf{pause}_{\mathsf{L02}}$ statement and emits the S1 signal. Simultaneously, thread 3 emits the S2 signal and throws the exception of the TP2 trap; and thread 4 also emits S3 signal and executes exit $\mathsf{TP1}_{\mathsf{L10}}$ statement to exit the TP1 trap. Hence, the outer trap TP1 takes priority. The control moves to the end of the scope of the corresponding trap TP1 declaration. During the above mentioned process, signal S1, S2, and S3 are emitted.

In the Trap2 module, the thread 1 throws the TP1 exception after emitting the S1 signal. At the same time, thread 3 executes the exit $\mathsf{TP2}_{\mathsf{L08}}$ statement to exit the TP2 trap. In this case, thread 2 is first weakly aborted, hence, control advances to the end of the scope of the TP2 trap, and then all of the instantaneous statements will be executed. That means the S signal will be set to presence via the emit $\mathsf{S}_{\mathsf{L13}}$ statement. Then the control responds to the exception of the T1 trap. During this period, signals S, S1, S2, and S3 are emitted.

Hence, although the nest structure is same in either case, the different way of threads throwing exceptions causes the different results.

The KEP uses some elaborate rules to accomplish this task. If the PC is located inside of the scope of a trap, the corresponding exception will be active. When several exceptions

```
          module Trap1:
          output R,S,T;
          output S1,S2,S3;
[L01,T0]  trap TP1 in
[L02,T1]    [ pause;
[L03,T1]      emit S1;
              % exit TP1;
            ||
[L04,T2]      trap TP2 in
[L05,T3]        [ pause;
[L06,T3]          emit S2;
[L07,T3]          exit TP2;
                ||
[L08,T4]          pause;
[L09,T4]          emit S3;
[L10,T4]          exit TP1;
                ];
[L11,T2]        emit R
[L12,T2]      end trap;
[L13,T2]      emit S ];
[L14,T0]    emit T
[L15,T0]  end trap;
[L16,T0]  halt;
          end module
```

```
          module Trap2:
          output R,S,T;
          output S1,S2,S3;
[L01,T0]  trap TP1 in
[L02,T1]    [ pause;
[L03,T1]      emit S1;
[L04,T1]      exit TP1;
            ||
[L05,T2]      trap TP2 in
[L06,T3]        [ pause;
[L07,T3]          emit S2;
[L08,T3]          exit TP2;
                ||
[L09,T4]          pause;
[L10,T4]          emit S3;
                  %exit TP1;
                ];
[L11,T2]        emit R
[L12,T2]      end trap;
[L13,T2]      emit S ];
[L14,T0]    emit T
[L15,T0]  end trap;
[L16,T0]  halt;
          end module
```

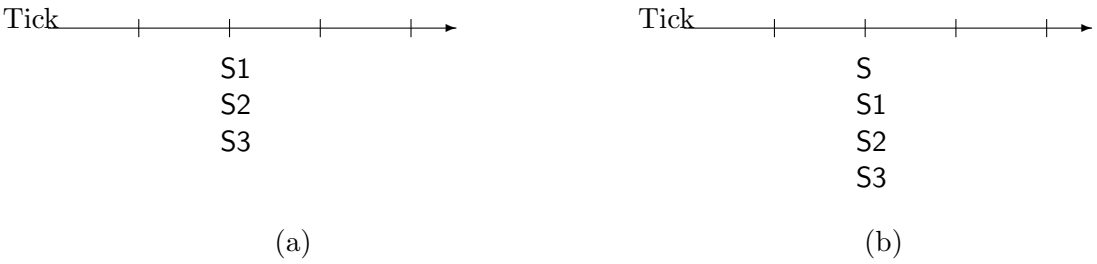


Figure 4.29: Esterel modules illustrating the trap nest, and possible execution trace. T0 denotes the initial thread, T1 is thread 1, etc.

are all active, the KEP will compare their *ParentThread* to estimate whether they belong to a group of "concurrent" threads. If they have a same parent thread, the outer trap will cancel the inner one. Or else, the control will respond the inner one at first.

The inherited strategy is used when a trap crosses several threads. At the join point, if a thread finds there is an active exception which is emitted by its child thread, it will inherit this exception by modifying the *ParentThread* of this exception as its parent thread.

Figure 4.30 shows KEP assembler programs for the Trap1 and Trap2 modules. For the KEP program of the Trap1 module, all of these threads have the same priority values as 1,

```
                        % KEP Assembler
                        % module Trap1:
                        OUTPUT R,S,T
                        OUTPUT S1,S2,S3
                        EMIT _TICKLEN,#15
[L01,T0] TP1S:          PAR 1,P1
[L02,T0]                PAR 1,P2
[L03,T0]                PARE P3
[L04,T1] P1:            PAUSE
[L05,T1]                EMIT S1
[L06,T2] P2: TP2S:      PAR 1,P4
[L07,T2]                PAR 1,P5
[L08,T2]                PARE P6
[L09,T3] P4:            PAUSE
[L10,T3]                EMIT S2
[L11,T3]                EXIT T2PE,TP2S
[L12,T4] P5:            PAUSE
[L13,T4]                EMIT S3
[L14,T4]                EXIT T1PE,TP1S
[L15,T2] P6:            JOIN
[L16,T2]                EMIT R
[L17,T2] TP2E:          EMIT S
[L18,T0] P3:            JOIN
[L19,T0]                EMIT T
[L20,T0] TP1E:          HALT
```

(a)

```
                        % KEP Assembler
                        % module Trap2:
                        OUTPUT R,S,T
                        OUTPUT S1,S2,S3
                        EMIT _TICKLEN,#15
[L01,T0] TP1S:          PAR 2,P1
[L02,T0]                PAR 1,P2
[L03,T0]                PARE P3
[L04,T1] P1:            PAUSE
[L05,T1]                EMIT S1
[L06,T2]                EXIT T1PE,TP1S
[L07,T2] P2: TP2S:      PAR 1,P4
[L08,T2]                PAR 1,P5
[L09,T2]                PARE P6
[L10,T3] P4:            PAUSE
[L11,T3]                EMIT S2
[L12,T3]                EXIT T2PE,TP2S
[L13,T4] P5:            PAUSE
[L14,T4]                EMIT S3
[L15,T2] P6:            JOIN
[L16,T2]                EMIT R
[L17,T2] TP2E:          EMIT S
[L18,T0] P3:            JOIN
[L19,T0]                EMIT T
[L20,T0] TP1E:          HALT
```

(b)

Figure 4.30: The KEP programs corresponding to the Trap1 and Trap2 modules (Figure 4.29).

except that the priority of the initial thread (thread 0) is always zero. In the second tick, the control starts from the $\mathsf{PAUSE}_{\mathsf{L12}}$ instruction of thread 4 due to the scheduling rule that was mentioned in Section 4.2.1. After the signal S3 is emitted by $\mathsf{EMIT\ S3}_{\mathsf{L13}}$, the $\mathsf{EXIT\ TP1E,TP1S}_{\mathsf{L14}}$ will throw an exception TP1. The first TRAP Register records the related information, *i. e.*, the start address TP1S, the end address TP1E, and the parent thread of the current thread, *i. e.*, the thread 2. Thread 4 is terminated immediately, and thread 3 is arranged to be executed. Similar to the above mentioned process, it emits S2 and throws another exception TP1, which ranges from TP2S to TP2E, and the *ParentThread* is thread 2 too. Hence, this exception is erased at once because the TP1 trap takes priority. With that, thread 2 responds the TP1 exception at the join point $\mathsf{JOIN}_{\mathsf{L15}}$, and modifies the *ParentThread* parameter as thread 0—the parent thread of thread 2. Since the target address TP1E of the exception exceeds the range of thread 2, thread 2 is terminated immediately at the join point. Thread 1 emits the signal S1 and then is terminated normally. Thread 0 responds to the active TP1 exception by jumping to TP1E, and then halts. Therefore, signals S1, S2, and S3 are emitted in this tick.

The process of the second tick of the KEP program of Trap2 module is different. To make it easier to understand, we use the similar execution order of exceptions as that

of the Trap1 program. Hence the priority of thread 1 is assigned to 2. So in the second tick, the control starts from the $\mathsf{PAUSE}_{\mathsf{L04}}$ instruction of thread 1, emits the signal S1, and then throws the TP1 exception. However, the *ParentThread* parameter should be recorded as thread 0. Then, thread 3 emits the signal S2 and executes EXIT $\mathsf{TP2E,TP2S}_{\mathsf{L12}}$ after thread 4 emits the signal S3 and terminates normally. The join point of thread 3 and thread 4 is the $\mathsf{JOIN}_{\mathsf{L15}}$ instruction of thread 2. This join point is located in either traps address ranges. Hence two exceptions are active at the same time. Note their *ParentThread* parameters are different, so these two exceptions will be active simultaneously. The control will respond to the inner one at first by jumping to the TP2E. As the control reaches the scope of the TP2 trap, the TP2 exception is terminated and its parameters will be cleared. However, the TP1 exception is still active. Before this thread is terminated, the instantaneous instruction EMIT $\mathsf{S}_{\mathsf{L17}}$ is executed. Now thread 0 will be scheduled and it will respond to exception TP1. Hence, signals S, S1, S2, and S3 are all emitted in this tick.

Figures 4.31 and 4.32 show the exception handling strategy of the KEP. The *EmptyTrapPoint* parameter indicates an empty Trap Register which can be written, and the *ClearTrap* parameter marks which Trap Register should be cleared. Finally, the *Trap.Active* parameter denotes whether there is an active exception or not, and the target address of the active exception is mapped to *Trap.Addr*.

For the Decoder & Controller of the KEP, the strategy of handling an active exception is similar to that of handling an active weak abortion. But how to respond when an exception and a weak abortion are both active? Due to the triggering mechanism of the exception and abortion, if they are active simultaneously, they are nested. Hence, the response depends on their address ranges — the inner one will be handled first.

There is another interesting situation which could make this mechanism fail. Considering the example in Figure 4.33(b), which is the corresponding KEP assembler program of the Esterel program shown in Figure 4.33(a), due to the scheduling method of the KEP, in the initial tick, the thread 3 emits the signal X and then is inactive. The thread 1 throws an exception, which terminates the thread 1 and thread 2 at their join point, *i. e.*, the $\mathsf{JOIN}_{\mathsf{L11}}$ instruction of the thread 0. Note the thread 3, which is nested in the thread 2, is inactive and will not be scheduled in this tick. Hence it will not receive this exception and respond to it. In the next tick, the un-terminated thread 3 will be active and emits signal X again. This result is unexpected.

Of course, in this case, the solution could be very simple: modify the priority of the thread 1 to 2 to let the exception be thrown at first. Therefore, all other threads could respond to this exception. However, it could be difficult to arrange the priority of a thread when the thread nest structure is complex and multiple exceptions could be thrown. The KEP employs a special hardware mechanism, which is named *Join Review*, to solve this problem.

The process of the *Join Review* consists of two steps. During the execution period of a tick, all of the threads which stay at the join point will be recorded. If an exception is thrown and it terminates some other threads, a flag will be set to trigger the *Join*

```
% For the Trap Element
if Instr.Op = EXIT then
  % Executing the EXIT instruction
  Trap_EmptyTrapPoint.StartAddr = ExitInstr.StartAddr
  Trap_EmptyTrapPoint.EndAddr = ExitInstr.EndAddr
  Trap_EmptyTrapPoint.ParentThread = Thread_ThreadID.ParentThread
  % Writing parameters to the Trap Registers
end if

for n = 0 to N_TrapNum −1 do
  if Trap_n.ParentThread = ThreadID
    % When the control arrives the corresponding parent thread.
    Trap_n.ParentThread = Thread_ThreadID.ParentThread
    % Replacing the ParentThread parameter (inherited strategy)
  end if

  if (PC > Trap_n.StartAddr) and (PC < Trap_n.EndAddr) then
    % Judging the active status of Traps
    Trap_n.Active = true
  end if

  if PC = Trap_n.EndAddr or ClearTrap_n = true then
    % When control arrives the scope of a Trap or a Trap is judged to be cleared
    Trap_n.StartAddr = 0
    Trap_n.EndAddr = 0
    % Erasing the Trap.
  end if

  if Trap_n.EndAddr = 0 then
    % When a Trap is not occupied
    EmptyTrapPoint = n
    % Setting the EmptyTrapPoint as to point to an empty Trap
  end if
end
```

Figure 4.31: Algorithm for setting and clearing an exception.

Review. If the flag is true after the initial thread is inactivated, the KEP will check all recorded threads again. A thread which takes lower ID will be checked first. If the parent thread is terminated, all of its child threads are also terminated. Note that since the ID of a child thread must be larger than that of its parent thread, this process will terminate all levels of child threads of a terminated thread step by step.

For the example in Figure 4.33(b), the KEP will record thread 2 as the thread which waits at the join point. Note the exception kills thread 2 at the $\mathsf{JOIN}_{\mathsf{L11}}$, hence, the *Join Review* mechanism is triggered. When the initial thread is inactive, the KEP will drive the control jump to the $\mathsf{JOIN}_{\mathsf{L10}}$. At this point, the disabled status of thread 2 will cause the termination of thread 3. Hence, no signal will be output in the next tick. The execution traces in Figure 4.33(c) show the additional process of the *Join Review* mechanism.

```
% For the Trap Element
% Judging which exception should be covered for concurrency exceptions
tmpEndAddr = 0
tmpn = 0
tmpInitFlag = 1
for n = 0 to N_Trap −1 do
  % Initializing parameters for the following process
  if Trap_n.Active = true then
    if tmpParentThread = Trap_n.ParentThread or tmpInitFlag = 1 then
      % Current active exception and the previous selected one are
      % concurrency exceptions
      if (tmpEndAddr > Thread_n.EndAddr) and tmpInitFlag = 0 then
        % The previous selected exception is the outer one
        ClearTrap_n = true
        % Setting the clear flag of the current exception as true
      else
        % The current exception is the outer one
        if tmpInitFlag = 0 then
          ClearTrap_tmpn = true
          % Setting the clear flag of the previous selected exception as true
        end if
        tmpInitFlag = 0
        tmpEndAddr = Trap_n.EndAddr
        tmpn = n
        tmpParentThread = Trap_n.ParentThread
        % Setting the current exception as the selected one
      end if
    end if
  end

% Judging which exception should take priority for non-concurrency exceptions
tmpEndAddr = 0
tmpn = 0
tmpInitFlag = 1
for n = 0 to N_Trap −1 do
  % Initializing parameters for the following process
  if Trap_n.Active = true then
    if tmpEndAddr > Thread_n.EndAddr or tmpInitFlag = 1 then
      % The previous selected exception is the outer one,
      tmpInitFlag = 0
      tmpEndAddr = Trap_n.EndAddr
      tmpn = n
      tmpParentThread = Trap_n.ParentThread
      % The inner exception will be selected and responded to first
    end if
  end if
end

if tmpEndAddr > 0 then
  % There is an active exception
  Trap.Addr = tmpEndAddr
  Trap.Active = true
  % Mapping parameters of the active exception to ports of the Trap Element
else
  Trap.Addr = 0
  Trap.Active = false
end if
```

Figure 4.32: Algorithm for covering and handling exceptions.

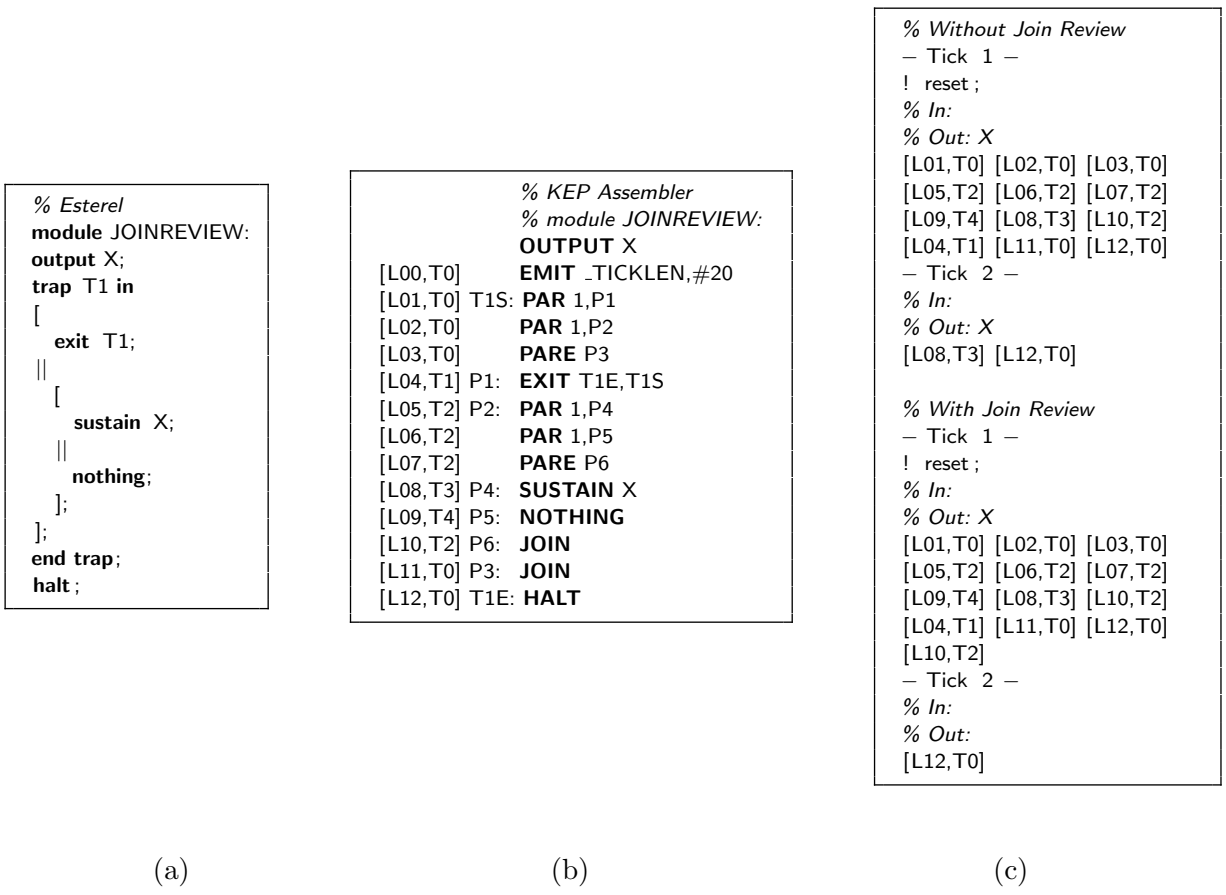


Figure 4.33: Algorithm for Join Review mechanism for handling exception.

4.2.3 Decoder & Controller

Similar to some other decoder blocks of processors, the Decoder & Controller block of the KEP accomplishes the *fetch*, *execution*, and *idle* functions as the execution steps of a processor. Furthermore, the structure of it also includes verbose "case branches" to handle various instructions and conditions. This thesis will not discuss those traditional implementations, which could be found in the standard literature [67, 4].

However, the Decoder & Controller block of the KEP has some novel features to meet the requirements of a reactive processor. In the previous sections, some functions of the Decoder & Controller were introduced. Some further features are explained in this section.

Before the Decoder & Controller executes an instruction, it will respond to some high priority events. First, due to the mechanism of the Thread Block of the KEP, a child thread should have higher or the same priority as its parent. However, for some program structures, it might be inefficient for the compiler to arrange the appropriate priority

value to satisfy this requirement. Hence, the KEP provides a mechanism to allow the priority of the parent thread to be higher than that of its child threads: when a thread executes a JOIN instruction (to check the status of its child threads) and at least one of its child threads is still active, the Decoder & Controller will inform the Thread Block to set the status of this thread as *inactive*. It does not mean that this thread will not be scheduled in this tick anymore. If any of its child threads' status changes from active to inactive/disabled, this thread will be *woken up* – its status will be set to active again.

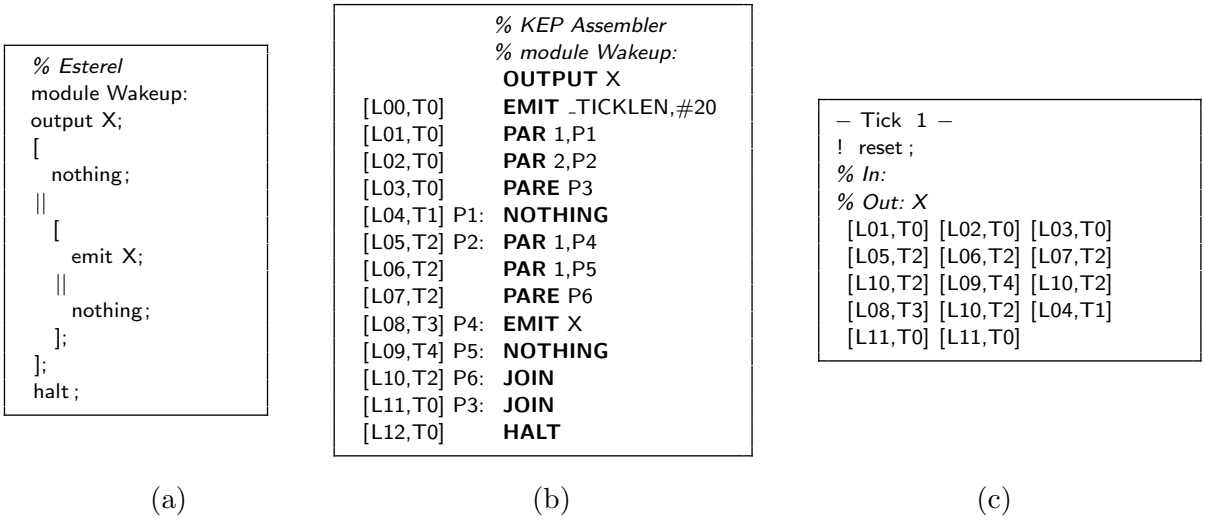


Figure 4.34: The example of waking up a thread.

An example in Figure 4.34(b) illustrates such a process. The priority of thread 2 is assigned to 2, which is the highest priority of all threads. Hence, after thread 2 has created its child threads, *i. e.*, thread 3 and thread 4, it is scheduled, and the $\mathsf{JOIN}_{\mathsf{L10}}$ instruction is executed. Since thread 3 and thread 4 are both active, thread 2 will become inactive. Then thread 4 executes $\mathsf{NOTHING}_{\mathsf{L09}}$ and terminates. This termination changes the status of thread 2 to be active. Hence, thread 2 is scheduled again. This process will continue until all its child threads are inactive or terminated. Figure 4.34(c) shows the execution trace of this program.

The Decoder & Controller handles other active preemptions in a certain order. Figure 4.35 shows a partial model of the Decoder & Controller, see [83] for details. Figure 4.36 describes the architecture of the Reactive Core of the KEP, which combines the aforementioned reactive blocks.

```
% For the Decoder & Controller
...
% Execution stage
if PreemptionElement.Type = suspension or \...
(Instr.Op = JOIN and ChildThreadInactive = false) then
  % When a suspension is active or a child thread of current thread is active
  DecoderAddr = PC
  TickFlag = false
  % The thread will be set as inactive
elsif PreemptionElement.Type = strong then
  % When a strong abortion is active
  DecoderAddr = PreemptionElement.Addr
  TickFlag = true
  % Responding to the preemption immediately
elsif rdAwait = true then
  % When a delay statement is terminated
  DecoderAddr = PC + 1
  TickFlag = true
  % Going through
else
  % A verbose case branches for executing various instructions
  ...
end if
```

Figure 4.35: Algorithm for the Decoder & Controller.

4.3 The Interface Block

The Interface Block is created as an interface layer of the reactive system. It supports the pre operation (introduced in Esterel V5.91), which allows to access the previous status and value of a signal, directly in hardware. There are two basic pre modes. One is the pre(S), which indicates the previous status of signal S, *i. e.*, its presence status in the previous instant. The other is pre(?S), which references the value of signal S in the previous instant. Figure 4.37 shows the architecture of an Interface Block.

In an earlier version [89, 88, 86], the KEP specified the number of the input and output signals, and connected a pair of input and output signals as an inner signal. Obviously, this architecture is easy to be handled, but is inefficient and inflexible. Hence, the KEP version 3a (KEP3a) [85] uses a Sinout port to present the input/output and inner signals together, *i. e.*, each bit of the Sinout port can be used as any kind of Esterel signal.

At the beginning of a tick, the KEP clears a signal status register (SinoutReg), and then stores the status of signals from the environment via the Sinout port. Hence, the status of input signals is sampled. During the period of a tick, the KEP could set or clear any bit of the SinoutReg to accomplish the function of signal emission or clearance (for inner signals). When the tick is finished, the content of the signal is mapped to Sinout, and the environment can access the status of any signal. Finally, before the next tick starts, the content of the SinoutReg will be loaded to the preSinoutReg as the pervious status of signals for corresponding pre operations in the oncoming tick.

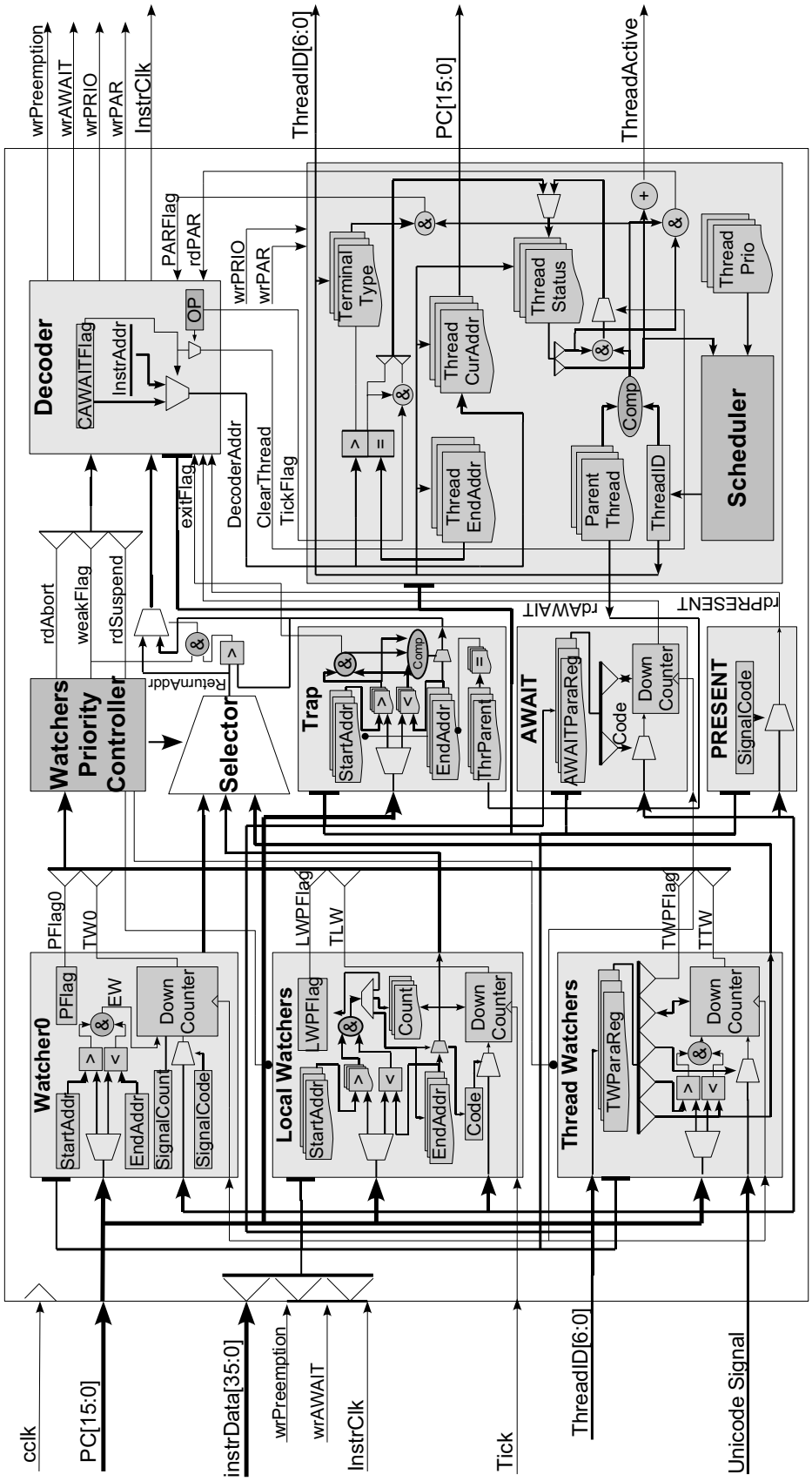


Figure 4.36: Architecture of the Reactive Core of the KEP.

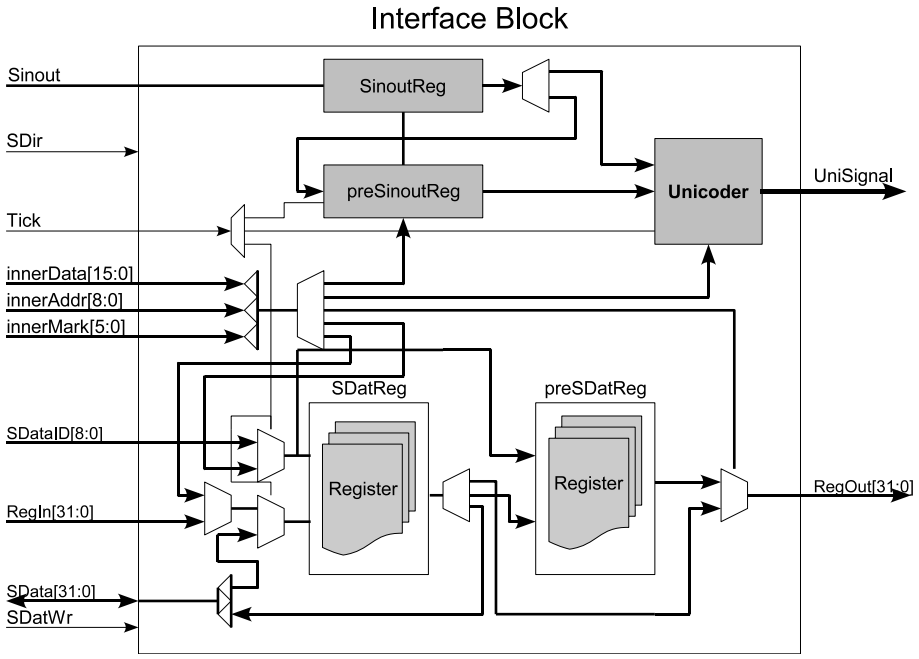


Figure 4.37: Architecture of an Interface Block.

As mentioned before, the UniSignal is the only signal bus of the KEP, and all input/output and inner signals are mapped to UniSignal. Figure 4.38(a) shows the interface definition of an example, and the corresponding signal codes are illustrated in Figure 4.38(b). The model of the UniSignal is visualized int Figure 4.39.

To satisfy the requirements of the implementation of Esterel control structures, *e. g.*, the watchers running concurrently, the bits of SinoutReg should be accessed in parallel. However, considering operations which are related to the carried data of a signal, it is unnecessary to fetch carried data of signals in parallel. Hence, carried data of signals are stored in a static random access memory (constituted by block RAM [72]) and can be accessed sequentially. Of course, the use of block RAM is an efficient way, but it raises the question how to handle the previous value of a signal, *i. e.*, pre(?S), since the method of handling statuses of signals could not be used.

The implementation of the Esterel pre(?S) benefits from one property of the signal values. Unlike the status of a signal, which will be cleared at the start of a tick [19], the value of a signal is kept until the next emission occurs. Hence, the pre(?S) could be implemented by sequential processes, too.

To illustrate the mechanism of the pre(?S) implementation, assume there is a valued signal A. In the KEP, a register $\mathsf{SDatReg}_A$ records the signal's value, and another register $\mathsf{preSDatReg}_A$ is employed to record the previous value of a signal. Furthermore, a flag

```
INPUT A, B
OUTPUT C
SIGNAL X
```

(a)

Signal	UniSignal	
	$d_8 - d_1$	d_0
_TICKLEN	00000000	0
TICK	00000000	0
A	00000001	0
B	00000010	0
C	00000011	0
X	00000100	0
PRE(A)	00000001	1
PRE(B)	00000010	1
PRE(C)	00000011	1
PRE(X)	00000100	1

(b)

Figure 4.38: The signal definition of an module (a), and the corresponding UniSignal codes of the signals (b).

```
% For the Interface Block
UniSignal_0 = true
UniSignal_1 = Tick.Pre
forall n ∈ N_SignalNum do
  % the SinoutReg and preSinoutReg signal are interlaced
  UniSignal_2n = SinoutReg_n
  UniSignal_2n+1 = preSinoutReg_n
end
```

Figure 4.39: Algorithm for building the UniSignal.

SDatRegFlag$_A$ is used to denote whether this signal has been emitted in the current tick or not.

Assume an EMIT A,#5 instruction was executed in a previous tick. Hence, the content of the SDatReg$_A$ is 5. When an instruction references ?A or PRE(?A), the content of the SDatReg$_A$ will be accessed. However, if another emission instruction, for example EMIT A,#3, emits the signal A with value 3, the previous value 5 will be stored in the preSDatReg$_A$ register, and the new value 3 will be saved in the SDatReg$_A$ register. At the same time, the SDatRegFlag$_A$ is also set to denote A was emitted in this tick. Now, if an instruction tries to get the value of ?A, the content of SDatReg$_A$ register (*i. e.*, 3) will be fetched; or else, if an instruction tries to get the value of PRE(?A), the content of the preSDatReg$_A$ register (*i. e.*, 5) will be accessed. Those processes are distinguished by the status of SDatRegFlag$_A$. Furthermore, if an instruction emits signal A in this tick

again (*e. g.*, for combined valued signals) [19], the content of $\mathsf{SDatReg}_A$ register will be modified but the $\mathsf{preSDatReg}_A$ register is not written.

To make the environment access the carried value, a port of the dual-port SRAM can be used by some outside circuit when a tick ends. Hence, the environment can read and write the value of corresponding signals when the Tick signal of the KEP is low, see also Chapter 5.

Figures 4.40 and 4.41 illustrate the behavior of the Interface Block.

```
% Definition of Parameters
% Sinout
  % The input/output port of the KEP. It also contains the inner signals.
% SinoutReg
  % A register to store the status of the input/output and inner signals.
% SinoutRegFlag
  % Denoting whether a signal was emitted in the current tick or not.
```

```
% For the Interface Block
if Tick.Start then
  % When a new tick starts
  preSinoutReg = SinoutReg
  % Loading the previous status of signals to the preSinoutReg register
  SinoutReg = false
  SinoutRegFlag = false
  % Clearing the signal and corresponding flags
  SinoutReg = Sinout
  % Reading the status of input signals from the environment
end if

if Tick.Finish then
  % When a tick finishes
  Sinout = SinoutReg
  % Putting the status of signals to the environment
end if
```

Figure 4.40: Algorithm for handling interface signals when a tick starts/finishes.

Esterel allows two variants of the valued signals. For *simple* valued signals, only one statement can emit the signal in an instant. For *combined* valued signals, multiple emitters are allowed. At the hardware level of the KEP, nothing can block a signal being emitted for multiple times in a tick. Hence, combined signals can be implemented simply by several KEP instructions. For example, Figure 4.42 illustrates a general translation method for a combined valued signal X: every X emission statement will be replaced by a sequence of instructions, and a temporary register X_REG is introduced. Whenever an X should be emitted, the KEP tests the status of signal X at first. If this signal is absent, the KEP will emit it with new value specified in the instruction; or else, the KEP will combine the value of signal X with the new value together into the temporary register, and then emits signal X with the content of the register.

```
% Definition of Parameters
% Instr.ExtOp.Reg
  % Extended operation code to denote whether the instruction is handling
  % a register in the Data Handling Block or not.
% Instr.SignalCode.Pre
  % The "PRE" flag of the signal code. See also Section 3.5.
% SDatReg
  % The carried data of a signal.
% preSDatReg
  % The previous carried data of a signal.
% RegIn
  % It comes from the Data Handling Block and presents the content of a register.
% RegOut
  % It puts the carried data of a signal to the Data Handling Block.
```

```
% For the Interface Block
if Instr.Op = EMIT or Instr.Op = SUSTAIN or Instr.Op = SETV then
  % Executing the signal emission or initializing instruction
  if Instr.Op != SETV then
    % Executing the signal emission instruction
    SinoutReg_{Instr.SignalCode} = true
    % Setting the presence of the signal as true
    if SinoutRegFlag_{Instr.SignalCode} = false then
      % If it is the first emission of this signal in the current tick
      SinoutRegFlag_{Instr.SignalCode} = true
      preSDatReg_{Instr.SignalCode} = SDatReg_{Instr.SignalCode}
      % The previous value of the signal will be pushed to the
      % corresponding preSDatReg register
    end if
  end if
  % Writing the carried data of the signal
  if Instr.ExtOp.Reg = true
    % The data comes from a register (e.g., EMIT A,Reg0)
    SDatReg_{Instr.SignalCode} = RegIn
  else
    % The data comes from the immediate data (e.g., EMIT A,#5)
    SDatReg_{Instr.SignalCode} = Instr.data
  end if
elsif Instr.Op = SIGNAL then
  % Handling the local signal
  SinoutReg_{Instr.SignalCode} = false
  preSinoutReg_{Instr.SignalCode} = false
  % Clearing previous/current status of the signal
end if

% Put the carried data of a signal to the RegOut port
if Op.SignalCode.Pre = true then
  % An instruction tries to fetch the pervious value (e.g., LOAD Reg0,PRE(?S))
  if SinoutRegFlag_{Instr.SignalCode} = false then
    RegOut = SDatReg_{Instr.SignalCode}
  else
    RegOut = preSDatReg_{Instr.SignalCode}
  end if
else
  RegOut = SDatReg_{Instr.SignalCode}
end if
```

Figure 4.41: Algorithm for executing the signal emission instruction.

```
% Esterel
output X := 0 : combine integer with +;
[
  emit X(5);
 ||
  emit X(3);
 ]
```

(a)

```
        % KEP Assembler
        OUTPUTV X
        VAR X_REG
        PAR 1,P1
        PAR 1,P2
        PARE P3
P1:     PRESENT X, P1A
        LOAD X_REG,?X
        ADD X_REG, #3
        GOTO P1B
P1A:    LOAD X_REG, #3
P1B:    EMIT X, X_REG
P2:     PRESENT X, P2A
        LOAD X_REG,?X
        ADD X_REG, #5
        GOTO P2B
P2A:    LOAD X_REG, #5
P2B:    EMIT X, X_REG
P3:     JOIN
```

(b)

Figure 4.42: Translating the Esterel combined valued signal (a) to the corresponding assembler code for KEP (b).

4.4 The Data Handling Block

The basic function of the Data Handling Block is very similar to the Arithmetic Logic Unit (ALU) of a processor. It performs all arithmetic computations, such as addition and multiplication, and all comparison operations. The implementation of the ALU is not a topic in this thesis. However, some features are also provided by the Data Handling Block and provide high-efficiency for the Esterel optimized signal and data handling. For example, the RegIn and RegOut ports, mentioned in Section 4.3, provide channels for communicating the Data Handling Block and the Interface Block.

Data Handling Block also provides the count value for the delay. Whenever a LOAD _COUNT,n instruction is executed, the value of n will be presented on the CountSpec.

However, assigning the count value for every instruction which has the delay expression is inefficient. Most of the delay/preemption instructions of an Esterel program just contain standard delays. Hence, the assigned data on the CountSpec will be kept for one instruction cycle to let the next instruction read it. Then the immediate data 1 will be presented on the CountSpec as the default data. In other words, for the standard delay expression, it is unnecessary to add a LOAD _COUNT, n instruction to assign a count value.

4.5 The Tick Manager and Energy Saving

One of the distinguishing features of the Kiel Esterel Processor is the Tick Manager, by which the KEP can autonomously ensure that logical ticks are computed at a fixed frequency. Furthermore, the Tick Manager internally monitors timing violations. The Tick Manager is activated by setting the pre-defined valued signal _TICKLEN to a certain value, typically at the beginning of the program. For the REINC example shown in Figure 3.14(b), the statement "EMIT _TICKLEN, #15" defines the length of a logical tick to be fifteen instruction cycles. Hence, if a tick is finished in less than _TICKLEN instruction cycles, KEP idles for the remaining cycles before starting the next tick. If, on the other hand, a tick is not finished within _TICKLEN cycles, this is considered a *tick length timing violation* [108]. As already described in Section 4.1, such timing violations are signaled to the environment via a special signal, TickWarn, with a dedicated output pin; this signal remains present until the next reset of the processor. As the KEP instruction cycles require a fixed number of clock cycles, providing a value for _TICKLEN alleviates the need for the environment to provide a timer that starts the ticks in regular intervals. Furthermore, the self-monitoring makes it easy for the environment to detect any timing violations. The Worst Case Reaction Time (WCRT) analysis presented in [86, 28, 27] arms to determine a conservative, yet tight value for _TICKLEN.

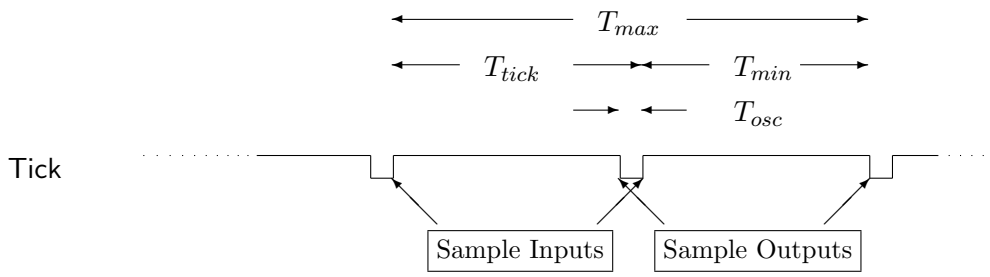


Figure 4.43: A waveform of the Tick signal and derived values.

To understand how a given value for _TICKLEN translates to concrete reaction times, we now consider the KEP instruction timing and signal sampling. Let T_{osc} be the basic clock rate supplied to the processor (via the Oscclk pin). The KEP signals the passage of logical ticks to the environment via a Tick signal, with a corresponding output pin. As illustrated in Figure 4.43, the KEP samples its inputs at rising edges of Tick. Furthermore, the KEP holds all outputs generated during a tick until the end of the tick, so the falling edges of Tick indicate when the environment should sample the outputs generated by the KEP. After the falling edge, Tick remains low for a gap of width T_{osc} before it rises again.

Let T_{react} be the time from the occurrence of an input signal (or combination thereof) until the generation of a corresponding output signal (or combination thereof). Assuming

that the environment may generate inputs at arbitrary times and that the KEP is computing reactions on its own, regular schedule, T_{react} may vary within an interval:

$$T_{min} \leq T_{react} < T_{max}. \tag{4.1}$$

Here, the lower bound T_{min} is determined by the time it takes to actually compute a reaction, *i. e.*, the length of a tick—which is the time from a rising edge of Tick to a falling edge of Tick. This reflects the case where an input is sampled *immediately* when it is generated; *i. e.*, an input happens to occur just when the KEP starts to compute a reaction. Let $V_{ticklen}$ be the maximal number of instructions that must be executed to compute a tick. Each KEP instruction takes three clock cycles. When a tick is finished, the Tick signal will be set to low for one instruction cycle. Hence, the lower bound on the reaction time can be computed as follows:

$$T_{min} = (3V_{ticklen} + 1) \times T_{osc}. \tag{4.2}$$

The upper bound T_{max} reflects the case when an input occurs just after inputs have been sampled, in which case the sampling of this input is delayed by the length to compute a tick, given by T_{min}, plus the gap T_{osc}. We denote the interval from one rising edge of Tick to the next rising edge by T_{tick} and obtain:

$$T_{tick} = T_{min} + T_{osc}, \tag{4.3}$$

$$T_{max} = T_{min} + T_{tick} (= (6V_{ticklen} + 3) \times T_{osc}). \tag{4.4}$$

For example, we obtain for a clock rate of $T_{osc} = 50ns$ and $V_{ticklen} = 8$ the following range for the reaction times:

$$1.3\mu s \leq T_{react} < 2.55\mu s. \tag{4.5}$$

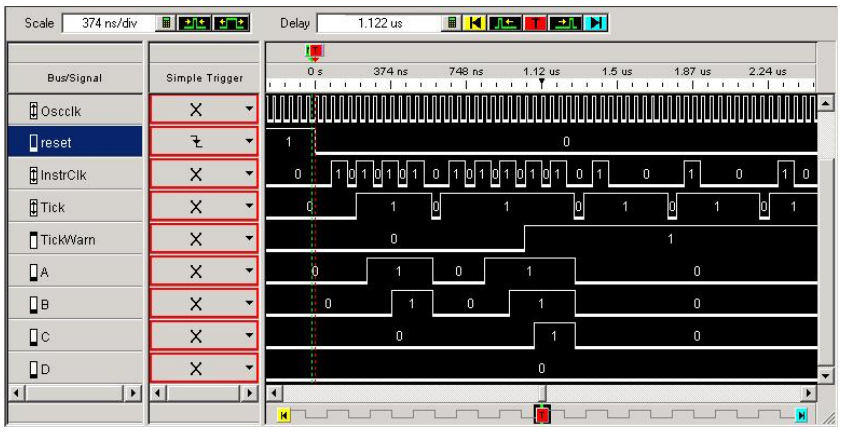
Figure 4.44 illustrates the KEP timing behavior for a small example. For the module OVERRUN in Figure 4.44(a) and some given input scenario (input signal D always absent), the KEP produces the timing shown in Figure 4.44(b). In this example, the program is running on a KEP implemented on a Memec V2MB1000 Development Board at a rate of $T_{osc} = 41.67ns$, the waveform was recorded by an Agilent 1683A Logic Analyzer. In OVERRUN, the first EMIT statement sets _TICKLEN to three; in other words, the module claims that $V_{ticklen}$, the maximal number of instructions executed within a tick, is at most three. If _TICKLEN is larger than $V_{ticklen}$, it means that the ticks are longer than necessary to finish the computations before the next tick starts; if _TICKLEN is smaller than $V_{ticklen}$ it means that we run the risk of timing violations.

Setting _TICKLEN to some value activates the Tick Manager. This from then on will on the one hand ensure that ticks which complete in less than, for example, three instructions will be padded until they are three instruction cycles long. On the other hand it will signal a timing violation if a tick is not completed within three instructions. In the example, the first logical tick lasts three instruction cycles. In the second tick,

```
% KEP Assembler
% module OVERRUN
INPUT D
OUTPUT A,B,C

EMIT _TICKLEN, #3
EMIT A
EMIT B
PAUSE
EMIT A
EMIT B
EMIT C
AWAIT D
```

(a)



(b)

Figure 4.44: An example KEP assembler code illustrating the Tick Manager (a), and a resulting timing diagram (b).

the controller has to execute five instructions until the AWAIT statement is executed. Hence, the TickWarn signal will be set high when the fourth instruction cycle is executed to indicate the tick length timing violation.

The goal of the WCRT analysis is to automatically deduce a value for _TICKLEN that is just large enough to never induce a timing violation; ideally, we achieve _TICKLEN = $V_{ticklen}$.

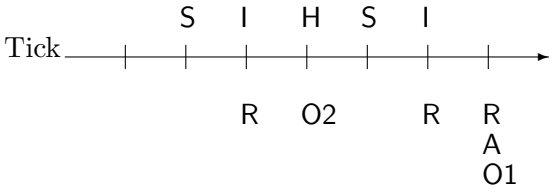
For controller programming, the main goal of Esterel, the control signals tend to be more often absent than present [19]. The condition of all signals being absent is called a *blank event*. For a reactive processor, very few instruction cycles are required for executing a blank event. To utilize this advantage of the KEP, when less than _TICKLEN instructions have been executed and there are no instructions needed for the current tick, *i. e.*, all threads are in inactive status, an IDLE signal will be broadcast to gate the clock of other elements for power reduction [8, 9, 97, 70].

4.6 Putting It All Altogether

To study how the KEP combines concurrency and preemption, it is instructive to work through the example code in Figure 4.45(a) (quoted from Figures 3.20(c) and 3.8(b)), and Figure 4.45(a) gives the trace of execution.

After starting the module, the initial thread (thread 0) is enabled and active. The EMIT _TICKLEN,#20$_{L00}$ instruction sets the length of a logical tick to twenty instruction cycles. The AWAIT S_{L01} configures the AWAIT Cell of thread 0, then this delay instruction clears

```
                    % KEP Assembler
                    % module EXAMPLE:
                    INPUT S, I, H
                    OUTPUT O1, O2
                    SIGNAL A, R
[L00,T0]            EMIT _TICKLEN, #20
[L01,T0]            AWAIT S
[L02,T0] A0:        ABORT S, A1
[L03,T0] TP1S: TP2S: PAR 3, P1
[L04,T0]            PAR 2, P2
[L05,T0]            PARE P3
[L06,T1] P1:        AWAIT I
[L07,T1]            WABORTI A, A2
[L08,T1]            SUSPEND H, A4
[L09,T1] A3:        EMIT R
[L10,T1]            PRIO 1
[L11,T1]            PRIO 3
[L12,T1]            PAUSE
[L13,T1]            GOTO A3
[L14,T1] A4: A2:    EMIT O1
[L15,T1]            EXIT TP1E,TP1S
[L16,T2] P2:        LOAD _COUNT, #2
[L17,T2]            AWAIT TICK
[L18,T2]            PRESENT R, A5
[L19,T2]            EMIT A
[L20,T2] A5:        EXIT TP2E,TP2S
[L21,T0] P3:        JOIN 0
[L22,T0] TP2E:      EMIT O2
[L23,T0] TP1E:      HALT
[L24,T0] A1:        GOTO A0
```



(a)

```
% Execution trace
– Tick 1 –
! reset ;
% In:
% Out:
[L00,T0] [L01,T0]
– Tick 2 –
% In: S
% Out:
[L01,T0] [L02,T0] [L03,T0] [L04,T0] [L05,T0]
[L06,T1] [L16,T2] [L17,T2] [L21,T0]
– Tick 3 –
% In: I
% Out: R
[L06,T1] [L07,T1] [L08,T1] [L09,T1] [L10,T1]
[L17,T2] [L11,T1] [L12,T1] [L21,T0]
– Tick 4 –
% In: H
% Out: O2
[L12,T1] [L17,T2] [L18,T2] [L20,T2] [L21,T0]
[L22,T0] [L23,T0]
– Tick 5 –
% In: S
% Out:
[L23,T0] [L24,T0] [L02,T0] [L03,T0] [L04,T0]
[L05,T0] [L06,T1] [L16,T2] [L17,T2] [L21,T0]
– Tick 6 –
% In: I
% Out: R
[L06,T1] [L07,T1] [L08,T1] [L09,T1] [L10,T1]
[L17,T2] [L11,T1] [L12,T1] [L21,T0]
– Tick 7 –
% In:
% Out: R A O1
[L12,T1] [L13,T1] [L09,T1] [L10,T1] [L17,T2]
[L18,T2] [L19,T2] [L20,T2] [L11,T1] [L12,T1]
[L14,T1] [L15,T1] [L21,T0] [L23,T0]
```

(b)

Figure 4.45: Execution the EXAMPLE program.

the TickFlag false to denote that this thread should be inactive. Although none of the threads are active, there is only one instruction cycle spent since the tick has started. Hence, this tick should be automatically extended until the remaining nine instruction cycles have passed after the IDLE signal is emitted to idle function blocks of the KEP to save energy.

For the example input trace, in the second tick, when the KEP fetches the AWAIT S_{L01}, the signal code of this instruction ($Instr.SignalCode$) makes the status of the S to be mapped to the rdPRESENT signal. Hence, the presence of rdPRESENT terminates AWAIT S_{L01}. Next, the ABORT S, $A1_{L02}$ configures the $Watcher_0$ to watch the signal S. The StartAddr of the $Watcher_0$ (abbreviated as W_0.StartAddr below) is $A0_{L02}$, and the W_0.EndAddr is $A1_{L24}$. Due to the nature of this instruction, the preemption type of the $Watcher_0$ (W_0.PreemptionFlag) is *strong*.

The following PAR/PARE instructions create two new threads. Thread 1 gets the initial priority 3. The local program counter ($ThreadCurAddr_1$) of thread 1 points to the upper bound of the thread scope ($P1_{L06}$). The lower bound of this thread ($P2_{L16}$) is stored in the corresponding $ThreadEndAddr_1$ register. Similarly, thread 2 gets the lower priority 2, and its range is from $P2_{L16}$ to $P3_{L21}$. Note that the created sub threads will not be scheduled until the $PARE_{L05}$ instruction ends the configuration of the last concurrent thread. Furthermore, the PARE $P3_{L05}$ instruction also makes the $ThreadCurAddr_0$ point to JOIN 0_{L21}. Now all active threads, *i. e.*, threads 0, 1, and 2, are handled by the Scheduler. Since thread 1 has the highest priority, it is scheduled first.

For thread 1, the non-instantaneous statement AWAIT I_{L06} causes it to become inactive, hence thread 2 is scheduled. At first, the *counter specification* instruction LOAD _COUNT,$\#2_{L16}$ loads the counter value for the next instruction, and then the AWAIT $TICK_{L17}$ configures the AWAIT Cell of thread 2 with the delay count 2, and then causes this thread to become inactive. The last active thread, *i. e.*, thread 0, executes the $JOIN_{L21}$ instruction to check the statuses of its incoming branched threads. Since those two threads are still enabled, the $JOIN_{L21}$ will not be terminated. Therefore, thread 0 turns to inactive, and the current tick is finished because all of threads are inactive.

When the third tick starts, all enabled threads are activated again. The Scheduler again starts with thread 1. The presence of signal I terminates AWAIT I_{L06}. Next, the WABORTI A, $A2_{L07}$ makes the $Watcher_1$ immediately watch the signal A; and SUSPEND H,$A4_{L08}$ configures the third watcher, *i. e.*, the $Watcher_2$, to be sensitive to signal H. Then the signal R is set by the EMIT R_{L09} instruction. The execution continues and the priority setting instruction PRIO 1_{L10} changes the priority of this thread (thread 1) to 1. Note that at this point the priority of thread 2 is 2, which is larger than that of thread 1, hence, the Scheduler pauses the execution of thread 1 and switches to thread 2. However, the address of the PRIO 3_{L11} – the next instruction of the PRIO 1_{L10}–will be stored at the $ThreadCurAddr_1$, and the status of thread 1, *i. e.*, active and enabled, is still kept.

When thread 2 turned to inactive in the previous tick, its $ThreadCurAddr$ pointed to the AWAIT $TICK_{L17}$. Hence, this instruction is executed. Remember that the count value of this AWAIT Cell was configured as 2 by the LOAD _COUNT, $\#2_{L16}$. At this time, the count value is decremented to 1. Since it does not equal zero, thread 2 will be inactive again. Hence, thread 1 resumes from the PRIO 3_{L17} instruction, which ensures that thread 1 is scheduled before thread 2 in the subsequent tick, before it becomes deactivated by $PAUSE_{L18}$.

In the fourth instant, the presence of signal H triggers the Watcher_2. Hence, the Preemption Element informs the Decoder & Controller that there is an active suspension. It causes the Decoder & Controller further to indicate to the Thread Block that the current thread should be inactive. These processes follow the description of those blocks, which are shown in Figure 4.26, 4.28, and 4.35.

Hence thread 1 is inactive but still enabled. Thread 2 terminates the $\text{AWAIT TICK}_{\text{L17}}$ because two ticks are escaped. The following signal test instruction $\text{PRESENT R,A5}_{\text{L18}}$ branches to A5 since signal R is not emitted. Then the $\text{EXIT TP2E,TP2S}_{\text{L20}}$ instruction configures the trap-handler cell Trap_0. The target address of the EXIT (TP2E_{L22}) is transferred to the Thread Block as the preferred address of the Decoder & Controller via the DecoderAddr port. However, it is greater than the end address of the current thread (P3_{L21}), hence, thread 2 is disabled by such a cross-boundary operation. Now only thread 0 is active, it executes the JOIN_{L21} instruction to test status of its sub threads. All child threads at the join point are terminated because there is an active exception. *i. e.*, at the join point, thread 1 is killed. Now all of the incoming branch threads are disabled, the control responds to the exception by jumping to TP2E_{L22}. Since the control arrives at the end of the Trap_0 scope, contents of the Trap_0 are immediately cleared. Then the O2 signal is emitted, and the control is halted by the HALT_{L23} instruction.

At the next tick, the disabled thread 1 and 2 will not be scheduled, and control starts from the terminated HALT_{L23} instruction. As S is present, the Watcher_0 is triggered. Since this is a strong abortion, the controller responds to it immediately. The W_0.EndAddr (A1_{L24}) is obtained as the program counter of the KEP. The control jumps to GOTO A0_{L24} and then continues as in the second tick again.

Similarly, in the seventh instant, the $\text{PRIO 1}_{\text{L10}}$ instruction causes the thread to be paused, thread 2 resumes from $\text{AWAIT TICK}_{\text{L17}}$ and executes $\text{PRESENT R,A5}_{\text{L18}}$ to test the presence of signal R. Since the signal R was emitted by thread 1, the PRESENT instruction will not cause the branch, and $\text{EMIT A}_{\text{L19}}$ is executed. Note that the Watcher_1 will not be triggered because the program counter (PC) of the KEP is outside of the watching scope of Watcher_1. Next, the $\text{EXIT TP2E,TP2S}_{\text{L20}}$ configures the Trap_0 and terminates thread 2. Hence, the control is handed over to thread 1 again. Note that the program counter is in the watching range of the Watcher_1, which is triggered by A. As this is a weak abort, the abort body is still executed until a delay type instruction is reached. Now the Trap_0 is still active. However, as mentioned in Section 4.2.2, the weak abort will get the priority because its body is nested in that trap. That is, the $\text{PRIO 3}_{\text{L11}}$ is executed, then the $\text{PAUSE}_{\text{L12}}$ is fetched. Since it is a non-instantaneous statement, it will be ignored, and control leaves the abortion block. Therefore, after the signal O1 is emitted, the $\text{EXIT TP1E,TP1S}_{\text{L15}}$ instruction, which corresponds to the exception TP1, configures the Trap_1. Note that exceptions TP1 and TP2 are thrown by thread 1 and thread 2, which have the same parent thread. Hence, the Trap_0—the inner one—is cleared immediately, following the rule mentioned in Figure 4.32. Hence, at the join point, only the second exception exists. The control responds to the exception, jumps to

$\mathsf{TP1E}_{\mathsf{L23}}$ and then halts. To conclude, the result of the KEP assembler program exactly corresponds to the expected result of the Esterel module.

4.7 Summary

This chapter has presented the architecture of the KEP. It provides a complete, semantically accurate implementation of all of the Esterel primitives, which include delay, concurrency, preemption, and exception.

A key concept realized in this architecture is that it offers concurrency orthogonally to the other reactive control flow behaviors, rather than providing concurrency on top of reactive behavior as is done in the multiprocessing approach. This is achieved by combining a single, sequential processing engine with separate control flow units for concurrency, preemption, signal testing, etc., which freely interact with each other according to the Esterel semantics.

Unlike patched processor solution, the KEP is a real reactive processor, which has a reactive kernel and adapted semi-custom (scalable) peripheral elements. It follows the exact definition of Esterel, processes valued signal and counter directly, and has configurable peripheral elements for making a processor series.

Chapter 5

Experimental Results

This chapter first describes the design flow of the KEP implementation of Esterel modules and the KEP Evaluation Platform, in Section 5.1. Then a comparison between the KEP and other execution platforms, such as the MicroBlaze, is given in Section 5.2. Section 5.3 represents the detailed experimental results for the KEP.

5.1 The KEP Evaluation Platform

To validate the correctness of the KEP and its compiler and to evaluate its performance, we employ an evaluation platform whose structure is shown in Figure 5.1. The user interacts via a host work station with an FPGA Board, which contains the KEP as well as some testing infrastructures.

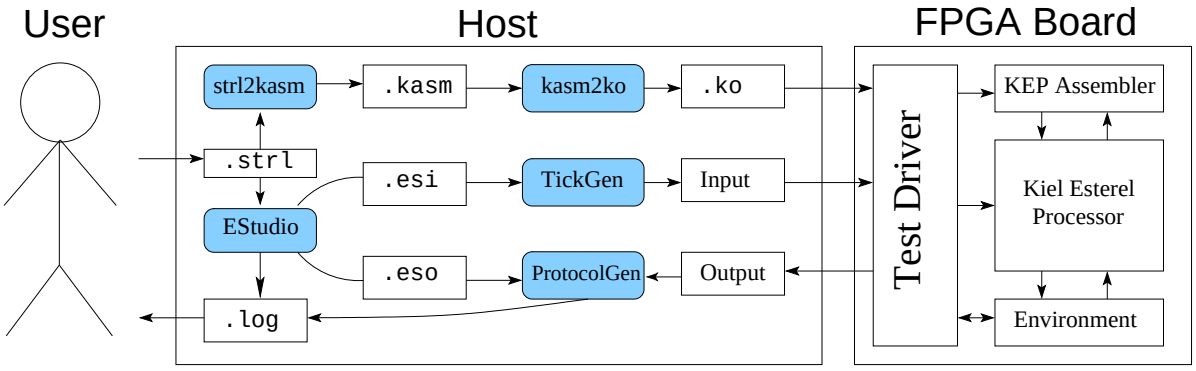


Figure 5.1: Structure of the KEP evaluation platform.

The implementation of an Esterel module on the KEP consists of several steps, which include compilation, implementation, and validation.

5.1.1 Compilation

Currently, the compiling process of the KEP is accomplished by two compilers: an Esterel module is compiled to a KEP assembler (.kasm) program by the *KEP (Esterel) compiler* (strl2kasm), and then the *KEP Assembler Compiler* (kasm2ko) compiles the assembler to the binary executable object code (.ko) of the KEP. Figure 5.2(a) illustrates the Esterel EXAMPLE module (cited from Figure 3.8(a)), the KEP assembler program generated by strl2kasm is shown in Figure 5.2(b), and Figure 5.2(c) shows the corresponding KEP machine code listing.

The strl2kasm compiler is based on the CEC compiler 0.3 (CEC) infrastructure [43].

The KEP Assembler Compiler (kasm2ko) in turn consists of two parts, the *KEP Machine Code Generator* (kasm2klst), and the *KEP Object Code Generator* (klst2ko). The KEP Machine Code Generator is responsible for refining the preemption instructions, to make optimal use of the different watcher types (see Section 4.2.2). The KEP Machine Code Generator will analyze the nesting relation of each preemption, and map the preemption to subclasses of the Watcher to be executed, *i. e.*, replace original [W]ABORT[I] instructions with refined T|L[W]ABORT[I] instructions. For this function, the compiler scans all abortion instructions in the program twice. At the first scanning, the compiler scans all abortion instructions. For each [W]ABORT[I] instruction, if there is neither a PAR/PARE instruction nor preemption instruction in its address range, it will be replaced with the T[W]ABORT[I], which denotes this abortion should be mapped to a TWatcher. For the second scanning, the remaining [W]ABORT[I] instructions will be analyzed again. If there is no [W]ABORT[I] instruction in its address range, it will be replaced with the L[W]ABORT[I]. Considering the address ranges of the nested abortions, this replacement strategy provides address ranges of L[W]ABORT[I]s that are exclusive.

5.1.2 Implementation

Currently, the KEP is written in VHDL [83]. Those VHDL programs are synthesized and implemented by the Xilinx ISE version 6.3.3, and all options use default values. Then the FPGA programming file can be downloaded to the corresponding FPGA chip. The content of the instruction memory, *i. e.*, the object code of a KEP program, can be written into the instruction memory and further downloaded to the FPGA together, or can be updated later.

The blocks of the KEP are scalable, so they can be reduced or extended for making a processor series. To generate the different configurations of the KEP, a program is developed to generate appropriate VHDL codes with given parameters. This program is integrated in the KEP assembler compiler. For details on its options, see Appendix B.1.

In addition to this, some additional hardware/software was also developed for the evaluation platform. On the host side, the KEP evaluation program, developed with Visual Basic 6.0, provides a graphical tool for running and debugging a KEP program. On the

```
% Esterel
module EXAMPLE:
input S, I, H;
output O1, O2;
signal A,R in
every S do
  trap T1 in
    trap T2 in
      [ await I;
        weak abort
          suspend
            sustain R;
          when H;
        when immediate A;
        emit O1;
        exit T1;
      ||
        await 2 tick;
        present R then
          emit A;
        end present;
        exit T2; ];
    end trap;
    emit O2;
  end trap;
end every;
end signal
end module
```

(a)

```
% Esterel LOC: 26
% Esterel LOC (expanded): 26
% Esterel LOC (expanded+kepdismantled): 49
%%% Esterel Module: EXAMPLE

%%%------I/O SIGNALS------
INPUT S,I,H
OUTPUT O1,O2
%%%------TOP LOCAL SIGNALS------
SIGNAL A,R
%%%------REGISTERS------
VAR COUNT
%%%------INTERFACE STATEMENTS------
EMIT _TICKLEN,#21

[L01,T0]                 AWAIT S
[L02,T0,P1]   A2:        ABORT S,A3
[L03,T0,P1]   A4:A5:     PAR 1,A6,1
[L04,T0,P1]              PAR 1,A7,2
[L05,T0,P1/1]            PARE A8,1
[L06,T1,P1]   A6:        ABORT I,A9
[L07,T1,P3]   A10:       PRIO 3
[L08,T1,P1/3]            PAUSE
[L09,T1,P1]              PRIO 1
[L10,T1,P1]              GOTO A10
[L11,T1,P3]   A9:        WABORTI A,A11
[L12,T1,P3]              SUSPEND H,A12
[L13,T1,P3]   A13:       EMIT R
[L14,T1,P2]              PRIO 2
[L15,T1,P3]              PRIO 3
[L16,T1,P2/3]            PAUSE
[L17,T1,P3]              GOTO A13
[L18,T1,P1]   A12:       NOTHING
[L19,T1,P2]   A11:       EMIT O1
[L20,T1,P2]              EXIT T1,A4
[L21,T2,P1]   A7:A14:    LOAD COUNT,#2
[L22,T2,P2]   A15:       PRIO 2
[L23,T2,P1/2]            PAUSE
[L24,T2,P2]              PRESENT TICK,A16
[L25,T2,P2]              SUB COUNT,#1
[L26,T2,P2]   A16:       CMPS COUNT,#0
[L27,T2,P2]              JW LE,A17
[L28,T2,P2]              EXIT AWAIT_DELAY,A14
[L29,T2,P1]   A17:       PRIO 1
[L30,T2,P1]              GOTO A15
[L31,T2,P2]   AWAIT_DELAY: PRESENT R,A18
[L32,T2,P2]              EMIT A
[L33,T2,P2]   A18:       EXIT T2,A5
[L34,T0,P1]   A8:        JOIN 0
[L35,T0,P1]   T2:        EMIT O2
[L36,T0]      T1:        HALT
[L37,T0,P1]   A3:        GOTO A2
```

(b)

```
%--------------------------------
% Generated by KEP4 Assembler Compiler Version 4.20
% Original file : EXAMPLE.opt.kep.kasm
%--------------------------------
% Esterel LOC: 26
% Esterel LOC (expanded): 26
% Esterel LOC (expanded+kepdismantled): 49
%%% Esterel Module: EXAMPLE
%%%-----I/O SIGNALS-----
INPUT S I H
OUTPUT O1 O2
VAR COUNT
% Signal codes
% Input ports (include local signals)
% [000000010] I/O(#1) S
% [000000100] I/O(#2) I
% [000000110] I/O(#3) H
% [000001100] I/O(#6) A %signal
% [000001110] I/O(#7) R %signal
% Output ports (include local signals)
% [000001000] I/O(#4) O1
% [000001010] I/O(#5) O2
% [000001100] I/O(#6) A %signal
% [000001110] I/O(#7) R %signal
% Variable
% [0000000010]    (1)   COUNT
%
% Summary:
% Input signals   :  3 (Pure: 3, Valued: 0)
% Output signals  :  2 (Pure: 2, Valued: 0)
% Local signals   :  2 (Pure: 2, Valued: 0)
% Variables       :  1
% RAM Usage (in byte): 4
% Code size (in byte): 171
% Code size (in word): 38
% Watchers needed:       3
% LWatchers needed:      0
% Preemption by TWatcher: 1
% Watcher Num if no L|TWatcher: 4
% Threads needed:  2
%
% Instruction code:
%--------------------------------
% Addr {Hex code} Label: Mnemonic
%--------------------------------
[00000] {40000003F}     EMIT _TICKLEN, #21 %T0 P1
[00001] {080800000}     AWAIT S %T0
[00002] {100800025} A2: ABORT S, A3 %T0 P1
[00003] {480810006} A4: A5: PAR 1, A6 (, 1) %T0 P1
[00004] {480820015}     PAR 1, A7 (, 2) %T0 P1
[00005] {480800022}     PARE A8, 1 %T0 P1/1
[00006] {70100000B} A6: TABORT I, A9 %T1 P3
[00007] {481800000} A10: PRIO 3 %T1 P3
[00008] {080000000}     PAUSE %T1 P1/3
[00009] {480800000}     PRIO 1 %T1 P1
[00010] {000010007}     GOTO A10 %T1 P1
[00011] {283010013} A9: WABORTI A, A11 %T1 P2
[00012] {301820012}     SUSPEND H, A12 %T1 P3
[00013] {403800000} A13: EMIT R %T1 P3
[00014] {481000000}     PRIO 2 %T1 P2
[00015] {481800000}     PRIO 3 %T1 P3
[00016] {080000000}     PAUSE %T1 P2/3
[00017] {00001000D}     GOTO A13 %T1 P3
[00018] {000000000} A12: NOTHING %T1 P1
[00019] {402000000} A11: EMIT O1 %T1 P2
[00020] {D00030024}     EXIT T1, A4 %T1 P2
[00021] {A00400002} A7: A14: LOAD COUNT, #2 %T2 P1
[00022] {481000000} A15: PRIO 2 %T2 P2
[00023] {080000000}     PAUSE %T2 P1/2
[00024] {00004001A}     PRESENT TICK, A16 %T2 P2
[00025] {A80480001}     SUB COUNT, #1 %T2 P2
[00026] {B80480000} A16: CMPS COUNT, #0 %T2 P2
[00027] {80005001D}     JW LE, A17 %T2 P2
[00028] {D0015001F}     EXIT AWAIT_DELAY, A14 %T2 P2
[00029] {480800000} A17: PRIO 1 %T2 P1
[00030] {000010016}     GOTO A15 %T2 P1
[00031] {003840021} AWAIT_DELAY: PRESENT R, A18 %T2 P2
[00032] {403000000}     EMIT A %T2 P2
[00033] {D00030023} A18: EXIT T2, A5 %T2 P2
[00034] {000200000} A8: JOIN 0 %T0 P1
[00035] {402800000} T2: EMIT O2 %T0 P1
[00036] {080010000} T1: HALT %T0
[00037] {000010002} A3: GOTO A2 %T0 P1
%--------------------------------
```

(c)

Figure 5.2: The EXAMPLE Esterel program: (a) Esterel; (b) KEP Assembler; (c) KEP Machine Code Listing.

FPGA board, to test and control the KEP, the TestDriver communicates with the host via the KEP evaluation program. It receives the commands and data from the host, completes the action according to the request, then returns the result to the host, see also Appendix B.2.

Due to the cooperation of the KEP evaluation program and the TestDriver, the user can easily generate the input event, send the tick, view the output event, and so on, via a GUI. Appendix B.3 gives more information.

5.1.3 Validation

An Esterel module, which is translated into KEP object code, is also used to generate the coverage scenario (with state and transition coverage) via Esterel Studio V5.0 (EStudio) to validate the correctness of the KEP and its compiler. The result is expressed as an Esterel Simulator Input (.esi) file, which defines sequential input events for the test module. We also employ EStudio to simulate this scenario and obtain the output trace (.eso) file.

The KEP validation method is evaluation-based [91, 29, 6]. First the executable code is loaded to the KEP instruction memory, and the KEP is reset. For every tick, the KEP evaluation program reads the input information from the input trace file (.esi), encodes the information and then transmits it to the TestDriver. The TestDriver decodes the information and sets the corresponding input signal to high if it should be present. If the signal is a valued signal, its carried value, which is stored in the block RAM of the Interface Block, is also modified. After all input signals are set as requested, the TestDriver releases the block of the clock of the KEP. Hence, the KEP runs immediately, and the Tick signal rises to denote that the KEP enters a logic tick period. During this period, the TestDriver records the program counter of the KEP via the instruction address bus, counts the number of instruction cycles, and watches the Tick signal of the KEP. Once the Tick signal falls, the TestDriver blocks the oscillator clock of the KEP to freeze it. A message will be sent to the host to notify that the current tick is finished. The TestDriver will capture the status of pins and transmits them back. The host receives the outputs and compares them with the EStudio's eso file for validation. Some other information, *e. g.*, the elapsed instruction cycle, etc., is also transferred to be further analyzed. The above mentioned processes will be repeated until the KEP evaluation program reaches the end of the input trace file. To test a batch of Esterel modules automatically, a Linux bash shell program is employed as batch program to handle all above mentioned processes together.

5.2 Comparison with Other Execution Platforms

To quantitatively compare the data handling abilities between the Esterel processor and other implementations, we use the CURVE module (contained in the mca200 test

	KEP (16-bit)	KEP (32-bit)	Hardware (32-bit)	Hardware (16-bit)	MCS51(1) (8-bit)	MicroBlaze(2) (32-bit)
Slices	870	1253	755	484	-	953
Code size (words)	191	191	-	-	1070(3)	436
Code size (bytes)	859	859	-	-	1636	1744
RAM Usage (words)	12	12	-	-	31	19
RAM Usage (bytes)	24	48	-	-	31	76

(1) Compiled by Keil C51 compiler V6.12. The level 8 (default) optimization is used.
(2) Compiled by gcc (for MicroBlaze) version 2.95.3-4. The level 2 (default) optimization is used.
(3) The lengths of MCS51's instructions is various; here, a *word* represents a complete assembler line.

Table 5.1: The code size and RAM usage (in word) comparison of the CURVE implementation between KEP, MCS51, and MicroBlaze.

bench [35]) as an example, since it is a typical module that includes varied data handling statements. Table 5.1 compares the resource usages of the KEP with different hard- and software implementations. For the hardware implementations, we synthesize the module to VHDL with the Esterel V7 compiler, because other hardware compilers cannot support valued signals. The V7 compiler does not provide a data ranging function, *i. e.*, an integer type valued input signal will always occupy a 32-bit bus to represent the carried value[1]. As an optimization, we manually resize all of the valued signals and variables to 16-bit width, since that range is sufficient for this Esterel module. Then those VHDL programs are implemented by the ISEV6.3.03, and the speed (default) optimization is used. For the software implementation, we use the CEC V0.3 compiler to synthesize the module to a C program, which is then compiled onto the 32-bit MicroBlaze soft processor core, and the MCS51, which is a classical widely used 8-bit processor.

As to be expected, the logic usage of the 32-bit KEP is more than that of the traditional 32-bit processor MicroBlaze. This result could be ascribed to the complex architecture of the Reactive Core of the KEP. However, considering the KEP is a multi-threaded architecture and it contains various components for handling all Esterel primitive statements directly, such a cost seems acceptable.

As mentioned in Section 2.3, the RePIC/EMPEROR are typical prototypes of the patched processor/multi-processing architecture for handling the Esterel program directly. Table 5.2 compares different KEP variants which offer similar functions to them, and they can be compared with RePIC/EMPEROR directly. Regarding the logic cell count, one should note that the RePIC is implemented on an ALTERA' EP20K200EFC484-2 FPGA chip. Hence, the Xilinx's XC2S200-6FG456 FPGA chip is chosen for the KEP's implementation. The basic units of those two chips have similar structures, functions, and speeds. Therefore, we can assume that logic cell counts are comparable.

[1]In the latest version (EStudio V5.4), the V7 compiler provides a data ranging function.

	KEP-a (8-bit)	KEP-b (8-bit)	KEP-c (8-bit)	KEP-d (16-bit)	KEP-e (32-bit)	RePIC (8-bit)	EMPEROR (8-bit)
Input/Output	24[1]	48	48	48	48	12/12[2]	24/24[3]
Thread Number	2[4]	2	32	2	2	1	2
Preemption Nest	4	6	6	6	6	4	4+4
Counter Value Range	1	1	1	1	1	1	1
Register Number	64	128	128	128	128	64	64+64
Logic Cells	1796	2048	4820	2484	3588	2068	4761
Max Osc Freq (MHz)	52.90	46.61	46.48	46.61	46.61	40.27	35.38
Instruction Freq (MHz)	17.63	15.54	15.49	15.54	15.54	10.1	8.84

(1) To the KEP, each interface signal could be used as input or output arbitrarily, and every signal could carry the data (valued signal).
(2) Includes one valued signal.
(3) Includes two valued signal.
(4) Due to the multi-threaded architecture, the KEP cannot provide a one thread version.

Table 5.2: Performance comparison between the KEP3 and EMPEROR.

Furthermore, every RePIC can handle an abortion nest of depth 4, but due to the architecture of EMPEROR, those abort handling elements cannot nest between different processors directly. Hence the EMPEROR2 contains eight abort handling elements, but can only deal with abortion nests of depth four. As an approximation, we compare this with the KEP3-E that offers a preemption nesting depth of six.

The RePIC uses four clock cycles to execute an instruction cycle, but the KEP uses only three clock cycles. When running on the same clock frequency, the KEP's instruction cycle period is just 75% of that of the RePIC's. Furthermore, the KEP typically takes significantly less instructions to implement the same behavior [89, 88, 87].

As a result, for the similar processor configuration as the EMPEROR, the KEP3-b uses 57% less resources and achieves a 1.7 times instruction clock speedup. The resource usage of the EMPEROR is higher than the 32-bit KEP-e which supports similar functions. Furthermore, the KEP-c occupies almost the same resources as that of the EMPEROR, however, it contains 36 threads: 18 times than that of the EMPEROR.

To further assess the resource efficiency of the multi-threaded approach, we have generated different KEP versions with a maximal thread number varying between 2 and 120. Since there are many different ways to calculate the hardware resouces utilized in a design, we choose both the *slice*[2] and *equivalent gate count* as metrics for hardware usage [96]. All versions are configured with 2 Watchers, 8 Local Watchers, and 48 valued I/O signals. The clock rate does not vary significantly, it is around 60 MHz; one instruction takes three clock cycles. Table 5.3 shows the corresponding resources usages. The hardware usage increases only 4x as the concurrency increases 60x when measured in slices, and even just 1.4x when measured in equivalent gates. The implementation

[2]One slice equals two logic cells.

Max. threads	2	10	20	40	60	80	100	120
Slices	1295	1566	1871	2369	3235	4035	4569	5233
Gates (k)	295	299	311	328	346	373	389	406

Table 5.3: Extending a KEP to different threads.

is based on the Xilinx 3S1500-4fg676 FPGA. For comparison, the MicroBlaze with the same memory size (BRAM) employs 309k gates.

5.3 Evaluation Results

The KEP evaluation suite currently contains more than 400 Esterel examples. To evaluate the performance of the KEP, we have present eleven standard test cases [18, 6, 35]. These benchmarks are typical Esterel applications, which not only contain reactive statements, but also include arithmetic and logical data handling. However, we leave out programs that make use of the pre operator, since the CEC compiler currently does not support it [43].

- abcd, abcdef, eight_but
  The abcd is a four-button user interface that locks out other buttons while one is pressed [45]. Here it is also extended to 6-button (abcdef) and 8-button (eight_but) modules.

- Chan_Prot
  The Chan_Prot is a rendezvous channel protocol used in the Communicating Reactive Processes concurrency model [11]. The channel protocol communicates via asynchronous FIFOs with two independent programs that request for rendezvous, and these requests may also be withdrawn at any time [117].

- reactor_ctrl
  The reactor_ctrl describes a simplified model of the reactor safety guard, which push in the rods when the reactor is overheating.

- runner
  The runner is an example from the Esterel tutorial [18], and it provides a process of morning exercise and alerts when exceptions happen.

- example
  The example [90] mixes nests of threads and preemptions.

- ww_button
  The ww_button is a part of the model of a wrist watch. This module performs

Module Name	Esterel				KEP			
	LOC	Threads			LOC	CKAG		
		Count	Max depth	Max conc.	(word)	Dep. count	Max priority	PRIO instr's
abcd	160	4	2	4	164	36	3	30
abcdef	236	6	2	6	244	90	3	48
eight_but	312	8	2	8	324	168	3	66
chan_prot	42	5	3	4	62	4	2	10
reactor_ctrl	27	3	2	3	34	5	1	0
runner	31	2	2	2	27	0	1	0
example	20	2	2	2	28	2	3	6
ww_button	76	13	3	4	95	0	1	0
greycounter	143	17	3	13	343	53	6	58
tcint	355	39	5	17	379	65	3	20
mca200	3090	59	5	49	8650	129	11	190

Table 5.4: Concurrency analysis of benchmarks.

mode handling and dynamic button renaming. It only handles pure signals, so that there are only input/output declarations [12].

- greycounter
  It represents a four-bit grey-code counter with an alarm [112].

- tcint
  The tcint is a medium-size Esterel program, which implements a TurboChannel bus interface [112].

- mca200
  The mca200 is an industry size model which models a shock absorber. It is one of the largest examples that are publicly available [39, 50].

To compare the performance of the KEP (including its compiler) with another platform, we chose the MicroBlaze 32-bit soft COTS RISC processor core as a reference. The MicroBlaze embedded soft core is a RISC processor which is optimized for implementation in Xilinx FPGAs. We use the CEC compiler 0.3 [43], the Esterel Compiler V5.92 [52], and the Esterel Compiler V7 to synthesize Esterel modules to C programs, which are then compiled onto the MicroBlaze via the gcc version 2.95.3-4, using the default level 2 optimization.

Module Name	KEP	MicroBlaze		
	Compiling time (sec)	Compiling time (sec)		
		V5	V7	CEC
abcd	0.15	0.12	0.09	0.30
abcdef	0.21	0.71	0.46	0.96
eight_but	0.26	0.99	0.54	1.25
chan_prot	0.07	0.35	0.35	0.43
reactor_ctrl	0.06	0.29	0.31	0.36
runner	0.05	0.30	0.34	0.40
example	0.05	0.28	0.31	0.31
ww_button	0.10	0.44	0.40	0.64
greycounter	0.34	0.57	0.43	0.75
tcint	0.34	0.41	0.52	1.11
mca200	11.25	69.81	12.99	7.37

Table 5.5: Comparison of compilation time of the benchmarks.

Analysis of the Concurrency

To characterize each benchmark with respect to its use of concurrency constructs, Table 5.4 lists the counts and depths of them. For the KEP, the table shows the number of dependencies found, the used number of priority levels (the KEP provides up to 255), and the number of used PRIO instructions. One can find that in most cases, the maximum priority used is three or less, indicating relatively few priority changes per tick. For example, eight_buttons has 168 dependencies, but the maximum priority used is 3. On the other hand, greycounter, with 53 dependencies, requires a maximum priority of 6.

The main mission of the KEP compiler is to assign appropriate priorities of threads to ensure those threads can be executed in a proper order. Since the architecture of the KEP can handle Esterel structures directly, the compilation of those control structures is a straightforward mapping strategy. This keeps the compilation cost of the KEP compilers low. Table 5.5 shows the comparison of compilation time, from Esterel code to machine code. In most used benchmarks, compilation time is significantly less than a second, and quite competitive with any of the compilers for the MicroBlaze.

Analysis of the Preemption Character

Table 5.6 gives an overview of the Esterel preemption structure of those benchmarks. Typically, the preemption constructs tend to be sequential or concurrent rather than being nested. For example, the mca200 employs 64 preemption statements, however, the maximum depth of the preemption nest is just 4. To assess the usefulness of providing

Module Name	Esterel		KEP			
	Preemptions		Preemption handled by			
	Count	Max depth	Only Watcher		Local Watcher	Thread Watcher
abcd	20	2	7	0	3	11
abcdef	30	2	11	0	5	17
eight_but	40	2	15	0	7	23
chan_prot	6	1	2	0	0	4
reactor_ctrl	5	1	3	0	0	4
runner	9	3	6	2	1	3
example	4	2	2	0	1	2
ww_button	27	2	15	0	5	10
greycounter	19	2	9	0	4	15
tcint	18	2	7	0	1	10
mca200	64	4	61	2	14	48

Table 5.6: Preemption character analysis of benchmarks.

different types of watchers, as has been described in Section 4.2.2, Table 5.6 also lists how many Watchers are necessary if there are no other watcher types (the reuse of Watchers are also considered), and how many watchers of each type are required. As it turns out, most of the preemptions can be handled by the cheapest Watcher type, the Thread Watcher.

Based on the watcher requirements in Table 5.6, we configured the Reactive Core with different parameters, and then synthesized these Reactive Cores to study the difference of preemption mapping strategies. In most cases, the refined watcher strategy wins in the maximum frequency speedup and economical hardware usage, and its benefits are more distinct for large scale modules. For the industry size module, *i. e.*, the mca200 benchmark, this strategy reduces hardware usage by 36%, and raises the maximum frequency also by 36%. Another benefit of the refined preemption handling architecture is that it keeps the performance stable. The extension of the Reactive Core from the simplest one to the largest and the most complex preemption structure slows down the maximum frequency about 19% (from 112MHz to 90MHz). However, even a 40% decease of the maximum frequency (from 112MHz to 66MHz) can occur, if there are no refined watcher types.

Analysis of Context Switches

Table 5.8 illustrates the analysis of the context switch (CS) activities in the benchmarks, for some specific test traces. For example, in the ww_button benchmark, a total of 292

Module Name	Preemption handled					
	Only by Watcher		by various watchers			
	HW (Slices)	Freq. (MHz)	HW (Slices)	Freq. (MHz)	HW Reduce (%)	Freq. Speedup (%)
abcd	376	88.04	332	105.80	12	20
abcdef	506	84.33	370	103.70	27	23
eight_but	623	83.06	418	100.87	33	21
chan_prot	254	111.73	270	111.73	-6	0
reactor_ctrl	276	107.68	259	111.37	6	3
runner	354	99.44	320	97.44	10	-2
example	249	111.73	283	109.49	-14	-2
ww_button	629	82.71	383	100.87	39	22
greycounter	470	87.64	412	106.17	12	21
tcint	433	90.49	422	99.89	3	10
mca200	1253	66.49	798	90.20	36	36

Table 5.7: Effects on the Reactive Core's cost/performance of the various watchers architecture.

Module Name	Instr's total	CSs total		CSs at same priority		PRIOs total		CSs due to PRIO		
	abs. [1]	abs. [2]	ratio [1]/[2]	abs. [3]	rel. [3]/[2]	abs. [4]	rel. [4]/[1]	abs. [5]	rel. [5]/[2]	rel. [5]/[4]
abcd	16513	3787	4.36	1521	0.40	3082	0.19	1243	0.33	0.40
abcdef	29531	7246	4.08	3302	0.46	6043	0.20	2519	0.35	0.42
eight_but	39048	10073	3.88	5356	0.53	8292	0.21	3698	0.37	0.45
chan_prot	5119	1740	2.94	707	0.41	990	0.19	438	0.25	0.44
reactor_ctrl	151	48	3.15	29	**0.60**	0	0	0	0	-
runner	5052	704	**7.18**	307	0.44	0	0	0	0	-
example	208	60	3.47	2	**0.30**	26	0.13	9	0.15	0.35
ww_button	292	156	**1.87**	92	0.59	0	0	0	0	-
greycounter	160052	34560	4.63	14043	0.41	26507	0.17	12725	0.37	**0.48**
tcint	80689	33610	2.4	16769	0.50	5116	0.06	2129	0.06	0.42
mca200	982417	256988	3.82	125055	0.49	242457	**0.25**	105258	**0.41**	0.43

Table 5.8: Analysis of context switches (CSs), in absolute numbers and relative. Minimal and maximal relative values are shown **bold**.

Module Name	Esterel LOC [1]	MicroBlaze Code+Data (b) V5 [2]	MicroBlaze Code+Data (b) V7 [2]	MicroBlaze Code+Data (b) CEC [2]	KEP Code (w) abs. [3]	KEP Code (w) rel. [3]/[1]	KEP Code+Data (b) abs. [4]	KEP Code+Data (b) rel. [4]/[2]
abcd	160	**6680**	7928	7212	164	1.03	738	0.11
abcdef	236	9352	9624	**9220**	244	1.03	1098	0.12
eight_but	312	12016	**11276**	11948	324	1.04	1458	0.13
chan_prot	42	3808	6204	**3364**	62	1.48	279	0.08
reactor_ctrl	27	2668	5504	**2460**	34	1.26	153	0.06
runner	31	3140	5940	**2824**	27	0.87	121	0.04
example	20	2480	5196	**2344**	28	1.4	126	0.05
ww_button	76	6112	7384	**5980**	95	1.25	427	0.07
greycounter	143	**7612**	7936	8688	343	2.4	1549	0.2
tcint	355	14860	11376	15340	379	1.07	1707	0.15
mca200	3090	104536	77112	**52998**	8650	2.8	39717	0.75

Table 5.9: Memory usage comparison between KEP and MicroBlaze implementations. "(b)" refers to measurements in bytes, "(w)" to words.

instructions are executed, there is a CS at about every other instruction, whereas for the runner benchmark, there is a CS roughly every seven instructions. This indicates that the fast, light-weight CS mechanism of the KEP is a key to performed once for executing these types of reactive programs. Overall, between 30 and 60% of the CSs took place at the same priority, that is, because threads became inactive and another thread at the same priority took over. Some benchmarks did not require any PRIO instructions, for others they constituted up to 25% of the instructions executed. Up to 37% of CSs were due to PRIO instructions. Finally, for those benchmarks that include PRIO statements, less than half of the PRIO instructions actually result in a CS, indicating that a static schedule would have been comparatively inefficient.

Comparison of the Memory Usage

Table 5.9 compares executable code size and RAM usage between the KEP and the MicroBlaze implementations. To assess the size of the KEP code related to the Esterel source, we compare the code size in words with the Esterel Lines of Code (LOC, before dismantling, without comments), and notice that the KEP code is very compact, with a word count close to the Esterel source. For comparison with the MicroBlaze, we compare the size of Code + Data, in bytes, and notice that the KEP code is typically an order of magnitude smaller than the MicroBlaze code. The KEP implementation results on average in an 83% reduction of memory usage (codes and RAM size) when compared with the best result of the MicroBlaze implementation. As for the mca200, the memory

Module Name	MicroBlaze						KEP			
	WCRT			ACRT			WCRT		ACRT	
	V5	V7	CEC	V5	V7	CEC	abs.	rel.	abs.	rel.
	[1] (**best**)			[2] (**best**)			[3]	[3]/[1]	[4]	[4]/[2]
abcd	1559	**954**	1476	1464	**828**	1057	135	0.14	84	0.11
abcdef	2281	**1462**	1714	2155	**1297**	1491	201	0.14	117	0.09
eight_but	3001	**1953**	2259	2833	**1730**	1931	267	0.14	153	0.09
chan_prot	754	**375**	623	683	**324**	435	117	0.31	54	0.17
reactor_ctrl	487	**230**	397	456	**214**	266	51	0.22	39	0.18
runner	566	**289**	657	512	**277**	419	30	0.10	6	0.02
example	467	**169**	439	404	**153**	228	42	0.25	24	0.16
ww_button	1185	**578**	979	1148	**570**	798	48	0.08	36	0.06
greycounter	1965	**1013**	2376	1851	**928**	1736	528	0.52	375	0.40
tcint	3580	**1878**	2350	3488	**1797**	2121	408	0.22	252	0.14
mca200	75488	29078	**12497**	73824	24056	**11479**	2862	0.23	1107	0.10

Table 5.10: The worst-/average-case reaction times (in clock cycles) for the KEP and MicroBlaze implementations, in absolute and relative values.

reduction of the KEP implementation is not so dramatic as that of other cases. The reason is that the mca200 contains lots of data handling—which is not a very strong point of the KEP.

Comparison of the Execution Time

The improvement in execution time of the KEP implementation is shown in Table 5.10. Comparing with the best result of the MicroBlaze implementations, the KEP typically obtains more than 4x speedup for the WCRT, and more than 5x for the Average Case Reation Time (ACRT). Note that for a fair comparison, the time is measured based on the system clock. If the comparison is based on the instruction cycles, the KEP will achieve 12x speedup for the WCRT and more than 15x for the ACRT.

The MicroBlaze uses several levels of memory [73]. Here we employed an FPGA chip which has a large scale on-chip memory to implement the MicroBlaze system. Hence, all of the MicroBlaze programs could be loaded into the on-chip memory to make sure that the memory access time is minimal. The MicroBlaze implementation benefits from this, because if the implementation is based on an FPGA which has smaller scale on-chip memory, the KEP program is likely to fit into the on-chip memory.

Module Name	MicroBlaze (82mW@50MHz) Blank	KEP (mW) Peak	KEP (mW) Blank	Ratio (KEP to MB) Peak	Ratio (KEP to MB) Blank
abcd	69	13	8	0.16	0.12
abcdef	74	13	7	0.16	0.09
eight_but	74	13	7	0.16	0.09
chan_prot	70	28	12	0.34	0.17
reactor_ctrl	76	20	13	0.24	0.17
runner	78	14	2	0.17	0.03
example	77	25	9	0.30	0.12
ww_button	81	13	4	0.16	0.05
greycounter	78	44	33	0.54	0.42

Table 5.11: The energy consumption comparison between KEP and MicroBlaze implementations.

Power Usage

To compare the energy consumptions, we choose the Xilinx 3S200-4ft256 as FPGA platform. This requires an additional 37mW as quiescent power for the chip itself. The MicroBlaze system is assumed to run at 50MHz, and the peak power of the MicroBlaze is calculated by the frequency and the hardware resources of the MicroBlaze system via Xilinx WebPower Version 8.1.01 [69]. Based on the findings presented in Table 5.10, we calculate the minimal clock frequencies of the KEP to achieve the same WCRT of corresponding MicroBlaze system for each benchmark, then calculate the peak power of the KEP implementation.

For most blank events, the action of an Esterel module is very simple–it tests the presence of awaited signals, and then finishes this tick because those await statements are not terminated. For the KEP, since the elapsed instruction cycle count—for those actions is far from the assigned tick length, the system will turn to the idle state for saving power. Although the MicroBlaze has no low-power operating mode that can be used to conserve processor energy [74] (*e. g.*, like the wait-state of the PowerPC405 [71]), we still assume it can use some additional circuit to manage its power usage by blocking its clock to satisfy the fixed tick length feature. Note that the real tick length for a blank event depends on the state of the program of the previous tick. The average power usage of blank events is also estimated by an extended esi file, which inserts a blank event between every two original ticks.

Table 5.11 shows that the KEP reduces energy usage on average by 75%. The reduction becomes even more significant if all environment inputs are absent, a rather frequent case. Energy consumptions of the MicroBlaze system are similar for different events.

However, the reactive architecture makes the power usage of the KEP 52% lower than its peak power. Hence, in this case, the KEP achieves 86% power savings.

5.4 Summary

This chapter presented the whole reactive processing design flow, and illustrated the evaluation platform of the KEP. Compared with other Esterel implementations, the KEP represents a good tradeoff between hardware (logic area) usage, memory usage, system flexibility, WCRT, and energy usage.

Since an Esterel program is implemented in software on a COTS processor in general, we compared the KEP implementation with the MicroBlaze implementation, which are both implemented on Xilinx FPGA. Although the MicroBlaze has been optimized for the Xilinx FPGA, the KEP still wins in terms of memory usage, execution period (WCRT and ACRT), energy consumption, etc.

To allow an efficient execution of concurrent Esterel programs, the KEP offers a very light-weight thread model. Furthermore, its preemption handling model is also optimized for the preemption structure of actual Esterel modules. Those novel methods provide an efficient solution for the direct handling of Esterel programs, and improve on other reactive processors in several areas, such as hardware cost and scalability.

Chapter 6

Conclusion and Outlook

Employing a reactive processor to execute Esterel programs is an efficient, predictable alternative to traditional hardware or software platforms. In this chapter we first summarize the features of the KEP, and discuss its pros and cons. In Section 6.2, some suggestions are presented for further research.

6.1 Conclusion

This thesis presents the KEP, a concurrent, configurable Esterel processor. It employs a multi-threaded reactive architecture which consists of a reactive core and an optimized data path for the direct execution of Esterel programs.

The KEP ISA is *complete* in that it allows a direct mapping of all Esterel statements onto KEP assembler. It supports Esterel's concurrency operator || in a very precise, direct and efficient way. It also supports full Esterel preemptions, *i. e.*, the delayed and immediate strong/weak abortion and suspension. All of other Esterel kernel statements, *e. g.*, the Esterel exception, delay, and signal emission, are also implemented directly and semantically accurate by the KEP. Furthermore, valued signals and signal counters, local signal declarations, and the pre operator are also supported directly. One of the strengths of Esterel is the clean orthogonalization of the different reactive control flow constructs, which allows to combine them in an arbitrary fashion; this is fully supported by the KEP.

The KEP not only supports common Esterel statements directly, but also takes into consideration the statement context. In particular, it provides preemption instructions that map onto different types of hardware units depending on whether preemptions are nested or not and whether they include single threads or multiple threads. Providing such a *refined* ISA further minimizes hardware usage while preserving the generality of the language.

The KEP uses a priority-based scheduler, which makes threads responsible for managing their own priorities. This scheme allows to keep the scheduler very light-weight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing a thread's priority costs an instruction. The presented priority assignment algorithm [84, 85] for the KEP makes use of a special concurrent control flow graph and has a complexity that is linear in the size of that graph, which in practice tends to be linear in the size of the program.

The KEP architecture is highly configurable and represents a whole family of processors. Hence, although each KEP has a specific maximal nesting depth of preemption constructs and a maximal number of threads, this number is configurable for a particular KEP.

The performance of the KEP is predictable. All instructions can be executed in a single instruction cycle. The predictability of the KEP also lends itself to an automated Worst Case Reaction Time analysis. To validate and evaluate the KEP, a evaluation platform was built. The user interacts via a Host work station with an FPGA Board, which contains the KEP as well as some testing infrastructure. Up to now, more than 400 test cases show the correctness and efficiency of the KEP implementation.

A frequently asked question is why a reactive processor should be used for handling Esterel programs, and what benefits can be taken from it. The answer is: the KEP makes a better trade-off between flexibility, speed, Esterel compliance, etc., than other implementations. As the experimental comparison with a 32-bit commercial RISC processor indicates, the KEP has advantages in terms of memory use, execution speed, and energy consumption due to its Esterel optimized control and data path. Of course, the KEP solution has its price: more logic area than a traditional processor, *e. g.*, the MicroBlaze, is needed. However, if we ignore the fact of that the KEP can be further optimized [101], the number of gates is not the most important issue for lots of application. Predictable WCRT, system-level power management, the memory usage—which further influence system performance and power—are all open problems to current embedded systems [122, 123, 94]. The multi-threaded reactive architecture of the KEP could provide a new impulse here.

One might argue that the multiprocessing approach has an efficiency advantage over a multi-threading approach, which still relies on sequential execution. In principle, it is possible that a multiprocessing architecture could achieve a speedup over a single-processor solution. However, even ignoring the limitations of the multiprocessing approach with respect to the ability to combine concurrency and preemption, one should note that threads in Esterel programs typically interact rather tightly, with signals communicating back and forth within a logical tick, imposing strong synchronization requirements. Unlike classical parallel programming, where an originally sequential algorithm is divided into coarse-grained code fragments that can be executed in parallel to achieve a speedup over a single processor implementation, the concurrent programming in Esterel mainly serves to separate concerns [34], not to improve efficiency. Quoting Girault [60]: "The source program is *parallel* and not sequential like in a classical programming lan-

guage ... But this *parallelism of expression* is used by the programmer to conceive his/her application in terms of parallel modules cooperating to achieve the desired behavior. It is therefore not related to the *parallelism of execution*, which is due to the fact that the target architecture is distributed." We suspect that for the type of concurrency found in synchronous languages such as Esterel, a sequential, multi-threaded architecture may very well lead to higher efficiency than a multiprocessing approach, due to the tight link between independent threads that allows very efficient synchronization among the threads. However, substantiating this would require a further systematic comparison of these approaches.

6.2 Recommendations for Further Research

This thesis introduced a multi-threaded reactive processor architecture as a novel Esterel implementation. We presented the implementation model of the KEP components, and also validated the correctness of them via the evaluation. However, the KEP is still an initial prototype. Hence, some function circuits could be further optimized or refined via advanced design methods [113, 25]. Furthermore, there is still some room to optimize or to enhance the KEP architecture and its compiler:

- Reconfigurable logic block
  Regarding further improvements of the KEP, a co-design approach is developed to accelerate reactive processing by using external logic blocks. Here a reconfigurable logic block allows the efficient detection of compound events, such as waiting for the simultaneous occurrence of two signals [76, 64, 121]. This goal is partially achieved [58].

- Mapping control and data handling to different address spaces
  The KEP uses absolute addresses to indicate the memory location of its program. However, it is not an efficient way for handling Esterel control structures. An alternative would be to separate the memory into a control-handling-program area and a data-handling-program area. Considering a program which contains some data handing processes, those processes could be moved to the data-handling-program area, with only the remaining parts participating in the address range checking (*e. g.*, the IOPRW mechanism mentioned in Section 3.2.2) of the KEP. This strategy could reduce the logic area and speed up the maximum frequency of the KEP, and also extend the address management ability of the KEP.

- Instruction set optimization
  The current KEP instruction set architecture has evolved from its previous versions [118]. Some details should be refined. For example, the number of signals (255) could be a limitation to handle some large scale module. On the other hand, the maximum priority (127) seems unproblematic.

Beyond those technical details, there is a more general issue. Up to now, we have implemented all Esterel kernel statements and some specific derived statements. The selection of the implemented Esterel-type statements is based on our experience, which is a common strategy for the ASIP design. However, we are interested in building an automated instruction selection strategy [120, 1, 36, 61], which would increase application performance while area constraints are still satisfied. Furthermore, some other Esterel statements and structures, *e. g.*, weak suspension [116], could also be implemented.

- Optimized and adapted compilation
  The KEP compiler translates the Esterel control statement to the KEP instructions word for word. As the results show in Section 5.3, this strategy keeps the compilation time low. However, for most modules, this straightforward method is not the most efficient. For example, in the tcint module, the weak abortion is expressed in primitive form [19], which is composed of two threads and a trap. For the KEP, the weak abortion could be implemented by a watcher directly. A manually optimized version of this module reduces thread numbers by 31% (from 39 to 27) and instruction memory usage by 14% (from 379 to 325 instruction words).

  This raises another interesting topic. Since an Esterel control statement could be replaced by the combination of other statements, the KEP compiler should "know" how to choose the right statement(s) to implement an Esterel statement. Furthermore, an adaptive compilation strategy should be considered [40]. For example, the compiler could implement a weak abortion by a watcher when watcher resources remain, or else, the compiler could translate the weak abortion to the trap+threads expressions. On the other hand, the compiler could also generate a new version of the KEP to fit the application [1, 36, 109, 5]. One could also envision a scheme to configure the KEP according to a desired WCRT or ACRT for a specific program.

- KEP in Esterel
  We are planing to implement the KEP in Esterel. Beyond the general interest in such an implementation—for example, as a good case study and benchmark of Esterel itself—the most significance of this design would be that it could provide a virtual machine that mediates in real time the interaction between software processes and physical processes, see also [68, 98].

- Multi-processor architecture
  Finally, we are also investigating to combine the KEP with a DSP as a multi-processor architecture [66, 81, 94, 82]. The KEP is designed to implement the Esterel control structures in a direct way. The data handling ability is not its strong point. A KEP+DSP hybrid system could integrate the advantage of either processor, and could also be a perfect implementation platform for the next generation Esterel language—the Esterel V7, which inherits the original Esterel control structures but enhances data processing functions [116].

As indicated above, there are still lots of research topics for the KEP architecture. We wish these possibilities can be explored in the near future. We also hope anyone who is interested in architecture feels free to discuss with us if they have any questions. We look forward to hear from you.

Appendix A

KEP Instruction Set

A.1 Esterel-type Instructions

A.1.1 Preemption

- [W]ABORT[I]

  Assembly syntax:

  ABORT *S*, *endAddr* (,*WatcherID*)

  ABORTI *S*, *endAddr* (,*WatcherID*)

  WABORT *S*, *endAddr* (,*WatcherID*)

  WABORTI *S*, *endAddr* (,*WatcherID*)

  S The name of the signal.

  endAddr The address behind the end of the abortion body; see Figure 3.11.

  WatcherID The Watcher identification. It could be assigned by the Esterel to KASM compiler (strl2kasm); or else, if it is not given, this parameter will be generated by the KEP Assembler Compiler (kasm2klst) automatically.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00ooo	sssssssss	wwwwww	AAAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode = ABORT: ooo=010; ABORTI: ooo=011; WABORT: ooo=100; WABORTI: ooo=101.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The index number of the Watcher, the KEP supports up to 64 Watchers.

[**15:00**] The address behind the end of the abortion body.

Note: as mentioned in Section 3.3, the count delay expression of an abortion instruction like ABORT *n*, *S*, *endAddr* (,*WatcherID*) will be divided automatically to two instructions. Furthermore, as in the original semantics of Esterel, the delay expression cannot be used for an immediate delay, *i. e.*, an ABORTI *n*, *S*, *endAddr* (,*WatcherID*) instruction is forbidden. However, if the user puts them together, for example, a LOAD _COUNT,#3 instruction, which is followed by an ABORTI S, A0, the KEP will ignore the counter specified instruction. In other words, the LOAD _COUNT,#3 instruction will do nothing.

- SUSPEND[I]

  Assembly syntax:

  SUSPEND *S*, *endAddr* (,*WatcherID*)

  SUSPENDI *S*, *endAddr* (,*WatcherID*)

  S The name of the signal.

  endAddr The address behind the end of the suspension body; see Figure 3.11.

  WatcherID The Watcher identification. It could be assigned by the Esterel to KASM compiler (strl2kasm); or else, if it is not given, this parameter will be generated by the KEP Assembler Compiler (kasm2klst) automatically.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	0011o	ssssssss	wwwwww	AAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode=SUSPEND when o=0; or SUSPENDI o=1.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The index number of the Watcher, the KEP supports up to 64 Watchers.

[**15:00**] The address behind the end of the abortion body.

Note: following the original semantics of the Esterel, the count delay expression cannot be used for the suspension, *i. e.*, an SUSPEND[I] *n*, *S*, *endAddr* (,*WatcherID*) instruction is unallowed. However, if the user puts them together, for example, a LOAD _COUNT,#3 instruction, which is followed by an SUSPEND[I] S, A0, the KEP will ignore the counter specified instruction.

- L[W]ABORT[I]

  Assembly syntax:

  LABORT *S*, *endAddr* (,*WatcherID*)

  LABORTI *S*, *endAddr* (,*WatcherID*)

  LWABORT *S*, *endAddr* (,*WatcherID*)

  LWABORTI *S*, *endAddr* (,*WatcherID*)

S The name of the signal.

endAddr The address behind the end of the abortion body; see Figure 3.11.

LWatcherID The LWatcher identification. It could be assigned by the Esterel to KASM compiler (strl2kasm); or else, if it is not given, this parameter will be generated by the KEP Assembler Compiler (kasm2klst) automatically.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01ooo	sssssssss	wwwwww	AAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode = LABORT: ooo=010; LABORTI: ooo=011; LWABORT: ooo=100; LWABORTI: ooo=101.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The index number of the Local Watcher, the KEP supports up to 64 Local Watchers.

[**15:00**] The address behind the end of the abortion body.

Note: the count delay expression for this instruction is similar as that of the [W]ABORT[I] instruction.

- T[W]ABORT[I]

  Assembly syntax:

  TABORT *S*, *endAddr*

  TABORTI *S*, *endAddr*

  TWABORT *S*, *endAddr*

  TWABORTI *S*, *endAddr*

  S The name of the signal.

  endAddr The address behind the end of the abortion body; see Figure 3.11.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01110	sssssssss	0000oo	AAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode = T[W]ABORT[I]

[**30:22**] The signal's unicode value. See also Section 4.3.

[**17:16**] The extended operation code. It indicates TABORT: oo=00; TABORTI: oo=01; TWABORT: oo=10; TWABORTI: oo=11.

[**15:00**] The address behind the end of the abortion body.

Note: unlike other preemption instructions, this instruction has no WatcherID parameter because it is indexed by the ID of current executed thread, see Section 4.2.2. The count delay expression for this instruction is similar as that of the [W]ABORT[I] instruction.

A.1.2 Exception

- EXIT

  Assembly syntax:

  EXIT *endAddr*, *startAddr*

  endAddr The target address of this exception, *i. e.*, the address behind the end of the corresponding trap body; see Figure 3.13.

  startAddr The start address of the corresponding trap body; see Figure 3.13.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	1101a	aaaaaaaaa	aaaaaa	AAAAAAAAAAAAAAAA

Instruction Encoding:

[35:32] Opcode = EXIT

[30:22] The address of the start of the corresponding trap body.

[15:00] The address behind the end of the corresponding trap body.

A.1.3 Concurrency

- PAR

  Assembly syntax:

  PAR *Prio*, *startAddr* [, *ThreadID*]

  Prio The initial priority value of the created sub thread.

  startAddr The start address of the body of the created sub thread; see Figure 3.10.

  ThreadID The identification of the created sub thread. It could be assigned by the Esterel to KASM compiler (strl2kasm); or else, if it is not given, this parameter will be generated by the KEP Assembler Compiler (kasm2klst) automatically.

	$d_{35} - d_{31}$	$d_{30} - d_{23}$	$d_{22} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01001	PPPPPPPP	IIIIIII	AAAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode=PAR/PARE/PRIO. Note that the PAR, PARE, and PRIO instructions share the same opcode, they are distinguished by the encoding of others fields of instruction.

[**30:23**] The priority of the created sub thread. See also Section 4.2.1.

[**22:16**] The identification of the created thread by this instruction. See also Section 4.2.1.

[**15:00**] The start address of the created sub thread.

- PARE

  Assembly syntax:

  PARE *endAddr*, [,*Prio*]

  endAddr The end address of created sub threads body.

  Prio The priority value of the current executed thread, *i. e.*, the parent thread of those created sub threads. It could be assigned by the Esterel to KASM compiler (strl2kasm); or else, if it is not given, this parameter will be generated by the KEP Assembler Compiler (kasm2klst) automatically as zero. If it equals zero, the thread priority will not be modified, *i. e.*, it keeps its current priority value.

	$d_{35} - d_{31}$	$d_{30} - d_{23}$	$d_{22} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01001	PPPPPPPP	0000000	AAAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode=PAR/PARE/PRIO.

[**30:23**] The priority value of the created sub thread. See also Section 4.2.1.

[**22:16**] The extended operation code. "0000000" indicates PAR. Note that for PAR and PRIO, this range is occupied by the thread ID. The created sub thread (for PAR) cannot be the thread 0; and it is also impossible to modify the priority value of the thread 0. Hence, for these two instructions, this range will not be zero.

[**15:00**] The end address of created sub threads body, *i. e.*, the address behind the end of the last sub thread body.

- JOIN

  Assembly syntax:

  JOIN [*Prio*]

  Prio The priority value of the current executed thread, *i. e.*, the parent thread of those created sub threads. It could be assigned by the Esterel to KASM compiler (strl2kasm); or else, if it is not given, this parameter will be generated

by the KEP Assembler Compiler (kasm2klst) automatically as zero. If it equals zero, the thread priority will not be modified, *i. e.*, it keeps its current priority value. One should note that the assignment of the priority will not happen until all of child threads of the current executed thread terminated. In other words, when the JOIN instruction terminates, the non-zero priority value will be assigned to the current executed thread before the execution continues.

	$d_{35} - d_{31}$	$d_{30} - d_{23}$	$d_{22} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00000	PPPPPPPP	0100000	0000000000000000

Instruction Encoding:

[35:31] Opcode=NOTHING/GOTO/CALL/RETURN/PRESENT/JOIN.

[30:23] The priority value of the current execute thread.

[22:16] The extended operation code. "0100000" indicates JOIN.

- PRIO

  Assembly syntax:

  PRIO *Prio* [,*ThreadID*]

  Prio The priority value. If the *ThreadID* equals zero, this value will be assigned to the current executed thread; or else, it will be assigned to the corresponding thread which specified by the *ThreadID* parameter.

  ThreadID The thread identification which will be assigned the new priority value. It could be assigned by the Esterel to KASM compiler (strl2kasm); or else, if it is not given, this parameter will be generated by the KEP Assembler Compiler (kasm2klst) automatically. The default value is zero, which implies this instruction just modifies the current executed thread.

	$d_{35} - d_{31}$	$d_{30} - d_{23}$	$d_{22} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01001	PPPPPPPP	IIIIIII	0000000000000000

Instruction Encoding:

[35:31] Opcode=PAR/PARE/PRIO.

[30:23] The priority value.

[22:16] The identification of the target thread.

A.1.4 Delay

- AWAIT[I]/PAUSE/HALT

  Assembly syntax:

AWAIT *S*

AWAITI *S*

PAUSE

S The name of the signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00001	ssssssss	0000o0	0000000000000000

Encoding:

[**35:31**] Opcode=AWAIT[I]/PAUSE/HALT/CAWAIT[S|I|E]

[**30:22**] The signal's unicode value. Since the PAUSE equals AWAIT TICK, for the PAUSE instruction, these bits are 000000000. See also Section 4.3.

[**21:16**] The extended operation code. 000000 indicates AWAIT; and 000010 indicates AWAITI.

Note: as mentioned in Section 3.3, the count delay expression of an abortion instruction like AWAIT *n, S* will be divided to two instructions automatically. Furthermore, as the original semantics of Esterel, the delay expression cannot be used for an immediate delay, *i. e.*, an AWAITI *n, S* instruction is not allowed. If the user puts them together, for example, a LOAD _COUNT,#3 instruction which followed by an AWAITI S, the KEP will ignore the counter specified instruction. In other words, the LOAD _COUNT,#3 instruction will do nothing.

- HALT

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00001	000000000	000001	0000000000000000

Encoding:

[**35:31**] Opcode=AWAIT[I]/PAUSE/HALT/CAWAIT[S|I|E]

[**21:16**] The extended operation code. 000001 indicates HALT.

- CAWAITS

Assembly syntax:

CAWAITS

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00001	000000000	110011	0000000000000000

Encoding:

[**35:31**] Opcode=AWAIT[I]/PAUSE/HALT/CAWAIT[S|I|E]

[**21:16**] The extended operation code. 110011 indicates CAWAITS.

- CAWAITS[I]

  Assembly syntax:

  CAWAIT *S*

  CAWAITI *S*

  S The name of the signal.

	$d_{35}-d_{31}$	$d_{30}-d_{22}$	$d_{21}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	00001	ssssssss	1100o0	0000000000000000

  Encoding:

  [**35:31**] Opcode=AWAIT[I]/PAUSE/HALT/CAWAIT[S|I|E]

  [**21:16**] The extended operation code. 110000 indicates CAWAIT; and 110010 indicates CAWAITI.

- CAWAITE *addr*

  addr The address of the first CAWAIT[I] instruction of corresponding await case branches. See also Figure 3.15.

	$d_{35}-d_{31}$	$d_{30}-d_{22}$	$d_{21}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	00001	000000000	110001	AAAAAAAAAAAAAAAA

  Encoding:

  [**35:31**] Opcode=AWAIT[I]/PAUSE/HALT/CAWAIT[S|I|E]

  [**21:16**] The extended operation code. 110001 indicates CAWAITE.

  [**15:00**] The address of the first CAWAIT[I] instruction of this CAWAITS/CAWAIT[I]/CAWAIT[instruction group. See also Figure 3.15.

  Note: unlike the AWAIT[I] instruction, the CAWAIT[I] does not support count delays. However, the user can use an inner signal and an additional thread to model the same behavior. See also Figure A.1.

A.1.5 Signal Emission and Testing

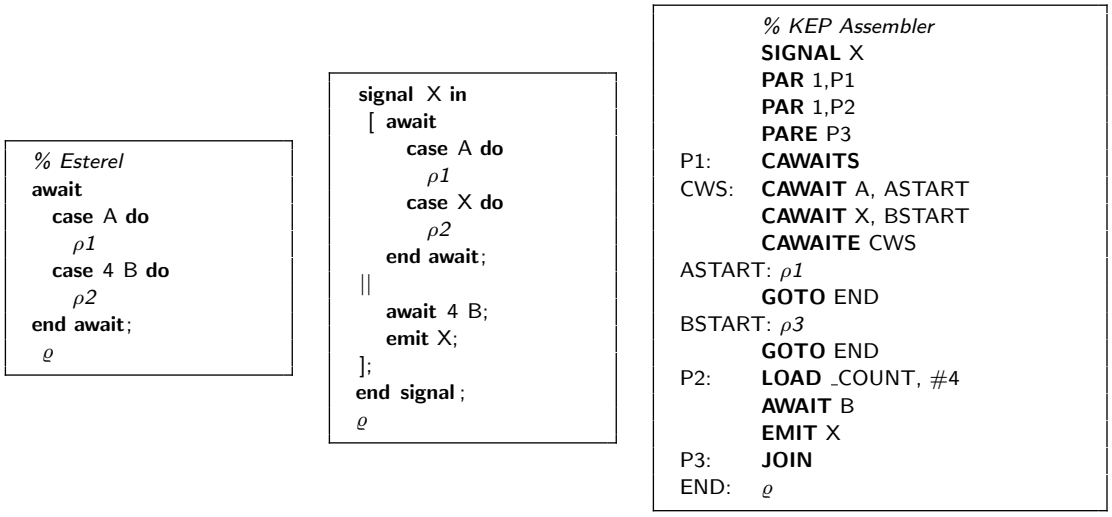
- SETV

  Assembly syntax:

  – SETV *S*, *#data*

    S The name of the valued signal.

    #data The operating 16-bit data.



(a) (b) (c)

Figure A.1: Translating an Esterel CountAwaitCase module (a) to its equivalent form (b), and the corresponding assembler code for KEP (c).

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01000	sssssssss	100000	nnnnnnnnnnnnnnn

Instruction Encoding:

[**35:31**] Operation code. 01000 indicates SETV/EMIT/SUSTAIN/SIGNAL, an instruction with a signal and immediate data.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 100000 indicates SETV.

[**15:00**] The 16-bit operating data.

– SETV *S*, *reg*

S The name of the valued signal.

reg The name of the target register.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10010	sssssssss	100000	000000rrrrrrrrrr

Instruction Encoding:

[**35:31**] Operation code. 10010 indicates SETV/EMIT/SUSTAIN/SIGNAL, an instruction with a signal and the content of a register.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 100000 indicates SETV.

[**09:00**] The index value of the source register.

- EMIT

  Assembly syntax:

  – EMIT *S*

    S The name of the valued signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01000	ssssssssss	000000	0000000000000000

    Instruction Encoding:

    [**35:31**] Operation code. 01000 indicates SETV/EMIT/SUSTAIN/SIGNAL.

    [**30:22**] The signal's unicode value. See also Section 4.3.

    [**21:16**] The extended operation code. 000000 indicates EMIT.

  – EMIT *S*, *#data*

    S The name of the valued signal.

    #data The operating 16-bit data.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01000	ssssssssss	000000	nnnnnnnnnnnnnnnn

    Instruction Encoding:

    [**35:31**] Operation code. 01000 indicates SETV/EMIT/SUSTAIN/SIGNAL, an instruction with a signal and immediate data.

    [**30:22**] The signal's unicode value. See also Section 4.3.

    [**21:16**] The extended operation code. 000000 indicates EMIT.

    [**15:00**] The 16-bit operating data.

  – EMIT *S*, *reg*

    S The name of the valued signal.

    reg The name of the target register.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10010	ssssssssss	000000	000000rrrrrrrrrr

    Instruction Encoding:

    [**35:31**] Operation code. 10010 indicates SETV/EMIT/SUSTAIN/SIGNAL, an instruction with a signal and the content of a register.

    [**30:22**] The signal's unicode value. See also Section 4.3.

    [**21:16**] The extended operation code. 000000 indicates EMIT.

    [**09:00**] The index value of the source register.

- SUSTAIN

  Assembly syntax:

  – SUSTAIN *S*

S The name of the valued signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01000	sssssssss	010000	0000000000000000

Instruction Encoding:

[**35:31**] Operation code. 01000 indicates SETV/EMIT/SUSTAIN/SIGNAL.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 010000 indicates SUSTAIN.

– SUSTAIN *S*, *#data*

S The name of the valued signal.

#data The operating 16-bit data.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01000	sssssssss	010000	nnnnnnnnnnnnnnnn

Instruction Encoding:

[**35:31**] Operation code. 01000 indicates SETV/EMIT/SUSTAIN/SIGNAL, an instruction with a signal and immediate data.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 010000 indicates SUSTAIN.

[**15:00**] The 16-bit operating data.

– SUSTAIN *S*, *reg*

S The name of the valued signal.

reg The name of the target register.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10010	sssssssss	010000	000000rrrrrrrrrr

Instruction Encoding:

[**35:31**] Operation code. 10010 indicates SETV/EMIT/SUSTAIN/SIGNAL, an instruction with a signal and the content of a register.

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 010000 indicates SUSTAIN.

[**09:00**] The index value of the source register.

- SIGNAL

Assembly syntax:

SIGNAL *S*

S The name of the signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	01000	sssssssss	110000	0000000000000000

Instruction Encoding:

[35:31] Operation code. 01000 indicates SETV/EMIT/SUSTAIN/SIGNAL.

[30:22] The (local) signal's unicode value. See also Section 4.3.

[21:16] The extended operation code. 110000 indicates SIGNAL.

- PRESENT

  Assembly syntax:

  PRESENT *S,elseAddr*

  S The name of the signal.

  elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00000	sssssssss	000100	AAAAAAAAAAAAAAA

Instruction Encoding:

[35:31] Opcode=NOTHING/GOTO/CALL/RETURN/PRESENT/JOIN.

[30:22] The signal's unicode value. See also Section 4.3.

[21:16] The extended operation code. 000100 indicates PRESENT.

[15:00] The address behind the end of the presentation body. If Signal S is presented at the current cycle, the KEP will execute the following instruction, or else it will jump to this indicated address.

A.1.6 Others

- NOTHING

  Assembly syntax:

  NOTHING

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00000	000000000	000000	0000000000000000

Instruction Encoding:

[35:31] Opcode=NOTHING/GOTO/CALL/RETURN/PRESENT/JOIN.

[21:16] The extended operation code. 000000 indicates NOTHING.

- GOTO

  Assembly syntax:

  GOTO *addr*

  addr The target address

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00000	000000000	000001	AAAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode=NOTHING/GOTO/CALL/RETURN/PRESENT/JOIN.

[**21:16**] The extended operation code. 000001 indicates GOTO.

[**15:00**] The target address. The KEP will go to this indicated address.

- CALL

  Assembly syntax:

  CALL *addr*

  addr The start address of the subroutine.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00000	000000000	000010	AAAAAAAAAAAAAAAA

Instruction Encoding:

[**35:31**] Opcode=NOTHING/GOTO/CALL/RETURN/PRESENT/JOIN.

[**21:16**] The extended operation code. 000010 indicates CALL.

[**15:00**] The start address of the subroutine. When a procedure calls a subroutine, the address behind current instruction will be pushed. Subsequently a jump to *addr* is preformed.

- RETURN

  Assembly syntax:

  RETURN

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	00000	000000000	000011	0000000000000000

Instruction Encoding:

[**35:31**] Opcode=NOTHING/GOTO/CALL/RETURN/PRESENT/JOIN.

[**21:16**] The extended operation code. 000011 indicates RETURN. Return to the procedure from a subroutine. The pushed address will be popped as the target address.

A.2 Classical Instructions

A.2.1 Program and Machine Control

- CMP[S]

  Assembly syntax:

  – CMP[S] *reg,#data*

  reg The name of the target register.

  data The operating data.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10111	RRRRRRRRRR	0o000	nnnnnnnnnnnnnnn

  Encoding:

  [**35:31**] Opcode=CMP

  [**30:21**] The index value of the target register.

  [**20:16**] The extended operation code. 00000 indicates CMP *reg,#data*, and 01000 indicates CMPS *reg,#data*. It means that the contents of the register *reg* will be compared with the value of the data. The result will affect the zero and carry bits. If the contents of the register *reg* equals to the value of the *data*, the zero bit will be set to '1', and the carry bit will be set to '0'. If the contents of the register *reg* is less than the value of the *data*, the carry bit will be set to '1', and the zero bit will be set to '0'. If the contents of the register *reg* is greater than the value of the *data*, the carry and equal bits will be set to '0'. The difference of the CMP and CMPS is that for the CMPS instruction the highest bit of the contents of the register *reg* is considered as a sign bit.

  [**15:00**] The value of the operating data.

  – CMP[S] *reg,REG*

  reg The name of the target register.

  REG The name of the source register.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10111	RRRRRRRRRR	0o001	000000rrrrrrrrr

  Encoding:

  [**35:31**] Opcode=CMP

  [**30:21**] The index value of the target register.

  [**20:16**] The extended operation code. 00001 indicates CMP *reg,REG*, and 01001 indicates CMPS *reg,REG*. It means that the contents of the register *reg* will be compared with the contents of the register *REG*. The result will affect the zero and carry bits.

[09:00] The index value of the target register.

– CMP *reg, ?S|PRE(?S)*

reg The name of the target register.

S The name of the valued signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10111	sssssssss	00o010	000000rrrrrrrrrr

Encoding:

[35:31] Opcode=CMP

[30:22] The index value of the target register.

[21:16] The extended operation code. 000010 indicates CMP *reg,?S|PRE(?S)*, and 001010 indicates CMPS *reg,?S|PRE(?S)*. It means that the contents of the register *reg* will be compared with the carried data of the valued signal *S*. The result will affect the zero and carry bits.

[09:00] The index value of the target register.

- JW

Assembly syntax:

– JW Z,*elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000000	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when not zero.* Set to "000000" when JW Z,*elseaddr*. It means that the KEP will test the zero bit. If the zero bit is '1' (true), the KEP will execute the following instruction, or else it will go to the branching address. The status of the zero bit depends on the last data, which was written into the register file.

[15:00] The branching address.

– JW L,*elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000001	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when not less.* Set to "000001" when JW L,*elseaddr*. It means that the KEP will test the carry bit. If the carry bit is '1' (true), the KEP will execute the following instruction, or else it will jump to the branching address. The status of the carry bit depends on the last statement which can affect the carry bit, *e. g.*, SUBC, ADDC, or CMP, etc. It also can be expressed as JC *elseaddr*.

[15:00] The branching address.

– JW G,*elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000011	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when not greater.* Set to "000011" when JW G,*elseaddr*. It means that the KEP will test the carry and zero bits. This instruction should be used right after a CMP instruction. If the result of the comparison is greater than, the KEP will execute the following instruction, or else it will jump to the branching address.

[15:00] The branching address.

– JW GE,*elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000100	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when not greater/equal.* Set to "000100" when JW GE,*elseaddr*. It means that the KEP will test the carry and zero bits. This instruction should be used right after a CMP instruction. If the result of the comparison is greater than or equal, the KEP will execute the following instruction, or else it will jump to the branching address.

[15:00] The branching address.

– JW LE,*elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000101	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when not less/equal.* Set to "000101" when JW LE,*elseaddr*. It means that the KEP will test the carry and zero bits. This instruction should be used right after a CMP instruction. If the result of the comparison is less than or equal, the KEP will execute the following instruction, or else it will jump to the branching address.

[15:00] The branching address.

– JW EE,*elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000110	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when not equal.* Set to "000110" when JW EE,*elseaddr*. It means that the KEP will test the zero bit. This instruction should be used right after a CMP instruction. If the result of the comparison is equal, the KEP will execute the following instruction, or else it will jump to the branching address.

[15:00] The branching address.

– JW NE,*elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000111	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when equal.* Set to "000111" when JW NE,*elseaddr*. It means that the KEP will test the zero bit. This instruction should be used right after a CMP instruction. If the result of the comparison is not equal, the KEP will execute the following instruction, or else it will jump to the branching address.

[15:00] The branching address.

- JNC

Assembly syntax:

JNC *elseaddr*

elseaddr The branching address.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10000	000000000	000010	AAAAAAAAAAAAAAAA

Encoding:

[35:31] Opcode=JW/JNC

[21:16] *Jump when carry bit is false.* Set to "000010" when JNC *elseaddr*. It means that the KEP will test the carry bit. If the carry bit is '0' (false), the KEP will execute the following instruction, or else it will jump to the branching address. This instruction and JW L,*elseaddr* are contrary terms. Note the JW L,*elseaddr* can also be expressed as JC *elseaddr*.

[15:00] The branching address.

A.2.2 Boolean Variable Manipulation

- CLRC

  Assembly syntax:

  CLRC

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10001	000000000	000000	0000000000000000

  Encoding:

  [35:31] Opcode=CLRC/SETC/SR[C]/SL[C]/NOTR

  [21:16] The extended operation code 000000 indicates CLRC. It means that the carry bit will be cleared.

- SETC

  Assembly syntax:

  SETC

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10001	000000000	000001	0000000000000000

  Encoding:

  [35:31] Opcode=CLRC/SETC/SR[C]/SL[C]/NOTR

  [21:16] The extended operation code. 000001 indicates SETC. It means that the carry bit will be set to '1'.

- SR[C]

  Assembly syntax:

  SR *reg*

  SRC *reg*

  reg The name of the register.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10001	000000000	00001o	000000rrrrrrrrr

Encoding:

[**35:31**] Opcode=CLRC/SETC/SR[C]/SL[C]/NOTR

[**21:16**] The extended operation code. 000010 indicates SR; and 000011 indicates SRC. It means that the contents of the register *reg* will be right shifted. The highest bit will be replaced by '0' for SR; or it will be replaced by the status of the carry bit for SRC.

[**09:00**] The index value of the target register.

- SL[C]

  Assembly syntax:

  SL *reg*

  SLC *reg*

  reg The name of the register.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10001	000000000	0001oo	000000rrrrrrrrr

Encoding:

[**35:31**] Opcode=CLRC/SETC/SR[C]/SL[C]/NOTR

[**21:16**] The extended operation code. 000101 indicates SL; and 000110 indicates SLC. It means that the contents of the register *reg* will be left shifted. The lowest bit will be replaced by '0' for SL; or it will be replaced by the status of the carry bit for SLC.

[**09:00**] The index value of the target register.

- NOTR

  Assembly syntax:

  NOTR *reg*

  reg The name of the register.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10001	000000000	000111	000000rrrrrrrrr

Encoding:

[**35:31**] Opcode=CLRC/SETC/SR[C]/SL[C]/NOTR

[**21:16**] The extended operation code. 000111 indicates NOTR. It means that the contents of the register *reg* will be complemented.

[**09:00**] The index value of the target register.

A.2.3 Data Transfer

- LOAD

  Assembly syntax:

  – LOAD *REG*, *#data*

  REG The name of the target register.

  data The data to be loaded into the register.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10100	RRRRRRRRRR	00000	nnnnnnnnnnnnnnn

  Encoding:

  [35:31] Opcode=LOAD

  [30:21] The index value of the target register.

  [20:16] The extended operation code. 00000 indicates LOAD *REG*, *#data*. It means that the contents of the register *REG* will be replaced by the value of data.

  [15:00] The value of the data.

  – LOAD *reg*, *REG*

  reg The name of the target register.

  REG The name of the source register.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10100	RRRRRRRRRR	00001	000000rrrrrrrrrr

  Encoding:

  [35:31] Opcode=LOAD

  [30:21] The index value of the source register.

  [20:16] The extended operation code. 00001 indicates LOAD *reg*, *REG*. It means that the contents of the register *reg* will be replaced by the contents of the register *REG*.

  [09:00] The index value of the target register.

  – LOAD *reg*, *?S|PRE(?S)*

  reg The name of the target register.

  S The name of the valued signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10100	sssssssss	000010	000000rrrrrrrrrr

  Encoding:

  [35:31] Opcode=LOAD

  [30:22] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 000010 indicates LOAD *reg*, *?S|PRE(?S)*. It means that the contents of the register *reg* will be replaced by the carried data of the valued signal *S* (for *?S*); or it will be replaced by the carried data of the valued signal *S* in the previous tick (for *PRE(?S)*).

[**09:00**] The index value of the target register.

- DEF32

  Assembly syntax:

  DEF32 *#data32*

  data32 The 32-bit data to be loaded.

	$d_{35}-d_{32}$	$d_{31}-d_{22}$	$d_{21}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	1100	nnnnnnnnnn	nnnnnn	nnnnnnnnnnnnnnnn

Encoding:

[**35:32**] Opcode=DEF32. This instruction loads a 32-bit data to a special register named _UINT32REG. The user can access this register to operate this data. See also Section 3.5.

[**31:00**] The value of the 32-bit data.

A.2.4 Arithmetic Operations

- ADD[C]

  Assembly syntax:

  - ADD *REG*, *#data*
  - ADDC *REG*, *#data*

    REG The name of the target register.

    data The data to be added into the register.

	$d_{35}-d_{31}$	$d_{30}-d_{21}$	$d_{20}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10101	RRRRRRRRRR	00o00	nnnnnnnnnnnnnnnn

Encoding:

[**35:31**] Opcode=ADD[C]/SUB[C]/MUL

[**30:21**] The index value of the target register.

[**20:16**] The extended operation code. 00000 indicates ADD *REG*, *#data*, and 00100 indicates ADDC *REG*, *#data*. It means that the contents of the register *REG* (and the value of the carry bit for ADDC) will be added with the value of the *data*, and the sum will be stored in the register *REG*.

[**15:00**] The value of the data.

– ADD[C] *reg*, *REG*

reg The name of the target register.

REG The name of the source register.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10101	RRRRRRRRRR	00o01	000000rrrrrrrrrr

Encoding:

[**35:31**] Opcode=ADD[C]/SUB[C]/MUL

[**30:21**] The index value of the source register.

[**20:16**] The extended operation code. 00001 indicates ADD *reg*, *REG*, and 00101 indicates ADDC *reg*, *REG*. It means that the contents of the register *REG* (and the value of the carry bit for ADDC) will be added with the contents of the register *reg*, and the sum will be stored in the register *reg*.

[**09:00**] The index value of the target (augend/sum) register.

– ADD[C] *reg*, *?S|PRE(?S)*

reg The name of the target register.

S The name of the valued signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10101	sssssssss	000o10	000000rrrrrrrrrr

Encoding:

[**35:31**] Opcode=ADD[C]/SUB[C]/MUL

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 000010 indicates ADD *reg*, *?S|PRE(?S)*, and 000110 indicates ADDC *reg*, *?S|PRE(?S)*. It means that the contents of the register *reg* will be added with the carried data (for *?S*) or the carried data in the previous tick (for *PRE(?S)*) of the valued signal *S* (and the value of the carry bit for ADDC), and the sum will be stored in the register *reg*.

[**09:00**] The index value of the target register.

- SUB[C]

Assembly syntax:

– SUB *REG*, *#data*

– SUBC *REG*, *#data*

reg The name of the target (minuend/difference) register.

data The value of subtrahend.

	$d_{35}-d_{31}$	$d_{30}-d_{21}$	$d_{20}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10101	RRRRRRRRRR	01o00	nnnnnnnnnnnnnnn

Encoding:

[**35:31**] Opcode=ADD[C]/SUB[C]/MUL

[**30:21**] The index value of the target (minuend/difference) register.

[**20:16**] The extended operation code. 01000 indicates SUB *REG*, *#data*, and 01100 indicates SUBC *REG*, *#data*. It means that the value of the *data* (and the value of the carry bit for SUBC) will be subtracted from the contents of the register *REG*, and the difference will be stored in the register *REG*.

[**15:00**] The value of the data.

– SUB[C] *reg*, *REG*

reg The name of the target (minuend/difference) register.

REG The name of the source (subtrahend) register.

	$d_{35}-d_{31}$	$d_{30}-d_{21}$	$d_{20}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10101	RRRRRRRRRR	01o01	000000rrrrrrrrrr

Encoding:

[**35:31**] Opcode=ADD[C]/SUB[C]/MUL

[**30:21**] The index value of the source (subtrahend) register.

[**20:16**] The extended operation code. 01001 indicates ADD *reg*, *REG*, and 01101 indicates SUBC *reg*, *REG*. It means that the contents of the register *REG* (and the value of the carry bit for SUBC) will be subtracted from the contents of the register *reg*, and the difference will be stored in the register *reg*.

[**09:00**] The index value of the target (minuend/difference) register.

– SUB[C] *reg*, *?S|PRE(?S)*

reg The name of the target (minuend/difference) register.

S The name of the valued signal (subtrahend).

	$d_{35}-d_{31}$	$d_{30}-d_{22}$	$d_{21}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10101	sssssssss	001o10	000000rrrrrrrrrr

Encoding:

[**35:31**] Opcode=ADD[C]/SUB[C]/MUL

[**30:22**] The signal's unicode value. See also Section 4.3.

[**21:16**] The extended operation code. 001010 indicates SUB *reg*, *?S|PRE(?S)*, and 001110 indicates SUBC *reg*, *?S|PRE(?S)*. It means that the carried data (for *?S*) or the carried data in the previous tick (for *PRE(?S)*) of the valued signal *S* (and the value of the carry bit for SUBC) will be subtracted from the contents of the register *reg*, and the difference will be stored in the register *reg*.

[09:00] The index value of the target (minuend/difference) register.

- MUL

  Assembly syntax:

  – MUL *REG, #data*

  reg The name of the target (multiplicand/product) register.

  data The value of multiplier.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10101	RRRRRRRRRR	10000	nnnnnnnnnnnnnnnn

  Encoding:

  [35:31] Opcode=ADD[C]/SUB[C]/MUL

  [30:21] The index value of the target (multiplicand/product) register.

  [20:16] The extended operation code. 10000 indicates MUL *reg*, *#data*. It means that the contents of the register *REG* will be multiplied by the value of the *data*, and the product will be stored in the register *REG*.

  [15:00] The value of the data.

  – MUL *reg, REG*

  reg The name of the target (multiplicand/product) register.

  REG The name of the source (multiplier) register.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10101	RRRRRRRRRR	10001	000000rrrrrrrrrr

  Encoding:

  [35:31] Opcode=ADD[C]/SUB[C]/MUL

  [30:21] The index value of the source (multiplier) register.

  [20:16] The extended operation code. 10001 indicates MUL *reg*, *REG*. It means that the contents of the register *reg* will be multiplied by the contents of the register *REG*, and the product will be stored in the register *reg*.

  [09:00] The index value of the target (multiplicand/product) register.

  – MUL *reg, ?S|PRE(?S)*

  reg The name of the target (multiplicand/product) register.

  S The name of the valued signal (multiplier).

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10101	sssssssss	010010	000000rrrrrrrrrr

  Encoding:

  [35:31] Opcode=ADD[C]/SUB[C]/MUL

  [30:22] The signal's unicode value. See also Section 4.3.

[21:16] The extended operation code. 010010 indicates MUL *reg*, *?S|PRE(?S)*. It means that the carried data (for *?S*) or the carried data in the previous tick (for *PRE(?S)*) of the valued signal *S* will be multiplied by the contents of the register *reg*, and the product will be stored in the register *reg*.

[09:00] The index value of the target (multiplicand/product) register.

A.2.5 Logical Operations

- ANDR

  Assembly syntax:

  – ANDR *REG*, *#data*

  REG The name of the target register.

  data The operating data.

	$d_{35}-d_{31}$	$d_{30}-d_{21}$	$d_{20}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10110	RRRRRRRRRR	00000	nnnnnnnnnnnnnnn

  Encoding:

  [35:31] Opcode=ANDR/ORR/XORR

  [30:21] The index value of the target register.

  [20:16] The extended operation code. 00000 indicates ANDR *reg*, *#data*. It means that the contents of the register *REG* will be ANDed with the value of the *data*, and the result will be stored in the register *REG*.

  [15:00] The value of the operating data.

  – ANDR *reg*, *REG*

  reg The name of the target register.

  REG The name of the source register.

	$d_{35}-d_{31}$	$d_{30}-d_{21}$	$d_{20}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10110	RRRRRRRRRR	00001	000000rrrrrrrrrr

  Encoding:

  [35:31] Opcode=ANDR/ORR/XORR

  [30:21] The index value of the source register.

  [20:16] The extended operation code. 00001 indicates ANDR *reg*, *REG*. It means that the contents of the register *reg* will be ANDed with the contents of the register *REG*, and the result will be stored in the register *reg*.

  [09:00] The index value of the target register.

  – ANDR *reg*, *?S|PRE(?S)*

reg The name of the target register.

S The name of the valued signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10110	sssssssss	000010	000000rrrrrrrrrr

Encoding:

[35:31] Opcode=ANDR/ORR/XORR

[30:22] The signal's unicode value. See also Section 4.3.

[21:16] The extended operation code. 000010 indicates ANDR *reg*, *?S|PRE(?S)*. It means that the contents of the register *reg* will be ANDed with the carried data (for *?S*) or the carried data in the previous tick (for *PRE(?S)*) of the valued signal *S*, and the result will be stored in the register *reg*.

[09:00] The index value of the target register.

- ORR

  Assembly syntax:

  – ORR *REG*, *#data*

  REG The name of the target register.

  data The operating data.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10110	RRRRRRRRRR	01000	nnnnnnnnnnnnnnnn

  Encoding:

  [35:31] Opcode=ANDR/ORR/XORR

  [30:21] The index value of the target register.

  [20:16] The extended operation code. 01000 indicates ORR *reg*, *#data*. It means that the contents of the register *REG* will be ORed with the value of the *data*, and the result will be stored in the register *REG*.

  [15:00] The value of the operating data.

  – ORR *reg*, *REG*

  reg The name of the target register.

  REG The name of the source register.

	$d_{35} - d_{31}$	$d_{30} - d_{21}$	$d_{20} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10110	RRRRRRRRRR	01001	000000rrrrrrrrrr

  Encoding:

  [35:31] Opcode=ANDR/ORR/XORR

  [30:21] The index value of the source register.

  [20:16] The extended operation code. 01001 indicates ORR *reg*, *REG*. It means that the contents of the register *reg* will be ORed with the contents of the register *REG*, and the result will be stored in the register *reg*.

[09:00] The index value of the target register.

– ORR *reg, ?S|PRE(?S)*

reg The name of the target register.

S The name of the valued signal.

	$d_{35}-d_{31}$	$d_{30}-d_{22}$	$d_{21}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10110	sssssssss	001010	000000rrrrrrrrrr

Encoding:

[35:31] Opcode=ANDR/ORR/XORR

[30:22] The signal's unicode value. See also Section 4.3.

[21:16] The extended operation code. 001010 indicates ORR *reg, ?S|PRE(?S)*. It means that the contents of the register *reg* will be ORed with the carried data (for *?S*) or the carried data in the previous tick (for *PRE(?S)*) of the valued signal *S*, and the result will be stored in the register *reg*.

[09:00] The index value of the target register.

- XORR

Assembly syntax:

– XORR *REG, #data*

REG The name of the target register.

data The operating data.

	$d_{35}-d_{31}$	$d_{30}-d_{21}$	$d_{20}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10110	RRRRRRRRRR	10000	nnnnnnnnnnnnnnnn

Encoding:

[35:31] Opcode=ANDR/ORR/XORR

[30:21] The index value of the target register.

[20:16] The extended operation code. 10000 indicates XORR *reg, #data*. It means that the contents of the register *REG* will be XORed with the value of the *data*, and the result will be stored in the register *REG*.

[15:00] The value of the operating data.

– XORR *reg, REG*

reg The name of the target register.

REG The name of the source register.

	$d_{35}-d_{31}$	$d_{30}-d_{21}$	$d_{20}-d_{16}$	$d_{15}-d_{00}$
Instr. Encoding	10110	RRRRRRRRRR	10001	000000rrrrrrrrrr

Encoding:

[35:31] Opcode=ANDR/ORR/XORR

[30:21] The index value of the source register.

[20:16] The extended operation code. 10001 indicates XORR *reg*, *REG*. It means that the contents of the register *reg* will be XORed with the contents of the register *REG*, and the result will be stored in the register *reg*.

[09:00] The index value of the target register.

– XORR *reg*, *?S|PRE(?S)*

reg The name of the target register.

S The name of the valued signal.

	$d_{35} - d_{31}$	$d_{30} - d_{22}$	$d_{21} - d_{16}$	$d_{15} - d_{00}$
Instr. Encoding	10110	sssssssss	010010	000000rrrrrrrrrr

Encoding:

[35:31] Opcode=ANDR/ORR/XORR

[30:22] The signal's unicode value. See also Section 4.3.

[21:16] The extended operation code. 010010 indicates XORR *reg*, *?S|PRE(?S)*. It means that the contents of the register *reg* will be XORed with the carried data (for *?S*) or the carried data in the previous tick (for *PRE(?S)*) of the valued signal *S*, and the result will be stored in the register *reg*.

[09:00] The index value of the target register.

Appendix B

An Introduction to the KEP Evaluation Platform

B.1 Function Description of the KEP Assembler Compiler

B.1.1 Options of the KEP Assembler Compiler

The KEP Assembler Compiler compiles a KEP assembler file into executable codes. The command for invoking the KEP compiler is:

kasm2klst [-d DebugLevel] -i *filename*

Options

- -d *DebugLevel*
  Specifies what file will be generated. When the *DebugLevel* is
  - 1
    The compiler will just generate the list file (.lst). The executable codes of the corresponding program is contained in the list file.
  - 2
    The compiler will generate the list file (.lst) and corresponding VHDL codes of the KEP's blocks. The parameters of these blocks depend on the values in the KEP configuration file (kep.cfg). See also Section B.1.2.
- -i *filename*
  Specifies the (input) KEP assembler language file.

B.1.2 The KEP Configuration File

The KEP configuration file defines some parameters to specify the generated VHDL code of the KEP blocks. This file is named kep.cfg and is at the same location as the KEP assembler compiler. If this file does not exist, the KEP assembler compiler will generate an initial template, the user can modify parameters in the template.

Following parameters can be assigned by the user:

I/O Signals
Specifies the amount of input, output, and inner signals. The maximum number is 127.

Datapath Width (bit)
Specifies the width of the data path of the KEP. It can be configured as 8, 16 or 32. The width of the data path will effect all data handling elements, *e. g.*, the Data Handling Block, the carried data width of valued signals which handled by the Interface Block.

Watcher Number
Specifies the amount of the general Watcher. The maximum number is 64.

Watcher Counter Width (bit)
Configures the maximum number of the count delay that can be handled by the Watcher directly. The range of this parameter is from 1 to 16 (16 bits maximum).

LWatcher Number
Specifies the number of general Local Watchers. The maximum number is 64.

LWatcher Counter Width (bit)
Configures the maximum count delay that can be handled by the Local Watcher directly. The range of this parameter is from 1 to 16 (65535 maximum).

TWatcher Counter Width (bit)
Configures the maximum count delay that can be handled by the Thread Watcher directly. The range of this parameter is from 1 to 16 (65535 maximum).

Counter Width (bit)
Configures the maximum count delay that can be handled by the AWAIT Element directly. The range of this parameter is from 1 to 16 (65535 maximum).

Thread Number
Specifies the amount of the threads, not including the initial thread (the thread 0). The range of the thread number is from 2 to 126.

Thread Priority Width
Configures the bit width of the thread priority. The range of this parameter is from 1 to 8, *i. e.*, the maximum priority of a thread can be up to 255.

Register Number
Specifies the maximum amount of KEP's registers. The register number ranges from 1 to 1024.

Tick Length
Specifies the maximum amount of KEP's tick length, see also Section 4.5. The tick length ranges from 1 to 21845 (*i. e.*, 65535/3).

Instruction Memory Size
Specifies the size of the instruction memory. The unit is the instruction word, *i. e.*, 36-bit. The address range of the instruction memory is from 512 (9-bit) to 65536 (16-bit) words.

Furthermore, the user can also decide which block should be generated by the KEP assembler compiler. This feature benefits for the VHDL synthesis – the corresponding VHDL synthesis tool can just re-synthesize modified blocks. Hence, the speed of the synthesis process can be improved. The following options can be assigned as YES or NO to enable or disable the generation of a block.

Generate blkcorev4
Switches whether the Decoder & Controller block should be re-generated or not. In most of conditions, this block is unnecessary to be re-generated.

Generate blkinterface
Switches whether the Interface Block should be re-generated or not. It should be set to YES whenever the amount of signals or the data path width is changed.

Generate blkreactive
Switches whether the Reactive Block should be re-generated or not. It should be set to YES whenever any parameter of reactive element, *e. g.*, Watcher number, counter width of AWAIT Element, or the thread number, is changed.

Generate blkthread
Switches whether the Thread Block should be re-generated or not. It should be set to YES whenever any thread related parameter is changed. Furthermore, when the tick length is modified, the Thread Block, which includes the code of Tick Manager, should also be updated.

Generate blkreg
Switches whether the Data Handling Block should be re-generated or not. It should be set to YES whenever any data related parameter is changed, *e. g.*, to define the new register number or data path width.

Generate KEPV4
Since all KEP blocks should be integrated and work together, a block which is named KEPV4 connects all others function blocks, instruction memory. To avoid some unwanted mistakes, we recommend it should always be set to YES.

In addition, to let the user know the basic requirement for an application, a kep.cfga file will be created whenever the compiler compiles an assembler file. The KEP Assembler Compiler will analyze the program and output the kep.cfga, which has the same format as the kep.cfg but sets parameters for minimum hardware usage for this specific program.

B.1.3 The Further Configuration

The configurability of the KEP goes beyond the above mentioned parameters. The user can modify some constants of the assembler compiler program for further improvements. The following descriptions illustrates some parameters that can be modified.

pre support
The pre expression can be directly supported by the KEP. However, this function is unnecessary in lots of applications. A boolean presupport parameter can be set to true or false to decide whether the hardware circuit for handling previous status and value of a signal should be created.

Allow lower priority parent thread
As mentioned in Section 4.2.3, the KEP provides a mechanism to allow the priority of the parent thread to be higher than that of its child threads. However, this mechanism is useless if the Esterel to KEP Assembler Compiler ensures this situation will not happen – which is the case under most conditions. Hence, the boolean parameter LowerPrioParentThreadSupport can be set to false to cancel this mechanism in order to reduce hardware usage.

Trap Number
Due to the exception handling mechanism of the KEP, several exceptions can be triggered simultaneously. The TrapNum parameter defines the maximum number of simultaneous running exceptions.

Show binary code
If the ShowBinCode is true, the binary code of instructions of a program will appear in the list file; or else, the list file will just contain the hexadecimal code of instructions.

The maximum length for recording the execution trace in a tick
During every tick, the TestDriver of the KEP Evaluation Platform will record the execution trace of the KEP. These information can be sent to the HOST for further analysis. The MaxTickLengthRecWidth parameter defines the maximum number of the recorded execution steps. The default value is 10 (10-bit), which means 1023 execution steps can be traced in maximum.

Debug mode
Generally, the KEP debugging depends on the HOST and TestDriver of the KEP Evaluation Platform by an evaluation strategy, see also Section 5.1.3. However,

the KEP also allows simulation strategy to debug its circuit by ModelSim. The SimDebugMethod parameter can be assigned as 0 to create the KEP Evaluation Platform; or else it will be switched to ModelSim simulation mode when it equals 1. In the ModelSim simulation mode, a 40-bit debug port will be created, and one can map any control signal of the KEP to this port for monitoring. In this mode, the instruction memory size cannot exceed 512 instruction words, and the execution code of the program will be initialized into the instruction memory.

B.2 Function Description of the TestDriver

The TestDriver is a part of the KEP Evaluation Platform on the FPGA board. After the program file (bit file) of the FPGA is downloaded into the chip, the TestDriver communicates with the HOST via the KEP evaluation program. It receives the commands and data from the HOST, tests and controls the KEP, completes the action according to the request, then returns the result to the HOST.

Consider a very simple KEP program, ABRO. We first compile it into executable code. Figure B.1 shows the KEP Machine Code Listing of this program. Following descriptions illustrate the functions of the TestDriver. The "->" means the HOST sends data to the TestDriver, and the "<-" means the TestDriver sends data to the HOST.

Verifying the communication

It verifies the communication between the TestDriver and the HOST:

```
-> "V"
<- "0123456789ABCDEFX" % The '0' will be sent first .
```

Getting the information of the on-board KEP

The TestDriver sends the information of the on-board KEP, which is controlled by the TestDriver to the HOST as a string, and then sends an "X" to denote this action is over. The information of the corresponding KEP is composed by 31 chars and depends on the configuration of the on-board KEP, see also Section B.1.2. The highest (31st) char codes a hex data to imply some further configuration of the KEP, see also Section B.1.3. Table B.1 illustrates the code format of this char.

Table B.2 illustrates the code format of the information string.

As described in Table B.2, the protocol is the following:

```
-> "N"
<- "02001E1404040C040408041F407D00AX"
   % The '0' will be sent first .
```

Writing the instruction memory

The TestDriver receives chars from the HOST, and decodes them to a binary code and then writes them to the instruction memory. A 56-bit word is used to describe

```
% ------------------------------------------------------------
% Generated by KEP4 Assembler Compiler Version 4.16
% Original file : abro.kasm
% ------------------------------------------------------------
INPUT A B R
OUTPUT O

% Signal codes

% Input ports (include local signals)
% [000000010] I/O(#1) A
% [000000100] I/O(#2) B
% [000000110] I/O(#3) R

% Output ports (include local signals)
% [000001000] I/O(#4) O

% Variable

% Summary:
% Input signals   :  3 (Pure: 3, Valued: 0)
% Output signals :  1 (Pure: 1, Valued: 0)
% Local signals   :  0 (Pure: 0, Valued: 0)
% Variables       :  0
%
% RAM Usage (in byte): 0
% Code size (in byte): 54
% Code size (in word): 12
%
% Watchers needed:        0
% LWatchers needed:       1
% Preemption by TWatcher: 0
%
% Watcher Num if no L|TWatcher: 1
%
% Threads needed:  2
%
% Instruction code:
% ------------------------------------------------------------
% Addr {Hex code}  Label:    Mnemonic
% ------------------------------------------------------------
[0000] {40000001E}          EMIT _TICKLEN, #10 %T0
[0001] {081800000}          AWAIT R %T0
[0002] {60180000B}     A0:  LWABORT R, A1 %T0
[0003] {481010006}          PAR 2, P1 (, 1) %T0
[0004] {481020007}          PAR 2, P2 (, 2) %T0
[0005] {480000008}          PARE P3, 0 %T0
[0006] {080800000}     P1:  AWAIT A %T1
[0007] {081000000}     P2:  AWAIT B %T2
[0008] {000200000}     P3:  JOIN 0 %T0
[0009] {402000000}          EMIT O %T0
[0010] {080010000}          HALT %T0
[0011] {000010002}     A1:  GOTO A0 %T0
% ------------------------------------------------------------
```

Figure B.1: The ABRO KEP Machine Code Listing.

Bit	Notes	Value	
		0	1
4th	KEP described language	VHDL	Esterel
3rd	Reserved		
2nd	Reserved		
1st	pre support	No	Yes

Table B.1: The code format of the 31st char of the information string.

Information string	Num of chars	Configuration parameter	Corresponding Hex codes
31st	1	KEP Type; *e. g.*, KEP without pre support	0
30th - 29th	2	Data path width; *e. g.*, 32-bit	20
28th - 26th	3	Signal number; *e. g.*, 30 signals in total	01E
25th - 24th	2	Thread number; *e. g.*, 20 thread in total	14
23rd - 22nd	2	Watcher number; *e. g.*, 4 Watcher	04
21st - 20th	2	Watcher counter width; *e. g.*, 4-bit	04
19th - 18th	2	LWatcher number; *e. g.*, 12 Local Watcher	0C
17th - 16th	2	LWatcher counter width; *e. g.*, 4-bit	04
15th - 14th	2	TWatcher counter width; *e. g.*, 4-bit	04
13th - 12th	2	Delay counter width; *e. g.*, 8-bit	08
11th - 10th	2	Prio value width; *e. g.*, 4-bit	04
9th - 7th	3	Register number; *e. g.*, 500	1F4
6th - 3rd	4	Tick length; *e. g.*, 2000	07D0
2nd - 1st	2	Instruction memory address width; *e. g.*, 10-bit	0A

Table B.2: The code format of the information string of the on-board KEP.

the content of an instruction memory cell. The first char encodes 4 bits as the control signal for writing them to the memory, which are followed by 4 chars (16 bits) to describe the address of this instruction. Then the code of the instruction (36 bits) will be translated. The protocol is the following:

```
−> "W"
−> "6000040000001E"    % Sent the instruction code at the first time
                       % The '6' will be sent first .
−> "7000040000001E"    % Sent the instruction code twice
                       % If they are not equal (except for the control char
                       % <−"X"
−> "60001081800000"    % AWAIT R (0x081800000) at 0001 address
−> "70001081800000"
...
−> "60010080010000"    % HALT (0x080010000) at 0010 address
−> "70010080010000"


−> "60011000010002"    % GOTO A0 (0x000010002) at 0011 address
−> "70011000010002"
−> "X"                 % End writing
```

The HOST will transfer every instruction code twice. At first, the control char is "6", and it will be changed to "7" the second time. The TestDriver will compare the remaining received data. If they are different, the TestDriver will report the error message by sending an "X" to the HOST.

Resetting the KEP

The TestDriver sets the Reset pin of the KEP and drives the clock of the KEP for some cycles, then sends an "X" to the HOST. However, the TestDriver will not release the Reset pin until the TestDriver receives a "T" char for starting a tick. The protocol is the following:

```
–> "R"          % Entering this function
<– "X"
```

Getting the status of input signals

The TestDriver receives chars from the HOST, and decodes them to a binary code and then maps these binary code to the Sinout port of the KEP. For example, assume the input signal B and R is present. The signal B is the 2nd signal of the Sinout port, and the signal R is the 3rd one. Hence, the status of input signals will be encoded as "110b", *i. e.*, "0x6" in hexadecimal. The HOST will send an "I" as the command to let the TestDriver enter this function. After the encode of the status of input signals is sent, an "X" will make the TestDriver exit this function. The protocol is the following:

```
–> "I"          % Entering this function
–> "6"
–> "X"
```

Note the lower char of the input signal status code will be transferred at first. For example, if the code is "0x5A3", then the "3" will be transferred at first, and the "5" will be sent finally.

Running a Tick

The TestDriver drives the clock signal of the KEP, stores the execution trace by saving the instruction memory address for every instruction cycle, watches the status of the Tick signal, and stops driving the clock signal when the Tick signal is down. Then the TestDriver sends the "X" to the HOST to denote this action is over. The protocol is the following:

```
–> "T"          % Entering this function
<– "X"
```

Sending the status of output signals

The TestDriver sends the status of the Sinout to the HOST. The form is similar as that of the input signals mentioned above. For example, assume the O signal is present, the code of output signal will equal "0x8" because the 4th bit of the Sinout port is set as '1'. Furthermore, the inner signal of the KEP will be regarded

as an output signal, and its status will also be encoded and sent to the HOST. The protocol is the following:

```
–> "O"          % Entering this function
<– "8"
<– "X"
```

Sending the valid tick length

The TestDriver transfers the information of valid tick length to the HOST. As mentioned in Section 4.5, although the KEP employs a specified fixed number of instruction cycles as its tick length, the actual executed number of instruction cycles of a tick can be different in every tick. It will be encoded and received by the HOST for further analysis. Assume a program executes 10 instruction cycles in a tick, the code of the valid tick length should be "000A", hexadecimal (always 16-bit). Note that the lower char of the tick length code will be transferred first. For example, if the code is "000A", then the "A" will be transferred first, and the "0" will be sent finally. The protocol is the following:

```
–> "L"          % Entering this function
<– "A000"
<– "X"
```

Sending the execution trace

The TestDriver sends the address of executed instructions of the last tick to the HOST. The width of the address code is 16 bits (LSB). For example, assume the KEP just executed two instructions in the last tick, and the order is first [0183] and then [0184]. Hence, the execution trace will be encoded as "38104801" – note the lower char will be transferred first. The protocol is the following:

```
–> "M"          % Entering this function
<– "38104810"
<– "X"
```

Choosing a valued signal

The TestDriver receives the index code of a signal from the HOST, and decodes and writes it to the corresponding port to index a valued signal. This function is started by a "D" char and finished by an "X" char. For example, if the HOST wants to read/write the carried data of the "0x1A" signal, the "A1" will be sent to the TestDriver – the lower char will be transferred first. The protocol is the following:

```
–> "D"          % Entering this function
<– "A1"
<– "X"
```

Getting the carried data of a valued signal

After the target valued signal was indexed by the *choosing a valued signal* function, the TestDriver could receive the content for this valued signal (32 bits) from

the HOST. For example, if the content of the "0x1A" signal should be set to "00ABCD53", the TestDriver will receive "35DCBA00" from the HOST and write it to the corresponding signal. The protocol is the following:

```
–> "G"          % Entering this function
–> "35DCBA00"
–> "X"
```

Sending the carried data of a valued signal

After the target valued signal was indexed by the *choosing a valued signal* function, the TestDriver could send the content (carried data) of the indexed valued signal (32 bits) to the HOST. For example, if the content of the "0x1A" signal is "00ABCD53", the TestDriver will send "35DCBA00" to the HOST.

```
–> "P"          % Entering this function
<– "35DCBA00"
<– "X"
```

B.3 Function Description of the KEP Evaluation Program

The KEP Evaluation Program is the main tool to evaluate and debug the KEP programs. Again we use ABRO as a running example.

B.3.1 Starting an Evaluation

To invoke the graphical KEP Evaluation Program, the following shell command should be executed:

KEPEvalBench.exe

The KEP Evaluation Program starts by displaying a window, as shown on Figure B.2. The left five icons from left to right correspond to these functions: load list file, verify communication, get information of the on-board KEP, write code into instruction memory, reset the KEP, and run a tick. The remaining five icons are used for validation. From left to right: load esi file, start validation, stop validation, save trace file, and compare the trace file with the eso which generated by the EStudio.

After a list file is loaded, input signals of this KEP program appear in the Input (signal) Window, and the output and inner signals are shown in the Output/Signal (signal) Window. The program, which contains address, hex code, and assembler instructions, is shown in the Program Window. The user can download executable codes of this program into the instruction memory of the KEP. Before the download process starts, the KEP Evaluation Program will check the configuration parameters of the on-board KEP and compare them with the required hardware resources to execute the current assembler

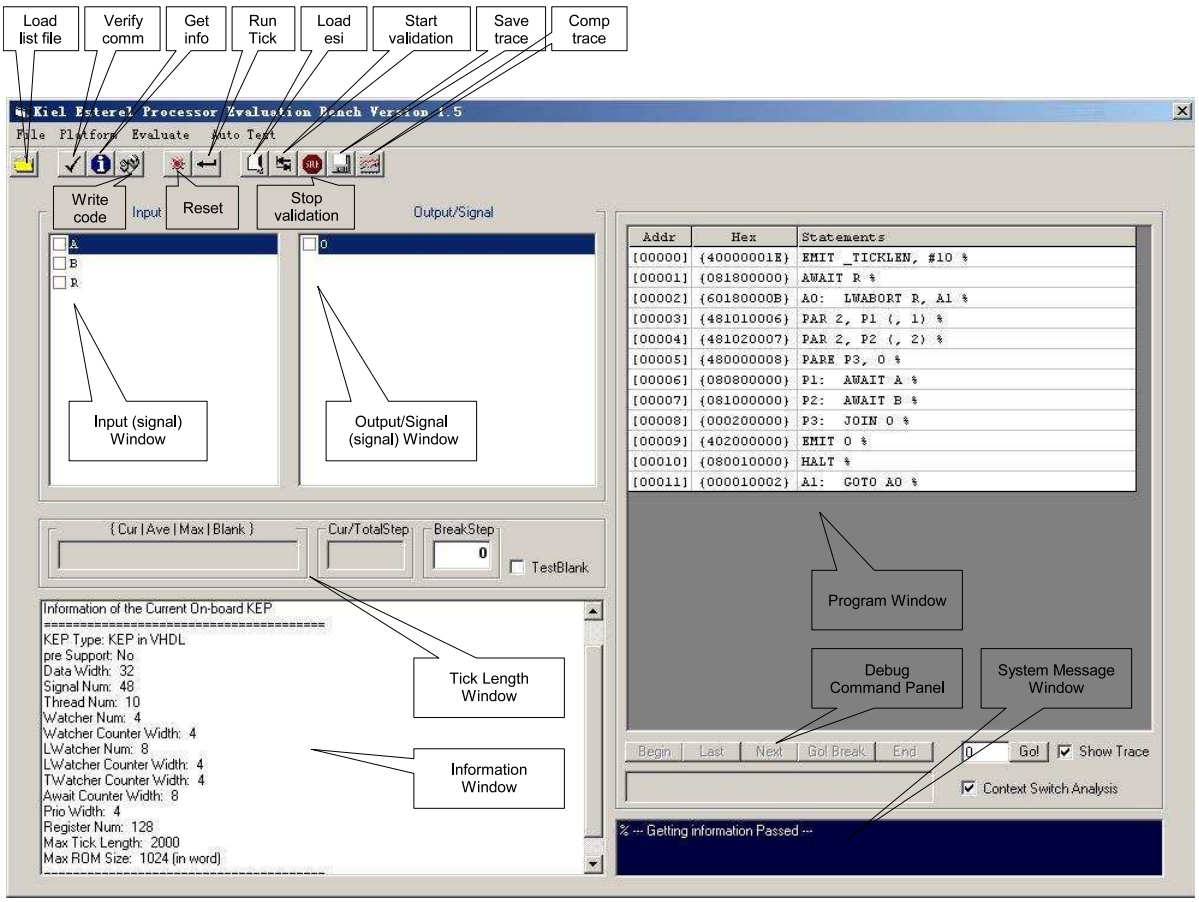


Figure B.2: The KEP Evaluation Program.

program. If the on-board KEP cannot handle this program, an error message appears to warn the user. The System Message Window, which is located at the bottom right of the panel, reports the corresponding information.

B.3.2 Debugging a Program

The KEP Evaluation Program allows the user to build his or her inputs, drive the KEP, and view the result of the program.

To build the input, one should set the status of input signals. In the Input Window, each item is labeled with the name of the signal. A little checkbox on the left of each item can be used to set the status. If the signal is a valued signal, it could be clicked with the right mouse button to popup a dialog box, where the user can write the value.

After the input event is built, the user can trigger the reaction by clicking the run a tick button. After the execution of this tick is finished, the checkbox of a present output

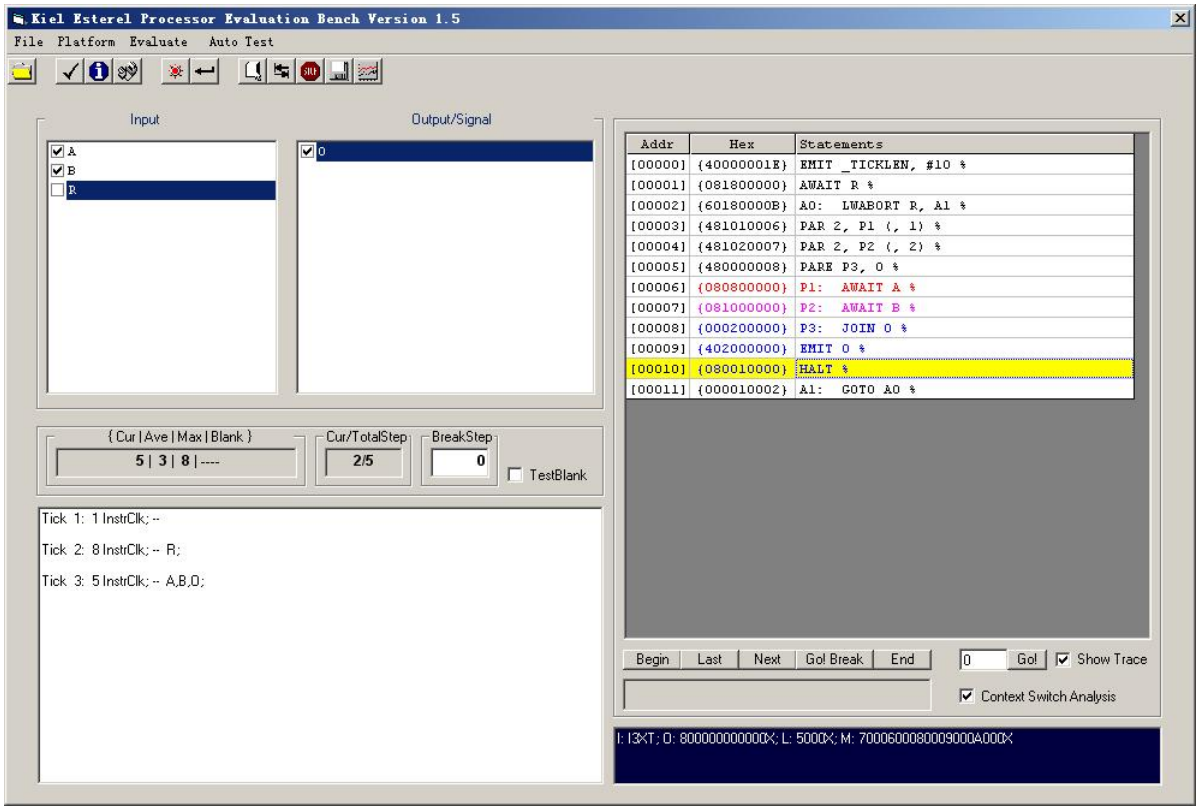


Figure B.3: Debugging a Program.

signal will be marked. The Tick Length Windows will show the information for the valid tick length. Corresponding information of the valid tick length and signal trace of every tick are also shown in the Information Window, as shown in Figure B.3.

To follow the execution trace of a tick, one can click the Begin button in the Debug Command Panel, which is below the Program Window, to put the control to the first executed instruction in this tick. Then the Next and Last buttons allow the control following the execution trace of current tick step by step. If an instruction has been executed in this tick, it will be marked by a different color. A pink instruction implies the control reached it; a blue instruction means the control has not reached it yet; and a red instruction indicates where the control is currently. Clicking a blue instruction sets a breakpoint indicated by a yellow background. The GoBreak button runs the program until reaching the break point.

The Tick Length Window will show some further information about instruction cycles. For example, in Figure B.3, the Tick Length Window notes that control rests at the 2nd instruction of this tick; and the maximum encountered number of instructions per tick is five. Note that this should never exceed the analyzed WCRT.

B.3.3 Validating a Program

To validate a KEP program, the user should first load the corresponding Esterel Simulator Input (.esi) file, and then start the validation. The KEP Evaluation Program will read the input trace, build the input event, send the tick, and get the result. The process will continue until all input traces in the .esi file are finished. One can save the execution trace (input and output of every tick) as a .keso file – which is compatible with the Esterel Simulator Output trace (.eso) file. The user can compare these two file – the .keso and the .eso to ensure the execution traces are exactly the same. The corresponding information appears in the Information Window.

To validate a batch of programs efficiently, the KEP Evaluation Program also provides a command line interface:

- KEPEvalBench.exe -A *filename*
  Execute the whole process of validation automatically: load the program, write the instruction memory, read the .esi file, send the ticks, and compare the results. The corresponding trace files of *filename.lst* a should be named *filename.esi* and *filename.eso*. The execution of these process will be reported in a _EXEPROC.LOG file, and the result of the comparison is given in a _CMP_*filename*.LOG file. If the validation fails, an _EVALERR.LOG file is generated to report the error.

- KEPEvalBench.exe -T *filename*
  Similar as the -A option. Furthermore, a *filename.ktr* file will be created, which contains the instruction execution trace of every tick.

- KEPEvalBench.exe -M *filename*
  Similar as the -A option. However, it targets a MicroBlaze platform as alternative the reference for the KEP.

- KEPEvalBench.exe -V
  Verify the communication between the TestDriver and the HOST. If the verification fails, an _EVALERR.LOG file will be generated to report the error.

- KEPEvalBench.exe -N
  Get the information of the on-board KEP. A _KEPINF.LOG file will be generated to report the information in detail.

- KEPEvalBench.exe -R
  Reset the KEP. If the process fails, an _EVALERR.LOG file will be generated to report the error.

- KEPEvalBench.exe -D *filename*
  Write the executable code of the *filename.lst* program into the instruction memory of the KEP. If the process fails, an _EVALERR.LOG file will be generated to report the error.

- KEPEvalBench.exe -S *filename*
  Evaluate the *filename.lst*. It loads the corresponding *filename.esi* file, runs the KEP, and saves the result into the *filename.keso* file.

- KEPEvalBench.exe -C *filename*
  Compare the *filename.keso* with the *filename.eso*. The result is given in a _CMP_*filename*.LOG file. If they do not match, an _EVALERR.LOG file is generated to report the error.

For the evaluation of a KEP program (-A or -S option), the KEP Evaluation Program will also write the information of the program and its execution to an autotestreport.tex file if the evaluation is successful. This file will be further compiled to a report after all programs are validated.

Bibliography

[1] Alauddin Alomary, Takeharu Nakata, Yoshimichi Honma, Masaharu Imai, and Nobuyuki Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 526–532, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. 20, 122

[2] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. `http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf`. 57

[3] L. Arditi, A. Bouali, H. Boufaied, G. Clave, M. Hadj-Chaib, and R. de Simone. Using Esterel and formal methods to increase the confidence in the functional validation of a commercial DSP. In *Proceedings of the 4th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'99)*, June 1999. `http://citeseer.ist.psu.edu/arditi99using.html`. 2

[4] Peter J. Ashenden. The VHDL cookbook, July 1990. `http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf`. 87

[5] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, 1993. `http://dx.doi.org/10.1109/2.204677`. 122

[6] Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen M. Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciono Lavagno, Alberto Sangiovanni-Vincentelli, and Kei Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach.* Kluwer Academic Publishers, April 1997. 8, 10, 106, 109

[7] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Patel, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *CODES'02: Proceedings of the Tenth International Symposium On Hardware/Software Codesign*, pages 151–156, New York, NY, USA, 2002. ACM Press. `http://doi.acm.org/10.1145/774789.774820`. 8, 9

[8] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. In *Readings in hardware/software co-design*, pages 231–248. Kluwer Academic Publishers, Norwell, MA, USA, 2002. 98

[9] Luca Benini and Giovanni De Micheli. Automatic synthesis of low-power gated-clock finite-state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*, 15:630 – 643, June 1996. 98

[10] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003. 2, 9, 10

[11] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–98, Charleston, South Carolina, 1993. 2, 20, 109

[12] Gérard Berry. Programming a digital watch in Esterel v3. Technical Report RR-1032, INRIA, 1988. `http://citeseer.ist.psu.edu/berry91programming.html`. 110

[13] Gérard Berry. A hardware implementation of pure ESTEREL. Technical Report de recherche 1479, INRIA, 1991. 20

[14] Gérard Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992. 7, 9

[15] Gérard Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–93, London, UK, 1993. Springer-Verlag. 25, 27

[16] Gérard Berry. The semantics of pure Esterel. In *Proceedings of the Marktoberdorf Intl. Summer School on Program Design Calculi*, Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1993. `http://citeseer.ist.psu.edu/berry93semantics.html`. 20

[17] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999. `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`. 10, 11, 20

[18] Gérard Berry. *The Esterel v5 Language Primer*, 1999. `ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps`. 1, 3, 22, 25, 109

[19] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000. `ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf`. 2, 4, 7, 20, 22, 23, 34, 41, 91, 93, 98, 122

[20] Gérard Berry. The Foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte. 42

[21] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *Lecture Notes in Computer Science (LNCS)*, pages 389–448. Springer-Verlag, 1984. 2

[22] Gérard Berry and Georges Gonthier. Incremental development of an HDLC entity in Esterel. *Computer Networks and ISDN Systems.*, 22(1):35–49, 1991. 2

[23] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992. `http://citeseer.nj.nec.com/berry92esterel.html`. 2, 3, 8, 9, 20, 54

[24] Gérard Berry and Ellen M. Sentovich. An implementation of constructive synchronous programs in polis. *Formal Methods in System Design*, 17(2):135–161, 2000. `http://dx.doi.org/10.1023/A:1008796718837`. 8

[25] Kevin Bixler and David Dye. Physical synthesis and optimization with ISE software. *Xcell Journal*, 2005. 121

[26] Marian Boldt. A compiler for the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, November 2007. To appear. 5, 13, 42

[27] Marian Boldt. Worst-case reaction time analysis for the KEP3. Study thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2007. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-st.pdf`. 5, 13, 96

[28] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. In *Proceedings of the Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P07)*, Braga, Portugal, March 2007. 5, 13, 44, 96

[29] Amar Bouali. XEVE: An Esterel verification environment. In *Proceedings of the 10th Internation Conference Computer-Aided Verification (CAV'98)*, volume 1427. Lecture Notes in Computer Science (LNCS), 1998. 106

[30] Hedi Boufaied, Arnaud Cavanie, Bernard Dion, Sylvan Dissoubray, Laurent Arditi, Gaël Clave, and Charles Andre. An experiment in using Esterel studio for modeling the control of mobile communication architectures. In *Proceedings of the Sophia-Antipolis forum on MicroElectronics (SAME) congress*, Sophia Antipolis, October 1999. 2

[31] Klaus Buchenrieder, Andreas Pyttel, and Christian Veith. Mapping statechart models onto an FPGA-based ASIP architecture. In *EURO-DAC '96/EURO-VHDL '96: Proceedings of the conference on European design automation*, pages 184–189, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. 8

[32] Annette Bunker, Ganesh Gopalakrishnan, and Sally A. Mckee. Formal hardware specification languages for protocol compliance verification. *ACM Transactions on Design Automation of Electronic Systems*, 9(1):1–32, 2004. `http://doi.acm.org/10.1145/966137.966138`. 7

[33] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 13–27, New York, NY, USA, 1989. ACM Press. `http://doi.acm.org/10.1145/73141.74820`. 44

[34] Paul Caspi, Alain Girault, and Daniel Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3), 1999. 120

[35] Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`. 107, 109

[36] Newton Cheung, Jorg Henkel, and Sri Parameswaran. Rapid configuration and instruction selection for an ASIP: A case study. In *Proceedings of Design, Automation and Test in Europe (DATE '03)*, pages 802–807, Los Alamitos, CA, USA, 2003. IEEE Computer Society. 122

[37] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, August 1994. `http://citeseer.ist.psu.edu/chiodo94formal.html`. 8

[38] C.M.Edmund Chow, Joyce S.Y.Tong, M.W.Sajeewa Dayaratne, Partha S Roop, and Zoran Salcic. RePIC - A New Processor Architecture Supporting Direct Esterel Execution. School of Engineering Report No. 612, University of Auckland, 2004. 7, 12, 64

[39] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In Florence Maraninchi, Alain Girault, and Eric Rutten, editors, *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002. `http://www.elsevier.com/gej-ng/31/29/23/117/53/34/65.5.010.pdf`. 8, 10, 43, 110

[40] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: Adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference*

on Languages, Compilers, and Tools for Embedded Systems (LCTES), pages 69–77, New York, NY, USA, 2005. ACM Press. `http://www.cs.rice.edu/~harv/lctes05.pdf`. 122

[41] M. W. Sajeewa Dayaratne, Partha S. Roop, and Zoran Salcic. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, April 2005. 12, 14

[42] Alexander G. Dean. Efficient real-time fine-grained concurrency on low-cost microcontrollers. *IEEE Micro*, 24(4):10–22, 2004. `http://dx.doi.org/10.1109/MM.2004.27`. 56

[43] Stephen A. Edwards. CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/~sedwards/cec/`. 42, 104, 109, 110

[44] Stephen A. Edwards. An Esterel compiler for a synchronous/reactive development system. Technical Report UCB/ERL M94/43, EECS Department, University of California, Berkeley, 1994. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1994/2572.html`. 8, 9

[45] Stephen A. Edwards. Compiling Esterel into Sequential Code. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES 99)*, May 1999. `http://www1.cs.columbia.edu/~sedwards/papers/edwards1999compiling.pdf`. 109

[46] Stephen A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), February 2002. 8, 10, 43

[47] Stephen A. Edwards. High-Level Synthesis from the Synchronous Language Esterel. *Proceedings of the International Workshop of Logic and Synthesis (IWLS)*, June 2002. 7, 9

[48] Stephen A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, April 2003. 9, 10

[49] Stephen A. Edwards, Nicholas Halbwachs, Reinhard von Hanxleden, and Thomas Stauner. 04491 executive summary – synchronous programming - synchron'04. In Stephen A. Edwards, Nicolas Halbwachs, Reinhard v. Hanxleden, and Thomas Stauner, editors, *Synchronous Programming - SYNCHRON'04*, number 04491 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/opus/volltexte/2005/195`. 1, 2

[50] Stephen A. Edwards, Vimal Kapadia, and Michael Halas. Compiling Esterel into static discrete-event code. In *International Workshop on Synchronous Languages,*

Applications, and Programming (SLAP'04), Barcelona, Spain, March 2004. 8, 10, 110

[51] Stephen A. Edwards and Jia Zeng. Code generation in the columbia esterel compiler. *EURASIP Journal on Embedded Systems*, pages Article ID 52651, 31 pages, 2007. http://www.hindawi.com/GetArticle.aspx?doi=10.1155/2007/52651. 10

[52] Esterel Technologies. Free Software Esterel Compiler. http://www.esterel-technologies.com/technology/free-software/esterel-compiler.html. 110

[53] Esterel Technologies. Company homepage. http://www.esterel-technologies.com. 42

[54] Esterel.org. Esterel history. http://www-sop.inria.fr/esterel.org/Html/History/History.htm. 9

[55] Jean-Daniel Fekete and Martin Richard. Esterel meets Java: Building reactive synchronous programs in Java. Technical report, École des Mines de Nantes, December 1998. http://www.lri.fr/~fekete/ps/EmeetsJ.pdf. 2

[56] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987. http://doi.acm.org/10.1145/24039.24041. 44

[57] Sascha Gädtke. Hardware/Software Co-Design für einen Reaktiven Prozessor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2007. 5, 13

[58] Sascha Gädtke, Xin Li, Marian Boldt, and Reinhard von Hanxleden. HW/SW Co-Design for a Reactive Processor. In *Proceedings of the Student Poster Session at the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006. With accompanying poster. 5, 13, 121

[59] Philippe Garrault and Brian Philofsky. HDL coding practices to accelerate design performance. Xilinx white paper wp231, Xilinx Inc, January 2006. http://direct.xilinx.com/bvdocs/whitepapers/wp231.pdf. 57

[60] Alain Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs (SLAP'05)*, Electronic Notes in Theoretical Computer Science, Edinburgh, UK, April 2005. Elsevier Science. http://www.inrialpes.fr/pop-art/people/girault/Publications/Slap05. 120

[61] Bita Gorjiara and Daniel D. Gajski. Custom processor design using NISC: A case-study on DCT algorithm. In *3rd Workshop on Embedded Systems for Real-Time Multimedia*, pages 55–60, 2005. 122

[62] Michael Gschwind. Instruction set selection for ASIP design. In *Proceedings of the seventh international workshop on Hardware/software codesign (CODES'99)*, pages 7–11, New York, NY, USA, 1999. ACM Press. `http://doi.acm.org/10.1145/301177.301187`. 20

[63] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9), September 1991. 2

[64] Sumit Gupta, Manev Luthra, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Hardware and interface synthesis of FPGA blocks using parallelizing code transformations. In *PDCS03: Proceedings of the Parallel and Distributed Computing and Systems*, November 2003. `http://citeseer.ist.psu.edu/659443.html`. 121

[65] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. `http://citeseer.nj.nec.com/halbwachs91synchronous.html`. 2

[66] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press. `citeseer.ist.psu.edu/hauser97garp.html`. 122

[67] Steve Heath. *Embedded Systems Design*. Butterworth-Heinemann, Newton, MA, USA, 1997. 87

[68] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 315–326, New York, NY, USA, 2002. ACM Press. `http://doi.acm.org/10.1145/512529.512567`. 122

[69] Xilinx Inc. Xilinx Spartan-3 web power tool version 8.1.01. `http://www.xilinx.com/cgi-bin/power_tool/power_Spartan3`. 116

[70] Xilinx Inc. *Low Power Design with CoolRunner-II CPLDs*, May 2002. `http://www.xilinx.com/bvdocs/appnotes/xapp377.pdf`. 98

[71] Xilinx Inc. *PowerPC Processor Reference Guide*, 2003. `http://www.xilinx.com/bvdocs/userguides/ug011.pdf`. 116

[72] Xilinx Inc. *Using Block RAM in Spartan-3 FPGAs*, 2003. http://www.xilinx.com/bvdocs/appnotes/xapp463.pdf. 91

[73] Xilinx Inc. *Platform Studio User Guide*, 2004. http://www.xilinx.com/ise/embedded/edk_docs.htm. 115

[74] Xilinx Inc. *MicroBlaze Processor Reference Guide*, 2005. http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf. 116

[75] Makiko Itoh, Shigeaki Higaki, Yoshinori Takeuchi, Akira Kitajima, Masaharu Imai, Jun Sato, and Akichika Shiomi. PEAS-III: An ASIP design environment. In *ICCD '00: Proceedings of the International Conference on Computer Design*, page 430, Los Alamitos, CA, USA, 2000. IEEE Computer Society. 8, 20

[76] Jeffrey A. Jacob and Paul Chow. Memory interfacing and instruction specification for reconfigurable processors. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 145–154, New York, NY, USA, 1999. ACM Press. http://doi.acm.org/10.1145/296399.296446. 121

[77] Hahnsang Kim and Thierry Turletti. An Esterel-based development environment for designing software radio applications. Technical report rr. 4256, INRIA, September 2001. http://citeseer.ist.psu.edu/kim01esterelbased.html. 2

[78] Tim Kogel and Heinrich Meyr. Heterogeneous MP-SoC: the solution to energy-efficient signal processing. In *DAC '04: Proceedings of the 41th Conference on Design Automation*, pages 686–691, 2004. 13

[79] T. John Koo, Bruno Sinopoli, Alberto Sangiovanni-Vincentelli, and Shankar Sastry. A formal approach to reactive system design: Unmanned aerial vehicle flight management system design example. In *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 522–527, August 1999. 8

[80] Kayhan Küçükçakar. An ASIP design methodology for embedded systems. In *Proceedings of the seventh international workshop on Hardware/software codesign (CODES'99)*, pages 17–21, New York, NY, USA, 1999. ACM Press. http://doi.acm.org/10.1145/301177.301190. 13

[81] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 302–313, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. http://doi.acm.org/10.1145/191995.192056. 122

[82] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The Dash prototype: Implementation and performance. In *ISCA '92: Proceedings of the 19th annual international symposium on computer architecture*, pages 92–103, New York, NY, USA, 1992. ACM Press. `http://doi.acm.org/10.1145/139669.139706`. 122

[83] Xin Li. The VHDL code of the Kiel Esterel Processor. `https://rtsys.informatik.uni-kiel.de/svn/kep/KEP/vhdl/`. 76, 88, 104

[84] Xin Li, Marian Boldt, and Reinhard von Hanxleden. Compiling Esterel for a multi-threaded reactive processor. Technical Report 0603, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2006. Revised September 2006. 13, 120

[85] Xin Li, Marian Boldt, and Reinhard von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, October 21–25 2006. 5, 13, 89, 120

[86] Xin Li, Jan Lukoschus, Marian Boldt, Michael Harder, and Reinhard von Hanxleden. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 225–236, New York, NY, USA, September 2005. ACM Press. 5, 12, 13, 33, 44, 89, 96

[87] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. Technical Report 0509, Christian-Albrechts-Universität Kiel, Department of Computer Science, November 2005. `http://www.informatik.uni-kiel.de/reports/2005/0509.html`. 108

[88] Xin Li and Reinhard von Hanxleden. KEP2 (Kiel Esterel Processor 2): The Esterel Processor. Technical Report 0506, Christian-Albrechts-Universität Kiel, Department of Computer Science, April 2005. `http://www.informatik.uni-kiel.de/reports/2005/0506.html`. 12, 33, 64, 89, 108

[89] Xin Li and Reinhard von Hanxleden. The Kiel Esterel Processor - a semi-custom, configurable reactive processor. In Stephen A. Edwards, Nicolas Halbwachs, Reinhard v. Hanxleden, and Thomas Stauner, editors, *Synchronous Programming - SYNCHRON'04*, number 04491 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/opus/volltexte/2005/159`. 12, 33, 64, 89, 108

[90] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006. 9, 13, 109

[91] Xin Li and Reinhard von Hanxleden. Light-weight, predictable reactive processing—the kiel esterel processor. In *Proceedings of the Design, Automation and Test in Europe University Booth (DATE'07)*, Nice, France, April 2007. With accompanying poster. 106

[92] Jan Lukoschus. *Removing Cycles in Esterel Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, July 2006. `http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_2015`. 42

[93] Manev Luthra, Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Interface synthesis using memory mapping for an fpga platform. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, page 140, Washington, DC, USA, 2003. IEEE Computer Society. 8

[94] Philippe Magarshack and Pierre G. Paulin. System-on-chip beyond the nanometer wall. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 419–424, New York, NY, USA, 2003. ACM Press. `http://doi.acm.org/10.1145/775832.775943`. 120, 122

[95] Sumio Morioka. CQPIC: PIC micro computer free soft IP. `http://www02.so-net.ne.jp/~morioka/cqpic.htm`. 12

[96] Matthew Ouellette and Dan Connors. Analysis of hardware acceleration in reconfigurable embedded systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 168.1, Washington, DC, USA, 2005. IEEE Computer Society. 108

[97] Massoud Pedram. Power optimization and management in embedded systems. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 239–244, New York, NY, USA, 2001. ACM Press. `http://doi.acm.org/10.1145/370155.370333`. 98

[98] Becky Plummer, Mukul Khajanchi, and Stephen A. Edwards. An Esterel virtual machine for embedded systems. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, March 2006. 122

[99] POLIS: a framework for hardware/software co-design of embedded systems. `http://embedded.eecs.berkeley.edu/research/hsc/polis_files.html`. 8

[100] Dumitru Potop-Butucaru and Robert de Simone. *Optimization for faster execution of Esterel programs*, pages 285–315. Kluwer Academic Publishers, Norwell, MA, USA, 2004. 10

[101] David J. Pursley and Brett L. Cline. A practical approach to hardware and software SoC tradeoffs using high-level synthesis for architectural exploration. In *Proceedings of the Global Signal Processing Expo (GSPx) Conference*, April 2003. 120

[102] P. S. Roop, Z. Salcic, and M. W. S. Dayaratne. Towards Direct Execution of Esterel Programs on Reactive Processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)*, Pisa, Italy, September 2004. 2

[103] Z. Salcic, P. S. Roop, M. Biglari-Abhari, and A. Bigdeli. REFLIX: A Processor Core with Native Support for Control Dominated Embedded Applications. *Elsevier Journal of Microprocessors and Microsystems*, 28:13–25, 2004. 12

[104] Zoran A. Salcic, Dong Hui, Partha S. Roop, and Morteza Biglari-Abhari. REMIC: Design of a reactive embedded microprocessor core. In *Proceedings of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 977–981, Shanghai, China, 2005. 12

[105] Zoran A. Salcic, Partha S. Roop, Morteza Biglari-Abhari, and Abbas Bigdeli. REFLIX: A processor core for reactive embedded applications. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Proceedings of the 12th International Conference on Filed Programmable Logic and Applications (FPL-02)*, volume 2438 of *Lecture Notes in Computer Science*, pages 945–945, Montpellier, France, September 2002. Springer. 9, 12

[106] Klaus Schneider, Jens Brandt, and Tobias Schuele. A verified compiler for synchronous programs with local declarations. In *Synchronous Languages, Applications and Programming (SLAP)*, Electronic Notes in Theoretical Computer Science, Barcelona, Spain, 2004. `http://citeseer.ist.psu.edu/article/schneider04verified.html`. 11, 12

[107] Dongwan Shin and Daniel Gajski. Interface synthesis from protocol specification. Technical Report CECS-02-13, University of California, Irvine, April 2002. 8

[108] R. K. Shyamasundar and J. V. Aghav. Validating real-time constraints in embedded systems. In *PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, page 347, Washington, DC, USA, 2001. IEEE Computer Society. 96

[109] Christian Siemers and Harald Richter. Reconfigurable Microprocessor and Microcontroller - Architectures and Classification. In *Proceedings Embedded World 2004 Conference*, pages 271–287, Nuremberg, Germany, February 2004. 122

[110] Cristian Soviani and Stephen A. Edwards. Challenges in synthesizing fast control-dominated circuits. In *International Workshop on Logic and Synthesis (IWLS)*, Lake Arrowhead, California, June 2005. 7

[111] Cristian Soviani, Jia Zeng, and Stephen A. Edwards. Improved controller synthesis from Esterel. Technical Report CUCS-015-04, Columbia University, 2004. 7

[112] Cristian Soviani, Jia Zeng, and Stephen A. Edwards. Sequential challenges in synthesizing Esterel. Technical Report CUCS-051-04, Columbia University, 2004. 110

[113] Synopsys Inc. *Guide to HDL Coding Styles for Synthesis*, 2005. 57, 121

[114] Olivier Tardieu and Robert de Simone. Instantaneous termination in pure Esterel. In *Static Analysis Symposium*, San Diego, California, June 2003. 11

[115] Olivier Tardieu and Robert de Simone. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign*, San Diego, CA, USA, 2004. 12

[116] Esterel Technologies. *The Esterel v7 Reference Manual Version v7_30 C initial IEEE standardization proposal*, November 2005. `http://www.esterel-technologies.com/files/Esterel-Language-v7-Ref-Man.pdf`. 7, 122

[117] Hervé Touati and Gérard Berry. Optimized controller synthesis using Esterel. In *International Workshop on Logic Synthesis (IWLS'93)*, 1993. `http://citeseer.ist.psu.edu/160147.html`. 7, 109

[118] Reinhard von Hanxleden and Xin Li. The Kiel Esterel Processor Homepage. `http://www.informatik.uni-kiel.de/rtsys/kep/`. 121

[119] Reinhard von Hanxleden, Xin Li, Partha Roop, Zoran Salcic, and Li Hsien Yoong. Reactive processing for reactive systems. *ERCIM News*, 66:28–29, October 2006. `http://www.ercim.org/publication/Ercim_News/EN67.pdf`. 12

[120] Michael J. Wirthlin and Brad L. Hutchings. A dynamic instruction set computer. In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, pages 99–107, Washington, DC, USA, 1995. IEEE Computer Society. 122

[121] Ralph D. Wittig and Paul Chow. OneChip: An FPGA processor with reconfigurable logic. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press. `citeseer.ist.psu.edu/wittig95onechip.html`. 121

[122] Wayne Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, 2003. 120

[123] Wayne Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: Proceedings of the 41th conference on Design automation*, New York, NY, USA, 2004. ACM Press. 120

[124] Li Hsien Yoong, Partha Roop, Zoran Salcic, and Flavius Gruian. Compiling Esterel for distributed execution. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, March 2006. 14

[125] Jia Zeng, Cristian Soviani, and Stephen A. Edwards. Generating Fast Code from Concurrent Program Dependence Graphs. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2004. 44