INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

KEP2 (Kiel Esterel Processor 2)
THE ESTEREL PROCESSOR

Xin Li, Reinhard von Hanxleden

Bericht Nr. 0506

June 2005

Revised August 2005

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

KEP2 (Kiel Esterel Processor 2)
THE ESTEREL PROCESSOR

Xin Li, Reinhard von Hanxleden

E-mail:
{xli,rvh}@informatik.uni-kiel.de

Abstract

The concurrent synchronous language Esterel allows to program reactive systems in an abstract, concise manner. Several preemption operators, such as suspension or weak and strong abortion, are provided to directly express reactive behavior with a deterministic, mathematically precise semantics, without the inherent need for the support of a resource-consuming, typically non-deterministic operating system.

An Esterel program is typically first translated into other, non-synchronous high-level languages, such as VHDL or C, and then compiled further into hardware or software. Another approach that has been proposed recently is the *direct execution* of Esterel-like instructions with a customized processor, which promises the flexibility of a software solution with an efficiency close to a hardware implementation. However, the instruction sets and implementations of the processor architectures proposed so far still have some limitations regarding their completeness, efficiency, and adherence to the original Esterel semantics. This paper presents a novel processor architecture, the Kiel Esterel Processor, which addresses these shortcomings. In particular, it provides a complete, semantically accurate implementation of the Esterel preemption primitives, most of which can be expressed directly with a single machine instruction.

Contents

List of Figures

List of Tables

1   Introduction

The synchronous language Esterel has been developed for modeling reactive systems [4]. As a system-level language, it gives an abstract, well-defined and executable description of the application, and can be synthesized into other high-level languages for further compilation.

A fundamental concept of Esterel is the *signal*; signals are used to communicate internally and with the environment. The execution of an Esterel program is divided into logical *instants*, or *ticks*, which also determine the sampling of input signals and the generation of output signals. The synchrony of Esterel implies that the outputs generated from given inputs occur at the same logical instant; that is, the generation of outputs is (conceptually) simultaneous with the inputs, and computations are (conceptually) instantaneous. Signals are *present* or *absent* throughout an instant, indicating the occurrence of certain events, and they may also carry a value.

As a system level language for programming control-dominated reactive systems, Esterel's control flow primitives are much richer than that of traditional, sequential programming languages. In particular, Esterel allows to express various types of preemption, including abortion and suspension [3]. 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 the control for a last time at the abortion time. A *suspension*, performed by `suspend`, freezes the state of a body for the instant when the trigger event occurs.

There are several methods to implement an Esterel program:

- A *hardware* implementation [2] has small memory requirements and low unit production costs. However, it is not flexible, and its resource usage increases rapidly when data path handling is needed.

- A *software* implementation [4, 8] on the other hand is a very flexible solution, and has low costs for the data path and arithmetic operations. However, common (COTS) processor architectures cannot handle reactive control constructs directly; therefore, handling these control constructs correctly turns out to be fairly expensive on classical software implementations. Moreover, the instruction and data memory requirements can be prohibitive for small, low-cost micro controllers.

- The *co-design* approach [1] partitions a model into hardware and software parts, trying to achieve a good balance of flexibility, performance and cost; for the software part, again a traditional μC is used.

Another approach to combine the advantages of custom hardware and traditional software is to implement an Esterel program on a *reactive processor* whose instruction set has been tailored to Esterel. We distinguish two variants of this approach.

- The *patched reactive processor* implementation combines a COTS processor core with an external hardware block, which implements additional Esterel-style instructions.

- A *custom reactive processor* implementation consists of a full-custom reactive core, whose instruction set and data path have been tailored exclusively for the processing of Esterel code.

Table 1 provides a high-level comparison of these implementation alternatives.

So far, there have been only limited and fairly recent investigations of the reactive processor approach. To our knowledge, the ReFLIX and RePIC architectures proposed by Dayaratne, Roop, Salcic *et al.* [11, 12, 7] are the only ones that fall into this category, and they both follow the patched processor strategy. Their results are fairly promising, illustrating the potential of
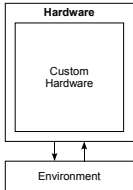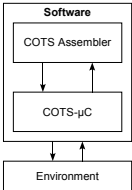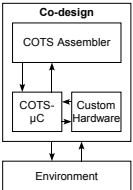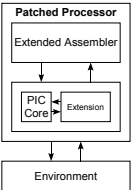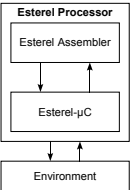
Architecture	Hardware	Software	Co-design	Patched Processor	Custom Processor
Speed	$++$	$-$	$+$	$+$	$+$
Flexibility	$--$	$++$	$-$	$+/-$	$+$
Esterel Compliance	$++$	$++$	$+/-$	$-$	$+/-$
Cost	$++$	$--$	$-$	$-$	$+$
Appl. Design Cycle	$--$	$++$	$+/-$	$++$	$++$

Table 1: Comparison of implementation alternatives
$++$ represents best; $--$ means worst, *e. g.*, Cost $++$ means very low production costs.

this approach. However, there are also certain limitations of the architectures proposed so far, for example regarding their support of the Esterel preemption primitives, as discussed further in the following section.

In this paper, we present an alternative architecture, the *Kiel Esterel Processor* (*KEP*), which is a custom reactive processor, to our knowledge the first of this kind. The architecture presented in this paper is version 2.0 of the Kiel Esterel Processor, hence we also refer to it as KEP2. Notable features of the KEP2 include the following:

1. It gives a complete, semantically accurate implementation of the Esterel preemption primitives, including weak and strong abortion and suspension.

2. As the instruction set and data path have been developed specifically for Esterel execution, the individual machine instructions can be executed fairly fast. Furthermore, most typical Esterel commands can be expressed directly with just a single KEP command, improving speed further and leading to minimal instruction and data memory usage.

3. The KEP also includes an interface block for handling input and output signals, which directly supports testing presence and values of signals across logical instants (corresponding to Esterel's `pre` operator).

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

The rest of this paper is organized as follows. The next section discusses related work. Section 3 presents the architecture of the KEP2, followed by a description of the KEP assembler. The experimental results are summarized in Section 4. The KEP2 is a single, sequential processor, and hence does not support Esterel's concurrency operator directly, which is probably its most significant limitation so far; this is addressed further in Section 5. Finally, Section 6 gives some concluding remarks and outlines future work. Appendix A elaborates on the instruction set in detail, Appendix B describes the synthesis of a specific KEP configuration, and Appendix C describes the translation of KEP2 assembler into executable codes.

```
% Esterel
...
weak abort
  ...
  emit Z;
  await B;
  emit X;
  await C;
  emit W;
when A;
emit Y;
...
```

```
% RePIC
   ...
4  ldaaddr 19
5  abort 0 A
   ...
8  emit Z
9  chkabort 0
10 await
11 present B
12 goto 10
13 emit X
14 chkabort 0
15 await
16 present C
17 goto 15
18 emit W
19 emit Y
   ...
```

```
% KEP ASM
   ...
   WABORT 1,A,A0
   ...
   EMIT Z
   AWAIT B
   EMIT X
   AWAIT C
   EMIT W
A0: EMIT Y
   ...
```

(a)   (b)   (c)
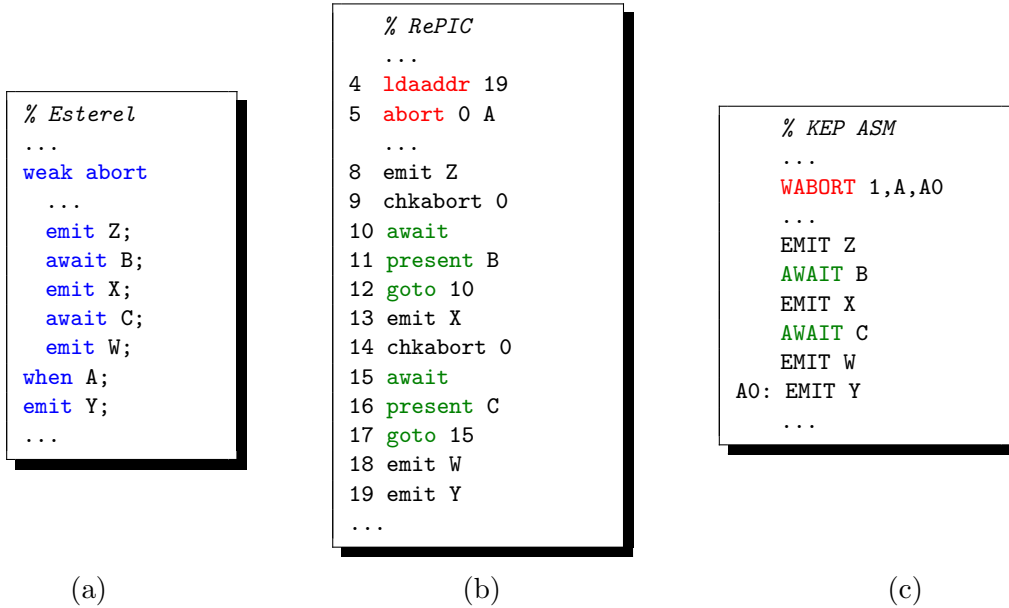
Figure 1: An Esterel code fragment involving a `weak abort` statement (a), and the corresponding assembler code for RePIC (b) and KEP (c).

2 Related Work

As mentioned in the introduction, the only other reactive processor proposals in the sense of Esterel that we are aware of are the ReFLIX and RePIC designs [11, 12, 7], of which RePIC is the more advanced. The RePIC includes an abort handling block, which is used for handling both strong and weak aborts; it does not handle suspension. These abort types are distinguished by the judicious placement of additional instructions in the assembler code. Figure 1(a)/(b) shows an example from Dayaratne *et al.* [7] for translating the Esterel `weak abort` statement to the RePIC assembler. The abort handler is configured with a pair of instructions; `ldaaddr` (line 4) specifies the continuation address, and `abort` (line 5) indicates the trigger signal. For weak abortion, the RePIC inserts a `chkabort` instruction before every `await` instruction within the abort body to determine whether control stays within the abort body in that logical instant. For example, after emitting signal Z (line 8), the `chkabort` (line 9) determines whether the "`await B`", encoded in lines 10–12, should be executed next, or whether a jump to the continuation address should be performed, thus aborting the body. As presented there, this would respond correctly to the presence of the abort signal A; however, if A is absent, execution would reach the `await`-loop in lines 10–12, and would in the following ticks only be sensitive to the awaited signal B, not to the abort trigger signal A. It seems that this could be remedied by including the `chkabort` instruction within the `await`-loop (*i.e.*, changing the "`goto 10`" in line 12 to "`goto 9`"). However, what we see as the more significant limitation is that this abort handling mechanism seems not as efficient as it could be, especially when considering nests of aborts. For comparison, the KEP handles aborts directly in hardware, without the need for additional assembler instructions to check the presence of abort signals at each control point, thus resulting in more compact and efficient code. To illustrate, consider the KEP assembler shown in Figure 1(c) for the same Esterel example; a single `WABORT` assembler instruction configures the abort handler, which is then active concurrently with the execution of the abort body, and which autonomously performs the necessary preemption of the abort body (correctly distinguishing between weak and strong aborts) when the abort signal occurs.

```
1  % Esterel
2  module SPEED:
3  input Centimeter,Second;
4  output Speed:integer;
5  loop
6    var Distance:=0:integer in
7      abort
8        every Centimeter do
9          Distance:=Distance+1
10       end every
11     when Second do
12       emit Speed(Distance)
13     end abort
14   end var
15 end.
```

```
1      % KEP ASM
2      % module SPEED
3      INPUT Centimeter,Second
4      OUTPUTV Speed
5      VAR Distance
6  A0: LOAD Distance,#0
7      ABORT 1,Second,A1
8      AWAIT Centimeter
9  A2: ABORT 1,Centimeter,A3
10     ADD Distance,#1
11     HALT
12 A3: GOTO A2
13 A1: EMITR Speed,Distance
14     GOTO A0
```

Figure 2: SPEED: an example module including valued signals and variables.

Based on RePIC, Dayaratne *et al.* [7] propose an extension to a multi-processor architecture, called EMPEROR, which allows the handling of Esterel's concurrency operator. This is an interesting approach, which could also be applied to the KEP2 to extend the range of acceptable Esterel programs; see also Section 5.

Finally, the KEP2 itself has evolved from earlier designs, the first of which being KEP version 0.1 [9]. This version already implemented the suspension primitives correctly, but did not include the interface block with the support of the `pre`-operator, did not support variables, and did not allow local signals.

3 The Kiel Esterel Processor Architecture

The KEP2 employs a 32-bit wide instruction word with a separate 16-bit wide inner data bus. This gives a range of up to 65535 for the signal counts, which indicate how often a signal must occur before for example an `abort` is triggered. The KEP assembler language contains thirty instructions. An overview is given in Tables 2 and 3. The most common Esterel statements, including a majority of the reactive kernel statements, can be represented directly. Other Esterel statements can be implemented by standard Esterel syntax translation. To illustrate the compactness of the KEP assembler, consider the Esterel module SPEED [3], shown in Figure 2. This module contains local variables as well as valued signals. To generate the assembler, the translation rules shown in Figure 3 were employed.

The architecture of the KEP2, shown in Figure 5, is inspired by the three layers that constitute a reactive program [4], *i.e.*, the *interface* layer, the *reactive kernel*, and the *data handling* layer. In the KEP, a Reactive Core decides what computations and what outputs must be generated when it reacts to inputs. An interface block handles input reception and output production. The classical computations are preformed by the data handling block. All of these blocks are scalable and optimized for the Esterel language.

3.1 The KEP2 Input/Output Interface

The top-level I/O signals of the KEP2 are illustrated in Figure 4(a). The environment can reset the processor via the Reset pin. An external clock must be connected to the OscClk pin; we use T_{osc} to denote the rate of that clock. ROMData and ROMAddr are data and address buses

Mnemonic, Operands	Description	Corresponding Esterel Statement	See Section	See page
ABORT $n,S,$ *endAddr*,*startAddr*	Configures the Watcher (the basic cell of the Preemption Element) in the Reactive Block	abort ...when n S	A.1.1	23
WABORT $n,S,$ *endAddr*,*startAddr*	Configures the Watcher in the Reactive Block	weak abort ...when n S	A.1.2	23
SUSPEND 1,$S,$ *endAddr*,*startAddr*	Configures the Watcher in the Reactive Block	suspend ...when S	A.1.3	24
AWAIT S	Configures the AWAIT Element in the Reactive Block	await S	A.2.1	25
AWAIT n,S	Similar to the above, n is the counter value	await n S	A.2.1	25
PAUSE (AWAIT TICK)	Similar to the above, delays for one instant	pause (await tick)	A.2.2	25
AWAIT $n,$TICK	Similar to the above, delays for n instants	await n tick	A.2.1	25
CAWAIT $S,$*SstartAddr*	Configures the CAWAIT Element	await case	A.3.1	26
CAWAITE $S,$*SstartAddr*	Configures the CAWAIT Element for the last case in the list	await case	A.3.2	26
EMIT S	Emits the signal S and keeps it during the current tick	emit S	A.4.1	27
EMIT $S,$#*data*	Emits the valued signal S with *data* and keeps it during the current tick	emit S(data)	A.4.1	27
EMITR $S,$*reg*	Emits the valued signal S with contents of the register *reg*	emit S(var_reg)	A.4.2	27
SUSTAIN S	Sustains the signal S	sustain S	A.5.1	28
SUSTAIN $S,$#*data*	Sustains the valued signal S with *data*	sustain S(data)	A.5.1	28
SUSTAINR $S,$*reg*	Sustains the valued signal S with contents of the register *reg*	sustain S(var_reg)	A.5.2	28
HALT	Halts the system	halt	A.6.1	29
NOTHING	Does nothing	nothing	A.7.1	29
PRESENT $S,$*elseAddr*	Tests signal S, goes to the address *elseAddr* if S is not presented	present S then ...end	A.8.1	29
SIGNAL S	Initializes a local signal S	signal S in ...end	A.9.1	30
SIGNAL PRE(S)	Executes right after the SIGNAL S instruction		A.9.1	30
CALL *addr*	Calls a subroutine which locates on *addr*	call *subroutine*	A.12.2	50
RET	Returns from a subroutine		A.12.3	50
GOTO *addr*	Goes to the address *addr*		A.12.1	49

Table 2: Overview of the instruction set architecture (part 1/2).

Mnemonic, Operands	Description	Corresponding Esterel Statement	See Section	See page
`JW Z,elseAddr`	Goes to the address *elseAddr* if the previous operating result is not *zero*. Generally it is used right after a `SUB` instruction to implement the `repeat` statement	`repeat n times` `...end`	A.11.1	47
`JW L,elseAddr`	Goes to the address *elseAddr* if the previous comparison operating result is not *less than*. Generally it is used right after a `CMP` instruction to implement the `if` statement	`if (A<B) then` `...else ...end`	A.11.1	47
`JW G,elseAddr`	Goes to the address *elseAddr* if the previous comparison operating result is not *greater than*	`if (A>B) then` `...else ...end`	A.11.1	47
`JW GE,elseAddr`	Goes to the address *elseAddr* if the previous comparison operating result is not *greater than* or *equal*	`if (A>=B) then` `...else ...end`	A.11.1	47
`JW LE,elseAddr`	Goes to the address *elseAddr* if the previous comparison operating result is not *less than* or *equal*	`if (A<=B) then` `...else ...end`	A.11.1	47
`JW EE,elseAddr`	Goes to the address *elseAddr* if the previous comparison operating result is not *equal*	`if (A=B) then` `...else ...end`	A.11.1	47
`JW NE,elseAddr`	Goes to the address *elseAddr* if the previous comparison operating result is *equal*	`if (A<>B) then` `...else ...end`	A.11.1	47
`CLRC`	Clears carry		A.10.1	30
`SETC`	Sets carry		A.10.2	31
`SR reg`	Right shifts the register *reg*		A.10.3	31
`SRC reg`	Right shifts the register *reg* with a carry		A.10.4	32
`NOTR reg`	Bitwise logical NOTs the register *reg*		A.10.5	32
`LOAD reg,val`	Loads the *val* to the register *reg*. The *val* can be one of the following. (1) `#data`: an immediate data (2) *reg*: the contents of a source register (3) `?S`: the value of signal *S* (4) `PRE(?S)`: the previous value of signal *S*	`reg:=val`	A.10.6	32
`ADD reg,val`	Adds the *val* to the register *reg*. See also the `LOAD` instruction.	`reg:=reg+val`	A.10.7	34
`ADDC reg,val`	Adds the *val* to the register *reg* with the carry. See also the `LOAD` instruction.		A.10.8	35
`SUB reg,val`	Subtracts the *val* from the register *reg*. See also the `LOAD` instruction.	`reg:=reg-val`	A.10.9	36
`SUBC reg,val`	Subtracts the *val* from the register *reg* with the carry. See also the `LOAD` instruction.		A.10.10	38
`MUL reg,val`	Multiplies the register *reg* by the *val*. See also the `LOAD` instruction.	`reg:=reg*val`	A.10.11	39
`ANDR reg,val`	Bitwise logical ANDs the register *reg* by the *val*. See also the `LOAD` instruction.		A.10.12	41
`ORR reg,val`	Bitwise logical ORs the register *reg* by the *val*. See also the `LOAD` instruction.		A.10.13	42
`XORR reg,val`	Bitwise logical XORs the register *reg* by the *val*. See also the `LOAD` instruction.		A.10.14	44
`CMP reg,val`	Compares the contents of the register *reg* and the *val*, affects equal and carry bits. See also the `LOAD` instruction.		A.10.15	45

Table 3: Overview of the instruction set architecture (part 2/2).
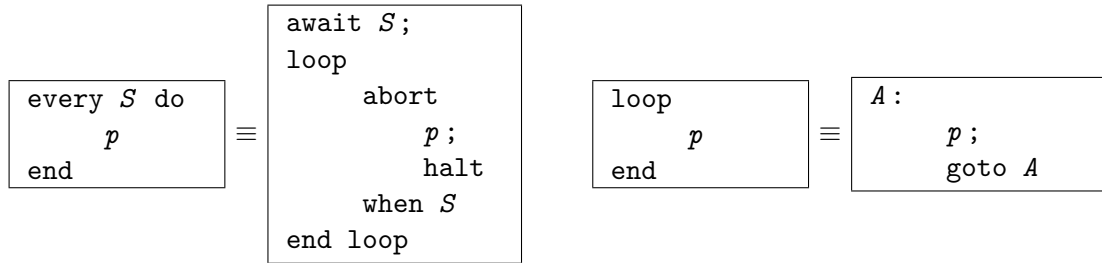
Figure 3: Translation rules of `every` and `loop` statements.

<div style="text-align:center">(a)</div>

<div style="text-align:center">(b)</div>
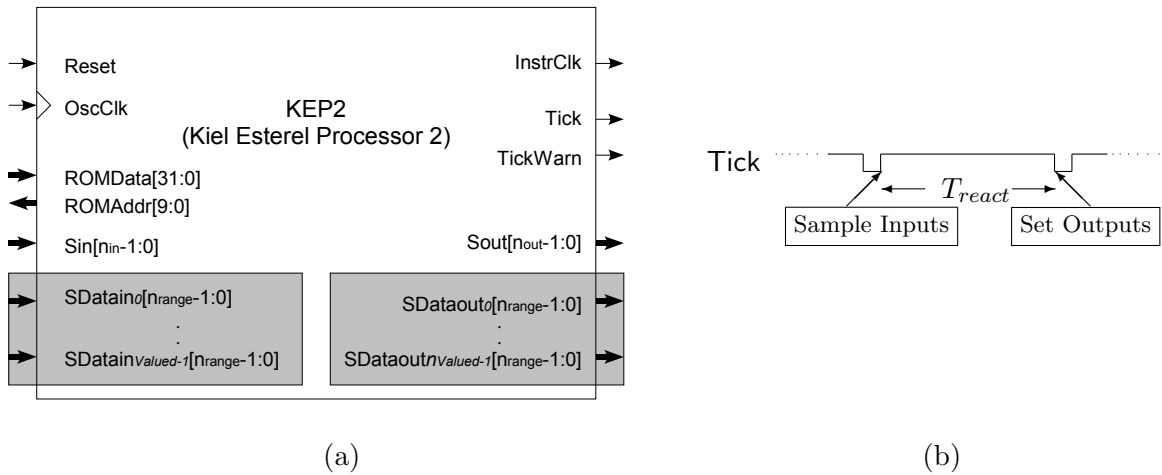
Figure 4: The interface connections (a), and the waveform of the Tick signal (b).

for the instruction memory. There are n_{in} pins Sin to signal the presence of input signals, and *Valued* data buses of width n_{range} to provide values for input signals. There are similar pins and buses for output signals. The InstrClk indicates the instruction clock; each instruction cycle lasts three OscClk cycles. A complete description of the I/O signals can be found in Table 4

Signal	Direction	Description
Reset	Input	To reset the KEP2 and to generate a Reset Event, assert this input High for at least one OscClk cycle. A Reset Event is automatically generated, and all outputs will be set to '0'.
OscClk	Input	The frequency may range from DC to the maximum operating frequency of the circuit, for example as reported by the FPGA development software. In KEP2, one instruction cycle equals three OscCLK cycles.
ROMAddr[9:0]	Output	The ROMAddr address bus is an output bus from the KEP2 and it indicates the instruction memory address. ROMAddr[9:0] will be set to "0000000000" if Reset is set.
ROMData[31:0]	Input	The ROMData data bus is an input bus to the KEP2 and it carries encoding data from instruction memory. When the ROMAddr changes, the new ROMData will be valid on the following clock rising edge.
InstrClk	Output	InstrClk is an output from KEP2 and it indicates the instruction cycle. It is set to '0' when Reset is set.
Tick	Output	Tick is an output from KEP2 and it indicates the logical tick of Esterel. It is set to '0' when Reset is set.
TickWarn	Output	TickWarn is an output from KEP2 and it is kept high once the physical time of an instant extends the preconcerted value.
Sin[$n_{in} - 1$:0]	Input	Presents and keeps valid input signals on this port during the Tick set (high) period. The n_{in} depends on the amount of input signals. *e. g.*, n_{in} equals 8 (Sin[7:0]) when there are 8 input signals. KEP2 supports up to 63 input signals.
SDatain$i$[$n_{range} - 1$:0]	Input	Carried data of valued signal S appears on the SDatain$i$[n_{range}:0] port. i ranges from 0 to $n_{valued} - 1$, where n_{valued} is the number of valued signals. n_{range} depends on the range of carried data. For example, assume there are two valued signals, and the range of carried data is from 0 to 200 (decimal). Two ports would be generated. One is SDatain0[7:0] which corresponds to Sout[0], and the other is SDataout1[7:0] which corresponds to Sout[1]. n_{range} is up to 16 (16-bit), which corresponds to values from 0 to 2^{16}-1 = 65535.
Sout[$n_{out} - 1$:0]	Output	Output signals appear on this port until the current Tick is finished (falling edge). The n_{out} depends on the amount of output signals, see also Sin[$n_{in} - 1$:0]. KEP2 supports up to 63 output signals.
SDataout$i$[$n_{range} - 1$:0]	Output	Carried data of valued signal S appears on the SDataout$i$[n_{range}:0] port. See also SDatain$i$[$n_{range} - 1$:0].

Table 4: KEP2 Interface Signals Descriptions.

The Tick pin indicates the logical tick of Esterel. Figure 4(b) shows a waveform of the Tick signal. The *activated period* (Tick high) is T_{react}, which indicates the reaction time of the module. A gap of length T_{osc} identifies when the inputs are sampled and when the outputs are generated by the reactive processor. The processor samples the inputs on the Tick rising edge, and holds the outputs generated during a logical tick until the Tick falling edge. The *deactivated*

KEP2 (Kiel Esterel Processor 2)
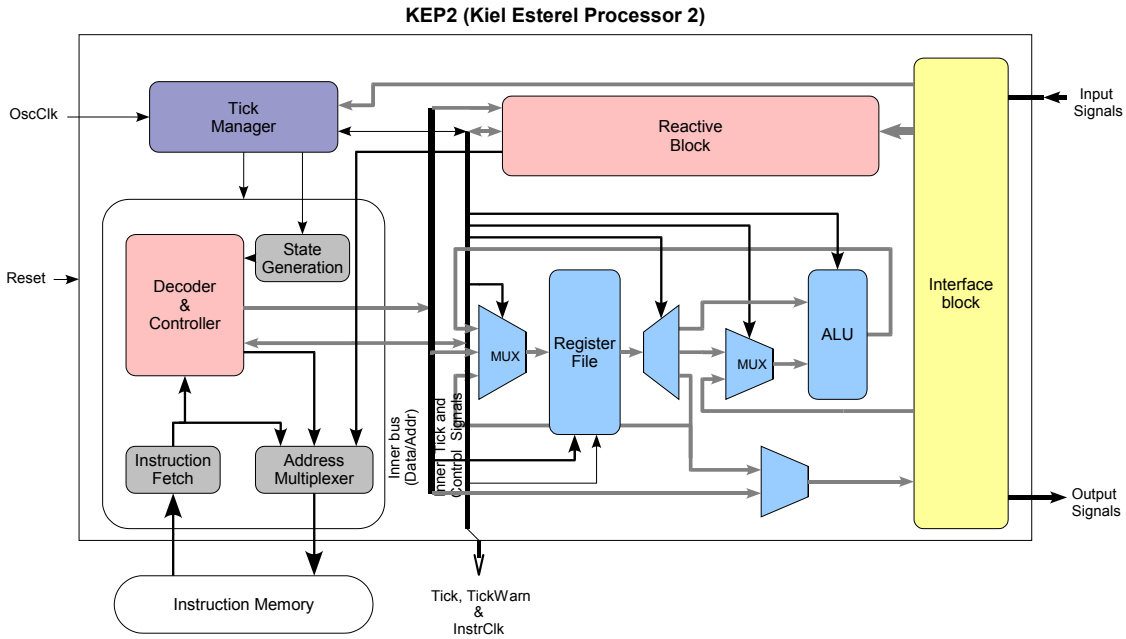
Figure 5: The architecture overview.

period (Tick low) leaves a period for the environment to modify the inputs.

3.2 The KEP Tick Manage

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" instruction, generally at the beginning of a module, this initializes the Tick Manager. When more than ⎵TICKLEN instructions have been executed since the last tick delimiting instruction (pause or await), the Tick Manager 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.

The programmer can try to find an appropriate value for ⎵TICKLEN and add a corresponding emit statement to the assembler program. However, we have also developed an analysis procedure that automatically performs this *Worst Case Reaction Time* (WCRT) analysis [10]. This analysis has been integrated into a compiler that translates Esterel to KEP assembler and automatically sets ⎵TICKLEN.

3.3 The Reactive Core

The implementation of Esterel's reactive statements relies on the cooperation of the KEP2's Decoder & Controller and the Reactive Block, which together form the Reactive Core. The Reactive Block contains a Preemption Element, which contains a configurable number of Watcher modules, that are responsible for implementing the preemption operations. The Reactive Block also contains the AWAIT and CAWAIT Elements, implementing single and concurrent signal awaiting, and the PRESENT Element, testing for signal presence.

9
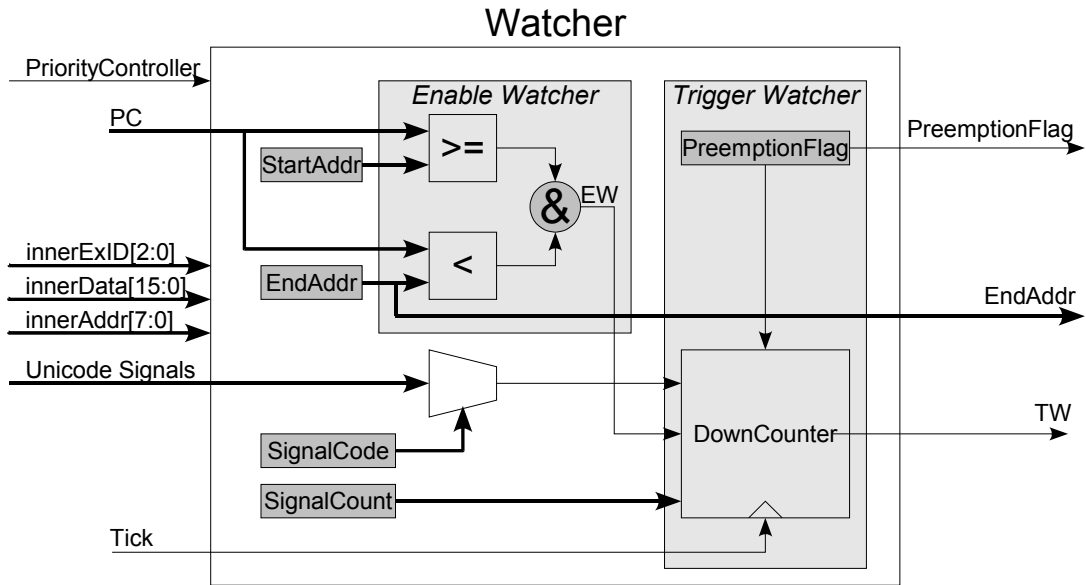
Figure 6: The structure of the Watcher.

3.3.1 The Preemption Element

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. We call this scheme *Inside/Outside Preemption Range Watching* (*IOPRW*).

A Watcher, shown in Figure 6, contains two functions to implement the IOPRW, the Enable Watcher and the Trigger Watcher. The *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.

The function of the *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, 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.

Figure 7 shows the architecture of a Reactive Block including three Watcher modules. To illustrate its operation, consider the Esterel module NESTED in Figure 8(a), which is an example of a nested preemption. After starting, the module watches C and A as the abortion trigger signals, and B as the suspension trigger signal. The execution stays on line 8 to wait for signal

10
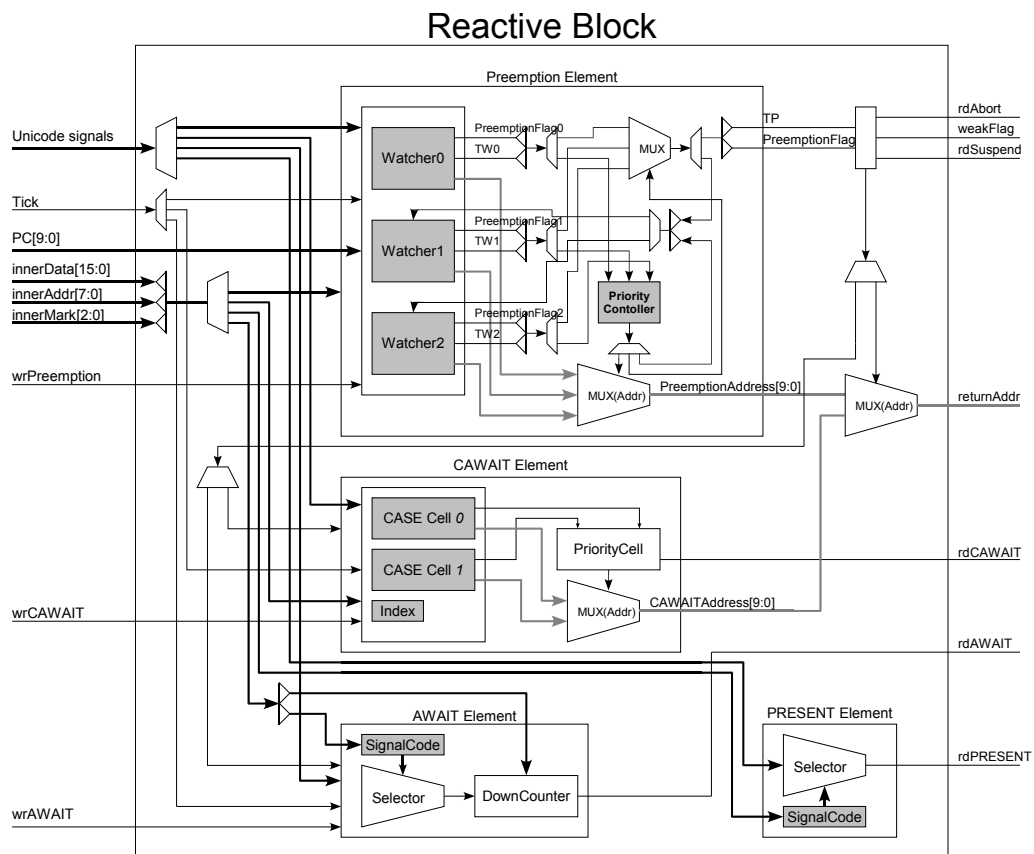
Reactive Block

Figure 7: Architecture of a Reactive Block with three Watchers.
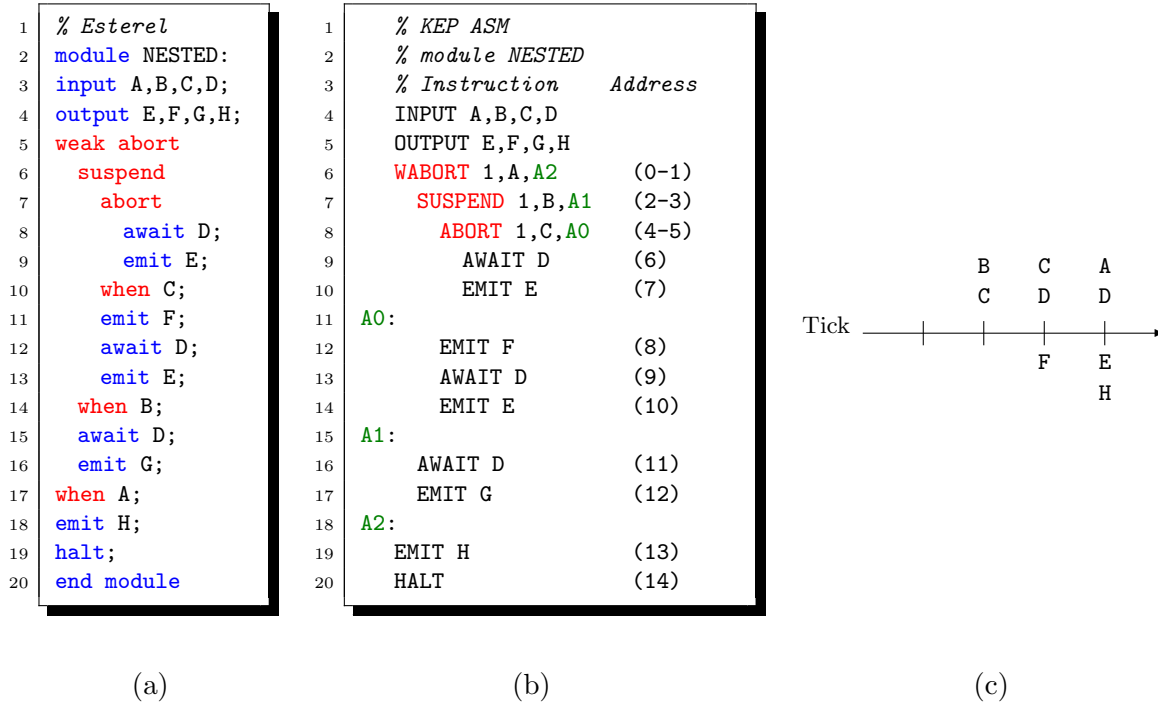
Figure 8: NESTED: the Esterel module illustrating the preemption statements (a), the KEP assembler program (b), and an execution trace (c).

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. Figure 8 (b) shows the corresponding KEP assembler program of the NESTED module. Figure 8 (c) shows a possible execution trace.

When executing NESTED, first the watchers Watcher0, Watcher1 and Watcher2 are configured via three preemption instructions (lines 6–8). The PC stays at address (6) (line 9) until any of the signals A, B, C, or D occur. Since this address is within each of the watchers' watching range, all of the watchers are enabled now.

If B and C occur simultaneously, TW1 and TW2 are set at the same time. The Preemption-Flag1 indicates a suspension, and PreemptionFlag2 indicates strong abortion. The PriorityController 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 PriorityController maps Watcher1'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. Watcher2 receives the result of the PriorityController, and since the current active preemption is a suspension, Watcher2 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 indicate the current active preemption for the Decoder & Controller. The Decoder & Controller checks the rdAbort, weakFlag, rdSuspend, 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 equals 8 (the next instruction address behind the body of abortion C), and the weakFlag is '0' to indicate a strong abortion type. Since the sensitive signal of the AWAIT statement is present, the AWAIT Element sets rdAWAIT to '1' to denote that AWAIT 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 to address 8 and executes "EMIT F" and "AWAIT D".

When signals A and D occur simultaneously in the following instant, weak abortion A takes priority. The PriorityController maps Watcher0's outputs to that of the Preemption Element, so the Reactive Block's rdAbort is '1' to denote an active abortion, the returnAddress equals 13 (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 instruction first. "EMIT E" (10) is executed and then the "AWAIT D" (11) is fetched. Since it is a non-instantaneous statement, the Reactive Core will ignore it, and instead respond to the weak abortion. The returnAddress is mapped to the PC, and then control jumps to (13). The KEP executes "EMIT H" (13) and HALT (14).

3.3.2 The AWAIT Element

The above case illustrates how the KEP deals with different preemption types. The KEP correctly and very efficiently implements abort/weak abort/suspend statements and allows arbitrary nesting. The Reactive Block includes further elements to directly implement the majority of reactive statements of Esterel.

The AWAIT and PAUSE instructions are handled by the AWAIT Element, which is shown in Figure 9. The AWAIT Element contains two parameter registers. One registers the watched signal coder value and the other registers the counter value. The element watches sensitive signal on Tick rising edge. If the watched signal is '1', the counter value will be deducted. When the counter value equals zero, the AWAIT Element sets rdAWAIT to '1' to denote the termination of AWAIT.

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

3.3.3 The CAWAIT Element

The CAWAIT Element watches several signals in parallel. The architecture of the CAWAIT Element is similar to that of the Preemption Element Element. Every CASE Cell includes two parameters, i.e. SignalCoder and CASEAddr. The CASEAddr registers the start address of the case body. The SignalCoder indicates which signal ought to be watched. Figure 11 illustrates a CAWAIT Element which includes three CASE Cell.

When several watched signals occur simultaneously, the corresponding CASE Cells are all active. The PriorityCell chooses the first active one in the list to take priority. In other words, the earlier case takes priority. The Decoder & Controller will respond to this event and jump to the registered address.

3.3.4 The PRESENT Element

The implementation of PRESENT statement depends on the PRESENT Element, which is shown in 11. The basic form of the Esterel PRESENT statement checks for one signal expression and
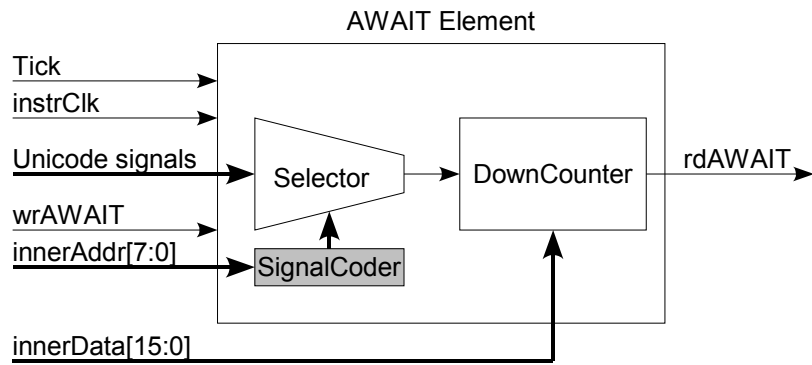
Figure 9: Architecture of the AWAIT Element.
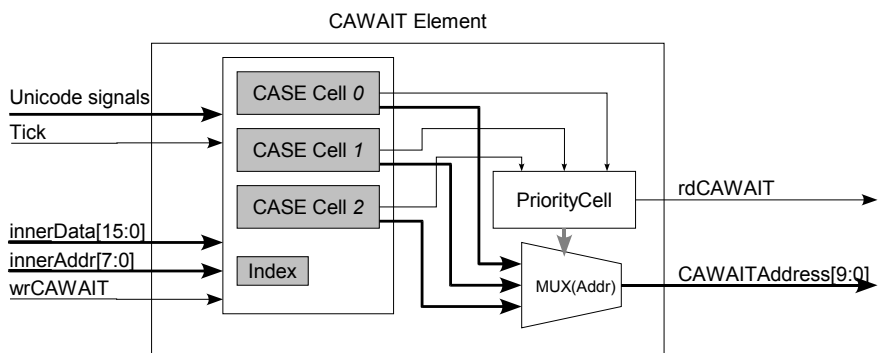
Figure 10: Architecture of a CAWAIT Element, with 3 CASE Cells.
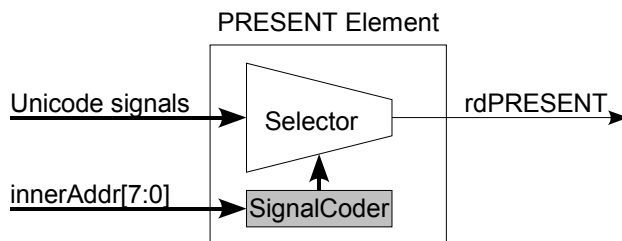
Figure 11: Architecture of the PRESENT Element.

performs binary branching. We map 26_{th} to 19_{th} bit of instruction encoding to the selected-signal-coder port of the PRESENT Element. The value of the selected signal is immediately put to an output port which is named as rdPRESENT.

Considering the stages of instruction execution, the selected signal's coder is emitted when KEP2 fetches a PRESENT instruction. When the Core executes the instruction, it just needs to test the rdPRESENT signal to decide whether to branch or not.

3.4 The Interface Block

The Interface Block is created as an interface layer of the reactive system. Figure 12 shows the architecture of an Interface Block, which here is configured with four pure and two valued inputs and outputs. The width of the valued signals' carried data is configured to be eight bits.

The Interface Block supports the pre operation (introduced in Esterel V5.91), which allows to access the previous presence 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 expresses the value of signal S in the previous instant.

Signal	$d_7 - d_2$	d_1	d_0
_TICKLEN	000000	0	0
A	000001	1	0
B	000010	1	0
C	000011	1	0
G	000001	0	0
H	000010	0	0
I	000011	0	0
PRE(A)	000001	1	1
PRE(I)	000011	0	1
X	000001		

(a)

Bit field	Value	Meaning
$d_{31} - d_{27}$	0011|1	Opcode
$d_{26} - d_{19}$	000|0010|0	The signal's unicode value
$d_{18} - d_{16}$	000	The extended code
$d_{15} - d_{00}$	0000|0000|0001|1001	The carried data

(b)

Table 5: The signal codes of the INOUT module (a), and the break down of the instruction encoding of the EMIT G,#25 instruction (hexadecimal 0x38200019) with indications for the half bytes (b).

To illustrate how the Interface Block deals with the input and output signals, consider the INOUT module shown in Figure 13(a), and the corresponding compiled KEP assembler program
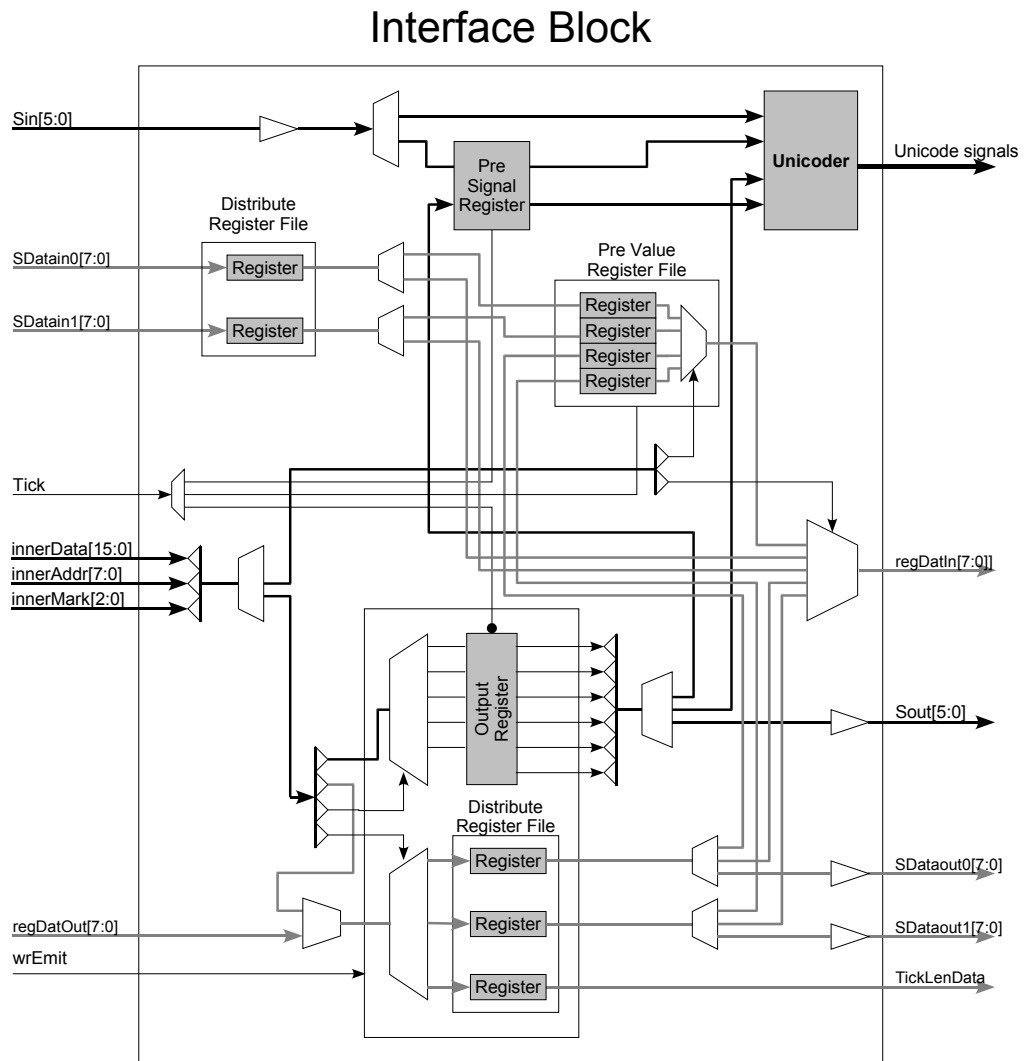
Interface Block

Figure 12: The architecture of an Interface Block.

```
1  % Esterel
2  module INOUT:
3  input A:integer;
4  input B:integer,C;
5  output G:integer;
6  output H:integer,I;
7  var X in
8  await A;
9  present C then
10    emit I;
11 end present;
12 await B;
13 present pre(I) then
14   emit G(25);
15   X:=pre(?A);
16   emit H(X);
17 end present;
18 halt;
19 end.
```

```
1                    INPUTV A,B
2                    INPUT C
3                    OUTPUTV G,H
4                    OUTPUT I
5                    VAR X
6  [0000] {38000005}    EMIT _TICKLEN #5
7  [0001] {30300001}    AWAIT A
8  [0002] {10700004}    PRESENT C,A0
9  [0003] {38600000}    EMIT I
10 [0004] {30500001} A0: AWAIT B
11 [0005] {10680009}    PRESENT PRE(I),A1
12 [0006] {38200019}    EMIT G,#25
13 [0007] {A03E0400}    LOAD X,PRE(?A)
14 [0008] {90400400}    EMITR H,X
15 [0009] {28000000} A1: HALT
```

(a)   (b)

Figure 13: `INOUT`: an Esterel module illustrating the processes of I/O statements (a), and the compiled KEP assembler program.

shown in Figure 13(b). Table 5(a) shows the signal codes of the `INOUT` module. The `_TICKLEN` is a reserved name of a valued signal for configuring the Tick Manager, see Section 3. The signal code is 8 bits wide. Bit 0 is used as the `pre` extended bit of the original signal. Bit 1 is used to describe the signal type, *i. e.*, '1' indicates an input type signal. Bits 7–2 further identify the signal. 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 and for variables (which are similar to valued signals, but do not carry a presence status). Table 5(b) shows the encoding of an `EMIT` instruction. Bits 26–19 of an instruction are the Unicode Signals code. In the KEP, all the input and output signals are recoded into a Unicode Signals bus, which can be accessed by the inner blocks and elements directly.

The first instruction in `INOUT` is "`EMIT _TICKLEN,#5`". The emitted value is transmitted to the TickLenData port and received by the Tick Manager for initialization. Then "`AWAIT A`" configures the AWAIT Element in the Reactive Block. Control stays there and waits for the trigger signal.

Assume the valued input signal `A` and pure signal `C` are both present after the initial instant, and `A` carries the value 20. The Reactive Block sets rdAWAIT to '1' to denote that `AWAIT` is terminated. The Unicode Signals are multiplexed to the Selector of the PRESENT Element in the Reactive Block. When the "`PRESENT C,A0`" instruction is fetched, the MUX maps signal `00001110b` to the rdPRESENT signal. Since `C` is present in this instant, the rdPRESENT is high to skip the branch. Then the "`EMIT I`" is executed, and the output register locks the presence of the emitted signal until the current tick ends. The execution of "`AWAIT B`" causes the KEP to wait for the next tick.

When the following instant starts, the old status of input and output signals will be written into the Pre Signal Register, and then regarded as the previous signal status ("`pre(S)`" in Esterel). The carried data of valued signals ("`?S`") will be written into the Pre Value register file, and can be accessed as the previous value ("`pre(?S)`"). Assuming that signal `B` is present, "`AWAIT B`" terminates. Since the signal `I` was emitted in the previous instant, `PRE(I)` is present. Therefore, the "`PRESENT PRE(I),A1`" instruction will not branch, and "`EMIT G,#25`" will be

```
1  % Esterel                          1      % KEP ASM
2  module REINC:                      2      % module REINC
3  input S;                           3      INPUT S
4  output O1,O2;                      4      OUTPUT O1,O2
5  loop                               5  A0:
6    signal S in                      6      SIGNAL S
7      present S then                 7        PRESENT S,A1
8        emit O1;                     8          EMIT O1
9      else                           9          GOTO A2
10       emit O2;                    10  A1:
11     end;                          11          EMIT O2;
12     pause;                        12  A2:
13     emit S                        13      PAUSE
14   end signal                      14      EMIT S
15 end loop                          15      GOTO A0
```

| (a) | (b) |

Figure 14: `REINC`: Translation of the Esterel `signal` declaration (a) into to the KEP `SIGNAL` instruction (b).

executed. The valued signal `G` consists of two interface hardware objects: a pure signal, which indicates signal presence or absence, and a signal bus, which conveys signal values. The inner buses of the KEP transmit the data to a MUX which selects the source of the data. The data on the inner data bus is selected and written into a distributed register file, and then issued as the value of `G`.

The instruction "`LOAD X,PRE(?A)`" causes the previously registered data of signal `A` to be taken from the Pre Value register file, and to be mapped to the regDatIn[7:0] port via MUXs. The Decoder & Controller will write the data into the register file of the processor. The implementation process of the "`EMITR H,X`" is similar to that of the "`EMIT G,#25`" instruction, except that the source of the data is the register file of the processor instead of the inner data bus.

3.5 Local Signals

Another feature of the Interface Block is the handling of local signals, which are declared in Esterel with a `signal` declaration. Unlike the (global) interface signals, a local signal has a limited scope. This may results in *reincarnation* [3]; *i. e.*, in case of an instantaneous repetition of loops, local signals can have several simultaneous instantiations.

The `SIGNAL` instruction implements a signal scope and correctly handles reincarnation. In the KEP, a local signal is implemented by a pair of connected I/O ports. When executing a "`SIGNAL S`" statement, the presence status and value of S is cleared, thus effectively introducing a fresh signal. To illustrate, consider the Esterel module `REINC`, shown in Figure 14 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.

4   Experimental Results

To quantitatively compare the data handling abilities between the Esterel processor and other implementations, we used the `CURVE` module (contained in the mca200 test bench [6]) as an example, since it is a typical module that includes varied data handling statements. Table 6 compares the resource usages of the KEP2 with different hard- and software implementations. For the hardware implementations, we synthesize the module to VHDL with the Esterel V7 compiler, since 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. As an optimization, we manually resized 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 ISE6.3, 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.

	KEP-C (16-bit)	Hardware (32-bit)	Hardware (16-bit)	MCS51[1] (8-bit)	Microblaze[2] (32-bit)
Logic Cells	1384	1510	968	-	1906
Code size (words)	185	-	-	1070[3]	436
Code size (bytes)	740	-	-	1636	1744
RAM Usage (words)	9	-	-	31	19
RAM Usage (bytes)	18	-	-	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 vary; here, a *word* represents a complete assembler line.

Table 6: The codes size and RAM usage (in word) comparison of `CURVE` implementation between KEP2, MCS51, and Microblaze.

To evaluate the performance of the KEP2, we use some standard test cases [3, 1, 6]. Those modules are typical Esterel applications, which not only contain the reactive statements, but also include arithmetic and logical data handling. However, we leave out the module which contains the `pre` operator, since the CEC compiler does not support it [5]. The module is first translated into the KEP assembler program and then compiled to the KEP executable codes. This is then compared with software synthesis results of the Esterel Compiler V5.92, the Esterel Compiler V7 and the CEC compiler 0.3. Table 6 reveals that the code for the 32-bit processor is smaller than that of a 8-bit processor (in words). Therefore, we use the Microblaze as reference point in the following.

Tables 7 and 8 illustrates the comparison of executable code size and RAM usage between the KEP implementation and the Microblaze software implementation. The optimized data path of the KEP results on average in a 89% reduction of codes size and 77% reduction of RAM usage in words (or 89% reduction in bytes) when compared with the best result of the Microblaze implementation.

The Esterel module's line count is very close to the KEP2's codes size (in words). This fact implies that the KEP handles the Esterel statements on a high level. Practically, the majority of Esterel statements can be translated into KEP assembler instructions word by word.

As mentioned in the introduction, the KEP has been designed to be highly configurable. Table 9 compares five different KEP2 variants which include different elements to target various applications. The KEP2-E offers similar functions to RePIC and can be compared with RePIC

Module name	Esterel lines	Microblaze (words[1])			KEP2 (words)
		V5	V7	CEC	
SPEED	11	276	1081	253	11
BELT_CONTROL	14	440	1169	340	18
TIMER	6	368	1160	295	9
CONTROLLER	26	560	1226	487	24
DEBOUNCE	31	392	1198	299	28
ALARM_COMPARE	16	315	1109	265	14
SPEEDOMETER	23	328	1145	293	20
DASHBOARD_TIMER	77	617	1388	541	65
FRC	26	375	1163	313	18
CURVE	190	1307	2017	436	185
BAT_DIAG	45	487	1274	378	63
VER_ACC_DIAG	38	433	1229	303	41
LONG_SPEED_STRAT	60	573	1306	319	56

(1) For the code size, one word equals four bytes.

Table 7: Comparison of the codes sizes, in words.

Module name	Microblaze (RAM words/bytes)			KEP2 (RAM words/bytes)
	V5	V7	CEC	
SPEED	13/52	12/48	9/36	1/2
BELT_CONTROL	16/64	18/72	10/40	0/0
TIMER	14/56	14/56	10/40	0/0
CONTROLLER	18/72	22/88	17/68	0/0
DEBOUNCE	18/72	17/68	12/48	4/8
ALARM_COMPARE	14/56	15/60	10/40	2/4
SPEEDOMETER	16/64	15/60	11/44	2/4
DASHBOARD_TIMER	25/100	24/96	17/68	8/16
FRC	19/76	20/80	12/48	2/4
CURVE	34/136	26/104	19/76	9/18
BAT_DIAG	20/80	19/76	14/56	6/12
VER_ACC_DIAG	18/72	17/68	13/52	4/8
LONG_SPEED_STRAT	20/80	18/72	13/52	4/8

Table 8: The RAM usage (in words/bytes) comparison of module implementations. One KEP word equals two bytes, and one Microblaze word equals four bytes.

	KEP2-A	KEP2-B	KEP2-C	KEP2-D	KEP2-E	RePIC
AWAIT CASE Number	2	2	2	2	2	2
Preemption Nest	2	2	2	4	4	4
Counter Value Range	1	255	1	255	1	1
Input/Output	11/11	16/16	11/11	16/16	12/12	12/12
Valued Input/Output	2/2	2/2	3/3	2/2	1/1	1/1
Datapath Width	8	8	16	16	8	8
Logic Cells	1092	1270	1384	1972	1488	2068
Max Osc Freq (MHz)	54.11	47.93	42.06	41.46	42.87	40.27
Instruction Freq (MHz)	18.04	15.93	14.02	13.82	14.29	10.1

Table 9: Performance comparison between the KEP2 series and RePIC.

directly.[1] RePIC uses four clock cycles to execute an instruction cycle, but the KEP2 uses only three clock cycles. When they run on the same clock frequency, the KEP2's instruction cycle period is just 75% of that of the RePIC's—and the KEP2 typically takes significantly less instructions to implement the same behavior.

5 Handling Concurrency

The KEP2 does already execute a number of operations concurrently that in traditional processors must be sequentialized. For example, it simultaneously watches all trigger signals that are active at any time, thus allowing very efficient and compact handling of preemption nests. However, as mentioned in the introduction, the KEP2 does not implement Esterel's concurrency operator ("||") yet, which so far significantly limits the range of programs that can be translated to KEP assembler directly.

There are a number of options to address this limitation:

Sequentialization It is always possible to translate an Esterel program into an equivalent program that has a flattened state space. As the KEP2 does already handle hierarchy, in this case the state space would not need to be flattened completely; however, true concurrency would need to be eliminated, with the corresponding potential state explosion.

Multiprocessing Following the EMPORER proposal of Dayaratne *et al.* [7], multiple KEP2 cores could be combined to execute multiple Esterel threads concurrently. This approach seems feasible; however, it is relatively hardware-intensive, in particular if one wants to scale up to high degrees of concurrency.

Interleaving In this approach, a single KEP core would be extended to handle concurrency by an *interleaved control flow*. Here, the Decoder & Controller would be combined with a newly created Thread Manager to alternate between concurrent blocks. This would be different from statically scheduled interleaving, which could for example be implemented with gotos, in that the threads would be assigned dynamic priorities, and the Decoder & Controller would run the individual threads accordingly.

In a future design of the KEP, we plan to explore the third of these options, the interleaving approach.

6 Conclusion and Outlook

This paper presents the KEP2, a semi-custom, configurable Esterel processor. It consists of a reactive core and an optimized data path for the direct execution of Esterel programs. The KEP supports full standard Esterel preemption statements, *i. e.*, abort, weak abort, and suspend, in a very precise, direct and efficient way. The maximal nesting depth of these constructs is given by the number of watchers that are provided by the KEP; however, this number is configurable for a particular KEP. Just as in the original Esterel, the KEP can nest and combine these constructs in an arbitrary fashion. The KEP supports valued signals and signal counters, local signal declarations, and the pre operator.

The performance of the KEP is predictable. All instructions can be executed in a single instruction cycle, except for the instructions that configure watchers (ABORT/WABORT/ SUSPEND)

---

[1]Regarding the logic cell count, one should note that the KEP2's implementation is based on a Xilinx's XC2S100-6TQ144 FPGA chip, and the RePIC is implemented on an ALTERA' EP20K200EFC484-2 FPGA chip. However, the basic units of those two chips have similar structures, functions, and speed. Therefore, we can assume that logic cell counts are comparable.

instructions, which require two instruction cycles. However, if there are enough watchers available such that they must not be reused among abort statements, it is also possible to configure the watchers just once, at the start of the program. In this case, all instructions executed after the initial instant after system resetting really require just one instruction cycle. The predictability of the KEP also lends itself to an automated Worst Case Reaction Time analysis [10].

As an initial prototype, the KEP2 can be further optimized. To extend the KEP to handly concurrency, we are currently investigating the interleaving architecture as described in the previous section. Other improvements concern the direct implementation of further Esterel constructs, such as the immediate signal triggering, that can already be handled by the current architecture, but require multiple instructions to do so.

References

[1] 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.

[2] Gérard Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

[3] Gérard Berry. *The Esterel v5 Language Primer*, 1999.

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

[5] CEC: The Columbia Esterel Compiler.

[6] Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[7] 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.

[8] 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.

[9] 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`.

[10] 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.

[11] P. S. Roop, Z. Salcic, M. Biglari-Abhari, and A. Bigdeli. A New Reactive Processor with Architecture Support for Control Dominated Embedded Systems. In *IEEE International Conference on VLSI Design*, pages 189–194. IEEE CS Press, January 2003.

[12] 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.

A The Instruction Set Architecture

A.1 Preemption (abort/weak abort/suspend)

A.1.1 ABORT

Assembly syntax:
   ABORT *n, S, endAddr* (*, startAddr*)

n Counter value (default: 1).

S The name of the signal.

endAddr The address behind the end of the abortion body; see Figure 15.

startAddr The start address of the abortion body. It is generated by the KEP2 compiler automatically, see Appendix C.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
First Instruction Encoding	00011	wwwwwwwa	aaa	aaaaaa	AAAAAAAAAA
Second Instruction Encoding	00011	SSSSSSSS	000	nnnnnn	nnnnnnnnnn

First Instruction Encoding:

[**31:27**] Opcode=ABORT/WABORT/SUSPEND

[**26:20**] The index number of the Watcher, the KEP2 supports up to 128 Watchers.

[**19:10**] The address behind the end of the abortion body.

[**09:00**] The start address of the abortion body.

Second Instruction Encoding:

[**31:27**] Opcode=ABORT/WABORT/SUSPEND.

[**26:19**] The signal's unicode value. See also Section 3.4.

[**18:16**] "000" indicates ABORT (strong abortion).

[**15:00**] The initialized counter value.

A.1.2 WABORT

Assembly syntax:
   WABORT *n, S, endAddr* (*, startAddr*)

n Counter value (default: 1).

S The name of the signal.

endAddr The address behind the end of the abortion body; see Figure 15.

startAddr The start address of the abortion body. It is generated by the KEP2 compiler automatically, see Appendix C.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
First Instruction Encoding	00011	wwwwwwwa	aaa	aaaaaa	AAAAAAAAAA
Second Instruction Encoding	00011	SSSSSSSS	100	nnnnnn	nnnnnnnnnn

First Instruction Encoding:

[31:27] Opcode=`ABORT/WABORT/SUSPEND`

[26:20] The index number of the Watcher, the KEP2 supports up to 128 Watchers.

[19:10] The address behind the end of the abortion body.

[09:00] The start address of the abortion body.

Second Instruction Encoding:

[31:27] Opcode=`ABORT/WABORT/SUSPEND`

[26:19] The signal's unicode value. See also Section 3.4.

[18:16] `"100"` indicates `WABORT` (weak abortion).

[15:00] The initialized counter value.

A.1.3 SUSPEND

Assembly syntax:

    SUSPEND 1, S, endAddr (, startAddr)

S The name of the signal.

endAddr The address behind the end of the suspension body; see Figure 15.

startAddr The start address of the suspension body. It is generated by the KEP2 compiler automatically, see Appendix C.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
First Instruction Encoding	00011	wwwwwwwa	aaa	aaaaaa	AAAAAAAAAA
Second Instruction Encoding	00011	SSSSSSSS	010	000000	0000000001

First Instruction Encoding:

[31:27] Opcode=`ABORT/WABORT/SUSPEND`

[26:20] The index number of the Watcher, the KEP2 supports up to 128 Watchers.

[19:10] The address behind the end of the suspension body.

[09:00] The start address of the suspension body.

Second Instruction Encoding:

[31:27] Opcode=`ABORT/WABORT/SUSPEND`

[26:19] The signal's unicode value. Total 63 signals can be supported. See also Section 3.4.

[18:16] `"010"` indicates `SUSPEND`.

[15:00] The initialized counter value is fixed in 1.

```
%Esterel
emit A;
abort
  emit B;
  suspend
    emit C;
    await D;
  when E;
  emit A;
when 3 F;
emit A;
weak abort
  emit H;
  await I;
when pre(J);
emit A;
```

```
%KEP ASM
EMIT A
ABORT 3,F,A0
  EMIT B
  SUSPEND 1,E,A1
    EMIT C
    AWAIT D
A1:
  EMIT A
A0:
  EMIT A
  WABORT 1,PRE(J),A2
    EMIT H
    AWAIT I
A2:
  EMIT A
```

Figure 15: Translating the Esterel `abort`/`weak abort`/`suspend` statements to the KEP2 instructions.

A.2 Signal awaiting (`await`, `pause`)

A.2.1 AWAIT

Assembly syntax:

```
AWAIT S
AWAIT n,S
AWAIT n,Tick
```

n Counter value.

S The name of the signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00110	SSSSSSSS	000	nnnnnn	nnnnnnnnnn

Encoding:

[23:19] Opcode=`AWAIT/PAUSE`

[26:19] The signal's unicode value. See also Section 3.4.

[15:00] The initialized counter value. For the `AWAIT S` expression, the value equals 1.

A.2.2 PAUSE

Assembly syntax:

```
PAUSE
```

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00110	00000000	000	000000	0000000001

Encoding:

[23:19] Opcode=`AWAIT/PAUSE`

[26:19] The unicode value of Tick. See also Section 3.4.

[15:00] The initialized counter value equals 1.

```
%Esterel
emit A;
await B;
emit C;
await 25 D;
pause;
await pre(B);
emit E;
```

```
%KEP ASM
EMIT A
AWAIT B
EMIT C
AWAIT 25,D
PAUSE
AWAIT PRE(B)
EMIT E
```

Figure 16: Translating the Esterel `await/pause` statements to the KEP2 `AWAIT/PAUSE` instructions.

A.3   Multiple signal awaiting (`await case`)

A.3.1   CAWAIT

Assembly syntax:

    CAWAIT S_n, $S_n startAddr$

S_n The name of the signal.

$S_n startAddr$ The start address of the CASE body.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00001	SSSSSSSS	000	000000	aaaaaaaaaa

Encoding:

[**31:27**] Opcode=`CAWAIT/CAWAITE`

[**26:19**] The signal's unicode value. See also Section 3.4.

[**18:16**] Being set to "000" when `CAWAIT`. It means this is not the last `CASE` of the `CASE` list.

[**09:00**] The start address of Signal S_n's `CASE` body.

A.3.2   CAWAITE

Assembly syntax:

    CAWAITE S_n, $S_n startAddr$

S_n The name of the signal.

$S_n startAddr$ The start address of the last CASE body.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00001	SSSSSSSS	100	000000	aaaaaaaaaa

Encoding:

[**31:27**] Opcode=`CAWAIT/CAWAITE`

[**26:19**] The signal's unicode value. See also Section 3.4.

[**18:16**] Being set to "100" when `CAWAITE`. It means this is the last `CASE` of the `CASE` list. See also Figure 17. KEP2 watches all of the `CASE` sensitivity signals concurrently until any of them is presented. If several signals occur simultaneously, the first relative `CASE` in the list will take priority and others will be ignored, as specified in Esterel [3].

```
                              %KEP ASM
                              EMIT G
                              CAWAIT A,A0
                              CAWAIT B,A1
                              CAWAITE C,A2
                          A0:
     %Esterel                  EMIT D
     emit G;                   GOTO A3
     await                 A1:
       case A do              EMIT E
         emit D;              GOTO A3
       case B do          A2:
         emit E;              EMIT F
       case C do             GOTO A3
         emit F;          A3:
     end await;               EMIT G
     emit G;
```

Figure 17: Translating the Esterel `await case` statements to the KEP2 `CAWAIT/CAWAITE` instructions.

[**09:00**] The start address of Signal S_n's `CASE` body.

A.4 Signal emission (`emit`)

A.4.1 EMIT

Assembly syntax:
  EMIT *S*
  EMIT *S*,#*data*

S The name of the signal.

data The carried data of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00111	SSSSSSSS	000	nnnnnn	nnnnnnnnnn

Encoding:

[**31:27**] Opcode=`EMIT`

[**26:19**] The signal's unicode value. See also Section 3.4.

[**15:00**] The carried data of valued signal, which ranges from 0 to 65535 (16-bit). Being set to "000000 0000000000" when the expression is `EMIT S` (pure signal).

A.4.2 EMITR

Assembly syntax:
  EMITR *S*,*reg*

S The name of the signal.

reg The name of the register.

```
%Esterel
emit A;
emit B(27);
emit C(REG_A);
```

```
%KEP ASM
EMIT A
EMIT B,#27
EMITR C,REG_A
```

Figure 18: Translating the Esterel `emit` statements to the KEP2 `EMIT/EMITR` instructions.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10010	SSSSSSSS	000	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=`EMITR`

[**26:19**] The signal's unicode value. See also Section 3.4.

[**15:10**] The index value of the register. The contents of the register *reg* will be output as the carried data of the valued signal.

A.5 Sustaining signals (`sustain`)

A.5.1 SUSTAIN

Assembly syntax:
    <u>SUSTAIN *S*</u>
    <u>SUSTAIN *S*,#*data*</u>

S The name of the signal.

data The carried data of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	01000	SSSSSSSS	000	nnnnnn	nnnnnnnnnn

Encoding:

[**31:27**] Opcode=`SUSTAIN`

[**26:19**] The signal's unicode value. See also Section 3.4.

[**15:00**] The carried data of the valued signal ranges from 0 to 65535 (16-bit). Being set to "000000 0000000000" when the expression is `SUSTAIN S` (pure signal).

A.5.2 SUSTAINR

Assembly syntax:
    SUSTAINR *S*,*reg*

S The name of the signal.

reg The name of the register. (See Figure 19)

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10011	SSSSSSSS	000	rrrrrr	0000000000

Encoding:

```
%Esterel
sustain A;
sustain B(27);
sustain C(REG_A);
```

```
%KEP ASM
SUSTAIN A
SUSTAIN B,#27
SUSTAINR C,REG_A
```

Figure 19: Translating the Esterel `sustain` statements to the KEP2 `SUSTAIN/SUSTAINR` instructions.

[**31:27**] Opcode=SUSTAINR

[**26:19**] The signal's unicode value. See also Section 3.4.

[**15:10**] The index value of the register. The contents of the register *reg* will be output as the carried data of the valued signal.

A.6   The `halt` statement

A.6.1   HALT

Assembly syntax:

HALT

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00101	00000000	000	000000	0000000000

Encoding:

[**31:27**] Opcode=HALT

A.7   The `nothing` statement

A.7.1   NOTHING

Assembly syntax:

NOTHING

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00000	00000000	000	000000	0000000000

Encoding:

[**23:19**] Opcode=NOTHING

A.8   Testing signal presence (`present`)

A.8.1   PRESENT

Assembly syntax:

PRESENT *S, elseaddr*

S The name of the signal.

elseaddr The branching address.

29

Figure 20: Translating the Esterel `present` statement to the KEP2 `PRESENT` instruction.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00010	SSSSSSSS	000	000000	aaaaaaaaaa

Encoding:

[**31:27**] Opcode=PRESENT

[**18:14**] The signal's unicode value. See also Section 3.4.

[**09:00**] The address behind the end of the presentation body. If Signal S is presented at the current cycle, the KEP2 will execute the following instruction, or else it will go to this indicated address.

A.9 Signal scoping (`signal`)

A.9.1 SIGNAL

Assembly syntax:
<u>SIGNAL *S*</u>
<u>SIGNAL PRE(*S*)</u>

S The name of the signal. If the following Esterel program accesses the previous S, the `SIGNAL PRE(S)` instruction is needed. See Figure 21

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	01001	SSSSSSSS	000	000000	0000000000

Encoding:

[**31:27**] Opcode=SIGNAL

[**18:14**] The local signal's unicode value. See also Section section 3.4.

A.10 Arithmetic and logical operations

A.10.1 CLRC

Assembly syntax:
<u>CLRC</u>

```
%Esterel
signal A in
  emit A;
end signal;
emit B;
pause;
signal A in
  present pre(A) then
    emit C;
  end present;
end signal;
halt;
```

```
%KEP ASM
SIGNAL A
%SIGNAL PRE(A)
  EMIT A
  EMIT B
SIGNAL A
SIGNAL PRE(A)
  PRESENT PRE(A),A1
    EMIT C
A1:
  HALT
```

Figure 21: Translating the Esterel `signal` statements to the KEP2 `SIGNAL` instructions.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10001	00000000	000	000000	0000000000

Encoding:

[**31:27**] Opcode=CLRC/SETC/SR/SRC/NOTR

[**18:16**] Being set to "000" when CLRC. It means the carry bit will be cleared.

A.10.2 SETC

Assembly syntax:

SETC

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10001	00000000	100	000000	0000000000

Encoding:

[**31:27**] Opcode=CLRC/SETC/SR/SRC/NOTR

[**18:16**] Being set to "100" when SETC. It means the carry bit will be set to '1'.

A.10.3 SR

Assembly syntax:

SR *reg*

reg The name of the register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10001	00000000	010	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=CLRC/SETC/SR/SRC/NOTR

[**18:16**] Being set to "010" when SR *reg*. It means the contents of the register *reg* will be right shifted. The highest bit will be replaced by '0'.

A.10.4 SRC

Assembly syntax:
  SRC *reg*

reg The name of the register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10001	00000000	110	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=CLRC/SETC/SR/SRC/NOTR

[**18:16**] Being set to "110" when SRC *reg*. It means the contents of the register *reg* will be right shifted. The highest bit will be replaced by the value of the carry bit, and the carry bit will be replaced by the lowest bit of the *reg*.

A.10.5 NOTR

Assembly syntax:
  NOTR *reg*

reg The name of the register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10001	00000000	001	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=CLRC/SETC/SR/SRC/NOTR

[**18:16**] Being set to "001" when NOTR *reg*. It means the contents of the register *reg* will be NOTed.

A.10.6 LOAD

Assembly syntax:

- LOAD *reg,#data*

  reg The name of the target register.

  data The data to be loaded into the register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10100	RRRRRR00	000	nnnnnn	nnnnnnnnnn

Encoding:

[**31:27**] Opcode=LOAD

[**26:21**] The index value of the target register.

[**18:16**] Being set to "000" when LOAD *REG,#data*. It means the contents of the register *reg* will be replaced by the value of data.

[**15:00**] The value of the data.

- `LOAD` *`reg,src`*

    `reg` The name of the target register.

    `src` The name of the source register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10100	RRRRRR00	100	rrrrrr	0000000000

    Encoding:

    [31:27] Opcode=`LOAD`

    [26:21] The index value of the source register.

    [18:16] Being set to "100" when `LOAD` *`reg,src`*. It means the contents of the register *`reg`* will be replaced by the contents of the register *`src`*.

    [15:10] The index value of the target register.

- `LOAD` *`reg,?S`*

    `reg` The name of the target register.

    `S` The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10100	SSSSSSSS	010	rrrrrr	0000000000

    Encoding:

    [31:27] Opcode=`LOAD`

    [26:19] The valued signal's unicode. See also Section 3.4.

    [18:16] Being set to "010" when `LOAD` *`reg,?S`*. It means the contents of the register *`reg`* will be replaced by the carried data of the valued signal *`S`*.

    [15:10] The index value of the target register.

- `LOAD` *`reg,`*`PRE(`*`?S`*`)`

    `reg` The name of the target register.

    `S` The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10100	SSSSSSSS	110	rrrrrr	0000000000

    Encoding:

    [31:27] Opcode=`LOAD`

    [26:19] The valued signal's unicode. See also Section 3.4.

    [18:16] Being set to "110" when `LOAD` *`reg,`*`PRE(`*`?S`*`)`. It means the contents of the register *`reg`* will be replaced by the carried data of the valued signal *`S`* in the previous tick.

    [15:10] The index value of the target register.

A.10.7 ADD

Assembly syntax:

- ADD *reg*,#*data*

  reg The name of the target (augend/sum) register.

  data The value of addend.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	RRRRRR00	000	nnnnnn	nnnnnnnnnn

  Encoding:

  [31:27] Opcode=ADD/ADDC

  [26:21] The index value of the target (augend/sum) register.

  [18:16] Being set to "000" when ADD *reg*,#*data*. It means the contents of the register *reg* will be added with the value of the *data*, and the sum will be stored into the register *reg*.

  [15:00] The value of the data.

- ADD *reg*,*src*

  reg The name of the target (augend/sum) register.

  src The name of the source (addend) register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	RRRRRR00	100	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=ADD/ADDC

  [26:21] The index value of the source (addend) register.

  [18:16] Being set to "100" when ADD *reg*,*src*. It means the contents of the register *src* will be added with the contents of the register *reg*, and the sum will be stored into the register *reg*.

  [15:10] The index value of the target (augend/sum) register.

- ADD *reg*,?*S*

  reg The name of the target (augend/sum) register.

  S The name of the valued signal (addend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	SSSSSSSS	010	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=ADD/ADDC

  [26:19] The valued signal's unicode. See also Section 3.4.

[18:16] Being set to "010" when ADD `reg,?S`. It means the contents of the register `reg` will be added with the carried data of the valued signal `S`, and the sum will be stored into the register `reg`.

[15:10] The index value of the target (augend/sum) register.

- ADD `reg`,PRE(?`S`)

  `reg` The name of the target (augend/sum) register.

  `S` The name of the valued signal (addend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	SSSSSSSS	110	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=ADD/ADDC

  [26:19] The valued signal's unicode. See also Section 3.4.

  [18:16] Being set to "110" when ADD `reg`,PRE(?`S`). It means the contents of the register `reg` will be added with the carried data of the valued signal `S` in the previous tick, and the sum will be stored into the register `reg`.

  [15:10] The index value of the target (augend/sum) register.

A.10.8 ADDC

Assembly syntax:

- ADDC `reg,#data`

  `reg` The name of the target (augend/sum) register.

  `data` The value of addend.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	RRRRRR00	001	nnnnnn	nnnnnnnnnn

  Encoding:

  [31:27] Opcode=ADD/ADDC

  [26:21] The index value of the target (augend/sum) register.

  [18:16] Being set to "001" when ADDC `reg,#data`. It means the contents of the register `reg` will be added with the value of the `data` and the value of the carry bit, and the sum will be stored into the register `reg`.

  [15:00] The value of the data.

- ADDC `reg,src`

  `reg` The name of the target (augend/sum) register.

  `src` The name of the source (addend) register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	RRRRRR00	101	rrrrrr	0000000000

  Encoding:

[**31:27**] Opcode=`ADD/ADDC`

[**26:21**] The index value of the source (addend) register.

[**18:16**] Being set to "101" when `ADDC reg,src`. It means the contents of the register *reg* will be added with the contents of the register *src* and the value of the carry bit, and the sum will be stored into the register *reg*.

[**15:10**] The index value of the target (augend/sum) register.

- `ADDC reg,?S`

  reg The name of the target (augend/sum) register.

  S The name of the valued signal (addend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	SSSSSSSS	010	rrrrrr	0000000000

  Encoding:

  [**31:27**] Opcode=`ADD/ADDC`

  [**26:19**] The valued signal's unicode. See also Section 3.4.

  [**18:16**] Being set to "011" when `ADDC reg,?S`. It means the contents of the register *reg* will be added with the carried data of the valued signal *S* and the value of the carry bit, and the sum will be stored into the register *reg*.

  [**15:10**] The index value of the target (augend/sum) register.

- `ADDC reg,PRE(?S)`

  reg The name of the target (augend/sum) register.

  S The name of the valued signal (addend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10101	SSSSSSSS	111	rrrrrr	0000000000

  Encoding:

  [**31:27**] Opcode=`ADD/ADDC`

  [**26:19**] The valued signal's unicode. See also Section 3.4.

  [**18:16**] Being set to "111" when `ADDC reg,PRE(?S)`. It means the contents of the register *reg* will be added with the carried data of the valued signal *S* in the previous tick and the value of the carry bit, and the sum will be stored into the register *reg*.

  [**15:10**] The index value of the target (augend/sum) register.

A.10.9 SUB

Assembly syntax:

- `SUB reg,#data`

  reg The name of the target (minuend/difference) register.

  data The value of subtrahend.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	RRRRRR00	000	nnnnnn	nnnnnnnnnn

Encoding:

[**31:27**] Opcode=SUB/SUBC

[**26:21**] The index value of the target (minuend/difference) register.

[**18:16**] Being set to "000" when SUB `reg,#data`. It means the value of the `data` will be subtracted from the contents of the register `reg`, and the difference will be stored into the register `reg`.

[**15:00**] The value of the data.

- SUB `reg,src`

  `reg` The name of the target (minuend/difference) register.

  `src` The name of the source (subtrahend) register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	RRRRRR00	100	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=SUB/SUBC

[**26:21**] The index value of the source (subtrahend) register.

[**18:16**] Being set to "100" when SUB `reg,src`. It means the contents of the register `src` will be subtracted from the contents of the register `reg`, and the difference will be stored into the register `reg`.

[**15:10**] The index value of the target (minuend/difference) register.

- SUB `reg,?S`

  `reg` The name of the target (minuend/difference) register.

  `S` The name of the valued signal (subtrahend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	SSSSSSSS	010	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=SUB/SUBC

[**26:19**] The valued signal's unicode. See also Section 3.4.

[**18:16**] Being set to "010" when SUB `reg,?S`. It means the carried data of the valued signal `S` will be subtracted from the contents of the register `reg`, and the difference will be stored into the register `reg`.

[**15:10**] The index value of the target (minuend/difference) register.

- SUB `reg,PRE(?S)`

  `reg` The name of the target (minuend/difference) register.

  `S` The name of the valued signal (subtrahend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	SSSSSSSS	110	rrrrrr	0000000000

Encoding:

[31:27] Opcode=SUB/SUBC

[26:19] The valued signal's unicode. See also Section 3.4.

[18:16] Being set to "110" when SUB `reg`,PRE(`?S`). It means the carried data of the valued signal `S` in the previous tick will be subtracted from the contents of the register `reg`, and the difference will be stored into the register `reg`.

[15:10] The index value of the target (minuend/difference) register.

A.10.10 SUBC

Assembly syntax:

- SUBC `reg`,#`data`

  `reg` The name of the target (minuend/difference) register.

  `data` The value of subtrahend.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	RRRRRR00	001	nnnnnn	nnnnnnnnnn

Encoding:

[31:27] Opcode=SUB/SUBC

[26:21] The index value of the target (minuend/difference) register.

[18:16] Being set to "001" when SUBC `reg`,#`data`. It means the value of the `data` and the value of the carry bit will be subtracted from the contents of the register `reg`, and the difference will be stored into the register `reg`.

[15:00] The value of the data.

- SUBC `reg`,`src`

  `reg` The name of the target (minuend/difference) register.

  `src` The name of the source (subtrahend) register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	RRRRRR00	101	rrrrrr	0000000000

Encoding:

[31:27] Opcode=SUB/SUBC

[26:21] The index value of the source (subtrahend) register.

[18:16] Being set to "101" when SUBC `reg`,`src`. It means the contents of the register `src` and the value of the carry bit will be subtracted from the contents of the register `reg`, and the difference will be stored into the register `reg`.

[15:10] The index value of the target (minuend/difference) register.

- SUBC `reg`,`?S`

reg The name of the target (minuend/difference) register.

S The name of the valued signal (subtrahend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	SSSSSSSS	011	rrrrrr	0000000000

Encoding:

[31:27] Opcode=SUB/SUBC

[26:19] The index value of the source valued signal.

[18:16] Being set to "011" when SUBC *reg,?S*. It means the carried data of the valued signal *S* and the value of the carry bit will be subtracted from the contents of the register *reg*, and the difference will be stored into the register *reg*.

[15:10] The index value of the target (minuend/difference) register.

- SUBC *reg*,PRE(?*S*)

  reg The name of the target (minuend/difference) register.

  S The name of the valued signal (subtrahend).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10110	SSSSSSSS	111	rrrrrr	0000000000

Encoding:

[31:27] Opcode=SUB/SUBC

[26:19] The index value of the source valued signal.

[18:16] Being set to "111" when SUBC *reg*,PRE(?*S*). It means the carried data of the valued signal *S* in the previous tick and the value of the carry bit will be subtracted from the contents of the register *reg*, and the difference will be stored into the register *reg*.

[15:10] The index value of the target (minuend/difference) register.

A.10.11   MUL

Assembly syntax:

- MUL *reg*,#*data*

  reg The name of the target (multiplicand/product) register.
  data The value of multiplier.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10111	RRRRRR00	000	nnnnnn	nnnnnnnnnn

Encoding:

[31:27] Opcode=MUL

[26:21] The index value of the target (multiplicand) register.

[18:16] Being set to "000" when MUL `reg,#data`. It means the contents of the register `reg` will be multiplied by the value of the `data`, and the product will be stored into the register `reg`.

[15:00] The value of the data.

- MUL `reg,src`

  `reg` The name of the target (multiplicand/product) register.

  `src` The name of the source (multiplier) register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10111	RRRRRR00	100	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=MUL

  [26:21] The index value of the source (multiplier) register.

  [18:16] Being set to "100" when MUL `reg,src`. It means the contents of the register `reg` will be multiplied by the the contents of the register `src`, and the product will be stored into the register `reg`.

  [15:10] The index value of the target (multiplicand/product) register.

- MUL `reg,?S`

  `reg` The name of the target (multiplicand/product) register.

  `S` The name of the valued signal (multiplier).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10111	SSSSSSSS	010	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=MUL

  [26:19] The valued signal's unicode. See also Section 3.4.

  [18:16] Being set to "010" when MUL `reg,?S`. It means the contents of the register `reg` will be multiplied by the carried data of the valued signal `S`, and the product will be stored into the register `reg`.

  [15:10] The index value of the target (multiplicand/product) register.

- MUL `reg,PRE(?S)`

  `reg` The name of the target (multiplicand/product) register.

  `S` The name of the valued signal (multiplier).

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10111	SSSSSSSS	110	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=MUL

  [26:19] The valued signal's unicode. See also Section 3.4.

[18:16] Being set to "110" when `MUL reg,PRE(?S)`. It means the contents of the register *reg* will be multiplied by the carried data of the valued signal *S* in the previous tick, and the product will be stored into the register *reg*.

[15:10] The index value of the target (multiplicand/product) register.

A.10.12   ANDR

Assembly syntax:

- `ANDR reg,#data`

  REG The name of the target register.

  data The operating data.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11000	RRRRRR00	000	nnnnnn	nnnnnnnnnn

  Encoding:

  [31:27] Opcode=`ANDR`

  [26:21] The index value of the target register.

  [18:16] Being set to "000" when `ANDR reg,#data`. It means the contents of the register *reg* will be ANDed with the value of the *data*, and the result will be stored into the register *reg*.

  [15:00] The value of the operating data.

- `ANDR reg,src`

  reg The name of the target register.

  src The name of the source register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11000	RRRRRR00	100	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=`ANDR`

  [26:21] The index value of the source register.

  [18:16] Being set to "100" when `ANDR reg,src`. It means the contents of the register *reg* will be ANDed with the contents of the register *src*, and the result will be stored into the register *reg*.

  [15:10] The index value of the target register.

- `ANDR reg,?S`

  reg The name of the target register.

  S The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11000	SSSSSSSS	010	rrrrrr	0000000000

  Encoding:

[31:27] Opcode=`ANDR`

[26:19] The valued signal's unicode. See also Section 3.4.

[18:16] Being set to "010" when `ANDR` *reg*,`?S`. It means the contents of the register *reg* will be ANDed with the carried data of the valued signal *S*, and the result will be stored into the register *reg*.

[15:10] The index value of the target register.

- `ANDR` *reg*,`PRE(?S)`

  reg The name of the target register.

  S The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11000	SSSSSSSS	110	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=`ANDR`

  [26:19] The index value of the source valued signal.

  [18:16] Being set to "110" when `ANDR` *reg*,`PRE(?S)`. It means the contents of the register *reg* will be ANDed with the carried data of the valued signal *S* in the previous tick, and the result will be stored into the register *reg*.

  [15:10] The index value of the target register.

A.10.13 ORR

Assembly syntax:

- `ORR` *reg*,`#`*data*

  reg The name of the target register.

  data The operating data.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11001	RRRRRR00	000	nnnnnn	nnnnnnnnnn

  Encoding:

  [31:27] Opcode=`ORR`

  [26:21] The index value of the target register.

  [18:16] Being set to "000" when `ORR` *reg*,`#`*data*. It means the contents of the register *reg* will be ORed with the value of the *data*, and the result will be stored into the register *reg*.

  [15:00] The value of the operating data.

- `ORR` *reg*,*src*

  reg The name of the target register.

  src The name of the source register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11001	RRRRRR00	100	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=`ORR`

[**26:21**] The index value of the source register.

[**18:16**] Being set to "100" when `ORR` `reg,src`. It means the contents of the register `reg` will be ORed with the contents of the register `src`, and the result will be stored into the register `reg`.

[**15:10**] The index value of the target register.

- `ORR` `reg,?S`

  `reg` The name of the target register.

  `S` The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11001	SSSSSSSS	010	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=`ORR`

[**26:19**] The valued signal's unicode. See also Section 3.4.

[**18:16**] Being set to "010" when `ORR` `reg,?S`. It means the contents of the register `reg` will be ORed with the carried data of the valued signal `S`, and the result will be stored into the register `reg`.

[**15:10**] The index value of the target register.

- `ORR` `reg,PRE(?S)`

  `reg` The name of the target register.

  `S` The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11001	SSSSSSSS	110	rrrrrr	0000000000

Encoding:

[**31:27**] Opcode=`ORR`

[**26:19**] The index value of the source valued signal.

[**18:16**] Being set to "110" when `ORR` `reg,PRE(?S)`. It means the contents of the register `reg` will be ORed with the carried data of the valued signal `S` in the previous tick, and the result will be stored into the register `reg`.

[**15:10**] The index value of the target register.

A.10.14 XORR

Assembly syntax:

- XORR *reg*,#*data*

  reg The name of the target register.

  data The operating data.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11010	RRRRRR00	000	nnnnnn	nnnnnnnnnn

  Encoding:

  [**31:27**] Opcode=XORR

  [**26:21**] The index value of the target register.

  [**18:16**] Being set to "000" when XORR *reg*,#*data*. It means the contents of the register *reg* will be XORed with the value of the *data*, and the result will be stored into the register *reg*.

  [**15:00**] The value of the operating data.

- XORR *reg*,*src*

  reg The name of the target register.

  src The name of the source register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11010	RRRRRR00	100	rrrrrr	0000000000

  Encoding:

  [**31:27**] Opcode=XORR

  [**26:21**] The index value of the source register.

  [**18:16**] Being set to "100" when XORR *reg*,*src*. It means the contents of the register *reg* will be XORed with the contents of the register *src*, and the result will be stored into the register *reg*.

  [**15:10**] The index value of the target register.

- XORR *reg*,?*S*

  reg The name of the target register.

  S The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11010	SSSSSSSS	010	rrrrrr	0000000000

  Encoding:

  [**31:27**] Opcode=XORR

  [**26:19**] The valued signal's unicode. See also Section 3.4.

[18:16] Being set to "010" when XORR `reg,?S`. It means the contents of the register `reg` will be XORed with the carried data of the valued signal `S`, and the result will be stored into the register `reg`.

[15:10] The index value of the target register.

- XORR `reg,PRE(?S)`

  `reg` The name of the target register.

  `S` The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11010	SSSSSSSS	110	rrrrrr	0000000000

  Encoding:

  [31:27] Opcode=XORR

  [26:19] The index value of the source valued signal.

  [18:16] Being set to "110" when XORR `reg,PRE(?S)`. It means the contents of the register `reg` will be XORed with the carried data of the valued signal `S` in the previous tick, and the result will be stored into the register `reg`.

  [15:10] The index value of the target register.

A.10.15 CMP

Assembly syntax:

- CMP `reg,#data`

  `reg` The name of the target register.

  `data` The operating data.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11011	RRRRRR00	000	nnnnnn	nnnnnnnnnn

  Encoding:

  [31:27] Opcode=CMP

  [26:21] The index value of the target register.

  [18:16] Being set to "000" when CMP `reg,#data`. It means 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'.

  [15:00] The value of the operating data.

- CMP `reg,src`

  `reg` The name of the target register.

  `src` The name of the source register.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11011	RRRRRR00	100	rrrrrr	0000000000

Encoding:

[31:27] Opcode=CMP

[26:21] The index value of the source register.

[18:16] Being set to "100" when CMP `reg,src`. It means the contents of the register `reg` will be compared with the contents of the register `src`. The result will affect the zero and carry bits.

[15:10] The index value of the target register.

- CMP `reg,?S`

  `reg` The name of the target register.

  `S` The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11011	SSSSSSSS	010	rrrrrr	0000000000

Encoding:

[31:27] Opcode=CMP

[26:19] The valued signal's unicode. See also Section 3.4.

[18:16] Being set to "010" when CMP `reg,?S`. It means 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.

[15:10] The index value of the target register.

- CMP `reg,PRE(?S)`

  `reg` The name of the target register.

  `S` The name of the valued signal.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	11011	SSSSSSSS	110	rrrrrr	0000000000

Encoding:

[31:27] Opcode=CMP

[26:19] The valued signal's unicode. See also Section 3.4.

[18:16] Being set to "110" when CMP `reg,PRE(?S)`. It means the contents of the register `reg` will be compared with the carried data of the valued signal `S` in the previous tick. The result will affect the zero and carry bits.

[15:10] The index value of the target register.

```
%Esterel
REG_A:=25;
REG_B:=REG_A+32;
emit C(REG_A);
emit D((?C)*REG_B);
halt;
```

```
%KEP ASM
LOAD REG_A, #25
LOAD REG_B, REG_A
ADD REG_B, #32
EMITR C, REG_A
LOAD REG_TMP, ?C
MUL REG_TMP, REG_B
EMITR D, REG_TMP
HALT
```

Figure 22: Translating the Esterel arithmetic statements to the KEP2 instructions.

A.11   The conditional branch statement

A.11.1   JW

Assembly syntax:

- JW Z,*elseaddr*

  elseaddr The branching address.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10000	00000000	000	000000	aaaaaaaaaa

  Encoding:

  [**31:27**] Opcode=JW

  [**18:16**] Being set to "000" when JW Z,*elseaddr*. It means that the KEP2 will test the zero bit. If the zero bit is '1' (true), the KEP2 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 is stored into the register file.

  [**09:00**] The branching address.

- JW L,*elseaddr*

  elseaddr The branching address.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10000	00000000	000	000000	aaaaaaaaaa

  Encoding:

  [**31:27**] Opcode=JW

  [**18:16**] Being set to "010" when JW L,*elseaddr*. It means that the KEP2 will test the carry bit. If the carry bit is '1' (true), the KEP2 will execute the following instruction, or else it will go to the branching address. The status of the zero 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 JW C,*elseaddr*.

  [**09:00**] The branching address.

- JW G,*elseaddr*

47

elseaddr The branching address.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10000	00000000	110	000000	aaaaaaaaaa

Encoding:

[**31:27**] Opcode=JW

[**18:16**] Being set to "110" when JW G,*elseaddr*. It means the KEP2 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 KEP2 will execute the following instruction, or else it will go to the branching address.

[**09:00**] The branching address.

- JW GE,*elseaddr*

  elseaddr The branching address.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10000	00000000	001	000000	aaaaaaaaaa

  Encoding:

  [**31:27**] Opcode=JW

  [**18:16**] Being set to "001" when JW GE,*elseaddr*. It means the KEP2 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 KEP2 will execute the following instruction, or else it will go to the branching address.

  [**09:00**] The branching address.

- JW LE,*elseaddr*

  elseaddr The branching address.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10000	00000000	101	000000	aaaaaaaaaa

  Encoding:

  [**31:27**] Opcode=JW

  [**18:16**] Being set to "101" when JW LE,*elseaddr*. It means the KEP2 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 KEP2 will execute the following instruction, or else it will go to the branching address.

  [**09:00**] The branching address.

- JW EE,*elseaddr*

  elseaddr The branching address.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10000	00000000	011	000000	aaaaaaaaaa

  Encoding:

```
%Esterel
REG_A:=?C;
if REG_A>=35 then
  emit C(REG_A);
else
  emit C(0);
end if;
pause;
emit D(pre(?C));
halt;
```

```
%KEP ASM
LOAD REG_A, ?C
CMP REG_A, #35
JW GE, AO
EMITR C, REG_A
GOTO A1
AO:
  EMIT C,#0
A1:
  PAUSE
  LOAD REG_TMP, PRE(?C)
  EMITR D, REG_TMP
  HALT
```

Figure 23: Translating the Esterel branch statements to the KEP2 branch instructions.

[**31:27**] Opcode=JW

[**18:16**] Being set to "011" when JW EE,*elseaddr*. It means the KEP2 will test the zero bit. This instruction should be used right after a CMP instruction. If the result of the comparison is equal, the KEP2 will execute the following instruction, or else it will go to the branching address.

[**09:00**] The branching address.

- JW NE,*elseaddr*

elseaddr The branching address.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	10000	00000000	111	000000	aaaaaaaaaa

Encoding:

[**31:27**] Opcode=JW

[**18:16**] Being set to "111" when JW NE,*elseaddr*. It means the KEP2 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 KEP2 will execute the following instruction, or else it will go to the branching address.

[**09:00**] The branching address.

A.12 Others

A.12.1 GOTO

Assembly syntax:

GOTO *addr*

addr The target address

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	00100	00000000	000	000000	aaaaaaaaaa

Encoding:

[31:27] Opcode=`GOTO`

[09:00] The target address. The KEP2 will go to this indicated address.

A.12.2 CALL

Assembly syntax:

    CALL *addr*

addr The start address of the subroutine.

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	01010	00000000	000	000000	aaaaaaaaaa

Encoding:

[31:27] Opcode=`CALL`

[09:00] 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.

A.12.3 RET

Assembly syntax:

    RET

	$d_{31} - d_{27}$	$d_{26} - d_{19}$	$d_{18} - d_{16}$	$d_{15} - d_{10}$	$d_{09} - d_{00}$
Encoding	01011	00000000	000	000000	0000000000

Encoding:

[31:27] Opcode=`RET` Return to the procedure from a subroutine. The pushed address will be popped as the target address.

B Synthesizing a KEP Configuration

We use several programs to generate the scalable blocks, which are then combined into a processor series to target different applications efficiently. Naturally, a processor which contains more input/output signals, supports wider data range, or includes more peripheral cells (*e. g.*, Watcher, Case Cell, etc.) will result in more resource usage and less system speed.

B.1 Reactive Block Generator

The `blkreactive` command invokes this generator, and creates a synthesizable VHDL program for implementing the Reactive Block.

```
blkreactive -s value [-l nestlevels] [-d datawidth] [-c casenums]
```

Options

- `-s value`
  Specifies the amount of Reactive Block's input signals. The `value` ranges from 1 to 63.

- `-l nestlevels`
  Specifies the amount of Reactive Block's nest levels. The `nestlevels` ranges from 1 to 127. The default value is 2.

- `-d datawidth`
  Specifies the bit width of Reactive Block's counters. The `datawidth` ranges from 0 to 15 (16-bit). The default value is 1.

- `-c casenums`
  Specifies the amount of Reactive Block's CASE Cells. The `casenums` ranges from 1 to MAXINT, the largest representable integer. The default value is 2.

B.2 Interface Block Generator

The `blkinterface` command invokes this generator, and creates a synthesizable VHDL program for implementing the Interface Block.

```
blkinterface -s value [-d datawidth] [-v valuedsignalnums]
```

Options

- `-s value`
  Specifies the amount of Interface Block's pure and valued input/output signals. The `value` ranges from 1 to 63. *e. g.*, 8 input signals and 8 output signals will be created when the `value` is 8.

- `-d datawidth`
  Specifies the carried data width of Interface Block's valued signals. To be compatible with the inner register file, the `datawidth` should be 8 or 16 (8-bit/16-bit). The default data width of valued signals is 16-bit, which ranges from 0 to 65535.

- `-v valuedsignalnums`
  Specifies the amount of Interface Block's valued input/output signals. *e. g.*, 1 valued input signal and 1 valued output signal will be created when the `valuedsignalnums` is 1.

B.3 **Datapath Block** Generator

The `blkreg` command invokes this generator, and creates a synthesizable VHDL program for implementing the Datapath Block, which includes register file, ALU, etc.

`blkreg -d` *datawidth*

Options

- `-d` *datawidth*
  Specify the data width of Datapath Block. It determines the data width for calculating, logical operation, and the data width of the contents of register file, etc. The *datawidth* should be 8 or 16 (8-bit/16-bit). The default data width is 16-bit, which ranges from 0 to 65535. We recommend that the *datawidth* parameter is consistent with others block generators' options.

B.4 **Innerconnection** Generator

The `blkkep` command invokes this generator, and creates a synthesizable VHDL program for implementing the innerconnection of KEP2's blocks.

`blkkep -s value` *value* `[-d` *datawidth*`] [-v` *valuedsignalnums*`]`

Options

- `-s` *value*
  Specifies the amount of KEP2's input/output signals. The *value* ranges from 1 to 63. It ought to be consistent with the *value* parameter of the Interface Block Generator.

- `-d` *datawidth*
  Specifies the data width of KEP2. To be compatible with the inner register file, the *datawidth* should be 8 or 16 (8-bit/16-bit). The default data width is 16-bit. We recommend that the *datawidth* parameter is consistent with others block generators' options.

- `-v` *valuedsignalnums*
  Specifies the amount of KEP2's valued input/output signals. It ought to be consistent with the *valuedsignalnums* option of Interface Block Generator.

C KEP2 Assembler Compiler

The KEP2 Assembler Compiler compiles a KEP2 assembler file into executable codes. The command for invoking KEP2 compiler is:

    kepcmp [-w resource|speed|all] [-s *value*] [-v *valuedsignalnum*] [-d *datawidth*]
-i *filename*

Options

- `-w resource|speed|all`
  Specifies the Watcher optimized strategy.

- `-s *value*`
  Specifies the maximum amount of KEP2's input/output signals. The compiler will give error message if the I/O limitation is not satisfied. This option can be omitted.

- `-v *valuedsignalnums*`
  Specifies the maximum amount of KEP2's valued input/output signals. The compiler will give error message if the valued I/O signal limitation is not satisfied. This option can be omitted.

- `-i *filename*`
  Specifies the (input) KEP2 assembler language file name.

A novelty of the compiler is the optimized strategy for Watcher initialization. To illustrate, considers the RUNNER [3] module shown in Figure 24.

Figure 24 (a) shows the Esterel version. The KEP does not support `every` statement directly, but it can be translated into the expression of `abort` statement according the syntax translation rules. The `-w speed` option of KEP2 compiler invokes *speed optimization*. The `-w resource` option makes the compiler generate the original assembler codes which is shown in Figure 24 (b). This strategy aims to save the Watcher usage and is called *resource optimization*.

For the original program, assume the KEP2 stays on the first `HALT` instruction and the `MORNING` signal is presented. The abortion `MORNING` will be activated. KEP2 kills this abortion and executes the `GOTO A0`. And then the execution will continue until the `SUSTAIN WALK` instruction is met. A total of 9 instruction cycles are needed.

According to KEP2's preemption implementing process, once the Watcher is configured, it can run automatically and there is no need to be managed any more. So the source assembler program can be transformed to the following style.

At first, the compiler marks the start addresses of the preemptions body, *i.e.*, `$0`, `$1`, `$2`, `$3` and `$4`. Then those parameters are combined with the original preemption instructions, see Figure 25. A combined preemption instruction contains all parameters for configuring the Watcher. That means the preemption instruction can be located at anywhere in the program. So the compiler moves all preemption instructions to the beginning of the module, out of the loop body. Since the Tick Manger will not work until its initialization instruction, *i.e.*, `EMIT _TICKLEN,#data`, is executed, the moved instructions ought to be before the `EMIT _TICKLEN,#data` instruction. The moved preemption instructions and the Tick Manger initialization instruction constitute the initializing part of the program. So the *data* in the `EMIT _TICKLEN,#data` instruction can also be re-assigned to indicate a shorter tick length.

For the optimized program, under the same condition, the required period is just 3 instruction cycles because all of the `ABORT` instructions need not be executed anymore. The result is shown in Figure 25 (b). The execution time speeds up 3 times. This strategy improves the execution time obviously and is called *speed optimization*.
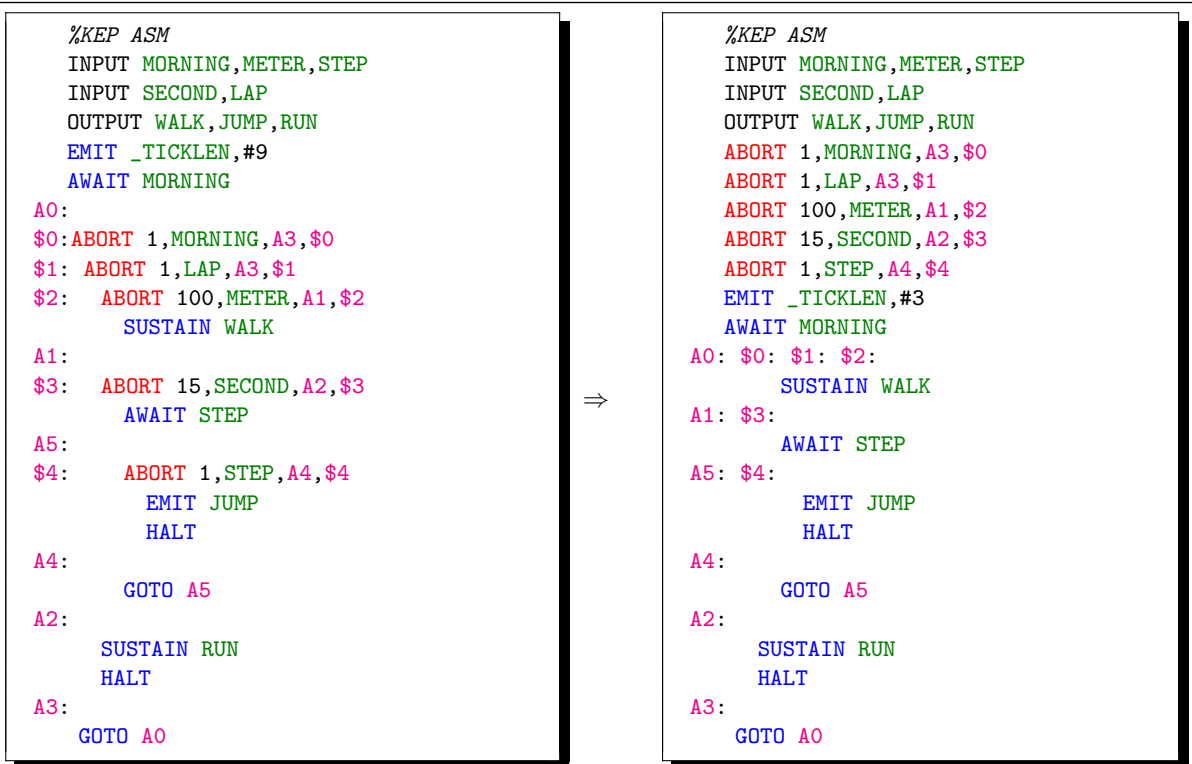
⇒

```
%KEP ASM
%MODULE RUNNER
INPUT MORNING,METER,STEP
INPUT SECOND,LAP
OUTPUT WALK,JUMP,RUN
EMIT _TICKLEN,#9
AWAIT MORNING
A0:
  ABORT 1,MORNING,A3
    ABORT 1,LAP,A3
      ABORT 100,METER,A1
        SUSTAIN WALK
A1:
        ABORT 15,SECOND,A2
          AWAIT STEP
A5:
          ABORT 1,STEP,A4
            EMIT JUMP
            HALT
A4:
          GOTO A5
A2:
      SUSTAIN RUN
      HALT
A3:
    GOTO A0
```

(a)   (b)

Figure 24: The program of the example RUNNER in Esterel (a) and KEP assembler (b).

```
%KEP ASM
INPUT MORNING,METER,STEP
INPUT SECOND,LAP
OUTPUT WALK,JUMP,RUN
EMIT _TICKLEN,#9
AWAIT MORNING
A0:
$0:ABORT 1,MORNING,A3,$0
$1: ABORT 1,LAP,A3,$1
$2:   ABORT 100,METER,A1,$2
         SUSTAIN WALK
A1:
$3:   ABORT 15,SECOND,A2,$3
         AWAIT STEP
A5:
$4:      ABORT 1,STEP,A4,$4
            EMIT JUMP
            HALT
A4:
         GOTO A5
A2:
      SUSTAIN RUN
      HALT
A3:
    GOTO A0
```

```
%KEP ASM
INPUT MORNING,METER,STEP
INPUT SECOND,LAP
OUTPUT WALK,JUMP,RUN
ABORT 1,MORNING,A3,$0
ABORT 1,LAP,A3,$1
ABORT 100,METER,A1,$2
ABORT 15,SECOND,A2,$3
ABORT 1,STEP,A4,$4
EMIT _TICKLEN,#3
AWAIT MORNING
A0: $0: $1: $2:
      SUSTAIN WALK
A1: $3:
      AWAIT STEP
A5: $4:
         EMIT JUMP
         HALT
A4:
      GOTO A5
A2:
   SUSTAIN RUN
   HALT
A3:
  GOTO A0
```

⇒

(a)   (b)

Figure 25: The speed optimization of the example RUNNER.
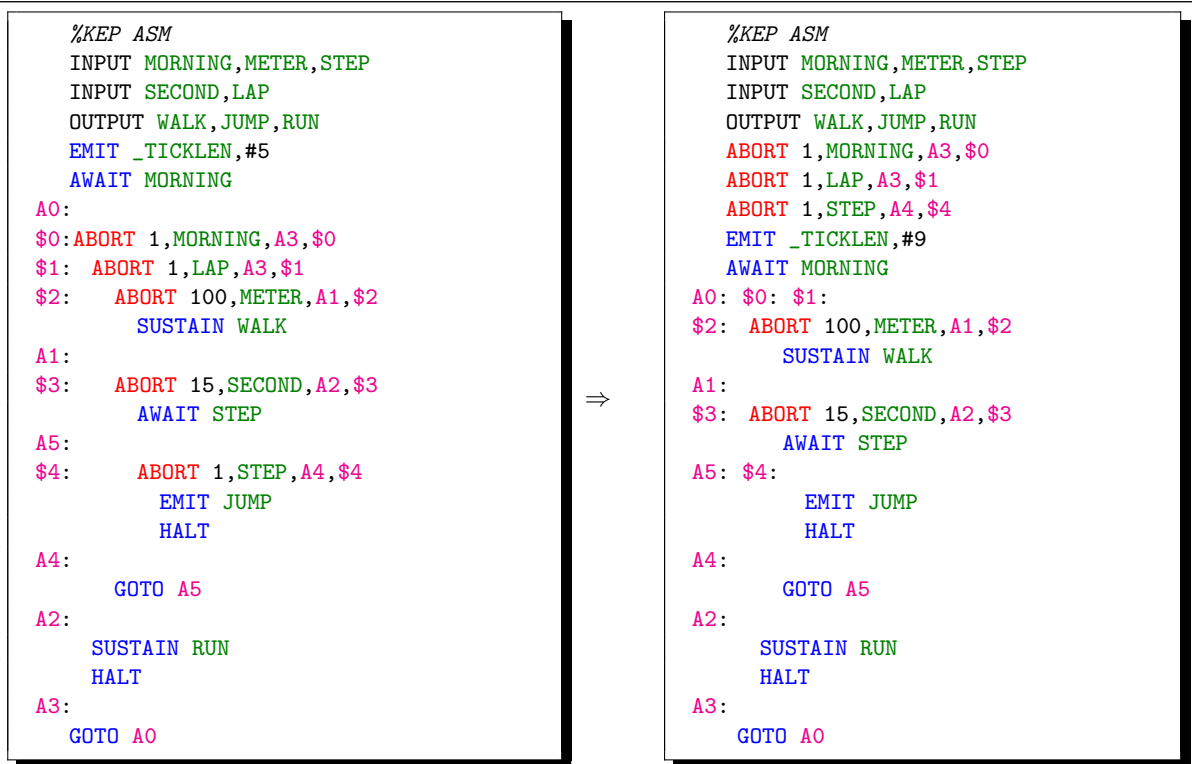
(a)   (b)

Figure 26: The tradeoff optimization of the example `RUNNER`.

The limitation of *speed optimization* is that every preemption instruction in the program occurs a Watcher. For example, since the deepest preemption nest is four levels, the original `RUNNER` assembler codes only need four Watchers in fact. But the speed optimized assembler codes of the `RUNNER` need five Watchers because the codes contain five preemption instructions.

For tradeoff between the shorter execution time and the lower Watcher usage, another optimized method is recommended. Reviewing the compiling process, the outer preemption is assigned to the higher priority Watcher. So the Watcher, which is not reused, can be judged by the corresponding instruction's priority.

For the `RUNNER` module, the abortion `MORNING` gets the highest priority since it locates in the outside. Thus this preemption is assigned to the Watcher0. For the similar reason, the abortion `LAP` is assigned to the Watcher1.

The abortion `SECOND` locates in the abortion `LAP`'s body. The Watcher2 is the corresponding Watcher. And the abortion `STEP` occurs Watcher3.

The abortion `METER` also locates in the abortion `LAP`'s body, so it gets the same priority as the abortion `SECOND`'s. Therefore, Watcher2 is used by two abortions, *i. e.*, abortion `METER` and abortion `SECOND`. That means it is a reused Watcher, and all instructions which are assigned to Watcher ought to be kept in the loop body for configuration during the execution period.

Figure 26 shows the `RUNNER` program codes which are optimized by the tradeoff strategy. Assume the KEP2 stays on the first `HALT` instruction and the `MORNING` signal is presented, totally 5 instruction cycles are needed for this instant. The execution time speeds up 1.8 times over the resource optimized codes, and the Watcher usage is less than that of the speed optimized codes. The `-w all` option of KEP2 compiler invokes the *tradeoff optimization*.

56