Esterel v7 for the Hardware Designer

Draft 1.1, unfinished because of the 2008 crisis
Version of March 20, 2008

Gérard Berry

Then at Esterel Technologies
Now at Collège de France, 11 place Marcelin Berthelot, 75005 Paris

French : https://www.college-de-france.fr/site/gerard-berry/index.htm
English : https://www.college-de-france.fr/site/en-gerard-berry/index.htm

e-mail : gerard.berry@college-de-france.fr

February 8, 2018

Contents

1 Introduction **5**

2 Control Signals and Control Behavior **7**

2.1 Signals, statements, and modules . . . 7
2.2 Pure control signals . . . 8
2.3 Emission, pausing, and sequencing . . . 9
2.4 Looping . . . 13
2.5 The sustain statement . . . 15
2.6 Signal emission equations . . . 15
2.7 Signal expressions . . . 16
2.8 Conditional branching . . . 17
2.9 Parallel execution . . . 19
2.10 Local signals and communication . . . 21
2.11 Trap and exit . . . 23

3 Temporal control **25**

3.1 The await statement . . . 25
3.2 Abortion statements . . . 26
3.3 Temporal loops . . . 29
3.4 Suspension and clock gating . . . 29
3.5 The ABRO example . . . 30
3.6 The Write Things Once principle . . . 35
3.7 The runner example . . . 35

4 Data Basics: a FIR Filter Example **37**

4.1 Data handling basics . . . 37
4.1.1 Exact unsigned arithmetic . . . 37
4.1.2 Valued signals . . . 38
4.1.3 Booleans, arrays, and bitvectors . . . 39
4.1.4 Conversions between bitvectors and numbers . . . 40
4.1.5 Interfacing valued signals at HDL level . . . 40
4.2 Basic filter specification . . . 40
4.2.1 Filtering algorithm . . . 40
4.2.2 Basic filter implementation constraints . . . 41
4.3 The BasicFilter_1 module . . . 41
4.3.1 The delay line . . . 42
4.3.2 The filter computation . . . 42
4.3.3 Datapath sizing and checking . . . 43

4.3.4 Correctness w.r.t. implementation constraints 44
4.3.5 Generated circuit structure 44
4.4 Basic filter specification updates 44
4.4.1 First specification update: initialization change 44
4.4.2 Second specification update: register the output 45
4.4.3 Third specification update: pipeline the design 46
4.5 Extracting pixels out of words 47
4.5.1 Word input specification 47
4.5.2 Using data and interface units to share declarations 47
4.5.3 Declaration refinement 48
4.5.4 Word data and interface 49
4.5.5 The WordFeeder module 49
4.5.6 The WorldFilter main module 51
4.6 Handling pixel lines 51
4.6.1 Line filter specification 51
4.6.2 From WordFilter to LineFilter 51
4.6.3 The LineFeeder module 52
4.7 Parametrizing the filter for reuse 52

Bibliography **52**

index **55**

Chapter 1

Introduction

My original goal for this book was to explain the Esterel v7 synchronous language to hardware designers by using a variety of examples. The book was intended to be a useful complement to the Esterel v7_60 Reference Manual, which I also made available this day from my Collège de France page at the following address:

Esterel v7, originally designed with Michael Kishivevsky (Intel Strategic CAD Lab, Portland, OR) [3], was a great language and toolset to elegantly design very efficient hardware circuits, as well as embedded reactive software. It has been successfully used to design a variety of research and commercial circuits in major companies.

Esterel v7 / Esterel Studio extended the previous version Esterel v5 language and system by many aspects, the most notable one being a fancy data path definition system sublanguage well illustrated in this draft. Programs were processed by a variety of tools, including a compiler to optimized Verilog or VHDL, a fancy symboloic debugger, and a formal verifier provided by the Swedish company Prover Technologies. Unfortunately, the 2008 crisis made several of our Esterel Technologies hardware customers disappear, which forced us to stop the development and cancel the whole project. Hopefully, the Esterel Technologies SCADE 6 / SCADE Studio counterpart for safety-critical software also developed by Esterel Technologies is doing great, more information at www.esterel-technologies.com.

I long hesitated in making this personal draft publicly available, but since I now teach Esterel again I think it has become appropriate to do it, even knowing that the text will probably not be made any longer for lack of time and motivation. Nevertheless, I hope it will stimulate new ideas elsewhere !

Unfortunately, the Esterel v7 compiler and the Esterel Studio toolset are no more available, since their new owner has decided to keep them deep frozen. But I can guarantee that they did work very well and that all the examples presented here did work as specified.

Gérard Berry, Paris, February 8th, 2018 - almost 10 years after the draft...

Chapter 2

Control Signals and Control Behavior

This chapter presents basics of control handling in ESTEREL v7. More elaborate temporal control will be explained in Chapter 3, and basics of data handling will be presented in Chapter 4.

2.1 Signals, statements, and modules

ESTEREL describes the design or its high-level model using communication signals and executable statements.

Signals serve as interfaces with the design environment and as internal communication objects between parts of the design that work concurrently. Interface signals. are declared with the `input` and `output` keywords. Signals can be also local to the design,then declared with the `signal` keyword.

A signal can be a pure control, e.g, a request, an acknowledge, or a valid bit indicating validity of information in an RTL design; it can be a data value, e.g., an address, a protocol payload, or an operand of an adder; it can also be a combined control-value pair, e.g., an operand of an adder together with a valid bit. Signals can be organized into homogeneous arrays or hierarchical ports with heterogeneous interfaces.

Statements control the behavior of the design and specify the data computations. They are imperative and operate temporally by controlling the state of the control flows and referring to the data path state. Some of the control statements of ESTEREL resemble explicit control propagation constructs found in C-like languages, e.g., statement sequencing, loops, and `if-then-else` and `switch` test statements. These standard constructs are are augmented by ESTEREL-specific parallel constructs, temporal constructs, and exception handling constructs. These specific constructs provide the user with a behavioral and hierarchical definition of sequential behavior, without resorting to manual state encoding. Implementation state encoding of the underlying controller is done by the compiler and can be further optimized with automatic tools.

Formally speaking, statements are divided into primitive statements and derived statements that can be macro-expanded into primitives ones. In this book, we are mostly concerned with the designer's point of view; the difference between between the primitive and derived statements is mostly irrelevant. See [2] and [5] for the definition of derived statements in terms of the primitive ones.

The main design unit is the module, which is composed of a module header that defines the signal interface and a behavioral statement called the module body:

```
module
  Header
  Body
end module
```

Here and in the remainder of this book, italicized names such as *Header* and *Body* are not ESTEREL objects, but auxiliary placeholders for declarations and statements to be described later on. Note that `module` is closed by "`end module`". This closing form will be used for all statements, e.g., "`end if`" to close `if`.

For the time being, we will only consider single-clock designs. Execution of a module is cycle-based and similar to zero-delay simulation of RTL designs. At each clock cycle, inputs are captured, and outputs and internal signals are computed using the current control state, data state, and input values. Concurrent components of the design are executed concurrently within the the same cycle. The language formal semantics and compiler implementation guarantee that control- and data-dependency constraints are respected throughout the computation.

The clock and reset signals typically present in RTL designs are implicit in a single-clock ESTEREL program and need not be declared. They become explicit in the generated HDL, see Section **??**. To refer to clock events, e.g., to count them, one uses a reserved signal name, `tick`.

A module can instantiate other modules using the `run` statement, which can be used at any place where any other ESTEREL statement can be used. Unlike for RTL designs, ESTEREL modules have execution lifetime since their behavior can be started and killed at will by surrounding temporal control statements. Therefore, an ESTEREL design allows the user to specify two types of hierarchy: a structural hierarchy, as in classical RTL design, and a behavioral hierarchy, specifically defined by ESTEREL control statements.

For better organization of large designs, the description of data objects and of interfaces can be separated from module definitions and described in separate hierarchical *data units* and *interface units*. This aspect will illustrated in Chapter 4 when designing a video filter.

2.2 Pure control signals

We start by the description of the pure control signals and of the statements that deal with them. A pure signal `S` is a Boolean signal whose value is called a *status*. At any clock tick, the signal can be *present* or *absent*. A signal `S` is present if input and set present by the environment, or if some "`emit S`" or "`sustain S`" statement is executed in the cycle; otherwise, it is absent. Throughout the book, we will identify *present* with having value 1 (or true, or high, or asserted) and *absent* with having value 0 (or false, or low, or false, or deasserted). This is the default used by the Esterel compiler when synthesizing the actual design. The compiler can also be instructed to reverse the encoding on a per-signal basis.

In addition signals can be qualified with temporal attributes that captures their behavior over time. By default signals are *immediate* (or combinational). If two modules are connected by an immediate signal, then a combinational path is crossing the module boundary, as in Mealy FSMs. Signals can be also specified as *registered* by declaring them with the `reg` or `reg1` keywords. In this case, a signal is captured using an automatically generated register with an initial value 0 (`reg`) or 1 (`reg1`). Connecting modules

through registered signals guarantees that no combinational path exists through the connection, as in Moore FSMs. Finally, pure control signals can be combined into vectors and multi-dimensional arrays with no restrictions on the number of dimensions. Vectors are simply single dimensional arrays of signals, not special "packed objects" as in HDLs. The following is an example of a signal declaration:

```
module M :
  input A, B[2];
  output X[4][5], Y;
  signal { RX, RY[32] } : reg in
    Body
  end signal
end module
```

Declarations in a comma-separated list are independent. One uses curly brackets "{}" to group declarations with the same qualifiers, here ": reg". Thus, A, B, X, and Y are immediate, while RX and RY are registered. Array components such as B[0] and B[1] can be viewed as completely independent signals. The declaration X[4][5] specifies a two-dimentional array of pure signals with 4 rows and 5 columns.

Pure signals are most often used for specifying control signals. As was mentioned before, a pure signal S is assumed to be implicitely deasserted (or 0, or false, or low) at each cycle, and it is asserted by executing an explicit emit S or sustain S emission statement as explained below.

2.3 Emission, pausing, and sequencing

The simplest statements are nothing, which does nothing in no time, emit, which asserts a signal, pause, which consumes one cycle, and the ';' statement sequencing operator[1]. Here is our first example:

```
module Ex1 :
  output X, Y, Z;
  emit X;
  pause;  // p1
  emit Y;
  pause;  // p2
  emit {X, Z}
end module
```

The body of this module shoul be read using C-like sequential control propagation where each statement starts when the previous one terminates, not as an HDL design where all statements are executed at each cycle. Figure 2.1 explains the behavior of the example. The circuit generated by the Esterel Studio compiler is shown in Figure 2.2 (before sequential optimization that reduces the number of registers by state reencoding, as explained in [4, 8] and Chapter **??**):

- at cycle 0, between the assertion of the reset and the first positive clock edge, the first "emit X" statement is executed and control passes to the *p1* "pause" statement. This statement terminates execution for the cycle and waits for the next clock edge.

[1] Formally, ';' is a separator and not a terminator. However, the Esterel compiler supports writing "emit X; emit Y;" using ';' as a C-like terminator.

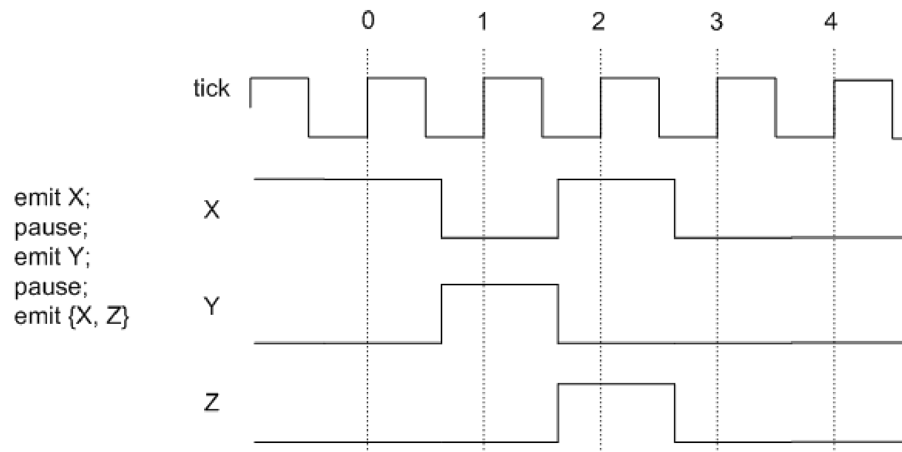

Figure 2.1: The behavior of Ex1

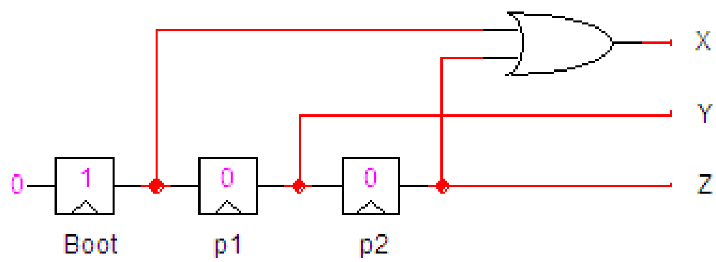

Figure 2.2: The circuit of Ex1

Therefore `X` is asserted during cycle 0, while `Y` and `Z` are deasserted since they have not been emitted during this cycle.

- at cycle 1, between the first and the second positive clock edges, control resumes from the *p1* pause statement; the "`emit Y`" statement is executed right away and pause *p2* is reached, which terminates execution for the cycle; thus, `Y` is asserted while `X` and `Z` are deasserted since they are not emitted;

- at cycle 2, control resumes from pause *p2* and the last `emit` statement is executed; this statement asserts `X` and `Z`, while `Y` is deasserted since not emitted;

- after the last `emit {X,Y}`, `end module` is reached, the module behavior is terminated, and all signals remain deasserted from then on.

The timing diagram pictured in Figure 2.1 is presented over continuous time. However, the only meaningful points in time are the clock rising edges pictured by vertical dotted lines[2], where the status of each signal should be read. Signal transitions can occur between clock rising edges as in conventional timing diagrams, but their actual positions between rising edges is immaterial. By convention, in the waveforms, input transitions are shown before the clock falling edge and output transitions are shown after the clock falling edge, to stress the fact that outputs may be caused by inputs.

Notice that `emit` is a purely combinational statement that consumes no cycle. The statement "`emit {X, Z}`" is equivalent to "`emit X; emit Z`".

Notice also that multiple emissions are allowed for pure signals. In `Ex1`, there are two `emit` statements for `X`, which happen not to act at the same time. It is also allowed to simultaneously emit the same signal several times: "`emit X; emit X`" is equivalent to "`emit X`". In the hardware circuit generated from ESTEREL, each pure signal is the output of an *or*-gate to which emission control wires are connected. See the gate with the output `X` in Figure 2.2. Combining multiple emissions of the same signal simplifies the specification: in the RTL specification one would need to speciy an explicit equation implementing the OR gate.

Here is a slighly modified example with registered outputs:

```
module Ex1R :
  output {RX, RY, RZ} : reg;
  emit next RX;
  pause;  // p1
  emit next RY;
  pause;  // p2
  emit next {RX, RZ}
end module
```

For registered signals, one must use the "`emit next`" form. Signal `RX` is asserted with a one-cycle delay, i.e., in the cycle that follows the cycle at which "`emit next RX`" is executed[3]. When running `Ex1R`, `X` is asserted at cycle 1 and 3, `Y` at cycle 2, and `Z` at cycle 3, as shown in Figure 2.3. The corresponding (sequentially unoptimized) circuit is shown in Figure 2.4. The `Ex1R` design is a Moore machine: the outputs of the module do not

[2] By setting the appropriate compiler option, it is possible to select the negative edges as tick boundaries in the implementation.

[3] The `next` keyword is redundant since the signal is declared `reg`. However, ESTEREL requires an explicit use of "`next`" to enhance readability and to stress the fact that the status will be set one cycle later.

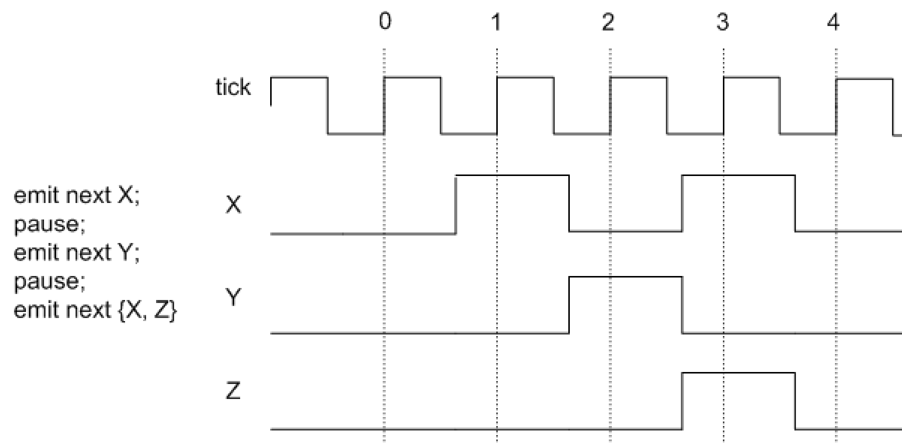

Figure 2.3: The behavior of Ex1R

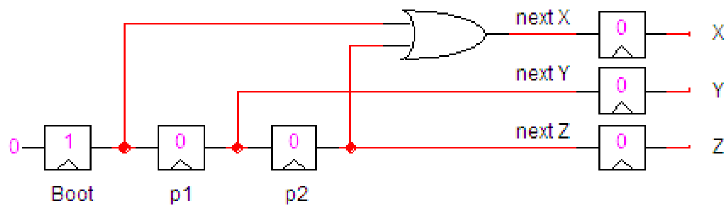

Figure 2.4: The circuit of Ex1R (before sequential optimization)

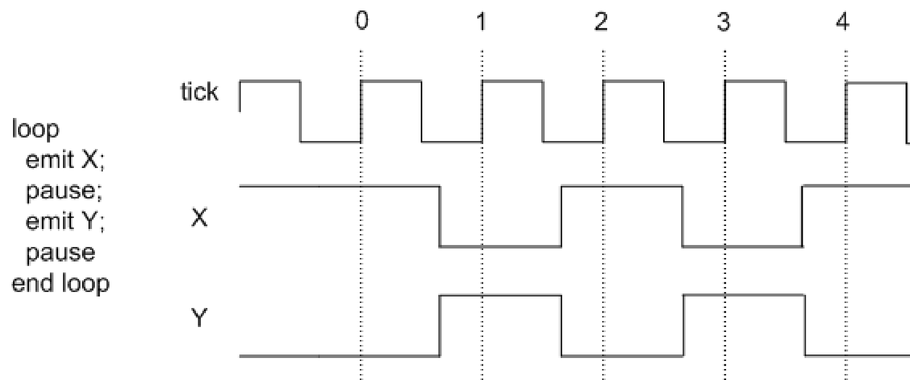

Figure 2.5: The behavior of Ex2

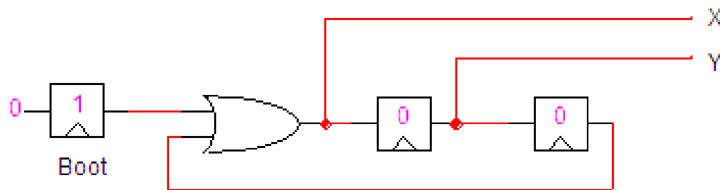

Figure 2.6: The circuit of Ex2 (before sequential optimization)

combinationally depend on inputs, unlike `Ex1`, which is a Mealy machine. Immediate and registered signals can be arbitrarily mixed in designs.

2.4 Looping

The `loop` statement repeats its body until this looping behavior is preempted from outside the loop or exited from inside the loop. At the end of the loop statement, control immediately returns to the beginning of the loop. Therefore, the looping operation is combinational and consumes no cycle by itself. Here is how to alternate between asserting two signals `X` and `Y`:

```
module Ex2 :
  output X, Y;
  loop
    emit X;
    pause;
    emit Y;
    pause
  end loop
end module
```

The `X` output is emitted at cycle 0 and at every other even cycle, while the `Y` signal is emitted at cycle 1 and at every other odd cycle. The behavior and the circuit of `Ex2` are shown in Figure 2.5 and Figure 2.6. Sequential optimization will merge the first (boot) register and the second pause register.

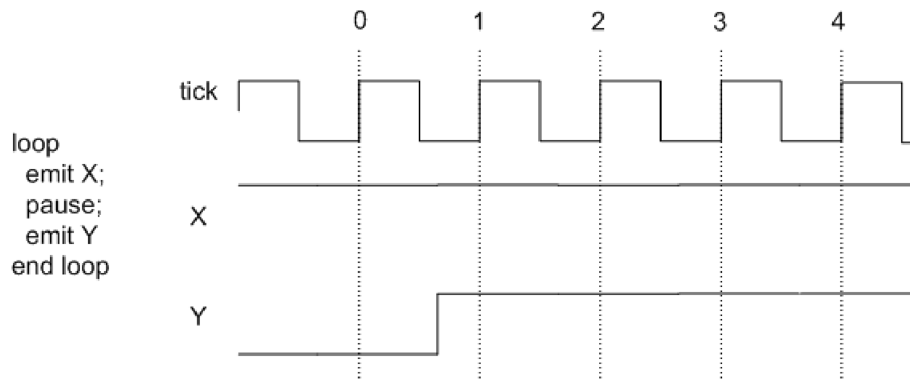

Figure 2.7: The behavior of Ex3

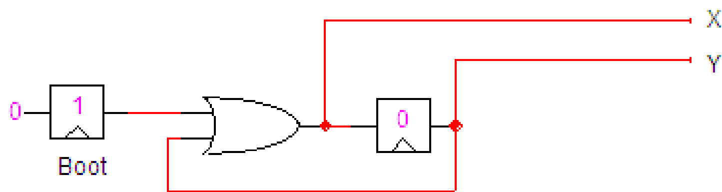

Figure 2.8: The circuit of Ex3

Notice that the second **pause** is essential to generate the X / Y alternation behavior. Consider the same program with the last **pause** removed:

```
module Ex3 :
  output X, Y;
  loop
    emit X;
    pause;
    emit Y
  end loop
end module
```

The behavior is quite different: X is emitted at all cycles and Y is emitted at all cycles but the first one. This is made clear by the following behavior-preserving loop unrolling:

```
emit X;
pause;
emit Y;
emit X; // loop
pause;
emit Y;
emit X; // loop
pause;
emit Y;
...
```

The behavior and circuit of Ex3 are pictured in Figure 2.7 and Figure 2.8.

The behavior of a `loop` statement is infinite by default. To terminate a loop, one can use the `trap-exit` construct described next in Section 2.11 or an enclosing abortion statement described in Section 3.2.

A different looping construct can be specified by the `repeat` statement that repeats a loop body a finite number of times:

```
repeat 5 times
  pause;
  emit O
end repeat
```

One can give a name to the index, which starts from 0:

```
repeat i < 5 times
  pause;
  emit O
end repeat
```

The index can be used inside the body (not shown in the example). The index can also be specified to span a range, see Section **??**.

Since the "`loop`" statement is immediate, the body of the loop should contain at least one sequential statement for every execution path. The compiler rejects programs with combinational loop bodies.

2.5 The sustain statement

The `sustain` statement "`sustain S`" or "`sustain {X,Y}`" keeps a signal or a list of signals asserted forever and never terminates, unless preempted, i.e., externally terminated by an enclosing or parallel statement, as described in Section 3.2. It is equivalent to a loop over an emission followed by a pause:

```
loop
  emit S;
  pause
end loop
```

2.6 Signal emission equations

It is often convenient to use a an equational version of the "`emit`" and "`sustain`" statements. Within `emit` and `sustain`, a signal can be conditionally emitted by defining an emission condition after the '`<=`' symbol.

Here is the way to permanently compute the conjunction and disjunction of two input signals `A` and `B`, with the disjunction registered:

```
module Ex4 :
  input A, B;
  output X, Y : reg;
  sustain {
    X <= A and B,
    next Y <= A or B
  }
end module
```

Equations within the same `emit` or `sustain` are executed concurrently and in no particular order, provided dependencies are preserved. A computed signals can be tested in another equation, as in the following example:

```
sustain {
  X <= A and B,
  Y <= X or C
}
```

The status of `X` referred to in `Y`'s equation right-hand-side is the one computed by the first equation. Scheduling is implicit from dependencies, not from equation ordering. It is equivalent to write the same equations in the reverse oder:

```
sustain {
  Y <= X or C,
  X <= A and B
}
```

Equations can also involve cases in their right-hand side, as in the following example:

```
sustain {
  X <= B if A
     | C and E if B
     | D and E
}
```

The cases are taken in order, and the first satisfied test determines the appropriate right-hand-side. The optional last case without an `if` is the default case.

Another way to embed case handling is by using `if-the-else` or `if-case` switches. Here is an example:

```
sustain {
  if S then
    X <= A and B,
    Y <= C and D
  else
    Z <= A or C
  end if
}
```

See [5] for more details.

2.7 Signal expressions

Signal expressions appearing on the right-hand-side of equations are denoted exp, exp_1, etc. They can be constructed from the following elementary expressions: a reference to a immediate or registered signal `S`, which denotes its current status; the `pre(S)` and `pre1(S)` delay operators for an immediate signal `S`, which return the status of a signal at the *previous* cycle with initial value 0 for `pre` and 1 for `pre1`[4]; the `next(R)` operator for a registered signal `R`, which returns true if an "`emit next R`" is executed in the current cycle.

[4] The `pre` operator is borrowed from synchronous language Lustre [6].

The logical operators include **and**, **or**, **not**, **xor**, **mux**, implication '=>', equivalence '<=>', and incompatibility '#', which is a n-ary operator with e_1#e_2#...#e_n true if at most one e_i is true.

The following equation specifies a rising edge detector for signal S, with initial value 0:

```
sustain RisingEdge <= S and not pre(S)
```

This exemplifies the fact that ESTEREL expressions can be sequential, not just combinational. Any access to `pre(S)` or `pre1(S)` for an immediate signal involves a one-cycle delay relatively to signal emission. Given a sequential equation the ESTEREL compiler will automatically generate an appropriate register, unlike for traditional HDL specifications where registers are manually allocated.

The `pre` operators can be applied to combinational expressions, see Section **??**. However, `pre`'s cannot be directly nested. To compute `pre` of `pre`, one needs to use an intermediate signal or better a delay line based on a signal array, as will be shown in Section 4.3.

Equations with immediate cyclic dependencies are rejected by the semantics and compiler as illegal combinational loops. Typical examples of rejected loops are "`emit X<=X`", "`emit X <= not X`", and "`emit {X<=Y, Y<=X}`". Here is an example of a rejected loop with two inverters:

```
sustain {
  Y <= not X,
  X <= not Y
}
```

See [2] and Chapter **??** for details on combinational dependencies and cycles[5].

2.8 Conditional branching

Conditional branching is performed by the `if-then-else` and `if-case` statements. These statements test Boolean signal expressions and branch to the corresponding statement. Here is an example:

```
module Ex5 :
  input A;
  output X, Y;
  loop
    if A then
      emit X; pause; emit X
    else
      emit Y
    end if;
    pause; emit Z
  end loop
end module
```

At first cycle, `A` is evaluated and the behavior branches as follows:

[5]The previous version ESTEREL v5 performed a deep semantical analysis of combinational loops to accept electrically stable ones. This analysis has not be retained for ESTEREL v7, because it does not scale up well enough

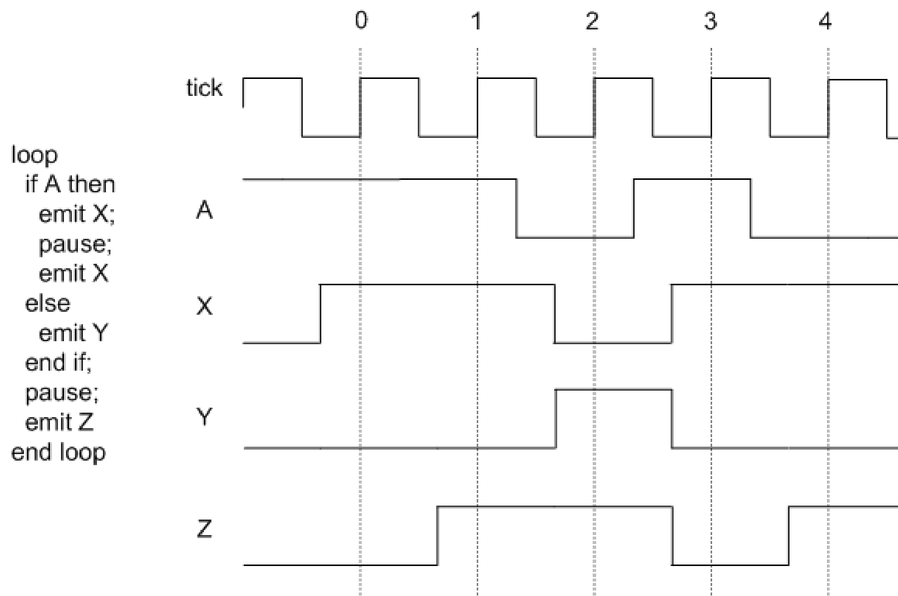

Figure 2.9: The behavior of Ex5

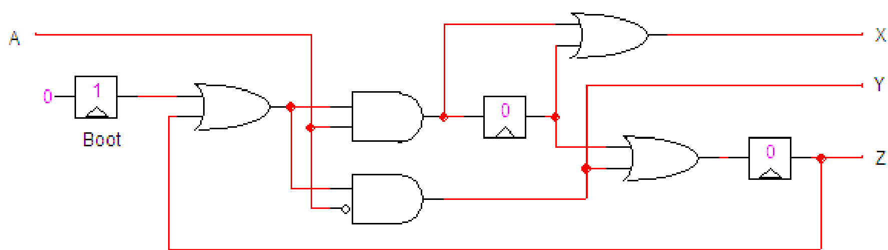

Figure 2.10: The circuit of Ex5

- if A is asserted, the `if` statement takes the `then` branch, which emits X in the cycle, pauses for one cycle, emits X again in the next cycle, and terminates;

- if A is deasserted, the `if` statement takes its `else` branch, which emit Y and immediately terminates.

Termination of the active branch provokes execution of the last `pause` statement, which make control pause for one cycle before emitting Z and looping back to the `if` statement.

The behavior is pictured in Figure 2.9 and the circuit in Figure 2.10. Notice that the test of an `if` statement is evaluated only when the `if` statement gets the control. While executing the branches, the test is not re-evaluated. Therefore, in the timing diagram of Figure 2.9, the status of A at cycles 1 and 4 is irrelevant, since it is not tested.

Multiple branching is possible using the `if-case` statement:

```
if
  case exp1 do
    stat1
  case exp2 do
    stat2
  [ default ] do
    statd
end if
```

The expressions are tested in order and the statement corresponding to the first true expression is taken. If there is a default, it is taken if no test expression is true. Otherwise, the `if-case` statement immediately terminates.

The `if` statement should not be confused with the use of `if` within `emit` and `sustain`: the former performs sequential control branching, while the latter is purely combinational.

2.9 Parallel execution

The parallel operator '`||`' places two ore more statements in parallel. Such parallel statements are each executed concurrently in each cycle, with input signals broadcasted to all parallel branches and output signals gathered from all parallel branches. A parallel statement immediately terminates when all its parallel branches have terminated. Sequencing '`;`' binds tighter than parallel '`||`', and blocks can be delimited by "`{}`" curly brackets. Here is an example:

```
module Ex6 :
  output X, Y, Z, T, U;
  {
    emit X; pause; emit Y
  ||
    pause; emit Z; pause; emit T
  };
  emit U
end module
```

Here, X is emitted at cycle 0, Y and Z are emitted at cycle 1, and T is emitted at cycle 2. The whole parallel statement terminates and emits U at cycle 2. Notice that the first parallel branch lasted 2 cycles while the second branch lasted 3 cycles. The behavior is shown in Figure 2.11 with the corresponding (semi-optimized) circuit in Figure 2.12. In

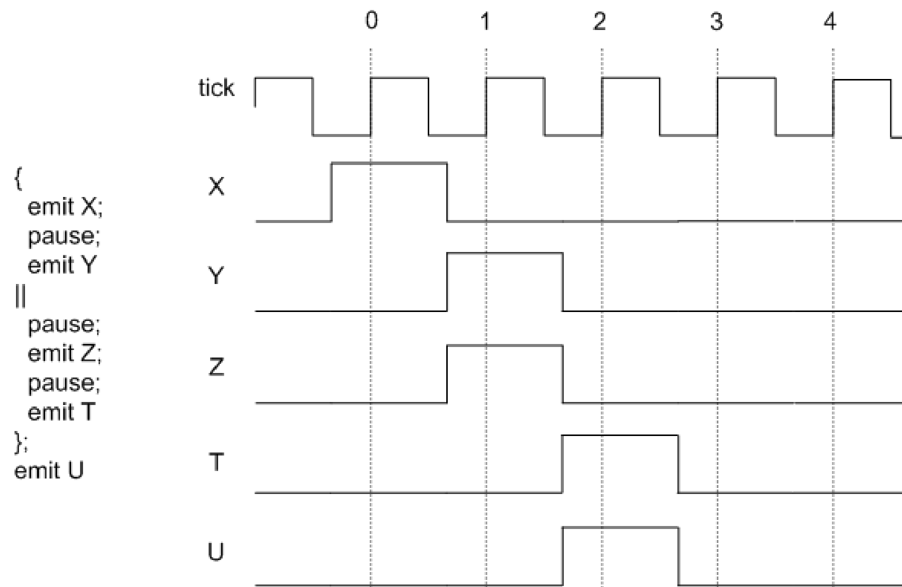

Figure 2.11: The behavior of Ex6

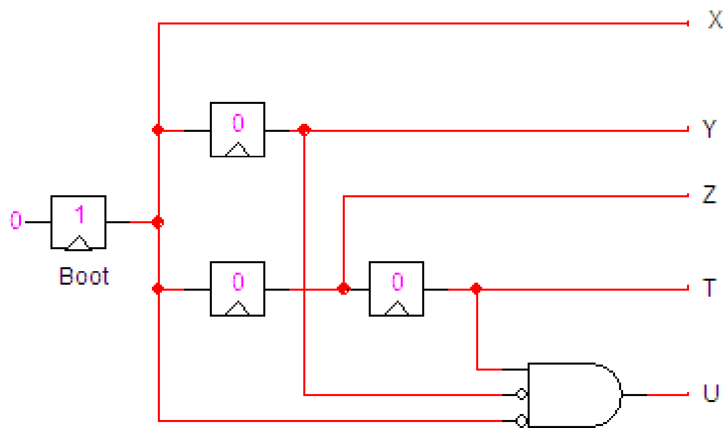

Figure 2.12: The circuit of Ex6

general, the circuit generated by a parallel statement may involve a non-trivial termination synchronizer that gathers termination of the branches, see Section 3.5, Chapter **??**, and [2]. The bottom and-gate of the circuit pictured in Figure 2.12 represents a partially optimized version of this synchronizer for this simple case. Termination occurs at cycle 2 cycle, with the first branch already terminated and the second branch terminating. Sequential optimization will wipe out this synchronization gate in this simple example.

There is no limit to the nesting and intertwining of parallel and sequencing, which makes it possible to construct arbitrarily complex sequential behaviors from their sub-behaviors.

2.10 Local signals and communication

Parallel statements can communicate using local signals, scoped by the local signal declaration statement:

```
signal S, T,
       R : reg in
  stat
end signal
```

Here, *stats* is an arbitrary statement called the body of the signal declaration. It is the scope of `S`. Scoping is static: a new declaration of a signal hides a previous declaration of a signal with the same name. Here is an example:

```
module Ex7 :
  input A, B;
  output X, Y;
  signal S in
     sustain S <= A and not pre(A)
  ||
     emit X;
     pause;
     emit Y <= S and B
  end signal
end module
```

The first branch emits `S` whenever `A` has a rising edge. The second branch emits `X` at first instant and `Y` at second instant if `A` has a rising edge and `B` is asserted. See Figure 2.13 and Figure 2.14 for the behavior and circuit. Note that the order of branches is irrelevant. As was discussed before for the case of multiple equation inside `emit` or `sustain` statement, parallel branches in-cycle scheduling is solely computed from signal dependencies induced by emissions and tests. Changing the order of parallel branches preserves the behavior.

With registered signals, there is of course a unit delay between emission and reception. Consider the following example:

```
module Ex8 :
  output XR : reg;
  signal LR : reg in
    emit next LR
  ||
    sustain next XR <= LR
  end signal
end module
```

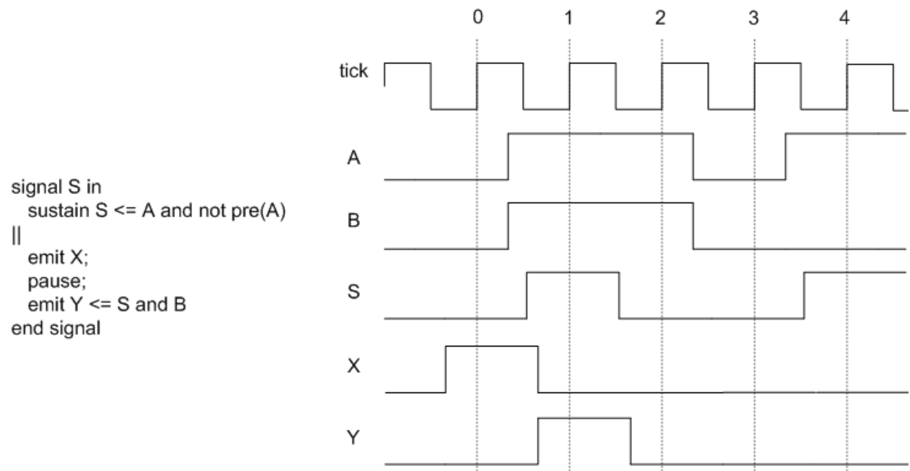

Figure 2.13: The behavior of Ex7

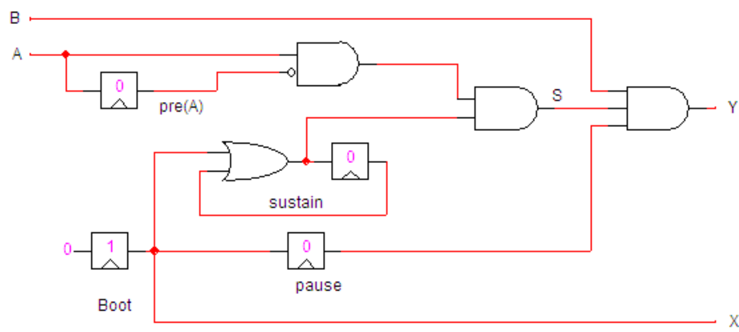

Figure 2.14: The circuit of Ex7

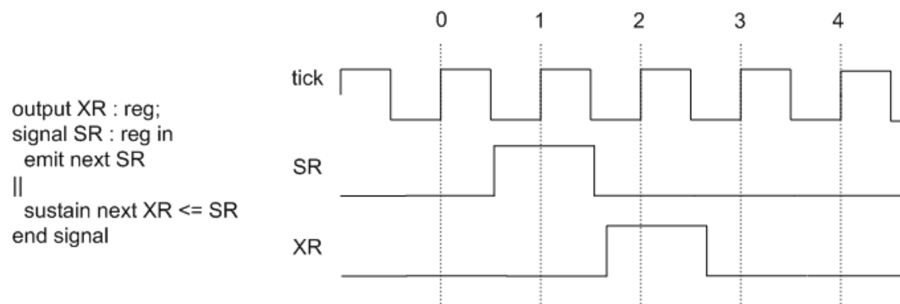

Figure 2.15: The behavior of Ex8

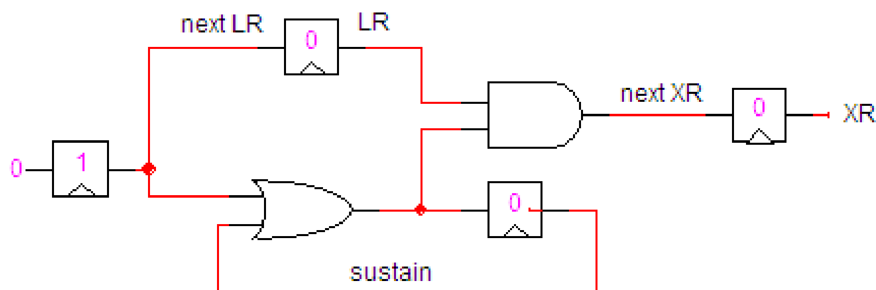

Figure 2.16: The circuit of Ex8

See Figure 2.15 and Figure 2.16 for the behavior and circuit of `Ex8`. Here, the local signal `LR` is emitted at cycle 0, thus asserted at cycle 1. Therefore, "`emit next XR`" is executed at cycle 1, which asserts `XR` at cycle 2.

The local signal declaration statement can be used wherever other statements can, which makes it possible to nest signal scopes at will.

2.11 Trap and exit

To continue or exit loops, C provides the user with `continue` and `break` constructs. These constructs are dangerous since unnamed, and some famous bugs are due to this.

GB: yet to be written.

Chapter 3

Temporal control

This chapter presents the statements specialized in temporal control handling, which is the main novelty of Esterel w.r.t. conventional languages. Two examples, `ABRO` and `Runner`, illustrate the power of these constructs. We discuss how Esterel makes it possible to obey a fundamental sanity principle we call *Write Things Once* when designing control-intensive applications.

3.1 The await statement

The `await` statement waits for am expression to become true and terminates. For instance, consider the statement

```
await S
```

When started, the statement waits for the next cycle where `S` is asserted, also called the next *occurrence* of `S`, and terminates. Notice that "`await tick`" is the same as `pause` since `tick` is always asserted.

Since it waits for the next occurrence of the condition, the `await` statement ignores the condition at starting instant. The added `immediate` keyword makes the `await` statement sensitive to the condition at first instant:

```
await immediate S
```

This statement immediately terminates if `S` is asserted at starting instant. Circuits for `await` and "`await immediate`" are pictured in Figure 3.1.

To wait for several occurrences of an expression, one can add an occurrence count:

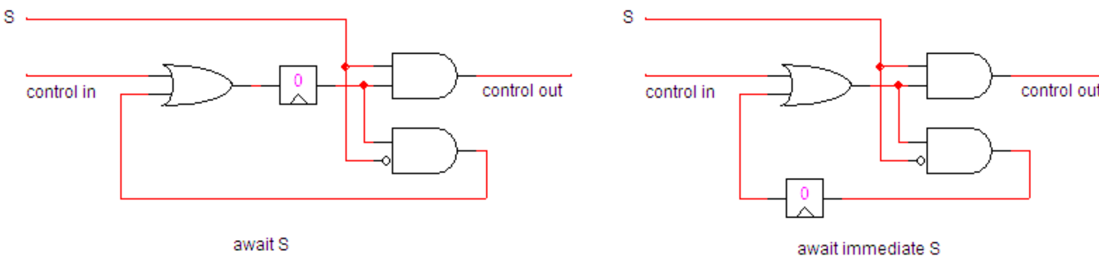

Figure 3.1: Circuits for await and await immediate

```
await 5 S;
await 10 times (S or T)
```

The `times` keyword is necessary when either the count or the signal expression is not trivial. The `immediate` keyword is incompatible with delay counters.

In the sequel, we call *delay* an occurrence designator of one of the forms *exp*, "*count* `times` *exp*", or "`immediate` *exp*". The first form is called a *simple delay*, the second form is called a *count delay*, and the third form is called an *immediate delay*. We use the symbol *dl* to denote a delay.

The `do` keyword can be added to start a statement when an `await` terminates:

```
await S do
  emit X;
  pause;
  emit Y
end do
```

The continuation statement between `do` and `end` is started when the delay elapses. It can be arbitrary. Finally, several delays can appear in an ordered case list.

```
await
  case dl1 do
    stat1
  case dl2 do
    stat2
  case dl3 do
    stat3
end await
```

Immediate delays are allowed in each case, and only them are tested when the `await` statement starts. There is no explicit `default` case for `await`, but "`case tick`" used in the last case has the same effect as `default` since the implicit `tick` clock event is always true.

3.2 Abortion statements

An abortion statement limit the timespan of their body, killing its behavior if a delay elapses. Let us start with the simplest of them, the `abort` statement:

```
abort stat when dl
```

where *stat* is an arbitrary statement and *dl* is an arbitrary delay. The `abort` statement starts is body *stat* and lets it execute at all subsequent cycles until one of two events occurs:

- the delay *dl* elapses; then *stat* is killed without being executed in the cycle, and the `abort` statement immediately terminates;
- the body *stat* terminates strictly before the delay elapses; then, the `abort` statement immediately terminates.

Notice the difference with `if`: the test is re-evaluated at each cycle while the body is alive for `abort`, while it is evaluated only when entering the statement for `if`.

Since the body is brutally preempted with no last chance for execution at abortion time, the default `abort` statement is called *strong*. The weak abortion form

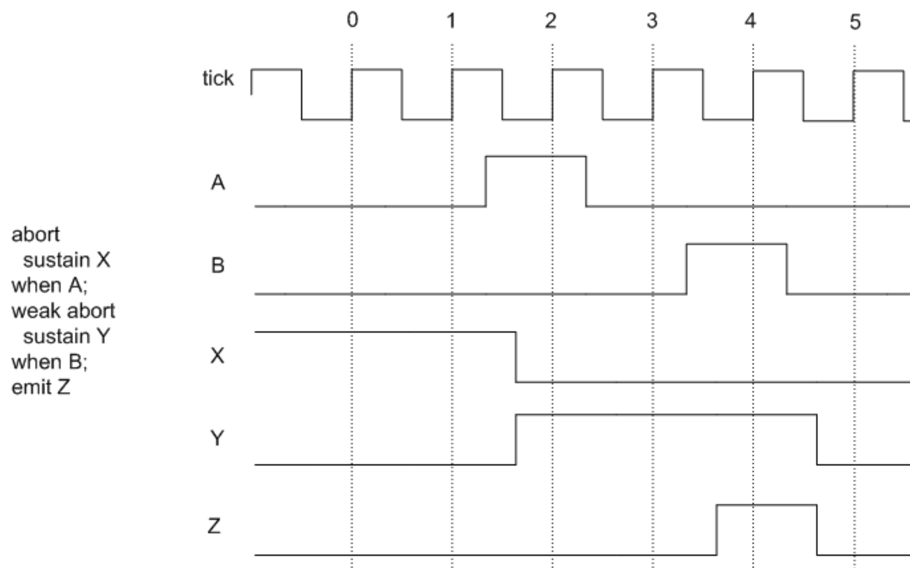

Figure 3.2: The behavior of Ex9 (abort and weak abort)

weak abort *stat* **when** *dl*

has the same behavior, except that *stat* is executed for a last time in the cycle where *dl* occurs. Here is an example involving the two forms of abortion:

```
module Ex9 :
  input A, B;
  output X, Y, Z;
  abort
    sustain X
  when A;
  weak abort
    sustain Y
  when B;
  emit Z
end module
```

Here, `X` is sustained until the first subsequent instant where `A` is asserted, this instant not included. Then, `Y` is sustained until the first subsequent instant where `B` is asserted, this instant included, and `Z` is emitted at the same instant. A timing diagram is pictured in Figure 3.2 and the generated circuit is pictured in Figure 3.3. Notice that occurrences of `B` are ignored when waiting for `A` and conversely.

To also watch for the condition at starting time, one adds the `immediate` keyword as for `await`:

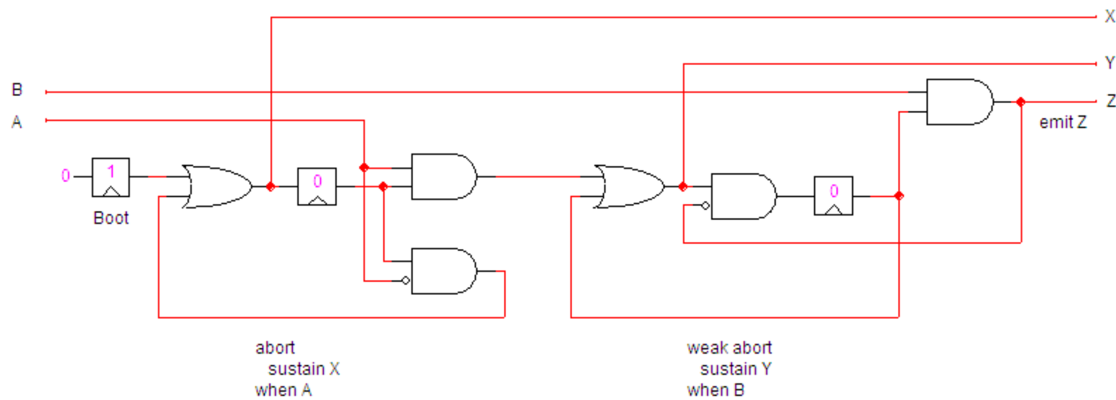

Figure 3.3: The circuit of Ex9 (abort and weak abort)

```
module Ex10 :
  input A, B;
  output X, Y, Z;
  abort
    sustain X
  when immediate A;
  weak abort
    sustain Y
  when immediate B;
  emit Z
end module
```

Here, if `A` and `B` occur at first cycle, the whole sequence terminates with `Y` and `Z` emitted. If `A` is not asserted at first cycle but `B` is asserted at first cycle where `A` is asserted, then the sequence immediately terminates with `Y` and `Z` emitted. In any other case, the `immediate` keyword has no effect and the behavior is as for strong `abort`. The circuit is a simple variant of the one pictured in Figure 3.3.

Cases can be added to abortion statements as for `await`:

```
[ weak ] abort
  stat
when
  case dl1 do
    stat1
  case dl2 do
    stat2
  case dl3 do
    stat3
end abort
```

Notice that `await` can be easily defined from `abort`, using the `halt` statement that never terminates and is an abbreviation for “`loop pause end`”:

```
await S = abort halt when S
```

3.3 Temporal loops

To enhance readability, temporal loops combine the `abort` statements and the `loop` statement into a single compound statement. The simplest temporal loop is `loop-each`:

```
loop
  stat
each dl
```

It is an abbreviation for

```
loop
  abort
    stat; halt
  when dl
end loop
```

The body *stat* is restarted afresh when *dl* is true, and aborted if not yet terminated. Because of the added `halt`, one waits for *dl* to restart *stat* if *stat* terminates before the delay elapses.

The other temporal loop uses the `every` keyword and differs from `loop-each` by the fact that one initially waits for *dl* to start *stat*. The statement

```
every [ immediate ] dl do
  stat
end every
```

is a shorthand for

```
await [ immediate ] dl;
loop
  stat
each dl
```

As usual, the immediate form tests for the condition at starting time.

3.4 Suspension and clock gating

Suspension is a milder form of preemption that suspends action of the body for the cycle without killing the body. As abortion, it comes into two forms, weak and strong, and each form can be delayed or immediate. The immediate weak form is important since it has exactly the effect of local clock gating, which is essential to save power in circuits.

The strong form is as follows:

```
suspend
  stat
when [ immediate ] exp
```

The body *stat* is only executed when the condition *exp* is false. In the default delayed form, the expression is tested from the cycle that follows the starting cycle on. In the immediate form, the expression is tested in the starting instant as well. The whole statement terminates if *stat* terminates when executed. For instance, consider the following modul:

```
module Ex11 :
  input A;
  output X;
  suspend
    sustain X
  when A
end module
```

the output X is emitted at first instant and at all subsequent instants where A is absent. The body can be any sequential statement, which acts only when the suspension signal is absent. Termination of the body when not suspended provokes termination of the whole `suspend` statement.

Notice that *exp* is an expression, not a general delay as for `abort`. Suspension is an instantaneous concern and "`suspend` *stat* `when 3 S`" is forbidden.

Weak suspension differs in the following way: when the expression is true, the combinational actions are performed but there is no sequential state change. Consider the following example:

```
module Ex12 :
  input A, B, C;
  input G;    // clock gater
  output X, Y, Z;
  weak suspend
    emit X;
    await A;
    emit Y <= not B;
    await C;
    emit Z
  when G
end module
```

A behavior ic pictured in Figure 3.4. Because of weak suspension by "G", state change occurs only when G is deasserted, while emissions and combinational control propagation take place normally. Thus, X is emitted when the program is started, and `await A` becomes active. From then on, Y becomes the negation of B whenever A is asserted, and "`await A`" stays active until G is deasserted again, which sets "`await C`" active. From then on, Z stays asserted whenever C is asserted and "`await C`" stays active until G is deasserted with C asserted, which makes the module terminate.

In the circuit translation, weak suspension can be implemented in two equivalent ways that both act on all registers generated by the body: by disabling the register inputs using muxes triggered by G; or by gating the register clock inputs by G. See Figure 3.5 for a clock-gated implementation.

Nesting weak suspension statements makes it possible to manage clock gating in a hierarchical and semantically well-defined way. Immediate weak suspension is also used to formally model the behavior of multiclock designs, see Chapter ?? and [5] for full details.

3.5 The ABRO example

Let us give a more elaborate example using the statements defined so far.

Specification ABRO:
Emit an output O *as soon as two inputs* A *and* B *have occurred. Reset this*

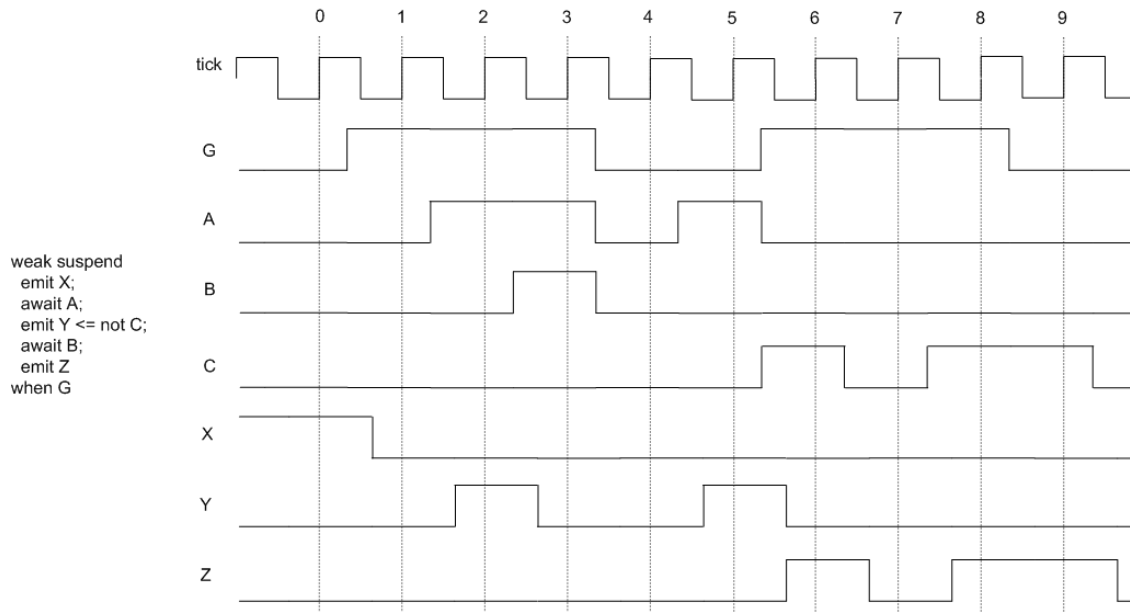

Figure 3.4: Behavior of Ex12

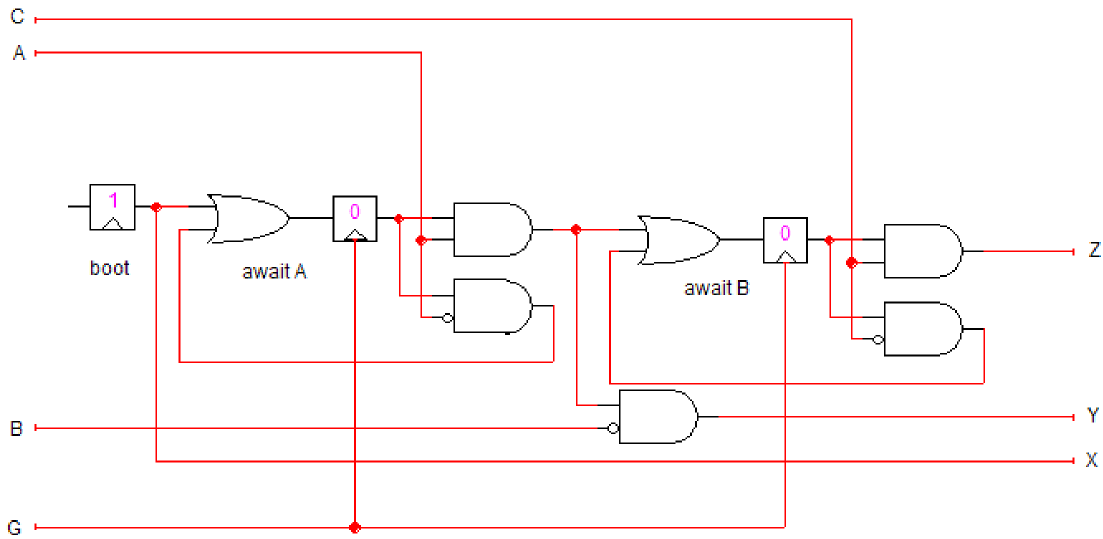

Figure 3.5: Circuits for Ex12; lines from G to register clock inputs indicate clock gating

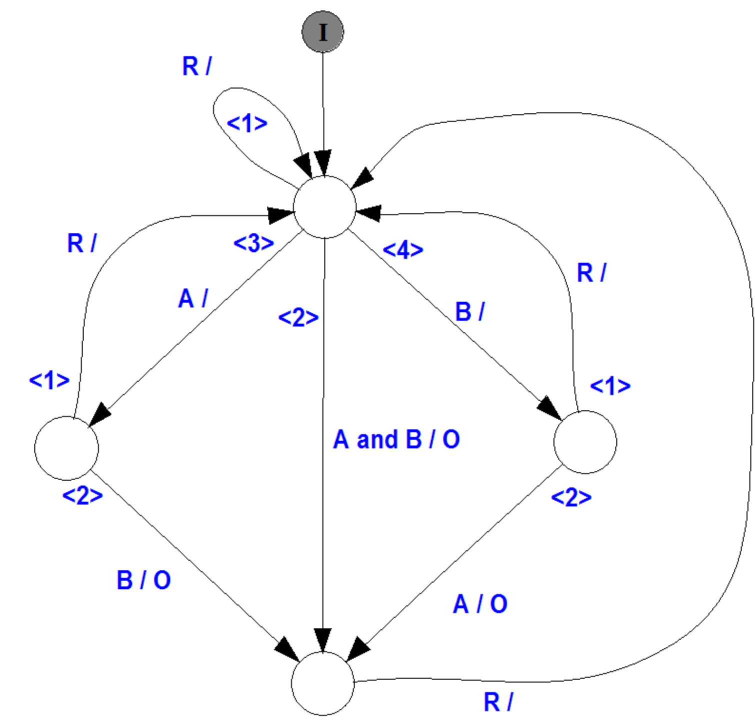

Figure 3.6: The ABRO Mealy machine

behavior each time the input `R` *occurs, wihout emitting* `O` *in the reset cycle.*

This simple specification actually represents a very common design pattern. A usage example is writing to memory, where `A` is address availability, `B` is data availability, `O` is the write command, and `R` is a reset command.

A common way of programming `ABRO` is to design a deterministic *Mealy machine* in state graph form, which is a deterministic finite automaton in which each transition arrow bears an input / output transition label. An Esterel Studio Mealy machine for `ABRO` is pictured in Figure 3.6. Transition labels have the form "E `/`" or "E `/ O`", where E is a possibly empty expression on inputs that triggers the transition and `O` is the optional transition output. For instance, the transition labeled "`A and B / O`" can be taken if `A` and `B` are both asserted, and, if taken, it provokes combinational emission of `O`. The numbers between angle brackets, such as `<1>`, are transition priorities that ensure determinism. Here, from the topmost state, the `R` transition has priority over the "`A and B / O`" transition, which itself has priority over the "`A /`" and "`B /`" transitions. This formal definition is of course a little heavier to draw than an approximate one as often found in documentations, but it is fully precise. Mealy machines can be easily encoded in Verilog or VHDL.

We now show that we can do a more readable, more scalable, and more efficient design in Esterel. Look at the automaton in Figure 3.6. Each signal appears several times, unlike in the original specification. For example, `A` appears 3 times: once on the left, as the first input, once on the right, as the second input, and once in the middle, in "`A and B`". If priorities were not used to disambiguate transition choice, `A` should also appear negatively on the first input transition on the right. The reset signal `R` appears 4 times, once per state; without priorities, it should also appear negatively on all other transitions. Finally,

the output O also appears 3 times, one for each possible sequencing of A and B. Consider now the problem ABCRO, where there is one more signal C to wait for before emitting O. We leave the automaton drawing to the reader: the automaton core now has the shape of a 3-D cube with 8 vertices, and much more signal replication. Of course, the n-signal problem yields a n-cube with 2^n vertices. The design is exponential in the size of the specification, which is not acceptable. Furthermore, automata are most often hard to draw and read if not ridiculously small, and they are very sensitive to specification changes. Thus, in practice, explicit Mealy machines are rarely good designs.

In contrast, the Esterel code for ABRO is as follows:

```
module ABRO:
  input A, B, R;
  output O;
  loop
     await A || await B ;
    emit O
  each R
end module
```

The two concurrent await statements respectively wait for A and B and terminate. The parallel operator that joins them terminates when both are terminated, i.e., in the cycle where the last of A and B is received. Since sequencing consumes no cycle, the output O is emitted in the same cycle, as requested by the specification. The "loop-each R" construct performs the reset in the right way, with priority of R over A and B since the body is not executed when R occurs. In the Esterel code, each signal appears *exactly once*, as in the specification and unlike in the Mealy machine. Furthermore, the code grows linearly with the size of the specification. With three signals S, B, and C, it simply becomes

```
loop
    await A || await B || await C ;
   emit O
each R
```

A similarly linear graphical version of ABRO is pictured in Figure 3.7. The formalism used is called *Synchronous state machines* or *SSM*. It is an evolution of André s SyncCharts [1], itself a synchronous version of Harel's Statecharts [7]. In a SSM, each state can be hierarchically refined into another SSM or a concurrent product of SSMs. A double circle indicates a terminal state, a state without exiting arrow is akin to halt, a dotted line specifies concurrency, a transition starting with a triangle specifies sequencing, and a transition starting with a circle specifies strong preemption. Therefore, in this example, the graphical syntax is simply a variant of the textual one.

The circuit generated by ABRO is pictured in Figure 3.8. It is also linear in terms of the number of inputs tested, unlike a circuit generated from an explicit Mealy machine which is of exponential size. Linearity also implie that each input is tested only once, instead of being tested by many individual transitions. The synchronizer subcircuit on the right-hand side performs synchronization betweem the parallel await arms: the parallel terminates if both arms terminate simultaneously or if one of them is already terminated when the other one terminates. See [2] for the exact structure and behavior of the synchronizer.

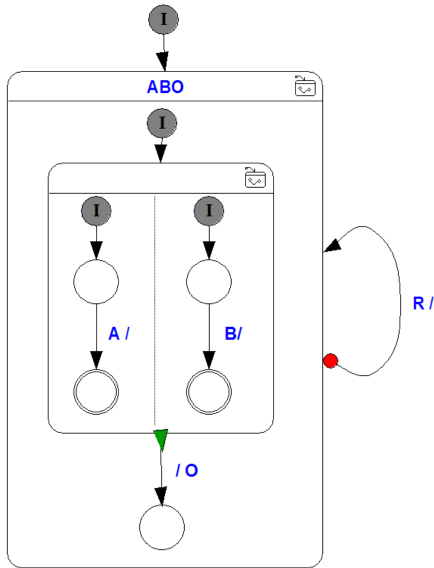

Figure 3.7: The ABRO SSM

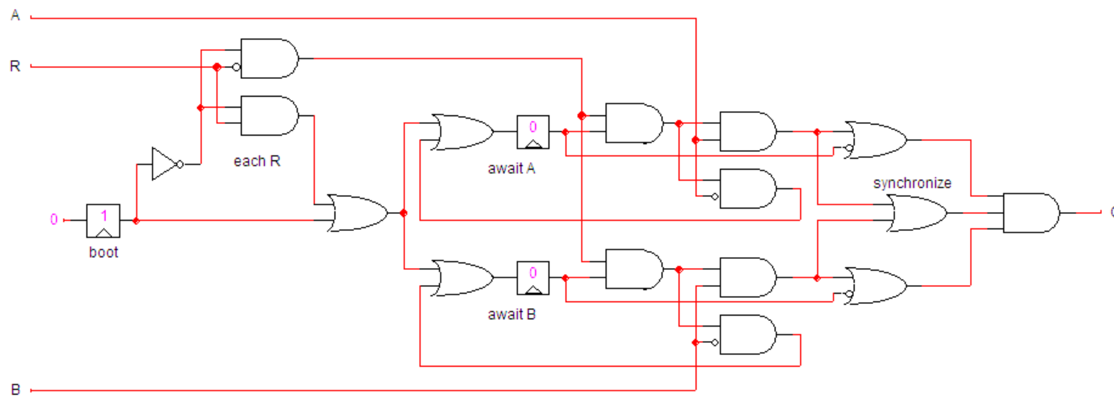

Figure 3.8: The ABRO generated circuit

3.6 The Write Things Once principle

The `ABRO` example illustrates the fundamental difference between Mealy machines and Esterel programs. Esterel statements and SSM constructs make it possible to replace *code replication* by *behavioral description* using concurrency, sequencing, and temporal primitives. Replacing code replication by behavioral structure is indeed the essence of language design: loops, functions, concurrency, objects, etc., all serve this purpose.

The real key to good programming is the *Write Things Once* or *WTO* principle: any part of the specification whould appear exactly once in the code. Any code replication makes the program harder to understand and to maintain, and yields a common source of bugs: the possibility of modifying some of the copies without modifying the other ones. We do not claim that Esterel fully achieves WTO. We only claim that the Esterel and SSM primitives help finding the real structure of reactive applications, which is the prerequisite to WTO, and that the appropriate primitives are absent from conventional C-like programming languages and from HDLs.

For `ABRO`, each construct contributes in its own way to WTO. Concurrency immediately saves an exponential. Synchronous sequencing is fundamental for using a single occurrence of `O` for all the termination cases of the parallel statement. The “`loop...each`” abortion statement makes it possible to preempt the body in any state using a single occurrence of `R`. All these constructs are *orthogonal*. This means that they can be freely mixed at any nesting depth without restriction. Here, the parallel statement appears within a sequence that itself appears within an abortion. Many languages limit concurrency to design toplevel, therefore loosing orthogonality. This is a sure way of not achieving WTO.

Another essential component of WTO is signal broadcasting, which is used for communication in Esterel. Input broadcasting was implicit when we said that concurrent statements evolve in lockstep in the same input environment. Broadcasting is extended to all output and local signals (and of course scoped for locals). For instance, assume that `ABRO` is a component of a wider system. Any process interested in knowing when `ABRO` emits `O` just wait for `O`. Such processes do not have to signal their identity to `ABRO`, the code of which does not depend on the number of receivers.

3.7 The runner example

Let us now give another example to illustrate temporal design, a runner controller. Inputs are supposed to come from appropriate sensors, and outputs are supposed to drive actuators. The runner should first run slowly during 100 meters, then jump and breathe synchronously at each step for 15 seconds, then run fast up to the end of the lap. The whole exercise is limited to 4 laps.One should check that the commands `RunSlowly`, `Jump`, and `RunFast` cannot be simultaneous.

```
module Runner :
  input Morning;
  input Second, Meter, Step, Lap;
  output RunSlowly, Jump, Breathe, RunFast;
  every Morning do
    abort // 4 laps
      loop
        abort
          sustain RunSlowly
        when 100 Meter;
        abort
          every Step do
            emit {Jump, Breathe}
          end every
        when 15 Second;
        sustain RunFast
      each Lap
    when 4 Lap
  end every
||
  sustain assert AtMostOneBehavior = RunSlowly # Jump # RunFast
end module
```

Note how direct the formal Esterel writing is: temporal behavior is written in a straightforward way from the specification, without resorting to explicit state encoding, and counter reset, decrementation, and test are handled automatically.

The `AtMostOneBehavior` assertion states that no two commands can be sent simultaneously, as requested by the specification. Such an assertion is defined by an equation in `emit` or `sustain` statement; it needs no other declaration. The assertion must be true whenever its `emit` oe `sustain` statement is active. Assertions are key to program correctness verification. They can be checked either by dynamic simulation or by static formal verification, see Chapter **??** for details.

All `Runner` sub-behaviors are not necessarily exercised. If some lap is shorter than 100 meters, then the runner will neither jump nor run fast in this lap (there is no assumption that all laps are of equal length). If some lap is longer than 100 meters but ends before the 15 seconds delay, the runner does not run fast in this lap. If the runner is real slow and a new morning occurs before the 4 laps are completed, the runner starts again without any rest.

Notice that nesting strong temporal statements automatically build priorities. Assume a lap occurs simultaneously with a step when the runner is in jump mode. Since `loop-each` is based on strong abortion, the internal “`every Step`” statement is strongly preempted by “`each Lap`”, which implies that `Jump` and `Breathe` are not emitted. Thus, `Lap` handling has priority over `Step` handling when in this mode.

Chapter 4

Data Basics: a FIR Filter Example

This chapter presents a first Esterel design, a video FIR filter similar to filters commonly found in video applications. After a short presentation of the required data-handling primitives, the design is done in an incremental way, starting from an initial simple specification and progressively augmenting the specification to build the final filter with its full computation, control, and pipelining aspects. This illustrates the flexibility of designing with Esterel.

During the design process, we put emphasis on program organization and readability. Nowadays, because of reuse needs, being able to easily read and modify specifications and programs is as important as being able to write them. This requires programs to be well architectured and well written, with the added benefit that well-written code is easier to debug and much less error-prone. However, the way to best write a program depends on its size. For instance, simple interface declarations are short and convenient for small modules, while hierarchically defined interfaces are much better for large modules. We show how to gradually transform lighweight "programming in the small" architectures into heavier but more manageable "programming in the large" architectures when needed.

Section 4.1 presents basic data in Esterel v7 and introduces arbitrary precision exact arithmetic. Section 4.2 presents the initial basic filter specification. Coding the basic filter, is done in Section 4.3. Section 4.4 presents the impact of three successive specification updates to the basic design. Section 4.5 introduces simple control by requiring pixels to be read from memory words. Section 4.6 introduces more elaborate hierarchical control by requiring the filter to handle lines of given length. Section **??** will later show how to make the design parametric to handle other pixel types, word types, and filter coefficients.

4.1 Data handling basics

We now turn to data handling, introducing the basic `unsigned` type and valued signals.

4.1.1 Exact unsigned arithmetic

Here data declarations we shall need for the video filter design presented in Chapter 4:

```
type Pixel = unsigned<[8]>;
constant Coef : unsigned<>[5] = {1, 4, 6, 4, 1};
```

The `Pixel` type is a synonym for 8-bit unsigned numbers, and the `Coef` constant array is the array of filter coefficients.

Let us explain what the `unsigned<[8]>` and `unsigned<>` declaration mean. Esterel v7 is very precise about arithmetic calculations. Unlike most software and harware description languages, it implements *arbitrary precision exact arithmetic* with *automatic data path sizing*. The basic arithmetic types are `unsigned<M>` and `signed<M>`, where M is an arbitrary positive integer. We will consider only unsigned numbers throughout this chapter, see Section **??** for signed numbers. The type `unsigned<M>` denotes the range $[0..M-1]$, which has exactly M elements. Addition is exact: if e_1 and e_2 are arbitrary expression of respective types `unsigned<M1>` and `unsigned<M2>` and of respective values v_1 and v_2, then the sum expression e_1+e_2 has value v_1+v_2 of type `unsigned<M1+M2-1>`. The expression type is determined by the fact that the biggest possible value of v_1+v_2 is $(M_1-1)+(M_2-1)=(M_1+M_2-1)-1$. This definition of numbers and operation departs from those used in C of HDLs, where numbers have given maximal bit with. In these languages, when overflow occurs, the value of e_1+e_2 is not v_1+v_2, which means that + is not mathematical addition. Esterel has no overflow, just as mathematics.

All signed and unsigned arithmetic operations are handled in the same way with exact results and types. Explicit saturation and truncation functions make it possible to saturate or truncate results when needed. As we will see later, arbitrary precision exact arithmetics makes data path sizing and formal verification much simpler.

Esterel v7 numbers are viewed as abstract and not necessarily implemented by binary bitvectors. In particular, the size parameter M needs not be a power of 2. For instance, `unsigned<23>` is the set of numbers from 0 to 22. However, the case where M is a power of 2 is of course very frequent, expecially for interface signals. If $M=2^N$, we use `unsigned<[N]>` as an abbreviation for `unsigned<M>` = `unsigned<2**N>`. In `FilterData`, pixels are of type `Pixel` = `unsigned<[8]>` = `unsigned<256>`[1].

For the `Coef` constant array above, we did not specify any size, because the Esterel type-checker will figure out the size by itself. Since the biggest value in the array is 6, the best possible type to cover all values is `unsigned<7>`. Writing this type explicitly is of course allowed, but being lazy is perfect.

Notice the advantage of dealing with exact ranges instead of bit witdh: if u has type `unsigned<6>` and v type `unsigned<5>`, both fit on 3 bits; their product has maximal value $5\times 4=20$ of type `unsigned<21>`, which fits on 5 bits. A bitwidth-only calculation would have given 3 bits $\times$ 3 bits = 6 bits.

4.1.2 Valued signals

Most designs are not limited to pure control but perform value computations. In Esterel v7, this is done through *valued signals*, which carry values of specified types. Valued signals can be automatically memorized and can obey a simple valid-bit protocol. A valued signal can be *value-only*, in which case it only carries a typed value, or *full*, in which case it also carries an implicit pure signal called its *status signal*. The operator '`?`' is used to refer to the value, called `?S` for a valued signal `S`. If `S` is full, its status is simply called `S` as for a pure signal. Valued signals are full by default. Value-only signals are declared using the `value` keyword.

By default, the value of a signal `S` is memorized between instants. This requires allocating memory elements in the hardware design, which may be too expensive. Memorization

[1] The notation is a little bit heavy, with a combination of '`<.>`' to recall exact arithmetic and '`[]`' to recall bitvectors, but it is non-ambiguous. Writing `unsigned[32]` would mean array of 32 unsigned, which is completely different.

can be suppressed by adding the `temp` keyword in the signal declaration. In that case, the value is undefined when the signal is not emitted, that is, when its status bit is deasserted.

A valued signal can also be declared registered using the `reg` keyword. Then, its value (and status for a full signal) is set one cycle after its emission, and the value is automatically memorized between cycles. Figure **??** presents the different possibilities. Here is a simple module with valued signals:

```
module Valued :
  type Pixel : unsigned<[8]>;
  input A : temp Pixel;
  output X : value Pixel;
  emit ?X <= 5;
  await 3 A;
  sustain ?X <= ?A if ?A > 10
}
```

Here, `A` is full temporary because of the `temp` declaration, while `X` is value-only memorized, because of the `value` declaration. The first `emit` statement emits `X` with value 5 at first cycle. Then, the `await` statement waits for 3 successive occurrences of `A`, with the value `?X` still 5 since `X` is memorized. Then, whenever `A` occurs with a value greater than 10, its value is re-emitted on `X`. In between, the value of `X` stays equal to its previous value.

Notice that the test in the `sustain` statement involves mixed status / value expressions. Notice also that we wrote two distinct emission statements for `X`. Because of the middle `await` statement that separates them in time, these emitters will never occur simultaneously and thus never conflict.

As for the pure signal case, `emit` and `sustain` can involve multiple equations computed concurrently. Furthermore, status and value equations .can be freely mixed in the same `emit` or `sustain`.

4.1.3 Booleans, arrays, and bitvectors

Esterel support data arrays of any number of dimensions. Arrays are like C arrays, with indices starting from 0. For instance, for a declaration "S : Pixel[3]", the value `?S` is an array of type `Pixel[3]` with three components, `?Pixel[0]`, `?Pixel[1]`, and `Pixel[2]`. One can take slices such as `[0..1]`. Arrays of any base type are allowed. Multidimensional arrays are allowed, but will not be used here; their type definition and indexation are as in C, for instance, a correct array type is `unsigned<[8]>[5][7]`, and indexation is `A[2][3]`; slicing can be done on one or several dimensions.

A very common kind of array is *bitvectors.* i.e., arrays of Booleans. The base type `bool` of `Word` is the Boolean type, with two values called `true` and `false` or `'0` and `'1`.

Here is a typical example:

```
type Word = bool[32];
```

The `type` declaration defines `Word` as a synonym to the array type `bool[32]`. Bitvector constants can be written in binary with higher-order bit first as usual, as for `'b10011`, in octal, as for `'o325`, or in hexadecimal, as for `'xFF4E`. They can also be written as Boolean arrays, with low-order bit first: `'b10011` and `{'1,'1,'0,'0,'1}` denote the same constant. Note that bitvectors are not special "packed array" type as in HDLs. Packing is done by the compiler.

Bitvectors support a map definition that gives names to slices. Here, we names the bytes in the word, which we will convert to pixels in the filter example:

The `map` declaration gives names to bits or fields of a bitvector type. Here, the map splits the word into four consecutive pixels named `p0`, `p1`, `p2`, and `p3`, lowest-order byte first as required by the specification. For an object `W : Word`, the expression `W.p0` is synonym to `W[0..7]`. Map fields can overlap, and several unnamed or named maps can be used for the same type, see Section ?? for more details.

```
    type Word = bool[32];
    map Word {
      p0[0..7], p1[8..15], p2[16..23], p3[24..31]
    };
  end data
```

4.1.4 Conversions between bitvectors and numbers

For the `Word` type and map, slices `p0` to `p3` are bitvectors of length 8. However, the the filter of Chapter 4 expects pixels as numbers of type `unsigned<[8]>`. Since it deals with numbers in a very precise and abstract way, Esterel does not identify bitvectors with numbers, and it does not even automatically convert bitvectors to numbers. One must use an explicit conversion function, here `bin2u` (binary to unsigned) to read the bits in binary format. Thus, the numerical pixels out of `p0` should be computed as `bin2u(p0)`, which is of type `unsigned<[8]>`. Other primitive functions are `bin2gray` to read the bits in Gray code format, `bin2onehot` to read the bits in onehot format, i.e. 0 = `'b1000`, 1 = `'b0100`, 2 = `'b0010`, and 3 = `'b0001` for 4-bits onehot encoding. Similar `bin2s`, etc., primitives make it possible to read the bitvector as signed instead of unsigned.

4.1.5 Interfacing valued signals at HDL level

In the HDL implementation gnerated by Esterel Studio, each main module pure interface signal `S` simply generates a Boolean HDL signal called S. Things are a little more involved for valued interface signals. Each Esterel data type is implemented c a corresponding HDL. A valued signal `S` of Esterel type `T` generates a HDL interface signal called `S_data`, with type HDL type the one associated with `T`. Esterel bitvectors become HDL bitvectors. Esterel unsigned numbers are assumed to be encoded binary, and signed numbers are assumed to be encoded in 2's complement form. In addition, if `S` is full, a Boolean signal `S` is generated for the status.

For instance, consider the following declarations:

```
type Pixel : unsigned<[8]>;
input InPixel : Pixel;
```

Then there are two HDL signals for `InPixel`: the 8-bit signal `InPixel_data` for the value, and the Boolean signal `InPixel` for the status.

4.2 Basic filter specification

4.2.1 Filtering algorithm

The filter should serially take an unbounded sequence of 8-bit pixels as input and serially return an unbounded sequence of 8-bit pixels as output. Let the input pixels be called

$x_0, x_1, \ldots, x_m, \ldots$ and the output pixels be called $y_0, y_1, \ldots, y_m, \ldots$ The recurrence filtering formula should be as follows, for $i \geq 2$:

$$y_i = (x_{i-2} + 4x_{i-1} + 6x_i + 4x_{i+1} + x_{i+2})/16$$

The filter should be started only when three input pixels have been read, with 0's for the missing values:

$$\begin{aligned} y_0 &= (6x_0 + 4x_1 + x_2)/16 \\ y_1 &= (4x_0 + 6x_1 + 4x_2 + x_3)/16 \end{aligned}$$

For instance, with input value sequence $x = 12, 23, 45, 127, 82, \ldots$, the output sequence should be $13, 30, 60, \ldots$.

4.2.2 Basic filter implementation constraints

Input pixels may not appear at every clock cycle. The filter circuit should have two inputs: an 8-bit signal `InPixel_data` carrying pixel values and a Boolean signal `InPixel` acting as a valid bit for `InPixel_data`. The value of `InPixel_data` should be meaningful only when `InPixel` is asserted (i.e., has value 1); it should be considered as irrelevant and potentially undefined if `InPixel` is deasserted. No pixel should be given by the environment at circuit reset time, i.e., before the first clock rising edge.

Symmetrically, the filter circuit should have two outputs: an 8-bit signal `OutPixel_data` carrying output pixel values and a Boolean signal `OutPixel` acting as a valid bit for `OutPixel_data`. The value of `OutPixel_data` should be defined when `OutPixel` is asserted and may be left undefined otherwise.

The output pixel should be computed combinationally for each new input pixel: for $i \geq 2$, the output of y_i should be synchronous with the input of x_{i+2}.

The final circuit should be single-clocked, with full synchronous reset, and fully DFT-compliant. A SystemC cycle-accurate model with behavior strictly idential to the circuit behavior should be made available.

4.3 The BasicFilter_1 module

It is obvious from the specification that the interface signals `InPixel` and `OutPixel` should be full temporary signals of type `Pixel`. Here is the corresponding module header:

```
module BasicFilter_1 :
  type Pixel = unsigned<[8]>
  input InPixel : temp Pixel;      // full temporary
  output OutPixel : temp Pixel;    // full temporary
  Body
end module
```

We use the name `BasicFilter_1` because we guess we will later receive specification updates, as usual.

We build the exectuable body by declaring a delay line signal array to hold 5 successive pixel values and by putting two behavioral components in parallel, the delay line handler, and the computation handler:

```
signal DelayLine : value Pixel[5] init 0 in
  DelayLineHandler
||
  Computation
end signal
```

The `DelayLine` signal is declared as an array of 5 pixels initialized to 0, shared between *DelayLineHandler* and *Computation*. We declare `DelayLine` as value-only using the `value` keyword, since its status will not tested in the rest of the design (they would be no harm in declaring it full, since the unused status would be swept at circuit synthesis time). Since it is not declared `temp`, the value of `DelayLine` will be memorized betweem cycles. Once again, note that memorization specification is *behavioral*: we state that the value should be memorized, but we do not specify how. Creating the appropriate memory elements is the job of the Esterel compiler.

4.3.1 The delay line

The *DelayLine* code is as follows:

```
sustain {
  if InPixel then
    ?DelayLine[0..3] <= pre(?DelayLine[1..4]),
    ?DelayLine[4] <= ?InPixel
  end if
}
```

The `sustain` statement contains two concurrent equations, executed only when `InPixel` is asserted, as stated by the `if` clause that tests the `InPixel` status. The first equation computes the value of `DelayLine[0..3]` using the sequential `pre` operator, which returns the value of a signal at previous cycle. Therefore, at each cycle where `InPixel` is asserted, the new value of the slice `DelayLine[0..3]` is defined as the old value of the slice `DelayLine[1..4]` at previous cycle. The second equation specifies that the new value of `?DelayLine[4]` is the current value of `InPixel`. The equations all define `DelayLine` array components, but do not conflict since there is no overlap in their right-hand-side indexation.

In the delay line definition, the second equation introduces a combinational dependency from `?DelayLine[4]` to `?InPixel`, while the first equation introduces no combinational dependency since there is a `pre` on the way.

4.3.2 The filter computation

The computation code is as follows:

```
signal Product : temp unsigned<[11]>[5] in
  await 3 InPixel;
  sustain ?Product <= Coef [*] ?DelayLine if InPixel
||
  sustain ?OutPixel <= assert<[8]>((?Product[0] + ?Product[1]
                                    + ?Product[2] + ?Product[3]
                                    + ?Product[4]) / 16)
                       if Product
end signal
```

We declare a full temporary signal `Product`, which is an array of 5 11-bit unsigned values. This array holds the products of the pixels times their coefficients. Basic product `*` is extended elementwise to arrays by the curly brackets in '`[*]`'. The `Product` signal has a single status, acting as a valid bit for each of the value array elements[2]. There are two parallel branches in the signal declaration body:

- The first branch is a sequence, whose first components waits for three occurrences of `InPixel` and terminates, and whose second component emits a product array whenever an `InPixel` is available.
- The second parallel branch immediately computes the output pixel whenever a new product array is available.

The `await` statement of the first branch implements the initial delay mentioned in the specification. The `[*]` array operator performs componentwise multiplication of the `Coef` and `?Product` arrays. Notice that the transmission of `Product` from the first parallel arm to the second parallel arm is combinational: the product values computed in the cycle are summed in the same cycle. The numerical assertion `assert<[8]>(...)` asserts that the argument value will always fit in the `Pixel`, i.e. is less than 256. It can be verified at simulation or formal verification time as explained below.

The code is written in a behavioral way. We use both concurrency and sequencing of behavior, and we write "`await 3 InPixel`" for the initial delay without specifying how the counter is implemented. The Esterel code can be viewed as a formal specification of the circuit behavior, with no precise specification of the circuit structure. Nevertheless, the generated circuit will be very similar to a manually designed one, whose source code would be much less readable.

4.3.3 Datapath sizing and checking

Let us now explain datapath sizing. We have set the unsigned size of `Product` to 11 bits. This value needs not be guessed since it can be computed by the Esterel compiler in the following way: first, set the size to a ridiculous `unsigned<[1]>`; call the compiler; the size-checker complains that one bit is to small and that the optimal size is `unsigned<1531>`, which requires 11 bits; then, correct the program, either by writing the optimal value `unsigned<1531>` or the upper approximation `unsigned<[11]>`[3]. The compiler finds the value 1531 by looking at the types of `Coef`, which is `unsigned<7>`, and that of `InPixel`, which is `unsigned<[8]>` = `unsigned<256>`; the maximal value of a product is $6 \times 255 = 1530$, whose best-fitting type is indeed `unsigned<1531>`.

With the above code, we can also check datapath correctness. For the final sum, the `assert<[8]>(...)` assertion asserts that the argument value will always fit in the `Pixel` type, i.e. is less than 256. This property cannot be computed solely by type-checking. Since the maximal value of a product is 1530, the maximal argument value computed at type-checking time for the output pixel is $5 \times 1530/16 = 478$. The type-checker is not able to use the fact that the sum of coefficients is equal to the dividend. There are two ways to check the assertion with Esterel Studio:

[2] The "`Product : unsigned<[11]>[5]`" declaration gives the whole array a single status. To get one individual status bit per array element, use "`Product[5] : unsigned<[11]>`".

[3] which makes no difference here; however, in general, optimal types are better if there is more propagation of size comptations.

- dynamically by simulation, using the Esterel Studio simulator, a C / SystemC simulator running the generated C / SystemC code, or an HDL simulator running the generated HDL code;
- by formal verification, using the Esterel Studio Design Verifier that understands both logic and numbers and can perform a static definitive proof of the assertion.

Once the assertion is proved valid, the truncation to 8 bits implied by the '`<=`' signal assignment symbol is proved arithmetically safe. This strongly constrats with conventional HDL practice where truncation would be built-in and not checked safe, leaving the possibility of nasty hidden bugs.

4.3.4 Correctness w.r.t. implementation constraints

We now explain how to meet the implementation constraints. First, the HDL interface is composed of signals `InPixel`, `InPixel_data`, `OutPixel`, and `OutPixel_data` as required by the specification, because of the way the Esterel compiler names generated signals, as explained in Section 4.1.5. Second, the design is combinational as required. The left-hand-side of an equation combinationally depends on the elements of the right-hand-side that are not within a `pre` delay operator. Thus, the status and data components of `OutPixel` depend on those of `Product`, which themselves depend on those of `DelayLine` through `DelayLine[4]`, which itself depends on the status and value of `InPixel`. This path can be visualized on the source code using Esterel Studio's colorful *browsing mode*.

After having verified the circuit using Esterel Studio simulation and formal verification, we generate two object models using Esterel Studio code generation: an (optimized) HDL model and a SystemC cycle-accurate model. Since both exactly respect the Esterel formal semantics, they have the same behavior by construction, as required by the last constraint.

4.3.5 Generated circuit structure

For HDL generation, there are settable compiling parameters for clock, reset, and DFT compliance that make respecting the constraints easy. We do not detail them here.

GB to complete

4.4 Basic filter specification updates

4.4.1 First specification update: initialization change

When using the filter, the architects realize that using 0's for missing initial values is not good enough. Therefore, they issue the following specification update:

The filter should be started only when three input pixels have been read, with the first read pixel used for for the two missing values:

$$\begin{aligned} y_0 &= (x_0 + 4x_0 + 6x_0 + 4x_1 + x_2)/16 \\ y_1 &= (x_0 + 4x_0 + 6x_1 + 4x_2 + x_3)/16 \end{aligned}$$

With the same value sequence $x = 12, 23, 45, 127, 82, \ldots$ as before, the output sequence should now be $16, 31, 60, \ldots$ instead of $13, 30, 60, \ldots$.

This update is implemented using only a small change to the delay line:

```
signal DelayLine : value Pixel[5] in
   await InPixel;
   for i < 3 dopar
      emit ?DelayLine[i+2] <= ?InPixel
   end for;
   pause;
   sustain {
     if InPixel then
       ?DelayLine[0..3] <= pre(?DelayLine[1..4]),
       ?DelayLine[4] <= ?InPixel
     end if
   }
||
   Computation
end signal
```

The newly added statements illustrate several important features of Esterel behavioral style.

- The initial "`await InPixel`" waits for the first pixel and terminates[4].

- Its termination immediately starts the `for-dopar` loop, which is static and conceptually replicates three parallel instances of its body, loading the initial pixels in `DelayLine[2]`, `DelayLine[3]`, and `DelayLine[4]`.

- The three replicated `emit` statements terminate immediately. Just as a parallel statement, a `for` loop computes the distributed termination of its instances. Therefore, the whole `for` loop terminates in the cycle. Therefore, the `pause` statement is started synchronously with first pixel reception.

- The `pause` statement pauses for one cycle and terminates, giving hand to the normal fifo handling `sustain` statement.

- Multiple emission of signals makes design much easier. Here, there are three statements emitting `DelayLine`, but conflict due to multiple simultaneous emissions for the same `DelayLine` position cannot occur. With Esterel Studio, that fact that can be checked either by dynamic simulation or by formal verification using Design Verifier. It follows from the fact that `pause` temporally separates the `for` loop from the `sustain` statement. (If needed, multiple simultaneous emission is allowed for *combined signal*, see Section **??**.)

In the generated circuit, the change is added muxes on positions 2 and 3 of the delay line and added control for these muxes.

4.4.2 Second specification update: register the output

The basic filter design was initially specified to be combinational. However, this happened to make its output much too late, electrically speaking. Therefore, the following specification update is issued:

[4] With `await`, a pixel present at reset time would be ignored. However, the specification states that there is no such pixel, and all is well. To accept a pixel at reset time, one should write "`await immediate`".

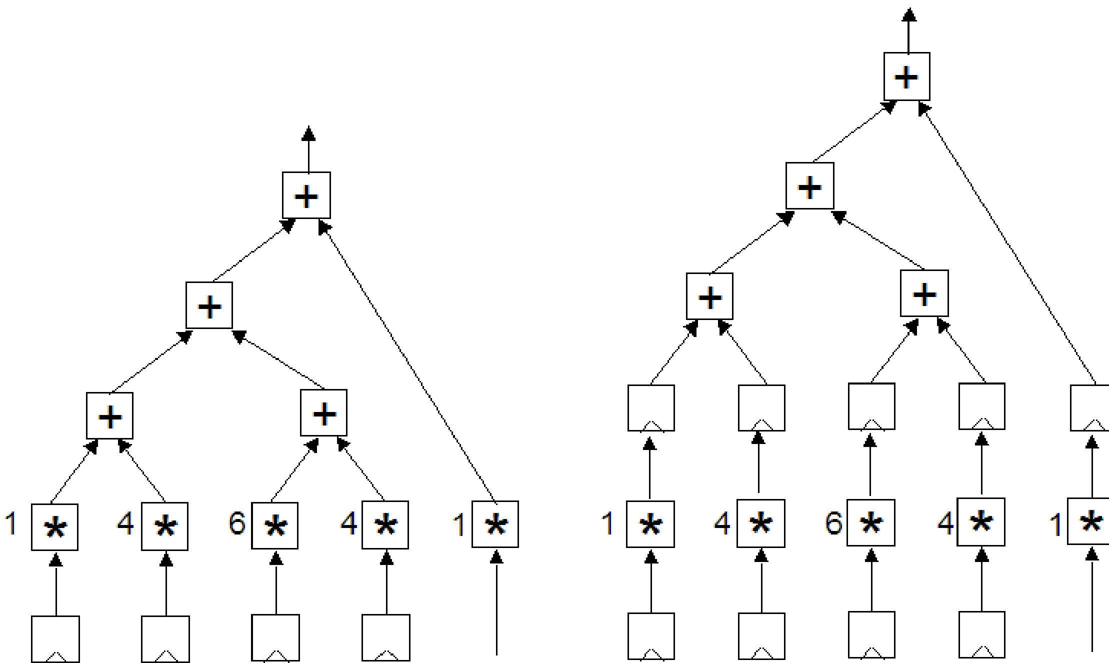

Figure 4.1: Combinational and retimed basic filter trees

The `Outpixel` *and* `OutPixel_data` *signals should be registered outputs. The design should have latency 1: after reading two pixels, whenever a new pixel is read,* `OutPixel` *and* `OutPixel_data` *should be output at next cycle.*

With Esterel, such a specification update is easily implemented using the notion of a registered signal presented in Section 2.2 and Section ??. Here is the appropriate changes in the module header:

```
module BasicFilter_3 :
   ...
   output OutPixel : reg Pixel;     // full registerd
   Body
end module
```

Here is the appropriate change in the executable code:

```
sustain next ?OutPixel <= assert<[8]>((?Product[0] + ?Product[1]
                                       + ?Product[2] + ?Product[3]
                                       + ?Product[4]) / 16)
                         if Product
```

Once again, the change is behavioral: only the `reg` and `next` keyword have been added. Actual register generation is the task of the compiler. In the generated HDL, the effect is imply to register the `OutPixel` and `OutPixel_data` signals.

4.4.3 Third specification update: pipeline the design

Further system-level benchmarks show that the cycle-time of `BasicFilter_2` is too long and should be shortened by pipelining, adding one extra latency cycle. As shown in the

left part of Figure 4.1, the computation parts stacks a tree of adders on top of an array of multipliers. The circuit pipelining solution will consist in inserting a register barrier between the multipliers and adders, as pictured in the right part of Figure 4.1. To obtain this effect, it suffices to delare the product signal registered, changing the computation part as follows:

```
module BasicFilter_4 :
  ...
  signal Product : reg unsigned<[11]>[5] in
    await 3 InPixel;
    sustain next ?Product <= Coef [*] ?DelayLine if InPixel
  ||
    sustain next ?OutPixel <= assert<[8]>((?Product[0] + ?Product[1]
                                         + ?Product[2] + ?Product[3]
                                         + ?Product[4]) / 16)
                              if Product
  end signal
end module
```

Now, there is one cycle latency for the transmission of `Product` from the first equation to the second one and one cycle latency for the output of `OutPixel`.

4.5 Extracting pixels out of words

The basic filter should now be linked with a control unit reading pixels from words, according to the following specification described below.

4.5.1 Word input specification

The full filter should feed the basic filter (in its lasr version) with pixels extracted from 32-bits input words, where each word contains 4 successives pixels ordered from low-order bits to high-order bits. The full filter circuit word interface will consists of three signals: an output `Ready` control bit, which, when asserted, tells the filter environment that the filter is ready to receive a word; a 32-bit wide signal `InWord_data`, which holds the input word value; and a bit `InWord` acting as a valid bit for `InWord_data`. When `InWord` is asserted by the environment, the filter can read a defined word from `InWord_data`. When `InWord` is deasserted, the environment can leave `InWord_data` undefined.

The `Ready` and `InWord` signals can act asynchronously. To make a new word available, the environment asserts `InWord`. When ready to process a word, the filter asserts `Ready`. These can occur in any order, and can be simultaneous. Actual read is performed when `InWord` and `Ready` are both asserted in the same cycle. Then, the filter should de-assert `Ready` and the environment should prepare a new word for the next read.

4.5.2 Using data and interface units to share declarations

The new filter module will be called `WordFilter_4`. It will consist of two concurrent components, a `WordFeeder` module whose role is to extract pixels out of words, and the corresponding basic filter module `BasicFilter_4`.

We now have to deal with three modules, which share data components and interface signals. For instance, the `Pixel` type should be declared in all three modules, the output signal `OutPixel` should be declared by `WordFilter_4` and `BasicFilter_4`, and the

InPixel signal should be an output of WordFeeder, an input of BasicFilter_4, and a local signal within WordFilter_4. In such a situation, it is fundamental to avoid copying the data and interface signal declarations, otherwise any change to a declaration should be done in several places; this is error-prone and one of the best known ways to make designs inconsistent. Sharing declarations is the goal of *data units*] and *interface units*, which we now introduce. Here is the way to factor out the previous declarations using these units:

```
data PixelData :
  type Pixel = unsigned<[8]>;
end data


interface InPixelIntf :
  extends data PixelData;
  input InPixel : temp Pixel;
end interface


interface OutPixelIntf :
  extends data PixelData;
  output OutPixel : temp Pixel;
end interface
```

The “extends data” directive imports all declarations of a data unit in the current unit. Here is the appropriate rewriting of PixelFilter_1, with body unchanged.

```
module PixelFilter_1 :
  extends interface InPixelIntf;
  extends interface OutPixelIntf;
  Body
end module
```

The “extends interface” directive imports all declarations of an interface in the current unit.

With extends, the data and interface secondary keywords are usually redundant and can be omitted. However, it can be useful to solely extends the data part of an interface Intf by explicitly writing “extends data Intf”. See Section ?? for details.

4.5.3 Declaration refinement

There is a fine point about interface extension. For PixelFilter_4, the OutPixel output should be declared registered. This cannot be done in the interface unit, since the fact that a signal is combinational or registered should not visible from users of the module: it is only an internal implementation property of the module. Therefore, the reg declaration should be performed in the module. Here is the way to do this using a *declaration refinement* that keeps the previously declared type of OutPixel and adds the reg features:

```
module PixelFilter_1 :
  extends interface InPixelIntf;
  extends interface OutPixelIntf;
  refine OutPixel : reg;
  Body
end module
```

4.5.4 Word data and interface

The data unit needed to process words has already been presented in Section **??**:

```
data WordData :
  type Word = bool[32];
  map Word {
    p0[0..7], p1[8..15], p2[16..23], p3[24..31]
  };
end data
```

The interface to read words is as follows:

```
interface WordIntf :
  extends data WordData;
  input InWord : Word;
end interface
```

The `InWord` Esterel input full signal groups `InWord` and `InWord_data` circuit inputs as usual.

4.5.5 The WordFeeder module

The `WordFeeder` module header is as follows:

```
module WordFeeder :
  extends interface WordIntf;
  extends mirror InPixelIntf;
  WordFeeder Body
end module
```

We extend the `InWord` input interface and we extend the *mirror* of the `InPixel` interface. Mirroring an interface simply consists in swapping inputs and outputs. This is needed here since `InPixel` has to be an output of `WordFeeder`.

The `?InWord` value is automatically memorized between instants since `InWord` is not declared `temp`. If the status `InWord` is asserted in the cycle, `?InWord` is the value provided by the environment. If `InWord` is de-asserted in the cycle, its value remains that of the previous cycle. This memorization is necessary to extract the four successive pixels, since the environmnent is not supposed to sustain the value of `InWord_data`. The body of `WordFeeder` can be specified in two different ways: textually using Esterel statements. or graphically as as state machine. Figure 4.2 presents the graphical version, and here is the textual version:

```
loop
  weak abort
    sustain Ready
  when InWord;
  emit ?OutPixel <= bin2u(?InWord.p0);
  pause;
  emit ?OutPixel <= bin2u(?InWord.p1);
  pause;
  emit ?OutPixel <= bin2u(?InWord.p2);
  pause;
  emit ?OutPixel <= bin2u(?InWord.p3);
  pause
end loop
```

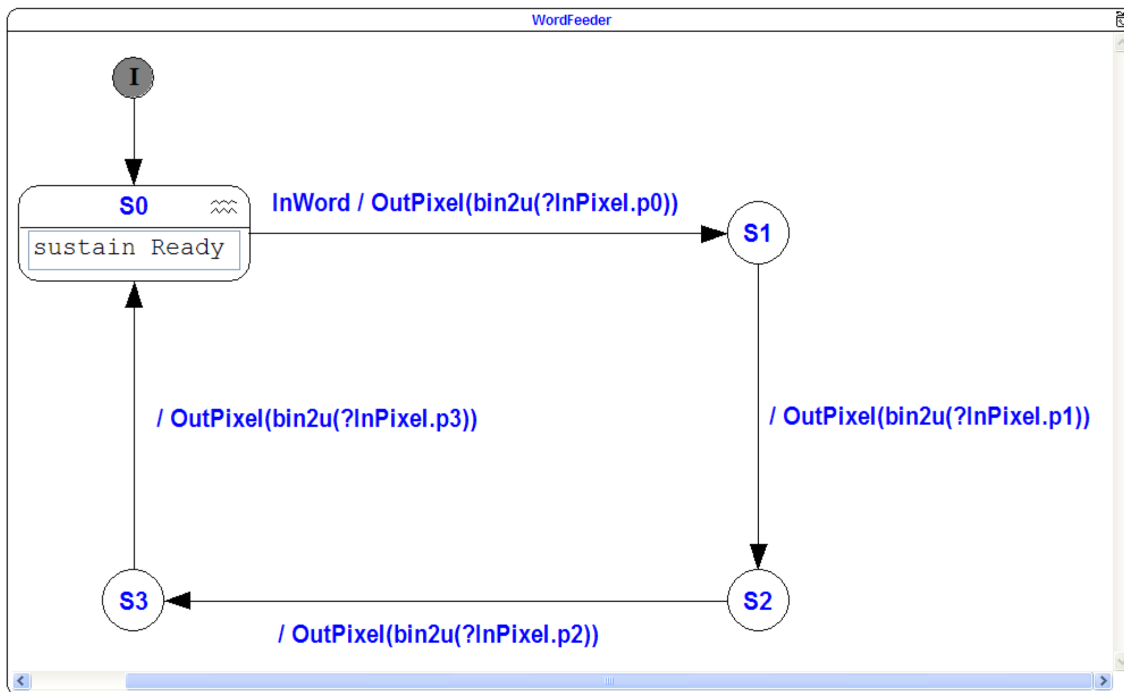

Figure 4.2: The WordFeeder state machine

The textual code is fairly obvious. The initial "`weak abort`" statement keeps `Ready` asserted until an `InWord` is received, this cycle included. Four pixels are extracted in four successive cycles separated by `pause` statements. Notice that `OutPixel` has multiple emitters, which never collide because they are separated by pauses. Running the textual code in the Esterel Studio simulator provides source code animation to make the state changes obvious.

Let us now comment the graphical version, which can also be animated by Esterel Studio. At initial instant, the `S0` state is entered and `Ready` is emitted by the internal `sustain` statement. The transition out of `S0` bears the following trigger / action pair:

```
InWord / OutPixel(bin2u(?InWord.p0)
```

The state machine remains in `S0` until the outgoing transition trigger becomes true, i.e., `InWord` becomes asserted. In that cycle, the transition is taken, the "`sustain Ready`" statement is killed, the transition effect emits `OutPixel` with value `?InWord.p0` converted from binary bitvector to unsigned, and `S1` becomes the new state. The transition is drawn as a simple line, which means that it weakly aborts the contents of `S0`, still emitting `Ready` as required. Strong abortion would be performed by a transition staring with a small red circle.

Pixels `p1`, `p2`, and `p3` are converted to unsigned and output in the next three successives clock cycle; the corresponding transitions are successively fired at each cycle since they have no trigger. At third cycle after `InWord`, the last transition brings the state back to `S0`, thus keeping `Ready` asserted again until the next `InWord` comes in.

Which is preferable from the graphical and textual designs depends on the user's point of view. To help the user change her or his mind, Esterel Studio embeds a *sequential equivalence checker* able to prove that two designs behave identically for any input sequence.

4.5.6 The WorldFilter main module

Building the `WordFilter` main module now simply consists in putting `WordFeeded` and `BasicFilter` in parallel, with local declaration of the intermediate `InPixel` signal. This declaration is performed extending the `InPixelIntf` interface:

```
module WordFilter_4 :
  extends WordIntf;
  extends OutPixelIntf;
  signal extends InPixelIntf in
    run WordFeeder
  ||
    run BasicFilter_4
  end signal
end module
```

Notice that signal directionalities are irrelevant in the local extension of `InPixelIntf`.

4.6 Handling pixel lines

The word filter should now be transformed into a line filter that processes image lines of fixed length.

4.6.1 Line filter specification

The filter should process input lines of 512 pixel and output lines of the same size. Since the first output pixel is output only when three input pixels have been read, two extra pixels should be symmetrically output after the end of the input line by copying the last input pixel:

$$
\begin{aligned}
y_{509} &= x_{507} + 4x_{508} + 6x_{509} + 4x_{510} + x_{511} \\
y_{510} &= x_{508} + 4x_{509} + 6x_{510} + 4x_{511} + x_{511} \\
y_{511} &= x_{509} + 4x_{510} + 6x_{511} + 4x_{511} + x_{511}
\end{aligned}
$$

The basic filter in use should be `BasicFilter_4`. Since it has been thoroughly validated It should be reused as such, without any modification

4.6.2 From WordFilter to LineFilter

The new module will be called `LineFilter_4`. There is no need for change in the data and interfaces. We simply replace `WordFeeder` by a new module `LineFeeder` described below and place `BasicFilter_4` within a `loop` statement with “`weak abort`” body:

```
module LineFilter_4 :
  extends InPixelIntf;
  extends OutPixelIntf;
  signal extends InPixelIntf in
    run LineFeeder
  ||
    loop
```

```
        weak abort
          BasicFilter_4
        when 512 OutPixel
      end loop
    end signal
  end module
```

Because of `loop` and "`weak abort`", the basic filter is automatically aborted, reset, and restarted when 512 pixels have been output. Because of input-output latency, it is important to count output pixels and not input pixels.

4.6.3 The LineFeeder module

The new `LineFeeder` module is obtained in a similar way from the existing `WordFeeder` module. After 512 input pixels sent, the `WordFeeder` module is weakly aborted and a padding state is entered for two ticks before the whole behavior is restarted afresh.

The textual version is written using a loop over a feed-pad sequence, where each element gets weakly aborted:

```
module LineFeeder :
  extends interface WordFeeder;
  loop
    abort
      run WordFeeder
    when 512 OutPixel;
    // padding
    weak abort
      pause; // skip a tick
      sustain ?OutPixel <= ?InWord.p3
    when 2 tick
  end loop
end module
```

Graphically, the same behavior is done using a *hierarchical state machine*. Such a machine can be designed in two ways. Figure 4.3 shows how to call `WordFeeder` using a *submodule run state* in `LineFeeder`. When the transition out of this state is taken, the `WordFeeder` submodule is weakly aborted and control goes to the padding state that resends the last pixel `?InWord.p3`. After 2 ticks, the `WordFeeder` submodule is restarted afresh.

An alternative pictured in Figure 4.3 is to directly embed the `WordFeeder` state machine into the first hierarchical state of `LineFeeder`, which is called a *macrostate*. Readability is improved, but direct reuse is decreased. Mixed graphical / textual design is also possible: the textual body of `WordFeeder` can be equivalently put in a textual macrostate, as shown in Figure 4.4.

4.7 Parametrizing the filter for reuse

FilterParam

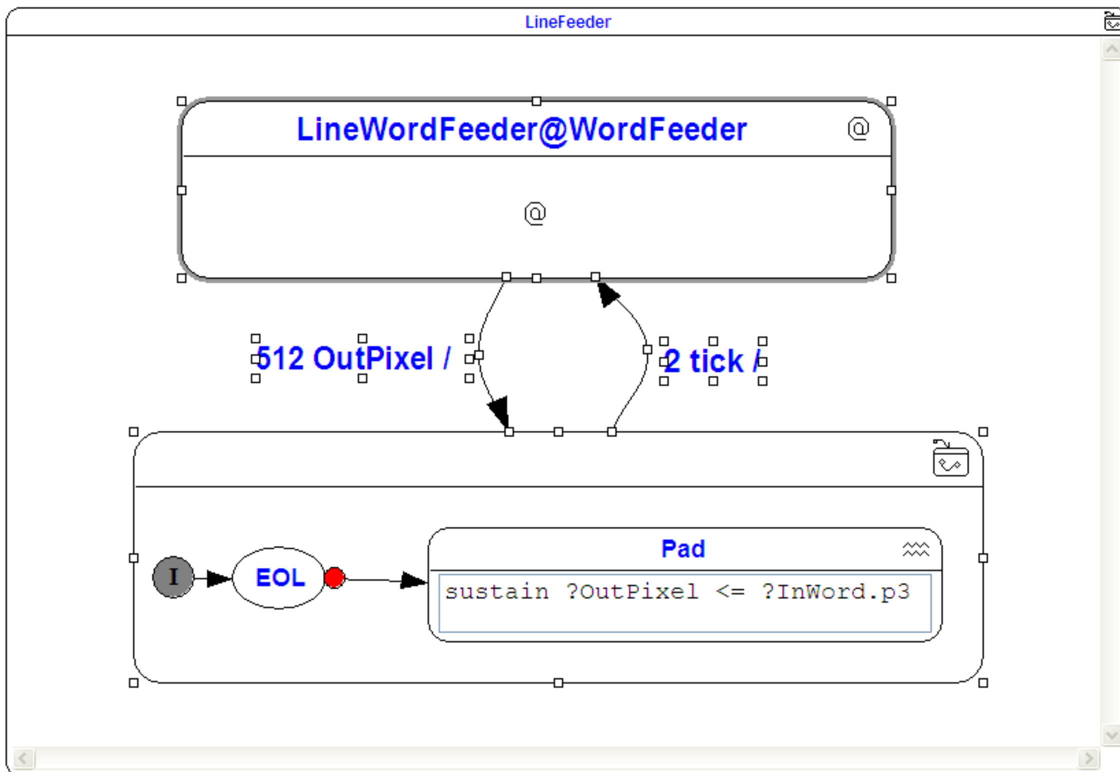

Figure 4.3: The LineFeeder state machine

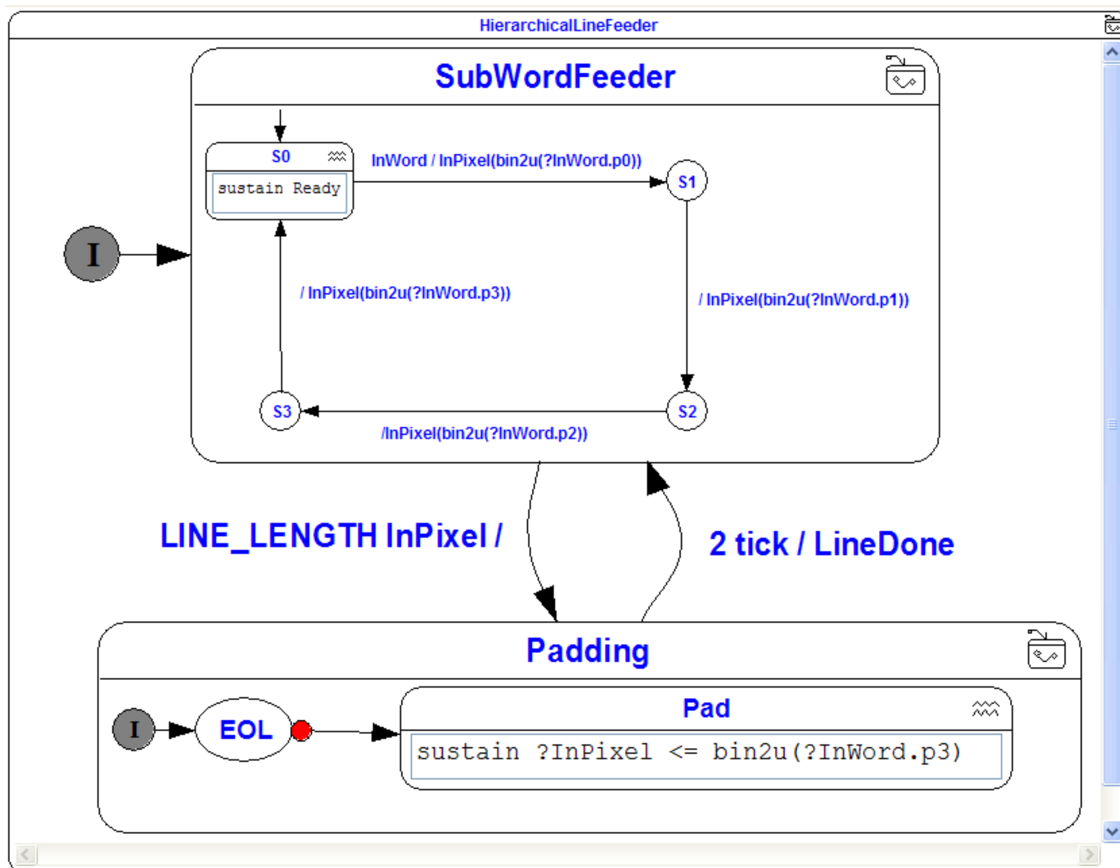

Figure 4.4: The LineFeeder hierarchical state machine

Bibliography

[1] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96, Lille, France*, July 1996.

[2] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft book, available in current form by download from web address https://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf, 2002.

[3] G. Berry, M. Kishinevsky, and S. Singh. System level design and verification using a synchronous language. In *Proc. International Conference on Integrated Circuit Design (ICCAD'03) San Jose, USA*, 2004.

[4] G. Berry and H. Touati. Optimized controller synthesis using Esterel. In *Proc. Intl. Workshop on Logic Synthesis, Lake Tahoe, USA*.

[5] G. Berry *et. al.* *The Esterel v7_60 Language Reference Manual*. Available at https://www-sop.inria.fr/members/Gerard.Berry/Papers/Esterelv7-refman76.pdf, 2007.

[6] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[7] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8:231–274, 1987.

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