The Esterel v7 Reference Manual
Version v7_60 for Esterel Studio 6.1

Esterel EDA Technologies
679 av. Dr. J. Lefebvre
06270 Villeneuve-Loubet, France

www.esterel-eda.com

1 November 2008

Contents

Chapter 1

Introduction

This document is the reference manual of the Esterel v7 language, release v7_60, supported by Esterel Studio 6.0. Esterel v7 is a major evolution of the previous version Esterel v5 [?, ?, ?], which makes it possible to design much richer systems and in particular hardware and software systems that combine complex data path and control path features.

1.1 The Esterel v7_60 version

Esterel v7_60 is a minor evolution of Esterel v7. Esterel v7_60 involves the following extensions:

- The '`_`' character can be used in non-prefixed unsigned literals, as for `18_500_000`.
- The `observe` keyword can be used in an interface extension or a port definition to import all the signals as inputs, see Section ??.
- Enum codes can be signed or unsigned numbers, see Section ??. In the previous version, enum codes had to be unsigned numbers only.
- Enum values can be converted to unsigned or signed numbers and conversely, see Section ??.
- The *switch expression* in a `switch` statement can be of signed type or bitvector type, see Section ??. In the previous version, only switch expressions of unsigned or enum types were allowed.
- Signal emissions in `emit` and `sustain` statements can be nested in `switch-case` structures, see Section ??.
- In a `run` statement, a submodule value-only input can be renamed by an expression, see Section ??. In the previous version, only submodule pure inputs could be renamed by an expression.
- In a `run` statement, a registered signal can rename a submodule output, see Section ??. In the previous version, a registered signal could not rename a submodule output.
- Bitvector maps can be defined in a sequenced way, see Section ??.

- The `onehot` predefined function can be used to check if a bitvector is onehot-encoded, see Section **??**.
- The `resize` and `resize` predefined functions can be used to resize a bitvector to a new length, see Section **??**. The `resize` and `resize` functions are more general than the `extend` and `sextend` functions and intended to replace them. The `extend` and `sextend` functions should be considered as obsolete.
- The `if static` instruction makes it possible to write configurable code and also enables static module recursion, see Section **??**.
- *Assumptions* are now built-in in the language and can be expressed with the `assume` keywords, see Section **??** and Section **??**.
- *Coverage points* make it possible to monitor Boolean expressions for design coverage, see Section **??** and Section **??**.
- Observers make it possible to write observation code independent from design code but connected to it for verification, see Chapter **??**.
- Local signals and ports can be declared directly in module headers, see Section **??**. This makes them visible from observers, see Section **??**.

1.1.1 Esterel v7_60 feature summary

Here is a summary of the main features of Esterel v7_60:

- Synthesizability of all programs in hardware and software, including multiclock designs.
- Separation of data, interface, and module units for better program structuring and reuse.
- Data genericity of interface and module units.
- Arrays of any dimension and arrays of arrays of any base type. Array expressions, with array slicing on any number of dimensions. Bitvectors seen as arrays of Booleans.
- Arbitrary precision exact arithmetic, dealing with abstract unsigned and signed numbers.
- Multiple number encodings to switch between numbers and their representations as bitvectors.
- Equational definitions of signals, which nicely complement Esterel imperative statements and are of major use in data path descriptions.
- Powerful temporal statements to handle complex control.
- Static replication of statements, fundamental for architectural design.
- Built-in assertions to check dynamic properties by simulation or formal verification. Built-in assumptions to express environment constraints.

- Built-on coverage points to enhance design verification during simulation and check coverability by formal verification.
- Observers for non-intrusive design verification.

1.2 Structure of the Language Reference Manual

We chose to present the manual in three parts: Part A deals with single-clock design, while Part B deals with multiclock design, and Part C is dedicated to observers. Multiclock design requires preliminary understanding of single-clock design. Therefore, we think it is wiser to present it separately in Part B. Observers presented in Part C are not intended for design building but for design verification. That is why they are described in a separate part.

For Part A, Chapter **??** presents the data, interface, and module units, and details the data genericity mechanism. Chapter **??** presents the data objects. Chapter **??** presents the unsigned and signed exact arithmetic. Chapter **??** presents signals and variables. Chapter **??** presents signal declarations in interfaces and modules. Chapter **??** presents signal and data expressions. Chapter **??** presents the Esterel v7 statements. Chapter **??** presents the submodule run statement. Chapter **??** present oracles, a special kind of uncontrolled signals to model non-determinism. Chapter **??** presents the macro-expansion of complex constructs into simpler ones. Finally, Chapter **??** presents the run-time errors that Esterel v7 programs can trigger.

For Part B, Chapter **??** presents the basics of multiclock design. Since multiclock design is much more delicate than single-clock design, the chapter presents a thorough studies of examples. Finally, Chapter **??** presents the syntax and constraints of multiclock units.

For Part C, Chapter **??** introduces observers, which are intended for non-intrusive design verification. The purpose of observers is to write verification code, in particular assumptions, assertions, and coverage points, which is independent of design code but connected to it for verification purpose. Since observers are brand new in v7_60, some examples are given at the end of the chapter in order to get users started.

1.3 Lexical issues

1.3.1 Identifiers and keywords

Identifiers start with a letter and can contain letter, digits, and '_' characters. They are case-sensitive and of unlimited length.

The list of keywords is as follows:

abort abs after always and assert assert_unsigned await
bin2u bin2s binsize bool bool2u
call case clock combine constant
data default do dodown dopar double doup
each else emit end enum every exit extends
false finalize float for function
generic gray2u
halt handle host
if immediate in init inout input inputoutput integer interface
lcat loop

```
main map mcat mcrun mem mirror mod module multiclock mux
never next not nothing
onehot2u open or oracle out output
pause port positive pre pre1 procedure
refine reg reg1 relation repeat reset reverse run
s2bin sat seq sextend signal sign signed string suspend sustain switch
temp then tick times trap true type
u2bin u2gray u2onehot unsigned upto
value var
weak when with
xor
```

It is possible to use a keyword as an identifier by prefixing it by '\'. For instance, '`\emit`' is a valid identifier whose actual name is `emit`.

1.3.2 Comments

Single-line comments start with '//', and multiline comments are bracketed by '/*' and '*/' (for compatibility with Esterel v5, one can also use '%' for single-line comments and '%' and '%' to bracket multiline comments). Comments starting by '///' or '/**' are propagated in special ways in generated code and documentation, see the compiler doceumentation for details.

1.3.3 Labels

One can use the '@' symbol after a keyword or operator to define a label to be propagated to the generated code in some compiler-defined way. Here are examples:

```
pause@WAIT1
halt@"hello␣␣␣world␣"
x +@smart y
```

The way labels are propagated is compiler-dependent, see the compiler documentation for details.

Acknowledgements: The Esterel v7 language is now being proposed for IEEE standardization under project P1778. A preliminary design was performed by G. Berry, from Esterel technologies, and M. Kishinevsky from Intel Strategic CAD Lab, Portland. An early document about this design is available on `www.esterel-eda.com`. Some constructs have been borrowed from the Esterel++ design done at Dassault Aviation. The language has benefited of many discussions with Stephen Edwards, Columbia University, Olivier Tardieu, INRIA, and many users at Intel, Philips (now NXP), ST Microelectronics, and Texas Instruments.

Chapter 2

Units and Genericity

2.1 Data, interface, and module units

An Esterel v7_60 program is composed of toplevel named *units*, which can be of three kinds: data units, interface units, and module units. The role of the different units is as follows:

- A *data unit* declares data types, constants, functions, or procedures. These objects can be either *defined* in place, *host*, i.e. defined in the host language to which Esterel is compiled, or *generic* for better reuse, as explained below. A data unit can *extend* one or several other data units by importing all the objects they declare. Simple or multiple data extension will be fully studied in Section **??**.
- An *interface unit* primarily declares input / output signals. It can extend other data and interface units by importing all their data and signal declarations. Extension can be selective, i.e. limited to the input or output signals of the extended interface. The *mirror* "`mirror Inf`" of an interface `Intf` is an interface with the same signals but input / output directionalities exchanged. An interface unit can also declare data objects and extend data units locally. Interface units can be generic w.r.t. data objects. Interface extension will be introduced in Section **??** and fully studied in Chapter **??**.
- A *module unit* defines a reactive behavior. It is composed of a *module header*, which contains declarations, and of a *body* which is an executable statement. As the other units, a module can be generic w.r.t. data objects. A module header can extend data and interface units by importing all the objects they declare. There is no module behavior extension, this notion being much less clear than data and interface extension. It can also declare locally data objects and interface signals.

Data genericity means that units and modules can be defined in an abstract way, depending on generic parameters. For instance, a module that delays a value by some number of ticks can be declared with data a generic value type `T` and a generic delay constant `D`. When instantiating the module, one can for instance specify the type `T` to be `bool` and the delay `D` to be 3. Genericity is studied in Section **??**.

Modules can instantiate the behavior of other modules using the `run` executable statement described in Chapter **??**. One of the modules must be called the `main` module. It is the one to be executed.

All the unit names share the same namespace. Therefore, data, interface, and module units must have distinct names. Units make object names visible either directly when using extension or through the dot notation '`X.A`' for port fields. There should also be no conflict between unit names and object names. The visibility rules will be made precise below for each kind of unit.

Before entering into details about units and genericity, we give two illustrative examples to gradually introduce these concepts.

2.2 A basic example

Here is a main module written in the most compact form, grouping all data and interface declarations in the module header. The module takes a signed number as input and doubles it on the output. There is precise size control on the input and output values, which are respectively in the ranges $[-8..7]$ and $[-16..15]$:

```
main module Double :
  constant N : unsigned = 8;
  input I : signed<N>;
  output O : signed<2*N>;
  every I do
    emit ?O <= 2 * ?I
  end every
end module
```

Splitting data and interface signals in separate units make it possible to separate the data / interface / behavior concerns for better readability and reuse. Here is the same example with maximally split data and interface units:

```
data SizeData :
  constant N : unsigned = 8;
end data

interface DoubleIntf :
  extends data SizeData;
  input I : signed<N>;
  output O : signed<2*N>;
end interface

module Double :
  extends interface DoubleIntf;
  every I do
    emit ?O <= 2 * ?I
  end every
end module
```

The `extends` declarations specify that all the objects defined in the extended unit are imported in the current unit, in a recursive way. Therefore, the constant definition `N = 8` is imported within `Double` since `Double` extends `DoubleIntf` which itself extends `SizeData`.

It is also possible and sometimes better to split concerns a little less, in order to avoid introducing too many small units. Here, we can suppress the data unit and directly declare the constant `N` in the interface:

```
interface DoubleIntf :
  constant N : unsigned = 8;
  input I : signed<N>;
  output O : signed<2*N>;
end interface
```

An interface can extend another interface by importing all its data and signal components:

```
interface ExtendedIntf :
  extends interface DoubleIntf;
  output X : bool[N];
end interface
```

The components of `ExtendedIntf` are `N`, `I`, `O`, and `X`. The latter is a bitvector of size `N`, see Chapter **??**.

If one is only interested in importing the constant `N`, one can extend only the data part of `DoubleIntf` using the "`extend data`" keywords, thus forgetting about the signals `I` and `O`:

```
data PartiallyExtendedIntf :
  extends data DoubleIntf;
  output X : bool[N];
end data
```

Then, the only components of `PartiallyExtendedIntf` are `N` and `X`.

2.2.1 Making the units generic

So far, we have hard-wired the value 8 for `N` in its declaration. It is possible to make the units *generic* w.r.t. the value of `N`, by prefixing the declaration with `generic` and not giving the value. Here is a generic interface declaration:

```
interface DoubleIntf :
  generic constant N : unsigned;
  input I : signed<N>;
  output O : signed<2*N>;
end interface
```

The generic constant `N` acts as a size parameter for the input signal `I` and the output signal `O`. The type `signed<N>` of `I` is that of all signed numbers from $-$`N` to `N`-1 included, while the type `signed<2*N>` of `O` is that of all signed numbers from $-2 \times$ `N` to $2 \times$ `N` $- 1$, see Section **??** below for details. These defined types themselves depend on the generic parameter `N`; we call them *defined generic* types.

Genericity is inherited by simple extension: without having to change its code, `Double` above automatically becomes generic in the same constant `N` since it imports the generic declaration of `N`.

Notice that the generic parameters are declared together with normal data objects, unlike in C++-like languages where generics are declared in a specific parameter list placed before the class declaration.

2.2.2 Instantiating a generic unit

The generic interface and module are instantiated by substituting actual objects for generic ones using the '`[./.]`' substitution notation below:

```
interface DoubleIntf8 :
  extends interface DoubleIntf [constant 8 / N]
end interface

module Double8 :
  extends interface DoubleIntf8;
  run Double [constant 8 / N]
end module
```

For the interface, the `extends` declaration simply includes the extended interface into the new one after substitution of actual data objects to generic ones. For the module, the `run` statement does more than instantiating the generic module: it also executes its code. Here, the `run` statement is executed at toplevel, realizing the simplest behavior instantiation. Generally speaking, `run` statements can appear anywhere a statement can.

2.3 A memory example

The next example is that of a simple memory. We start with a non-generic memory, then make it generic, and instantiate it in different ways.

2.3.1 A non-generic memory

A non-generic memory may have object type, size, and address type, explicitly defined as follows:

```
data MemData :
  type ObjType = bool[32];
  constant MemSize : unsigned = 256;
  type Address = unsigned<MemSize>;
end data
```

The memory interface is declared as follows:

```
interface MemIntf :
  extends data MemData;
  input Write : Address;
  input DataIn : value ObjType;
  input Read : Address;
  output DataOut : value ObjType;
end interface
```

The “`extends data`” declaration imports all data objects of `MemData` into `MemIntf`. The other objects declared in `MemIntf` are *interface signals*. Valued signals such as `Read` or `Write` carry two kinds of information: a Boolean *status*, absent or present (also called low or high, unset or set, 0 or 1, etc.), and a value of the type that follows the colon. The status is tested by giving the name of the signal, as in “`if Write then...`”, and the value is read using the ‘`?`’ operator, as for `?Write` which denotes the address carried by `Write`. A value-only signal such as `DataOut` is declared with the additional `value` keyword. It has no status.

In `MemIntf`, the status of `Write` is used as a write enable for the value `DataIn` at address `?Write`, while that of `Read` is used as a read enable at address `?Read`, with the read operation returning the value `?DataOut`. The memory behavior is written as follows:

```
module Mem :
extends interface MemIntf;
var MemArray : ObjType[MemSize] in
  always
    if Write then
      MemArray[?Write] := ?DataIn
    end if;
    if Read then
      emit ?DataOut <= MemArray[?Read]
    end if
  end always
end var
end module
```

The semicolon ';' denotes sequencing: if write and read occur at the same address, the value read is the one just written.

2.3.2 Making the memory generic

Making the memory generic consists in specifying the value type and memory size as generic parameters, without changing the interface and module codes. For this, we simply write the following:

```
data MemData :
  generic type ObjType;
  generic constant MemSize : unsigned;
  type Address = unsigned<MemSize>;
end data
```

In `MemData`, the generic parameters are explicitly indicated by the `generic` keyword. The `Address` defined generic type is not a generic parameter, but it is indirectly generic in some sense since it depends on the generic parameter `ObjType`.

When making the data unit generic, the memory interface and behavior module automatically become generic in the same parameters since basic extension of a data unit simply imports its declaration. One can also make interfaces and modules directly generic by declaring generic objects within them, without resorting to a separate data unit.

2.3.3 Reconstructing a specific memory

Our initial word memory interface can be reconstructed as follows, with more explicit type and size names:

```
data WordData :
  type Word = bool[32];
  constant WordMemSize: unsigned = 256;
end data


interface WordMemIntf :
  extends WordData;
  extends MemIntf [type Word / ObjType;
                   constant WordMemSize / MemSize]
end interface
```

Here, extension of `MemIntf` substitutes actuals for generic parameters and imports the other data and signal objects from `MemIntf` after this substitution. Therefore, the visible data objects within `WordIntf` are the type `Word`, the constant `WordMemSize`, and the `Address` type defined by `Address = unsigned<WordMemSize>`. The generic parameter names `Objtype` and `MemSize` are not visible any more. The signals of `MemIntf` are as follows:

```
input Write : Address;
input DataIn : value Word;
input Read : Address;
output DataOut : value Word;
```

Here is the instantiation of the behavior module:

```
module WordMem :
  extends interface WordMemIntf;
  run Mem [type Word / ObjType;
           constant WordMemSize / MemSize];
end module
```

Because we have split the interface and module declaration , we need to repeat the substitution in the `run` executable statement; otherwise, that statement would still have generics `ObjType` and `MemSize`. Substitution in the interface extension from `MemIntf` to `WordMemIntf` has no effect on the `Mem` module, which still has `MemIntf` interface: `extends` and `run` are uncorrelated.

2.3.4 Ports and run statements

Assume we want to use a pair of word memories in an application. It is clearly not possible to directly extend twice `WordMemIntf`, since this would doubly declare its component with the same names. We do not provide ways of renaming signals in interface instantiation. The solution we propose is to use *ports*, which are groups of signals typed by an interface. One uses a dot-notation to access port fields, as for conventional record fields. Here is a compound of two concurrent word memories accessed by two ports:

```
module DoubleWordMem :
  port P1 : WordMemIntf;
  port P2 : WordMemIntf;
{
  run WordMem1 / WordMem [P1.Write / Write, P1.DataIn / DataIn,
                          P1.Read / Read, P1.DataOut / DataOut]
||
  run WordMem2 / WordMem [P2.Write / Write, P2.DataIn / DataIn,
                          P2.Read / Read, P2.DataOut / DataOut]
}
end module
```

As far as behavior is concerned. two instances of the `WordMem` module are run in parallel using the Esterel '`||`' concurrency operator. To distinguish the instances, they are renamed into `WordMem1` and `WordMem2` in the `run` statement. One writes into memory 1 using `P1.Write`, and one reads from memory 2 using `P2.Read`. In the `run` statements, the '`/`' character is used to connect signals in the caller to interface signals in the callee. For instance, the port field `P1.Write` declared in `DoubleWordMem` is connected to the `Write` input of `WordMem1`.

To further simplify the writing, on can use the `open` statement described in Section **??** that gives direct access to port fields:

```
{
  open P1 in
    run WordMem1 / WordMem
  end open
||
  open P2 in
    WordMem2 / WordMem
  end open
}
```

In the body of the "`open P1`" port opening statement,simple identifiers such as `Write` become synonyms to the field `P1.Write`. Therefore, the implicit renaming "`Write / Write`" silently performs the actual renaming to "`P1.Write / Write`".

Notice that signal instantiation uses the same syntax as data instantiation: the actual signal is on the left of the '`/`' instantiation symbol and the instantiated interface parameter is on the right. This is true independently of the input / output directionality of the interface signals. Data instantiation and signal instantiation can both appear in an instantiation list, with data instantiation first, see Chapter **??**..

Port declaration and run statements are not extensions

A port declaration refers to an interface but it does not provoke extension of this interface. Similarly, a `run` statement does not extend the data and interface of the module it refers to. Therefore, in the body of `DoubleWordMem`, the data object `Word`, `WordMemSize`, and `Address` are not visible. This is not a problem here since we do not need to refer to them. However, we might need to be able to use them, for instance to code additional check or trace operations. This is easily done by adding the following declaration:

```
extends data WordMemIntf;
```

2.3.5 Multiple instantiation of generic data: renaming defined objects

Substituting actual data to generic parameters is not enough to deal with multiple heterogeneous instantiation of a generic unit. For instance, assume we want to define an interface for a pair of memories of different types and sizes, with an access port for a 128 bytes memory and an access port for a 256 words memory. A fully explicit description would be as follows:

```
data ByteMemData :
  type Byte = bool[8];
  constant ByteMemSize = 128;
  type ByteAddress = unsigned<ByteMemSize>;
end data

data WordMemData :
  type Word = bool[32];
  constant WordMemSize = 256;
  type WordAddress = unsigned<WordMemSize>;
end data
```

```
interface ByteMemIntf :
  extends ByteMemData;
  input Write : ByteAddress;
  input DataIn : value Byte;
  input Read : ByteAddress;
  output DataOut : value Byte;
end interface

interface WordMemIntf :
  extends WordMemData;
  input Write : WordAddress;
  input DataIn : value Word;
  input Read : WordAddress;
  output DataOut : value Word;
end interface

interface MemoryPair :
  extends ByteMemData;
  extends WordMemData;
  port BytePort : ByteMemIntf;
  port WordPort : WordMemIntf;
end interface
```

Obviously, the goal of genericity is to factor out these declarations by instantiating the same generic units twice. The first obvious idea would be to write the data units hierarchically as follows:

```
data ByteMemData :
  type Byte = bool[8];
  constant ByteMemSize = 128;
  extends MemData [type Byte / ObjType;
                   constant ByteMemSize / MemSize]
end data

data WordMemData :
  type Word = bool[8];
  constant WordMemSize = 256;
  extends MemData [type Word / ObjType;
                   constant WordMemSize / MemSize]
end data



data DoubleMemData :
  extends ByteMemData;
  extends WordMemData;
end data
```

with automatic building of address types by the extension scheme. But this simple method would not work since it would provoke a double declaration of the `Address` type:

```
type Address = unsigned<ByteMemSize>;  // from ByteMemData
type Address = unsigned<WordMemSize>;  // from WordMemData
```

Therefore, we need a way to distinguish between byte and word addresses as we did in the non-generic design. To solve this problem, Esterel provides the user with the possibility of renaming defined types at extension time:

```
data ByteMemData :
  type Byte = bool[8];
  extends data MemData [type Byte / ObjType,
                             ByteAddress / Address;
                        constant ByteMemSize / MemSize]
end data

data WordMemData :
  type Word = bool[32];
  extends data MemData [type Word / ObjType,
                             WordAddress / Address;
  constant WordMemSize / MemSize]
end data

data DoubleMemData :
  extends data ByteMemData;
  extends data WordMemData;
end data
```

Although they use the same substitution symbol '/', the substitution "`Byte / ObjType`" and "`ByteAddress / Address`" are not quite of the same nature: the first one passes the actual type `Byte` defined in `ByteMemData` to replace the generic type `ObjType` of `MemData`, while the second one renames the defined generic type `Address` generated by `MemData` into `ByteAddress` for inclusion in `DoubleMemData`.

To avoid performing the same substitution again for interfaces, one can do the substitution directly for both data and signals at interface level:

```
interface ByteMemIntf :
   extends interface MemIntf [type Byte / Obj,
                                   ByteAddress / Address;
                              constant ByteMemSize / MemSize]
end interface

interface WordMemIntf :
   extends interface MemIntf [type Word / Obj,
                                   WordAddress / Address;
                              constant WordMemSize / MemSize]
end interface


module ByteWordMem :
  extends data ByteMemIntf;  // forgetting signals
  extends data WordMemIntf;  // forgetting signals
  port BytePort : ByteMemIntf;
  port WordPort : PortMemIntf;
  run Mem [type Byte / ObjType;
           constant ByteMemSize / MemSize;
           signal BytePort.Write / Write,
                  BytePort.DataIn / Write,
                  BytePort.Read / Read,
                  BytePort.DataOut / DataOut]
```

```
||
  run Mem [type Word / ObjType;
           constant WordMemSize / MemSize;
           signal WordPort.Write / Write,
                  WordPort.DataIn / Write,
                  WordPort.Read / Read,
                  WordPort.DataOut / DataOut]
end module
```

2.3.6 Composing genericity

The byte and word memories share a common property: they both store bitsets of different lengths. Therefore, it can also be useful to define bitset memories of variable length as intermediate structure. Here is how to do this:

```
data BitsetMemData :
  generic  constant BitsetLength : unsigned;
  type BitsetType = bool[BitsetLength];
end data

interface BitsetMemIntf :
  extends BitsetMemData;
  extends MemIntf [type BitsetType / ObjType];
end interface

module BitsetMem :
  extends BitsetMemIntf;
  run Mem  [type BitsetType / ObjType];
end module
```

Here, we have changed the level of genericity: the former generic type `ObjType` has been replaced by a generic parameter `BitSetLength` determining the size of the bitset type. This is what we call genericity composition. The other generic parameter was `MemSize`. Since it has not been instantiated, its declaration is simply imported in `BitSetMemIntf` and `BitSetMem` and it remains generic there. The implicitly created interface unit is as follows:

```
interface BitsetMemIntf :
  generic constant BitsetLength : unsigned;
  type BitsetType = bool[BitsetLength];
  generic constant MemSize : unsigned;
  type Address = unsigned<MemSize>;
  input Write : Address;
  input DataIn : value BitsetType;
  input Read : Address;
  output DataOut : value BitsetType;
end data
```

2.3.7 Implicit generic instantiation

Finally, if there is only one instantiation of the memory in the design, simply defines the generic parameters with the same names before instantiating the design:

```
interface WordMemIntf :
  constant MemSize = 256;
  type Address = unsigned<MemSize>
  type ObjType = bool[32];
  extends MemIntf;
end interface
```

Here, the simple extension "`extends MemIntf`" acts as a short-hand for

```
  extends WordMemIntf [ type Addess / Address,
                        ObjType / ObjType;
                        constant MemSize / MemSize]
```

This capture-by-name mechanism is very convenient for large argument lists.

2.4 Data genericity specification

2.4.1 Data declaration

Units declare data objects of four different *kinds*: type, constant, function, and procedure. Data objects can be declared in three different ways:

- *Generic*, declared with the `generic` keyword. Generic objects only have a kind, a name, and possibly a type.
- *Fully defined*, with no reference to generic objects. They can be given a definition in the data unit, or they can be declared *host*, in which case their definition is to be given in the host language.
- *Defined generic*, with a definition, that refers to other generic or indirectly generic object. Host objects cannot be defined generic; in particular they cannot have generic types.

Consider the following example:

```
data D :
  generic constant WordLength : unsigned;
  type Word = bool[WordLength];
  host type Time;
  generic type T;
  generic constant A : T;
  generic function F (T, T) : T;
  constant C : T = F(A,A);
end data
```

Here, `WordLength`, `T`, `A`, and `F` are generic, while `Time` is fully defined and `Word` and `C` are defined generic. There is an additional dependency of `A`, `F`, and `C` on `T` since their types involves `T`; these types will be viewed as a constraints to be respected at instantiation time. For instance, if one writes "`bool / T`", one must substitute `A` by a Boolean constant and `F` by a Boolean binary function. In that case, `C` will indeed type-check as a `bool`.

2.4.2 Generic data extension

Data extension of a generic unit `Sub` within a current unit `U` is peformed by declaration of the form "`extends data Sub`", optionally with a bracketed substitution list. A substitution list has semicolon-separated sections, each identified by a keyword `type`, `constant`, `function`, or `procedure`. There can be serveral sections starting with the same keyword. Within each section, Substitutions are comma-separated. A substitution has the form "`X / Y`" where `X` is a data object already declared in `U` and `Y` is a data object of the same kind declared in `Sub`. The rules are as follows:

- If `Y` is a generic parameter in `Sub`, there are two subcases:
  - If there is already an object called `Y` in `U`, then this object is substituted to `Y` in the definition of all the subsequent data objects imported from `Sub`.
  - Otherwise, `Y` becomes a generic parameter of `U`.
- If `Y` is a defined or defined generic object in `Sub`, its definition in `Sub` becomes the definition of `X` in `U` after generic parameter substitution.

2.4.3 Generic extension example

Generic extension is mostly useful for interfaces and modules. Consider the following generic interface that extends `D` above:

```
interface Intf1 :
  extends D;
  input I : T;
  output O : Word;
end interface
```

A partial extension of `Intf1` can be defined as follows:

```
interface Intf2 :
  extends Intf1 [type bool / T;
                 constant true / A;
                 function and / F]
end interface
```

Then `Intf2` is actually the following interface:

```
interface Intf2 :
  generic constant WordLength : unsigned;
  type Word = bool[WordLength];
  host type Time;
  constant C : bool = true;
  input I : bool;
  output O : Word;
end interface
```

Notice that the generic parameter names `T`, `A`, and `F` do not occur any more in `Intf2` since they have been instantiated. From `Intf2`, we can derive a fully defined interface `Intf3` as follows:

```
interface Intf3 :
  extends Intf2 [constant 32 / WordLength]
end interface
```

This makes the name WordLength disappear in Intf3. To keep this convenient name alive, one can write the extension as follows:

```
interface Intf4 :
  constant WordLength : unsigned = 32;
  extends Intf2;
end interface
```

Then, WordLength is defined when Intf2 is extended and it replaces Intf2's WordLength generic parameter during the extension. The final interface Intf4 is fully defined, and it could have been written as follows:

```
interface Intf4 :
  constant WordLength : unsigned = 32;
  type Word = bool[WordLength];
  host type Time;
  constant C : bool = and(true, true);
  input I : bool;
  output O : Word;
end interface
```

If needed, one could have renamed some defined components to give them more explicit names:

```
interface Intf5 :
  constant WordLength : unsigned = 32;
  extends Intf2 [type Word32 / Word]
end interface
```

Then, the resulting interface would have been:

```
interface Intf5 :
  constant WordLength : unsigned = 32;
  type Word32= bool[WordLength];
  host type Time;
  constant C : bool = and(true, true);
  input I : bool;
  output O : Word32;
end interface
```

2.4.4 Using interfaces in ports and runs

When an interface Intf is refererred to in a port declaration "port P : Intf", it is not considered as being extended. Therefore, all its generic parameters must be either already defined in the context or renamed in the port declaration, as in the following declarations:

```
type Word = bool[32];
port P : Intf [type Word / ObjType]
```

Since there is no name exported from Intf to the current context, defined objects of Intf cannot be renamed in a port interface substitution list.

Similarly, the interface of a module M is not extended by a "run M [...]" statement. All generic parameters must be either already defined or explicitly instantiated, and no defined object can be renamed.

2.4.5 Name spaces, visibility, and declaration uniqueness

The named objects in a program are units, data objects, signals, ports, variables, and traps. Units and statements define the *visibility scope* at each point of the program. All the unit names are visible from everywhere in a program. Data objects are visible at a point if they were already declared in the current unit or imported from other units, this directly or indirectly. Signals, variables, and traps follow classical static scoping visibility rules for statements, explained in Chapter **??**.

At each point, there is a single namespace for all the visible objects. Therefore, no two distinct objects can have the same name if they are visible from a common point. Consider the following program:

```
data D1 :
  constant C : unsigned = 213;
end data;



module M1 :
  extends D1;
  output O : unsigned;
  emit ?O <= C
end module



data D2 :
  constant C : unsigned = 347;
end data;

module M2 :
  extends D2;
  output O : unsigned;
  emit ?O <= C
end module

main module Main :
  output O : unsigned;
  run M1;
  pause;
  run M2
end module
```

Since there is no point at which the two different declarations of the `C` identifier are visible, the program is legal and emits 213 at first instant and 347 at second instant. It would be illegal to write:

```
main module Main :
  extends data D1;
  extends data D2;
  output O : unsigned;
  run M1;
  pause;
  run M2
end module
```

because the two distinct definitions of a constant named `C` would be visible in the same scope.

Nevertheless, relying only on visibility rules to give the same name to two different constants can be confusing for the program reader and should we done with care.

Multiple data extension

It is legal to import twice the very same declaration during unit extension, as in the following *multiple extension* scheme:

```
data D :
  constant N : integer = 3;
end data

data D1 :
  extends D;
  type T1 = unsigned<N>;
end data

data D2 :
  extends D;
  type T2 = unsigned<N>;
end data

data D3 :
  extends D1;
  extends D2;
end data
```

Since `D1` and `D2` both extend `D`, they share the very same definition of `N`. On the contrary, the following scheme is illegal:

```
data D1 :
  constant N : integer = 3;
  type T1 : unsigned<N>;
end data

data D2 :
  constant N : integer = 3;
  type T2 : unsigned<N>;
end data

data D3 :
  extends D1;
  extends D2;
end data
```

Here, there are two objects called `N`, which is forbidden even though their definitions happen to coincide.

2.4.6 Host objects are global

Host objects are considered as global to the whole application (in practice, they are most often defined with their name in the host language, although this is not strictly mandatory). Therefore, it is forbidden to declare two host objects with the same name, even if there is no visibility scope where both declarations are visible.

2.5 Forgetting unit components

There is a strict hierarchy between the three kinds of units: any interface unit can be viewed as a data unit of the same name by *forgetting* its signals, and any module unit can be viewed as an interface unit by *forgetting* its behavior. A `data` or `interface` keyword specifies what is retained (not forgotten). Here is an example:

```
interface Intf1 :
  generic type T1;
  input I : T1;
  output O : T1;
end interface

data D2 :
  extends data Intf1;  // forget I and O
  host type T2;
end data

module M3 :
  extends data D2;
  input I : T1;
  output O : T2;
  ...
end module

module M4 :
  extends data M3;  // forget I and O of Intf1 and M3
  input J : T1;
  ...
end module
```

For `D2`, the “`extends data Intf1`” keywords are used to specify that we extend only the data part of `Intf1`, forgetting about its interface signals. Similarly, “`extends data M3`” extends only the data part of `M3` and disregard its interface. Notice that there is no double declaration of `I` and `O` in `M3` since their declarations in `Intf1` were forgotten.

To shorten writing, the keywords `data` or `interface` can be omitted in an `extends` declaration when there is no possible confusion, The rules are as follows, where `Intf` is an interface and `M` is a module assumed to be already declared, for any unit `U` and any unit `V` that contains “`extends U`”:

- If `U` is a data unit, then “`extends U`” is the same as `extends data U`.
- Otherwise, if `V` is a data unit, then “`extends U`” is the same as “`extends data U`”.
- Otherwise, `U` and `V` are interfaces or modules, and “`extends U`” is the same as “`extends interface U`”.

Chapter 3

Data

The data objects handled in Esterel v7_60 are types, constants, functions, and procedures. These objects can be defined either directly in the Esterel program or in the host language to which the Esterel program is compiled. In the latter case, the data objects are called *host data objects* and are known only by their name and type at Esterel level. The value of host constants and the body of host functions and procedures is supposed to be defined in the host language.

Furthermore, Esterel supports generic data definitions for parametric design, as explained in Section **??**.

3.1 Types

Types can be of four kinds:

- *primitive*, i.e. defined by the language;
- *host*, i.e. defined in the host language;
- *generic*, i.e. type parameters of a unit.
- *defined*, i.e. defined in the program from other types and constants.

3.1.1 The bool primitive type

The `bool` type contains the true and false Boolean values, with constants named `true` or `'1` and `false` or `'0`. It is used in test expressions and is also the basis of bitvectors, see Section **??**.

The primitive function `bool2u` takes a `bool` argument and returns the unsigned number 0 for argument `false` and the unsigned number `1` for argument `true`. Its result type is `unsigned<2>`, see Section **??** below.

3.1.2 The unsigned primitive types

For each natural number $M > 0$, the type `unsigned<`M`>` is a primitive type that denotes natural numbers from 0 to $M-1$ included. There is no maximum to the range since Esterel supports arbitrary precision exact unsigned arithmetic, see Section **??**.

Note that M represents the size of the set being encoded, not the number of bits in binary encoding. Therefore, `unsigned<21>` represents all the natural numbers from 0 to 20 included, while the set of numbers one can write with 21 bits is `unsigned<2**21>`.

Of course, it is very common to take a power of 2 for N. We provide the user with the syntactic sugar `unsigned<[`M`]>` for `unsigned<2**`M`>`. Therefore, `unsigned<2**21>` can also be written `unsigned<[21]>`. The number of bits it takes to implement `unsigned<M>` can be computed as `binsize(M-1)`, where the primitive function `binsize(`M`)` returns the number of bits it takes to write the number M in binary, see Section **??**.

To be compatible with more classical type declarations, one can also omit the size declaration altogether and write `unsigned` as a shorth-hand for `unsigned<[32]>`.

In designs, the unsigned types play two different roles:

- At run-time, they are fundamental to implement various kinds of calculations: counting, addressing, coloring pictures, etc.
- At compile-time, they are fundamental to define size arrays, index arrays, count events, parametrize designs, etc.

Esterel unsigned arithmetic is *exact*, which means that '`+`' always means exact addition, '`-`' always means exact subtraction, etc. There is no overfflow in exact arithmetic: operations never drop bits in an implicit way, unlike in classical software languages or HDLs. All bit-dropping calculations must be explicitly done by the user. Unsigned arithmetic is presented in details in Section **??**.

Abstract numbers vs. bit encodings

Esterel unsigned numbers are defined in an abtract way, *independently of any internal representation*. The user or the compiler may choose to use other internal representations than classical binary: onehot, Gray, signed-digit Avizienis encoding, etc. This is why the size parameter M of the `unsigned<`M`>` declaration is a set size and not a bit size: the number 21 takes 5 bits in binary or Gray encoding, 22 bits in onehot encoding, and 10 bits in Avizienis redundant binary encodings.

Of course, bitvector manipulations are also essential, for instance because addresses or values must be extracted from bit frames. The user can convert numbers to bitvector representations and conversely using *encoding* and *decoding* functions such as `u2bin` or `bin2u` for binary encoding, see Section **??**. Three primitive encodings are predefined in Esterel together with their conversion functions: binary, onehot, and Gray. Specific encodings can be added by the user, for example signed digit encoding. Distinguishing the semantics of numbers from their internal representation makes programming higher-level, since algorithms are expressed at a more abstract level. It also yields more optimization potential and an easier link to the newest verification systems that do understand numbers and not only bits: such mathematical algorithms based systems definitely prefer exact arithmetic to truncated one.

Unsigned literals

Unsigned literals can be written in various numbering systems:

- *decimal*, either without prefix as in `15` or with a '`0d`' prefix as in `0d15`;

- *binary*, with a '`0b`' prefix, as in `0b1111`;
- *octal*, with a '`0o`' prefix, as in `0o17`;
- *hexadecimal*, with a '`0x`' prefix, as in `0xF`.

For all forms, one can use '`_`' underline characters to make the value more readable, as for `123_456` and `0b_110_001`. This character can appear at any place. It is ignored when computing the constant value.

The type of an unsigned constant M declared in Esterel is the minimal unsigned type that contains it, which is `unsigned<`M`+1>`. This type is automatically extended as needed to larger unsigned types during arithmetic operations, see Section **??**.

Only the numerical value of a literal matters to define its type. For instance, since leading 0's don't change the value, the literals `0xA` and `0x0A` have the same type `unsigned<11>`. See Section **??** for the difference with bitvector constants, which are not numerically evaluated: `'xA` has type `bool[4]` while `'x0A` has type `bool[8]`.

Unsigned constant tight declarations

When declaring an unsigned defined constant, one gives the value of the constant after the '`=`' sign. To control intermediate result sizes in expressions that use this constant, it is always a good idea to declare the type of the constant as the smallest unsigned type that contains its value. This type is `unsigned<`N`+1>` if the value is N. Here are explicit declarations of this kind, where `A` and `B` are already declared unsigned constants:

```
constant WordSize : unsigned<33> = 32;
constant C : unsigned<A+B+2> = A+B+1;
```

To make such declarations simpler, Esterel v7 provides the user with a simpler defined unsigned constant *tight declaration syntax*, where the type is simply declared as `unsigned<>` to abbreviate `unsigned<`N`+1>` if the value is N. The previous examples can be rewritten as follows:

```
constant WordSize : unsigned<> = 32;
constant C : unsigned<> = A+B+1;
```

The tight declaration syntax is available only for defined unsigned constant declaration and cannot be used elsewhere. Any constant expression can be used in the defining expression.

3.1.3 The signed primitive types

Given a positive integer M, the type `signed<`M`>` denotes the set positive or negative integers x such that $-M \leq x < M$, i.e. the range $[-M, M-1]$. Note that the cardinality of this set is $2M$. The notation `signed<[`M`]>` is a shorthand for `signed<2**(`M`-1)>`, which exactly denotes the set of signed integers one can represent with M bits in 2's complement binary notation. One can use `integer` as a synonym for `signed<[32]>`.

As for unsigned arithmetic, signed arithmetic is always exact and no implicit bit-dropping is performed. Signed arithmetic is presented in Section **??**.

Although the language does not enforce signed numbers to be represented in 2's complement binary form, the bounds are chosen to conform to it, and predefined functions `s2bin` and `bin2s` converts signed numbers to and from bitvectors in this form.

There are no signed literals. Signed constants can be built from unsigned literals by adding the `+` or `-` signs, as for `+3` or `-5`, see Section ??.

For compatibility with common use, we provide the user with the `integer` keyword as an abbreviation of `signed<[32]>`. Beware if you lazily use this type for synthesis, you may generate unwanted 32-bit registers.

Signed constants tight declarations

Signed constants tight declarations are similar to unsigned constants tight declarations described in Section ??. When declaring an signed defined constant, one can use the tight type denotation `signed<>` to mean the smallest signed type that contains the value of the expression. If the expression is E, that type is computed as follows:

- If E evaluates to a positive number n, the type is `signed<`n`+1>`
- If E evaluates to a negative number $-n, n > 0$, the type is `signed<`n`>`.
- If E cannot be completely evaluated in the current scope because it contains generic constants, the type is `signed<abs(`E`)+1>`, where `abs` be the is the absolute value function described in Section ?? below. the '`+1`' term is needed to handle the case where E is positive.

Assume that `A` and `B` are previously declared signed constants with known values, and consider the following declarations:

```
constant C : signed<> = 5;
constant D : signed<> = -5;
constant E : signed<> = A+B;
```

The type of `C` is `signed<6>`, while the type of D is `signed<5>`. If the value v of `A+B` is positive, the type of `E` is `unsigned<`$v + 1$`>`. If v is negative, let $w = -v$; then the type of `E` is `unsigned<`w`>`.

The tight signed declaration syntax is available only for defined constant declaration and cannot be used elsewhere.

3.1.4 The float primitive type

The `float` type denotes positive or negative floating-point numbers. The constants are written as in C++ or Java: `12.3f`, `.123E2F`, or `-1.23E-1F`. The representation and range of supported floats depends on the host language and CPU architecture. It should agree with the IEEE floating point norm.

3.1.5 The double primitive type

The `double` type denotes positive or negative double-precision floating-point numbers. The constants are written as in C++ or Java: `12.3`, `.123E2`, or `-1.23E-1`. The representation and range of supported doubles depends on the host language and architecture.

.

3.1.6 The string primitive type

The `string` type is a primitive type. It denotes character strings. Constant strings are written between double quotes: `"a␣string"`, with doubled double quotes as in `"a␣""␣double␣quote"`. The empty string is written `""`.

3.1.7 Enum types

An *enum* type defines a list of distinct values, which are given symbolic names and called *enum constants*. For example:

```
type Color = enum {Red, Blue, Green};
```

Enum constants are associated with signed or unsigned numbers, which are called *enum codes*, and which must be all distinct. By default, with no explicit enum code declaration, enum codes shall be the unsigned numbers 0, 1, 2, 3, etc. For example, the `Color` type above has code 0 for `Red`, code 1 for `Blue`, and code 2 for `Green`.

The user may explicitly specify the enum code of every enum constant by specifying an unsigned literal preceded by an optional '`+`' or '`-`' symbol:

```
type CardSuit = enum {
  Clubs = 1,  // explicit code 1
  Diamonds,   // implicit code 2
  Hearts = 4, // explicit code 4
  Spades      // implicit code 5
};

type CarManualGear = enum {
  Rear,      // implicit code -1
  Idle = +0, // explicit code 0
  First,     // implicit code +1
  Second,    // implicit code +2
  Third,     // implicit code +3
  Fourth,    // implicit code +4
  Fifth      // implicit code +5
};

type CarAutomaticGear = enum {
  Rear,          // implicit code -1
  Park = +0,     // explicit code 0
  Standard = +2, // explicit code +2
  Sport          // implicit code +3
};
```

If any of the user enum codes uses a '`+`' or '`-`' symbol, then the type is a called *signed enum type*. Otherwise, the type is an *unsigned enum type*. If no code is specified, the type is considered as an *unsigned enum type*.

For an unsigned enum type, the code of the first constant shall be 0 if it is not specified by the user. For any other constant, if the code is not specified by the user, it shall be the code of the previous constant in the list incremented by one; if the code is specified by the user, it shall be strictly superior to the explicitly or implicitly specified code of the previous constant in the list.

For a signed enum type, let us note C the first constant whose code is specified by the user and let us note x this code. The code of the constant that is positioned just before

C in the list shall be *x-1*, the code of the constant just before the latter constant shall be *x-2*, and so on until the beginning of the list. The codes of enum constants after *C* shall be incremented by one from *C* until another user-specified code is encountered; in that case the user code shall be strictly superior to the explicit or implicit code of the previous constant. The incrementation by 1 shall restart from the new user-defined value.

We define the *size* of an unsigned (resp. signed) enum type as the size of the smallest unsigned (resp. signed) type that can hold all the enum codes. For example, the `Color` type above is an unsigned enum type of size 3; the `CardSuit` type above is an unsigned enum type of size 6; the `CarManualGear` type above is a signed enum type of size 6; the `CarAutomaticGear` type above is a signed enum type of size 4.

Equality '`=`' and unequality '`<>`' can be used on an `enum` type: "`Red = Red`" is true, "`Red <> Red`" is false, "`Red = Green`" is false, and "`Red <> Green`" is true.

The `mux` operator can be used to build enum expressions, see Section **??**. Here is an example:

```
output TrafficLight : Color;
...
emit ?TrafficLight <= mux(isGreen, Green, Red);
```

Conversions between unsigned enum types and unsigned types can be performed using the functions `enum2u` and `u2enum`; conversions between signed enum types and signed types can be performed using the functions `enum2s` and `s2enum`, see Section **??**.

3.1.8 Host types

An *host type* is a type whose contents and implementation are not specified in Esterel but only in the host language to which the type is compiled. A host type is declared using the `host` keyword, as follows:

```
host type Time;
```

The `Time` type can be implemented in C by a structure with hours, minutes, and seconds fields, using the usual Babylonian carry propagation. The interest of host types is that such datapath details can be irrelevant for the behavior of an Esterel program and better handled in separate host libraries to separate concerns.

One can declare a constant, a signal, or a variable of host type. These objects are then called *host objects*, and they are the only way to manipulate objects of the type. One can compare abstract objects by '`=`' or '`<>`', and pass host objects to functions, procedures, or units. Here are examples:

```
module M :
  host type T;
  host constant C : T;
  host function F (T) : T;
  input I : T;
  output O : T;
  sustain ?O <= F(?I) if ?I <> C
end module
```

3.1.9 Defined types

A *defined type* is a type declared to be equal to another type, which must be primitive or declared beforehand:

```
type Temperature = unsigned<90>;
type Word = bool[32];
```

The types are identical for any usage, except for bitvector types for which *maps* can be defined as explained in Section **??**.

3.1.10 Array types

Data arrays can be defined from types by adding dimensions. The dimension expressions must be statically evaluable unsigned or positive signed numbers, see Section **??**. Arrays and arrays of arrays are supported up to any size and depth.

Array types can be named or unnamed. An unnamed array type directly appears in the declaration of an Esterel object:

```
input Opcode : bool[8];
var X : integer[3][5] in ... end
```

A named array type is an array type appearing in a type definition:

```
type Image = unsigned<[8]> [100][100];
type Word = bool[32];
constant MemorySize : unsigned;
type Memory = Word[MemorySize];
```

Notice that objects of `Memory` type are arrays of arrays declared in an indirect way. When indexing such indirectly defined arrays, the indices appear in the definition dependency order: `Memory[i]` is a `Word`, and `Memory[i][j]` is the `j`-th bit of this word. This indexing scheme defines the *unfolded type* of the indirect type, which is obtained by successively removing all defined type declarations. Here, the unfolded type of `Memory` is `bool[MemorySize][32]`, *not* `bool[32][MemorySize]`.

Because of the type-prefix notation `bool[32]`, array type unfolding is a little awkward:

```
   Word[MemorySize]
→  (bool[32])[MemorySize]
→  bool[MemorySize][32]
```

To avoid any confusion, we forbid to define arrays from direct arrays, thus to write types of the form `(bool[32])[MemorySize]`. We enforce the use of intermediate defined array types such as `Word`.

Notice that the same unfolding rules are clearer with a type-postfix notation *a la* VHDL. There, one simply has

```
   array MemorySize of (array 32 of bool)
→  array MemorySize of array 32 of bool
```

Two array types are considered identical if and only if they have the same base type, the same number of dimensions and the same dimensions, this after unfolding for indirect array types.

Data arrays can be indexed and sliced as explained in Section **??**. Their relation with signal arrays is presented in Section **??**.

Array literals

Array literals are used to define values for constant arrays, and they can be used in array expressions. Their syntax follows that of System Verilog [**?**]. First, for single-dimensional arrays, one can first give a list of values between curly brackets, the length of the list being exactly the array dimension:

```
constant C : bool[3] = {'0, '1, '0};
```

One can repeat a subrange by adding a constant unsigned expression acting as a repetition factor followed by a bracket and a sublist:

```
constant C1 : bool[3] = {'1, 2{'0}};    // same as {'1,'0,'0}
constant C2 : bool[N] = {(N/2){'0,'1}}
                        // same as {'0,'1,'0,'1,'0,'1...}
                        // provided N is even
```

After expansion of the repeated elements, the literal size must match the array size.

For arrays of arrays, one nests subarray definitions, first index varying slowest, with mandatory dimension match at all levels. Repetition factors can be applied to whole subarrays. Here are examples:

```
var V : unsigned[2][2] = { {1,2}, {3,4} } in ... end
    // V[0][0]=1, V[0][1]=2, V[1][0]=3, V[1][1]=4

signal S : integer[5][5] init {4{{5{0}}}, {-1,-2,-3,-4,-5}} in
             // lines 0-3 filled with 0's
             //line 4 filled by decreasing numbers from -1 on
 pause;
 emit ?S <= {2{{5{-1}}}, 3{{5{0}}}}
   // lines 0-1 filled with -1's, lines 2-4 filled with 0's
end signal
```

For definition of constants and for initialization of variables and signals, one can use a single literal instead of an array constant to initialize all components of the array with a single value:

```
constant C : unsigned[10] = 0; // shorthand for {10{0}}
type Word = bool[32];
constant B : Word[100] = '0;   // shorthand for {100{{32{'0}}}
signal Mem : Word[100] init '0 in ... end
var V: Word[100] := INIT_WORD in ... end
```

The initialization literal is limited to be either a data constant literal such as `0`, `true`, or `'x01` or a constant name such as `INIT_WORD`. Its type must match the array base type.

3.1.11 Bitvectors

Single dimensional arrays of Booleans are called *bitvectors*. They play a very special role in hardware or low-level software designs.

Bitvector literals

For bitvector constants, one can use an alternative syntax similar to the unsigned constant syntax presented in Section ??, replacing the leading `0` by a quote symbol. The `'b`, `'o`, and `'x` forms are allowed, while `'d` is disallowed. As for unsigned constants, one can use the '`_`' dummy separators, as in `'b00_01_10_11`.

The bit width can also be given explicitly before the quote symbol: `8'b11` is the same as `'b00000011`, `5'o23` is the same as `'b10011`, and `10'xF37` is the same as `'b1100110111`; note that the unused higher-order 1's of `F` are simply discarded in the latter case and provoke no error.

If the bit width is not explicitly given, it is computed by multiplying the number of digits in the constant by the basis bit width, taking account of the leading `0`'s (unlike for unsigned constants). Therefore, `'o27` is six-bits wide, while `'x01234567` is 32-bits wide.

There is a well-known but unfortunate reversal of endianness w.r.t. the C-like notation.

- The literal `'b1110001` is a synonym to the array literal `{'1,'0,'0,'0,'1,'1,'1}`. Notice the bit reversal w.r.t. array literals: low-order bits are first in the C-based array literal notation, last in the bitvector array notation;
- The literal `'o027` is a synonym to the array literal `{'1,'1,'1,'0,'1,'0, '0, '0, '0}`. Notice that the first 0's do matter, unlike for the unsigned constant 0o027;
- The literal `7'x6E` is a synonym to the array literal `{'0,'1,'1,'1,'1,'1,'0}`.

3.1.12 Generic types

A *generic type* is declared as follows:

```
generic type T;
```

If a data, interface, or module unit contains a generic type declaration, it becomes a generic unit. The generic type can be instantiated as explained in Section ??. All generic types must be instantiated for the main module to be executable.

3.2 Constants

Constants can be defined, host, or generic, just as types. A constant has a type, which must be primitive or declared beforehand.

3.2.1 Defined constants

A *defined constant* is a constant whose value is given at declaration time, after the type and a '`=`' symbol.

```
constant Threshold : unsigned = 223;
constant GlobalSize : unsigned = MemorySize * WordSize;
```

The value is given by a static expression containing only literals, constants, and primitive operators, with all objects declared beforehand.

There is a special tight notation described in Section ?? and Section ?? to help sizing unsigned and signed constants.

3.2.2 Host constants

A *host constant* is a constant which will be defined in the host language. Its type cannot be generic. A host constant is declared using the `host` keyword:

```
host constant Threshold : unsigned<[16]>;
host constant Midnight : Time;
```

A host constant is known only by its name, nothing being known about the value.

3.2.3 Generic constants

A *generic constant* is a constant declared by its name and type after the `generic` keyword:

```
generic constant C : T;
```

The type itself can be defined, host, or generic.

If a data, interface, or module unit contains a generic constant declaration, it becomes a generic unit. The generic constant can be instantiated as explained in Section **??**. All generic constants must be instantiated for the main module to be executable.

3.3 Functions

A function takes typed arguments by value and returns a typed value. A function is supposed to be computed instantaneously and to have no side-effects.

The argument types are in a parenthesized list, the result type appearing after a colon. Non-primitive types must be declared beforehand. They can be arbitrary, including generic. There are predefined functions, host functions, and generic functions.

3.3.1 Predefined functions

The predefined functions are as follows:

- `binsize`(m), where m is a natural number, returns the number of bits it takes to write m in binary.
- `bool2u` converts a Boolean to an unsigned number, see Section **??**.
- `bin2u`, `onehot2u`, `code2u`, `u2bin`, `u2onehot`, and `u2code`, convert bitvectors to unsigned numbers and conversely, see Section **??**.
- `bin2s` and `s2bin` convert bitvectors to signed numbers and conversely, see Section **??**.
- `mux` chooses between values according to a condition, see Section **??**.
- `lcat` and `mcat` concatenate bitvectors, see Section **??**.
- `reverse` reverses bitvectors, see Section **??**.

3.3.2 Host functions

A *host function* is a function whose body is defined externally in the host language and unknown at Esterel level. The arguments and result types cannot be generic: they must be primitive or declared beforehand. A host function is declared using the `host` keyword:

```
host function IncrementTime (Time) : Time;
```

The declaration declares the name and type of the host function, its definition remaining unknown at Esterel level.

3.3.3 Generic functions

A *generic function* is a function declared after the `generic` keyword by its name and its argument and result types, which can be generic as well:

```
generic function Fun (T1, T2) : T;
```

The argument and result types can also involve generic constants. Thus, the following is allowed:

```
generic type T;
generic constant Size : unsigned;
generic function Transpose (T[Size, Size]) : T[Size, Size];
```

If a data, interface, or module unit contains a generic function declaration, it becomes a generic unit. The generic function can be instantiated as explained in Section **??** provided types match at instantiation time. All generic functions must be instantiated for the main module to be executable.

3.4 Procedures

A procedure has a list of `in`, `out`, and `inout` arguments; it is declared by its name, and its arguments types annotated with the corresponding access mode keyword. The procedure is supposed to be executed instantaneously. When executed, it reads its `in` and `inout` arguments and modifies (side-effects) its `out` and `inout` arguments, which must be variables. There are host procedures and generic procedures. A procedure is called by the `call` Esterel statement, see Section **??**. There is no provision to define the code of a procedure in Esterel. Therefore, the code of the procedure will always be written in the host language. Because Esterel is concurrent, side effects other than assigning out or inout parameters are fully under the user control, and their effect cannot be predicted if procedures sharing user data are called concurrently.

3.4.1 Host procedures

A *host procedure* is a procedure whose body is defined externally in the host language and unknown at Esterel level. The arguments types cannot be generic: they must be primitive or declared beforehand. A host procedure is declared using the `host` keyword:

```
host procedure UpdateTime (inout Time, in integer);
```

3.4.2 Generic procedures

A *generic procedure* is a procedure declared using the `generic` keyword:

```
generic procedure P (in T1, out T2, inout unsigned);
```

The types can be generic. If a data, interface, or module unit contains a generic procedure declaration, it becomes a generic unit. The generic procedure can be instantiated as explained in Section **??** provided types match at instantiation time. All generic procedures must be instantiated for the main module to be executable.

Chapter 4

Arithmetic, Enum, and Bitvectors operations

4.1 Boolean operators

Remember that the Boolean literals are `false` or `'0` and `true` or `'1`. The Boolean operations are unary `not`, left-associative '`and`', '`or`', and '`xor`', implication '`=>`', equivalence '`<=>`', exclusion '`#`', plus the generic `mux` operator.

Conjunction `and` and disjunction `or` are left-associative, with higher priority for `and` as usual. Implication is right-associative: x `=>` y `=>` z means x `=>` $(y$ `=>` $z)$. Equivalence '`<=>`' is synonym to equality '`=`' but makes use of the often clearer standard mathematical notation. It is left associative, just as for '`=`'.

Exclusion is truly n-ary: x `#` y `#` z means that x, y, and z are exclusive, i.e. that no two of them can be true at the same time. Beware, x `#` y `#` z is completely different from x `#` $(y$ `#` $z)$, which means "x is exclusive with the result of the exclusivity predicate of y and z".

The $\texttt{mux}(b, e_1, e_2)$ operator takes a Boolean expression b and two expressions e_1 and e_2 of the same arbitrary type, which can also be an array type. It returns the value of e_1 if b evaluates to true, e_2 otherwise.

$$\texttt{mux}(b, e_1, e_2) \quad = \quad \begin{cases} e_1 & \text{if } b = \texttt{true} \\ e_2 & \text{if } b = \texttt{false} \end{cases}$$

If the expressions e_1 and e_2 are arrays, an array is returned and the whole `mux` expression is considered to be an array expression, see Section **??**.

Note that the first argument is a single Boolean expression, not an array expression. Extension to Boolean arrays as first argument requires the `[mux]` array operator notation presented in Section **??**.

The `bool2u` function takes a `bool` and returns an unsigned number of type `unsigned<2>`, defined as follows:

$$\begin{aligned} \texttt{bool2u('0)} &= 0 \\ \texttt{bool2u('1)} &= 1 \end{aligned}$$

Notice the equality $\texttt{bool2u}(b) = \texttt{mux}(b, 1, 0)$.

4.2 Unsigned arithmetic

The unsigned types support the following internal operations that return unsigned numbers: addition '`+`', subtraction '`-`', multiplication '`*`', division '`/`', modulo (remainder) '`mod`', power '`**`', binary size '`binsize`', saturation `sat<`M`>` and binary truncation `trunc<[`M`]>`, up to any size. The size assertion operation `assert<`M`>` described in Section **??** asserts that a value is in a given unsigned type and provokes an error otherwise. Unsigned numbers support all standard Boolean comparisons `=`, `<>`, `<`, `<=`, `>`, and `>=`. Conversions to and from signed numbers and bitvectors will be studied in Section **??** and Section **??**. Note that there is no `+` and `-` unary operations on unsigned numbers, since these operations convert unsigned numbers into signed ones. All operations can be applied to unsigned of any size.

We use m, n to denote unsigned numbers, M, N to denote type sizes, and e_m, e_n to denote unsigned expressions of respective types `unsigned<`M`>` and `unsigned<`N`>`. Expression typing is denoted e_m : `unsigned<`M`>`. The result type is always computed to be the smallest unsigned type that fits the result, with the rules given below. Therefore, size extension can be viewed as automatic.

4.2.1 Unsigned addition

Unsigned addition '`+`' takes two unsigned m, n and returns an unsigned with value $m+n$. For expression type-checking, the result type is the smallest possible to accomodate the result, the worst case being $(M-1) + (N-1)$:

$$e_m : \texttt{unsigned<}M\texttt{>},\ e_n : \texttt{unsigned<}N\texttt{>} \quad\rightarrow\quad e_m \texttt{+} e_n : \texttt{unsigned<}M+N-1\texttt{>}$$

Note that addition never overflows, for any unsigned size. There is no implicit "bit-dropping". The sizing rule being associative, the size of the result of a n-ary addition is independent of any subexpression calculation order. Therefore, there is no difference in value and size between $(e_m + e_n) + e_p$ and $e_m + (e_n + e_p)$.

4.2.2 Unsigned subtraction

Unsigned subtraction '`-`' takes two unsigned m, n such that $m \geq n$ and returns an unsigned with value $m-n$. If m is strictly smaller than n, then m `-` n is undefined and the operation raises a run-time error, see Section **??**.

The result type is the smallest possible to accomodate the result, i.e. that of the first expression since the second one can be 0:

$$e_m : \texttt{unsigned<}M\texttt{>},\ e_n : \texttt{unsigned<}N\texttt{>} \quad\rightarrow\quad e_m \texttt{-} e_n : \texttt{unsigned<}M\texttt{>}$$

Unsigned subtraction is left-associative: $e_m \texttt{-} e_n \texttt{-} e_p$ means $(e_m \texttt{-} e_n) \texttt{-} e_p$.

4.2.3 Unsigned multiplication

Unsigned multiplication takes two unsigned m, n and returns an unsigned with value $m \times n$. For the result type size, the worst case is $(M-1) * (N-1)$:

$$e_m : \texttt{unsigned<}M\texttt{>},\ e_n : \texttt{unsigned<}N\texttt{>} \quad\rightarrow\quad e_m * e_n : \texttt{unsigned<}(M-1)(N-1)+1\texttt{>}$$

As for addition, multiplication never fails and is always exact, without any bit loss. It is fully associative.

4.2.4 Unsigned power

Unsigned power takes two unsigned m, n and returns an unsigned with value m^n. For the result type size, the worst case is $(M-1)\,\texttt{**}\,(N-1)$:

$$e_m : \texttt{unsigned<}M\texttt{>},\ e_n : \texttt{unsigned<}N\texttt{>} \quad\rightarrow\quad e_m \,\texttt{**}\, e_n : \texttt{unsigned<}(M-1)^{(N-1)}+1\texttt{>}$$

Power associates to the right: $e_m \,\texttt{**}\, e_n \,\texttt{**}\, e_p$ is $e_m \,\texttt{**}\, (e_n \,\texttt{**} e_p)$

4.2.5 Unsigned division

Unsigned division '`/`' takes two unsigned m, n such that $n \neq 0$ and returns an unsigned with value the integer quotient m/n. If n has value 0, then $m\,\texttt{/}\,n$ is undefined and raises a run-time error, see Section **??**.

For subexpression typing, the result type is that of the first expression, since the second one can be 1:

$$e_m : \texttt{unsigned<}M\texttt{>},\ e_n : \texttt{unsigned<}N\texttt{>} \quad\rightarrow\quad e_m \,\texttt{/}\, e_n : \texttt{unsigned<}M\texttt{>}$$

4.2.6 Unsigned modulo

Unsigned modulo `mod` takes two unsigned m, n such that $n \neq 0$ and returns the remainder of the division of m by n. If n has value 0, then $m\,\texttt{/}\,n$ is undefined and raises a run-time error, see Section **??**.

For subexpression typing, since $m \,\texttt{mod}\, n$ is strictly less than n and n is strictly less than N, the size of the result type is $N-1$:

$$e_m : \texttt{unsigned<}M\texttt{>},\ e_n : \texttt{unsigned<}N\texttt{>} \quad\rightarrow\quad e_m \,\texttt{mod}\, e_n : \texttt{unsigned<}N-1\texttt{>}$$

4.2.7 Unsigned binary size

Unsigned binary size $\texttt{binsize}(m)$ takes an unsigned m and returns the number of bits necessary to write m in binary:

$$\texttt{binsize}(m) \quad=\quad \begin{cases} 1 & \text{if } m = 0 \\ n & \text{if } 2^{n-1} \leq m < 2^n \end{cases}$$

The `binsize` function is mostly used at compile-time to compute the size of the bitvector implementing an unsigned type $\texttt{unsigned<}M\texttt{>}$ in binary, which is $\texttt{binsize}(M-1)$. Notice that $\texttt{binsize}(2^M-1) = M$, which is consistent with the fact that M bits are needed to implement the type $\texttt{unsigned<[}M\texttt{]>} = \texttt{unsigned<}2^M\texttt{>}$.

Of course, the `binsize` function can also be used for other purposes.

The typing rule for type-checking `binsize` subexpressions itself uses the function `binsize`:

$$e_m : \texttt{unsigned<}M\texttt{>} \quad\rightarrow\quad \texttt{binsize(}e_m\texttt{)} : \texttt{unsigned<binsize}(M-1)+1\texttt{>}$$

4.2.8 Unsigned saturation

Unsigned saturation `sat<`N`>` takes a positive compile-time constant N and an unsigned m and returns the largest unsigned in `unsigned<`N`>` that is less than or equal to m. Therefore, the definition is as follows:

$$\texttt{sat<}N\texttt{>}(m) \quad = \quad \begin{cases} m & \text{if } m < N \\ N-1 & \text{if } m \geq N \end{cases}$$

The type-checking rule is as follows:

$$e_m : \texttt{unsigned<}M\texttt{>} \quad \rightarrow \quad \texttt{sat<}N\texttt{>(}e_m\texttt{)} : \texttt{unsigned<}N\texttt{>}$$

Unsigned saturation is important for saturated arithmetic, for instance color handling in photofinishing.

The `sat` operator can also be used to change the type of an unsigned value into a bigger one. Given an expression e of type `unsigned<M>` and given `N` $>$ `M`, the expression `sat<N>(`e`)` has the same value as e but has type `unsigned<N>`

4.2.9 Unsigned binary truncation

Remember that `unsigned<[`N`]>` is a shorthand for `unsigned<2**`N`>`. Binary truncation `trunc<[`N`]>` takes a positive constant N and an unsigned m of any size and returns the remainder of the division of m by 2^N, i.e. the value of "m `mod (2**` N`)`". In the binary representation, one simply drops the bits above N, brutally bringing back m to N binary bits. Truncation is easy to express using bitvector conversions:

$$\texttt{trunc<[}N\texttt{]>}(m) \quad = \quad \begin{cases} m & \text{if } m < 2^N \\ \texttt{bin2u}(\texttt{u2bin}(m)[0..N-1]) & \text{if } m \geq 2^N \end{cases}$$

The `trunc` operation is only available for sizes that are powers of 2, and we make the special brackets `<[ ]>` mandatory here to avoid any confusion.

The type-checking rule is as follows:

$$e_m : \texttt{unsigned<}M\texttt{>} \quad \rightarrow \quad \texttt{trunc<[}N\texttt{]>(}m\texttt{)} : \texttt{unsigned<[}N\texttt{]>}$$

For an application example, consider addition of two N-bits numbers. The default Esterel calculation is exact:

$$m, n : \texttt{unsigned<[}N\texttt{]>} \quad \rightarrow \quad m\texttt{+}n : \texttt{unsigned<}\,2^{N+1}-1\texttt{>}$$

for instance, if $N = 3$, one computes $7 + 6 = 13$: `unsigned<15>`. On the contrary, the usual HDL carry-dropping calculation returns 5 : `unsigned<8>` if the result is to be put on 3 bits. In Esterel, we must explicitly use `trunc` for such a truncated addition, writing `trunc<[3]>(7+6)` $= 5$. The associated type formula is:

$$m, n : \texttt{unsigned<[}N\texttt{]>} \quad \rightarrow \quad \texttt{trunc<[}N\texttt{]>(}m\,\texttt{+}\,n\texttt{)} : \texttt{unsigned<[}N\texttt{]>}$$

Since $7 + 6 = 5$ is not very good for program verification, we definitely prefer arithmetic to be exact by default. Therefore, we require carry-dropping to be explicit.

4.2.10 Unsigned assert

Given two natural numbers M, N and an expression e_u of type `unsigned<`N`>` of value v, the expression `assert<`M`>(`e_u`)` is of type `unsigned<`M`>`. It returns v if $v < M$ and provokes a run-time error otherwise, see Section **??**.

Numerical `assert<M>` is used to shrink down the type of a computation result for which an upper bound is known for dynamic reasons that are outside the scope of static type-checking. Unlike C-like casting, `assert<`M`>` is completely safe since it does not change the meaning of its argument and generates a verification condition.

For a typical usage instance, consider converting a 4-bit bitvector `B` that represents the binary writing of an unsigned number m into that number. The conversion expression is `u2bin(B)`, detailed in Section **??**. Its type is `unsigned<[4]>` = `unsigned<16>`. Assume now that $0 \leq m < 13$ is guaranteed by the design specification. We want the result to be in `unsigned<13>`. By writing `assert<13>(u2bin(B))`, we appropriately restrict the type in a safe way. If, because of some bug, `B` does not respect the specification, we can detect that fact at run-time since a run-time error is generated or statically using formal verification.

For another usage instance, consider a signal `S` of type `unsigned<`M`>`. One often writes emissions like

```
emit ?S <= pre(?S)+1
```

Strictly speaking, these expressions are ill-typed, since the right-hand side is in the type `unsigned<`$M+1$`>` while the left-hand side is in `unsigned<`M`>`. With rigid type-checking, the program should be rejected. One should write the following:

```
emit ?S <= assert<M-1>(pre(?S))+1
```

In tolerant type-checking, the language tolerates the emission but implicitly encloses the right-hand side within `assert<`M`>`:

```
emit ?S <= assert<M>(pre(?S)+1)
```

which is actually less clever than the form above since the adder may need to use one more bit.

A similar case is the indexing of an array of size M with an unsigned expression of a bigger type, which is tolerated in tolerant type-checking and for which compilers should generate verification conditions for simulators or formal verifiers.

4.2.11 Unsigned comparisons

The comparison operators are classical: `=`, `<>` (not equal), `<`, `<=`, `>`, and `>=`. They take two unsigned numbers of any size and return a Boolean.

4.2.12 Unsigned size checks

There are five cases where the value of an unsigned expression *exp* can be assigned to or passed to an object of unsigned type: variable assignment, signal emission, function call, procedure call, and module run statement:

```
V := exp                  // V unsigned variable
emit ?S <= exp            // S unsigned signal
...f(exp, ...)...         // unsigned argument
```

```
call P(..., exp, ...)     // unsigned argument
run Sub [S / I]           // S unsigned signal
```

In practice, it would be too cumbersome to systematically add `sat` or `trunc` operations to guarantee that no run-time overflow can occur. Therefore, the language allows the user to write an assignment of a value v in `unsigned<M>` to an object of type `unsigned<N>` with M $>$ N. Such an assignment is correct only if $v <$ N. Compilers and programming environments may check value assignment correctness in two ways:

- *statically*, by checking at compile-time that the value received is guaranteed to be in the required range. This is obvious if the value type is smaller than the receiver type; it can also be checked using partial evaluation techniques in other cases.
- *dynamically*, by providing run-time checks (at least in simulation mode). This should be done only if static checks are not sufficient.

Similarly, when indexing a signal or data array as for `S[`m`]`, the index can be in a type larger than the index range of the array dimension, but it is an out-of-bounds error if its actual value is bigger than or equal to the dimension, see Section **??**.

Nevertheless, compilers should always be able to issue warnings for dubious operations.

4.3 Signed arithmetic

The signed types support the following internal operations: unary plus '`+`', unary minus '`-`', addition '`+`', subtraction '`-`', multiplication '`*`', division '`/`', power '`**`', saturation '`sat<`N`>`', and binary-truncation '`trunc<[`N`]>`', up to any size. Notice that there is no signed modulo operation, since this operation has no universally adopted definition yet. Signed types support the assertion `assert<`M`>` to assert that a signed value is within a signed type and the assertion `assert_unsigned<`N`>` to assert that a signed value v satisfies $0 \leq v < N$.

Signed numbers support all classical Boolean comparisons. All operations can be applied to signed of any size, with exact signed arithmetic.

In practice, one most often uses full binary-encoded signed sets, i.e. types `signed<[`M`]>`. However, finer signed types naturally appear when tightly typing subexpression, and they can also be useful in applications.

The relation between signed numbers and unsigned numbers is as follows: the unary operations '`+`' and '`-`' convert unsigned numbers into signed ones; the `abs`(x) absolute value function returns the unsigned absolute value of x; the x`>=0` test returns the sign of x as a Boolean (one can use `mux` to convert it to a 0 / 1 sign if needed).

Unsigned numbers are automatically converted to signed numbers of the appropriate size by the unary operations '`+`' and '`-`' and when used together with signed numbers in arithmetic operations. Conversions to and from bitvectors by `s2bin` and `bin2s` is studied in Section **??**.

We use x, y to denote signed numbers and M, N to denote their type size. We use e_x and e_y to denote signed subexpressions of respective types `signed<`M`>` and `signed<`N`>`, denoting subexpression typing by e_x : `signed<`M`>`. As for unsigned numbers, the result signed type of an expression is always computed to be the smallest signed type that fits the result, with the rules given below. Therefore, size extension can be viewed as automatic.

4.3.1 Signed unary plus and minus

When applied to a signed number x : `signed<`M`>`, unary plus `+`x simply returns x in the same type; when applied to an unsigned number m : `unsigned<`M`>`, unary plus `+`m returns the signed value m; that value belongs to the type `signed<`M`>`, which is the smallest one to contain it. Here are the typing rules:

$$\begin{aligned} e_x : \texttt{signed<}M\texttt{>} &= \texttt{+}\,e_x : \texttt{signed<}M\texttt{>} \\ e_m : \texttt{unsigned<}M\texttt{>} &= \texttt{+}\,e_m : \texttt{signed<}M\texttt{>} \end{aligned}$$

When applied to a signed number x : `signed<`M`>`, unary minus `-`x returns the opposite of x, which is in `signed<`$M+1$`>` since that type is the smallest one to contain $M = -(-M)$. When applied to an unsigned number m : `unsigned<`M`>`, unary minus `-`m returns the opposite of m, which is in `signed<`$M-1$`>` since the worst case is $-(M-1)$. Here are the typing rules:

$$\begin{aligned} e_x : \texttt{signed<}M\texttt{>} &= \texttt{-}\,e_x : \texttt{signed<}M+1\texttt{>} \\ m : \texttt{unsigned<}M\texttt{>} &= \texttt{-}\,m : \texttt{signed<}M-1\texttt{>} \end{aligned}$$

4.3.2 Signed addition and subtraction

Signed addition '`+`' and subtraction '`-`' take two signed numbers x, y and returns their sum $x+y$ and their difference $x-y$. The result type is `signed<`$M+N$`>` in both cases since the worst cases are respectively $(-M)$`+`$(-N)$ and $(M-1)$`-`$(-N)$:

$$\begin{aligned} e_x : \texttt{signed<}M\texttt{>},\ e_y : \texttt{signed<}N\texttt{>} &\rightarrow\ e_x \texttt{+} e_y : \texttt{signed<}M+N\texttt{>} \\ &\rightarrow\ e_x \texttt{-} e_y : \texttt{signed<}M+N\texttt{>} \end{aligned}$$

Notice that e_x`+`e_y has type `signed<[`$N+1$`]>` for e_x, e_y : `signed<[`N`]>`, and that the same property holds for subtraction, which is always defined. This contrasts signed number vs. unsigned ones, since none of these properties is true for the latter.

The sizing rule being associative, the size of the result of a n-ary addition is independent of any subexpression calculation order. Therefore, there is no difference in value and size between `(`e_x`+`e_y`)+`e_z and e_x`+(`e_y`+`e_z`)`. As for unsigned addition, signed subtraction is left-associative: e_x`-`e_y`-`e_z is `(`e_x`-`e_y`)-`e_z.

4.3.3 Signed multiplication

Signed multiplication '`*`' takes two unsigned x, y and returns their product value $x \times y$. The result type is `signed<`$M\,N+1$`>` since the worst case is $(-M)$`*`$(-N) = MN$:

$$e_x : \texttt{signed<}M\texttt{>},\ e_y : \texttt{signed<}N\texttt{>}\ \rightarrow\ e_x \texttt{*} e_y : \texttt{signed<}MN+1\texttt{>}$$

Signed multiplication is left-associative.

4.3.4 Signed division

Signed division '`/`' takes two unsigned x, y such that $y \neq 0$ and returns their signed integer quotient value x/y. If y has value 0, then x`/`y is undefined and raises a run-time error, see Section **??**.

The result type is `signed<`$M+1$`>` since the worst case is $(-M)\,/\,(-1) = M$:

$$e_x : \texttt{signed<}M\texttt{>},\ e_y : \texttt{signed<}N\texttt{>} \quad\rightarrow\quad e_x\,/\,e_y : \texttt{signed<}M+1\texttt{>}$$

Signed division is left-associative.

4.3.5 Signed saturation

Signed saturation `sat<`N`>` takes a positive constant N, a signed x of any size, and returns the largest signed in `signed<`N`>` that is less than or equal to x if $x > 0$ or the smallest signed in `signed<`N`>` that is bigger than or equal to x if $x < 0$. Therefore, the definition is as follows:

$$\texttt{sat<}N\texttt{>}(x) \quad = \quad \begin{cases} x & \text{if } -N \leq x < N \\ N-1 & \text{if } x \geq N \\ -N & \text{if } x < -N \end{cases}$$

The type-checking rule is as follows:

$$e_x : \texttt{signed<}M\texttt{>} \quad\rightarrow\quad \texttt{sat<}N\texttt{>(}e_x\texttt{)} : \texttt{signed<}N\texttt{>}$$

4.3.6 Signed binary truncation

Remember that `signed<[`N`]>` is a shorthand for `signed< 2**`$(N-1)$`>`, which denotes the set of signed numbers writable with N bits in 2's complement representation. Binary truncation `trunc<[`N`]>` takes a positive constant N, a signed x of any size, and returns a signed obtained by dropping the higher-order bits in the 2's-complement binary representation of x, including the sign bit. It brutally brings back x to N bits. The formal definition is easiest using bitvector conversions:

$$\texttt{trunc<[}N\texttt{]>}(x) \quad = \quad \begin{cases} x & \text{if } M \leq N \\ \texttt{bin2s}(\texttt{s2bin}(x)[0..N-1]) & \text{if } M > N \end{cases}$$

There is no simple arithmetic formula to express numerically what `trunc` does on signed numbers.

For an usage example, consider addition of two signed N-bits expressions. The default Esterel calculation is exact:

$$e_x, e_y : \texttt{signed<[}N\texttt{]>} \quad\rightarrow\quad e_x\texttt{+}e_y : \texttt{signed<[}N\texttt{+1]>}$$

For instance, with $N = 4$, one has `(-8) + (-8)` $= -16$: `signed<[5]>`. On the contrary, a 4-bits carry-dropping usual HDL calculation returns 0. In Esterel, such a truncated calculation must involve an explicit `trunc` truncation:

$$e_x, e_y : \texttt{unsigned<[}N\texttt{]>} \quad\rightarrow\quad \texttt{trunc<[}N\texttt{]>(}e_x\,\texttt{+}\,e_y\texttt{)} : \texttt{signed<[}N\texttt{]>}$$

As for unsigned numbers, we prefer arithmetic to be exact by default since $-8-8 = 0$ is a formula that does not quite help formal verification. Therefore, we require carry-dropping to be explicit, writing `trunc<3>((-8)+(-8))` $= 0$.

4.3.7 Signed absolute value

The absolute value `abs`(x) of a signed number x is an unsigned number with value x is $x \geq 0$ and $-x$ if $x < 0$. Expression type-checking is as follows:

$$e_x : \texttt{signed<}M\texttt{>} \quad \rightarrow \quad \texttt{abs(}e_x\texttt{)} : \texttt{unsigned<}M+1\texttt{>}$$

Here, $M + 1$ is needed because of the worst case $e_x = -M$.

4.3.8 Signed comparisons

The comparison operators are classical: `=`, `<>` (not equal), `<`, `<=`, `>`, and `>=`. They take two signed numbers of any size and return a Boolean.

4.3.9 Signed assert

Signed assertion is similar to unsigned assertion described in Section **??**, and it uses the same syntax. Given two natural numbers M, N and a signed expression e_x of type `signed<`N`>`, the expression `assert<`M`>(`e_x`)` has type `signed<`M`>`. It returns the value v of e_x if that value satisfies $-M \leq v < M$ and provokes a run-time error otherwise, see Section **??**.

4.3.10 Signed assert_unsigned

Given two natural numbers M, N and a signed expression e_x of type `signed<`N`>`, the expression `assert_unsigned<`M`>(`e_x`)` has type `unsigned<`M`>`. It returns the value v of e_x if that value satisfies $0 \leq v < M$ and provokes a run-time error otherwise, see Section **??**.

4.3.11 Mixed signed / unsigned arithmetic

Unsigned and signed can be combined in operations, as for $u + x$ where u is an unsigned and x is a signed. In this case, the unsigned is converted into a signed, viewing the operation as $(+u) + x$ with the corresponding size computation rules.

Conversion is done in a way that respects the operator associativity rules presented in Section **??**, and leaving intermediate results unsigned as long as possible. For instance, with u, v unsigned and x signed, $u + v + x$ is evaluated as $(+(u + v)) + x$, while $u + x + v$ is evaluated as $((+u) + x) + (+v)$.

4.4 Bitvector operations

Bitvectors are arrays of Booleans. Bitvector literals were presented in Section **??**. We describe here the specific bitvector operations, including the conversion from signed and unsigned numbers to and from bitvectors.

A cumbersome issue we cannot escape is that we have to deal with two traditional ways of seeing the same bitvector: least significant bit (lsb) first, as for the array notation `{0,1,1}`, or most significant (msb) first, as in 'b110 which is the same bitvector. In Esterel, we do not identify bitvectors with numbers, but we still have to follow the conventions. We shall be precise about what operators mean in both presentations.

4.4.1 Bitvector equality

Equality '=' and unequality <> are available for bitvectors. Two bitvectors are equal if and only if they have the same length and the same bits.

Equality is not available for general arrays, and in particular for arrays of bitvectors. It needs to be programmed explicitly.

4.4.2 Bitvector onehot and onehot0 predicates

The `onehot` predefined function takes a bitvector argument of arbitrary length and returns true if exactly one bit is one in the bitvector and false otherwise. The `onehot0` predefined function takes a bitvector argument of arbitrary length and returns true if the bitvector is onehot-encoded or filled only with 0's and false otherwise. Example:

```
module Arbiter :
input  Req [4];   // request vector
output Grant [4]; // grant vector
  ... // arbiter code
||
  sustain {
    // check there is always at most one grant
    assert GrantIsOnehot0 = onehot0(Grant),
    // check there is exactly one grant when there is some request
    assert GrantIsOnehotIfReq = onehot(Grant) if SomeReq
  }
end module
```

4.4.3 Bitvector shifts

There are four bitvector shift operations: unsigned right shift '>>', unsigned left shift '<<', signed right shift '>>>', and signed left shift '<<<'. Signed operations preserve the sign if one sees the bitvector as the binary encoding of a signed number.

The shift operations take a bitvector expression as left argument and an unsigned expression as right argument, as for `X >> E`. If the type of the first argument is `bool[`M`]`, the type of the second argument must be `unsigned<`N`>` with $N <= M + 1$, otherwise there is a type-check error. That is, a bitvector of type `bool[4]` can be shifted 4 times in any direction but not 5 times. The size of the result is equal to the size of the bitvector argument. The shift direction is indicated in the numerical bit order, i.e. with most significant bit on the left, opposite to the array order. For instance, the same right-shift operation can be written in two ways:

```
'b00110100 >> 1 = 'b00011010
{'0,'0,'1,'0,'1,'1,'0,'0} >> 1 = {'0,'1,'0,'1,'1,'0,'0,'0}
```

Shift operations are formally defined as follows, assuming that the bitvector expression X evaluates to a bitvector B of size n and the shift argument E evaluates to $k \leq n$:

$$(B >> k)[i] = \begin{cases} B[i+k] & \text{if } i < n-k \\ 0 & \text{if } n-k \leq i < n \end{cases}$$

$$(B << k)[i] = \begin{cases} 0 & \text{if } i < k \\ B[i-k] & \text{if } k \leq i < n \end{cases}$$

$$(B \texttt{>>>} k)[i] = \begin{cases} B[i+k] & \text{if } i < n-k \\ B[n-1] & \text{if } n-k \leq i < n \end{cases}$$

$$(B \texttt{<<<} k)[i] = \begin{cases} 0 & \text{if } i < k \\ B[i-k] & \text{if } k \leq i < n-1 \\ B[n-1] & \text{if } i = n-1 \end{cases}$$

4.4.4 Bitvector concatenation

The `lcat` and `mcat` concatenation operators take n bitvectors and concatenate them. For `lcat`, which can be read *lsb-cat*, the lsb bit of the result is that of the first argument. For `mcat`, which can be read *msb-cat*, the msb bit of the result is that of the first argument. One has the following equality:

$$\texttt{lcat}(B_0, B_1, \ldots, B_n) = \texttt{mcat}(B_n, \ldots, B_1, B_0)$$

Of course `lcat` is easiest to understand using the array notation and `mcat` is easiest using the bitvector notation. Here is an example:

```
lcat({'0,'0}, {'1,'0,'0}, {'0,'1}} = {'0,'0,'1,'0,'0,'0,'1}
mcat('b11, 'b001, b'10) = 'b1100110
```

The same examples can be written as follows using the other notation:

```
lcat('b00, 'b001, 'b10} = 'b1000100
mcat({'1,'1}, {'1,'0,'0}, {'0,'1}) = {'0,'1,'1,'0,'0,'1,'1}
```

We formally defined concatenation of two bitvectors B and C of respective lengths k and l, extension to the n-ary case being trivial by associativity:

$$(\texttt{lcat}(B, C))[i] = \begin{cases} B[i] & \text{if } i < k \\ C[i-k] & \text{if } k \leq i < k+l \end{cases}$$

$$(\texttt{mcat}(B, C))[i] = \begin{cases} C[i] & \text{if } i < k \\ B[i-k] & \text{if } k \leq i < k+l \end{cases}$$

4.4.5 Bitvector extension

There are two extension functions that take a bitvector `B` of size M and an unsigned statically evaluable expression E: `extend` and `sextend`. It is required that E statically evaluates to a value $N \geq M$, otherwise there is a type error. The definition is as follows:

- $\texttt{extend}(B, E)$ is a bitvector C of size N such that $C[0..M-1] = B[0..M-1]$ and $C[M..N-1] = \texttt{'0}$.
- $\texttt{sextend}(B, E)$ is a bitvector C of size N such that $C\texttt{[}0..M-1\texttt{]} = B\texttt{[}0..M-1\texttt{]}$ and $C\texttt{[}M..N-1\texttt{]]} = B\texttt{[}M-1\texttt{]}$.

$$\texttt{bin2u}\,(\texttt{extend}(B, E)) = \texttt{bin2u}\,(B)$$
$$\texttt{bin2s}\,(\texttt{sextend}(B, E)) = \texttt{bin2s}\,(B)$$

where `bin2u` and `bin2s` are the conversions to unsigned and signed numbers presented in Section **??** below.

For instance, `extend('b110,5)` = `'b00110` and `sextend('b110,5)` = `'b11110`.

Note: `extend` *and* `sextend` *functions should be considered as obsolete and so they should no longer be used. Instead,* `resize` *and* `sresize` *functions should be used.*

4.4.6 Bitvector resizing

There are two bitvector resizing functions that take a bitvector `B` of size M and an unsigned statically evaluable expression E: `resize` and `sresize`. It is required that E statically evaluates to a value $N \neq 0$, otherwise there is a type error. The definition is as follows:

- `resize`(B, E) is a bitvector C of size N such that:
  - $C[0..min(M, N) - 1] = B[0..min(M, N) - 1]$;
  - if $M < N$, then $C[M..N - 1] =$ `'0`.
- `sresize`(B, E) is a bitvector C of size N such that:
  - $C[0..min(M, N) - 1] = B[0..min(M, N) - 1]$;
  - if $M < N$, then $C[M..N - 1] = B[M - 1]$.

$$\begin{aligned}\texttt{bin2u}\,(\texttt{resize}(B, E)) &= \texttt{trunc} < [E] > (\texttt{bin2u}\,(B))\\ \texttt{bin2s}\,(\texttt{sresize}(B, E)) &= \texttt{trunc} < [E] > (\texttt{bin2s}\,(B))\end{aligned}$$

where `bin2u` and `bin2s` are the conversions to unsigned and signed numbers presented in Section **??** below.

For instance, `resize('b110,5)` = `'b00110`, `resize('b110,2)` = `'b10`, and `sresize('b110,5)` = `'b11110`, `sresize('b110,2)` = `'b10`.

4.4.7 Bitvector reverse

A bitvector expression of value B and size k is reversed by applying the `reverse` predefined function. The formal definition is as follows:

$$(\texttt{reverse}(B))[i] \quad = \quad B[k - i - 1] \quad \text{for } i < k$$

4.5 Bitvector maps

4.5.1 Bitvector map declaration

A *map* is a set of aliases for bits and slices of a named bitvector type. It can be unnamed or named. Maps can be defined only for defined types (see Section **??**). An alias for a bit or a slice is called a *map field*. Slices and bits can overlap within a map, and several maps can be declared for a single type. Map field identifiers must be unique for a single type. The same field identifier can occur in two maps only if they do not bear on the same type. Maps must be defined for a given type only and must not follow structural type equivalence: a map defined for a type `Word` declared as `bool[32]` must not be applicable to any `bool[32]` nor for any other declared type of type `bool[32]`. Since map is a data declaration, it is inherited by data extension.

There are two ways of specifying maps:

- In a *concurrent map declaration*, a single-bit field is defined by giving the field name and the bit index in the bitvector; a slice field is defined by giving the field name, the slice first index, and the slice last index in the bitvector.

- In a *sequenced map declaration*, a single-bit field is defined by giving the field name; a slice field is defined by giving the field name and slice size. Map fields are defined in sequence in the bitvector: the first field starts at index 0, any other field starts at the index following the previous field. Notice that map fields in a sequenced map declaration cannot overlap by construction.

Here is an example of concurrent map declaration:

```
type Word = bool[32];

map Word {       // unnamed
  Low[0..15],
  High[16..31]
};

map Instruction : Word {
  Opcode[0..7],
  Immediate[8],    // single bit
  Register[9..15],
  Address[16..31]
};

map SignedNumber : Word {
  Sign[31],        // single bit
  Abs[0..30]
};
```

Notice that `Sign[31]` is a single bit, while `Sign[31..31]` would be a bitvector of dimension 1.

The following sequenced map declaration is fully equivalent to the precedent one:

```
map Word seq {
  Low[16],
  High[16]
};

map Instruction : Word seq {
  Opcode[8],
  Immediate,
  Register[7],
  Address[16]
};

map SignedNumber : Word seq {
  Abs[31],
    Sign
};
```

A map name such as `Instruction` has no semantic meaning. It is only meant to be used by programming and debugging environments, typically to display information in a nicer way.

4.5.2 Bitvector map field usage

A mapped bit or slice is referred to using the dot-field notation:

```
emit {
  ?W.Immediate,
  ?W.Opcode <= 42
}
```

is the same as

```
emit {
  ?W[8] <= 1,
  ?W[0..7] <= 42
}
```

Maps can themselves be reindexed or sliced: `?W.Register[1]` is the same as `?W[10]` and `?W.Register[1..3]` is the same as `?W[10..12]`.

4.6 Encoders: from unsigned to bitvectors and back

An *unsigned encoding* translates an unsigned expression into a bitvector according to a given mathematical encoding. An *unsigned decoding* performs the converse operation. Esterel v7 provides the user with three predefined codes, *binary* (classical binary writing, least significant bit 0), *onehot* (exactly one 1 bit per value, for instance $0 \rightarrow 0b001$, $1 \rightarrow 0b010$, $2 \rightarrow 0b100$), and *Gray*. The user can also define her or his own code.

Unsigned encoding and decoding can be applied to any numerical type, not just types defined by powers of 2. However, in dense encodings such as binary and Gray, a number in `unsigned<[`M`]>` is encoded by a bitvector in `bool[`M`]` and conversely, while a number in `signed<[`M`]>` is binary encoded by a bitvector in `bool[`$M+1$`]` and conversely. This justifies the `<[]>` notation for signed and unsigned types, which facilitates the handling of powers of 2.

For arbitrary encodings, the size correspondence may be more complex. For instance, in onehot encoding, `unsigned<`M`>` is in bijection with `bool[`M`]`, which implies that `unsigned<[`M`]>` is in bijection with `bool[`2^M`]`.

4.6.1 Binary encoding / decoding

To turn an unsigned expression e_u into a binary-encoded bitvector, one can use one of the two forms

`u2bin(`e_u`)`
`u2bin(`e_u`, `b`)`

Assume that the type of e_u is `unsigned<`M`>`. In the first form, the result has type `bool[binsize(`$M-1$`)]`. For instance, one has `u2bin(6)` = `'b110` = {`'0, '1, '1` } with type `bool[3]`. In the second form, b is the size of the resulting bitvector; it must be statically evaluable and satisfy $b \geq$ `binsize(`$M - 1$`)`. For instance, to put a value `v` of type `unsigned<`M`>` with $M < 2^{32}$ on a 32-bit bus, one writes

```
output Bus : bool[32];
...
emit ?Bus <= u2bin(v, 32)
```

Notice that the second argument is not really needed. One could obtain the same effect with the unary `u2bin`, by writing `u2bin(assert<[32]>(v))`. However, this form would be far less readable.

According to our exact arithmetic philosophy, there is no implicit number bit-dropping for `u2bin`: the resulting bitvector must be always big enough to hold the result. If bit-dropping is desired, it can be performed either before or after the call to `u2bin`. For instance, to get the `N` least significant bits of `v` written in binary, one can write one of the two following statements:

```
emit Y <= u2bin(v mod 2**N)

emit {
  X <= u2bin(v),
  Y <= X[0..N-1]
}
```

Binary decoding is symmetrical. The unsigned value of a bitvector `B` read as a binary-encoded number is the result of the following expressions:

`bin2u(`B`)`
`bin2u(`B`, `u`)`

In the first form, the type of the result is `unsigned<`2^M`>` = `unsigned<[`M`]>` if `B` is of type `bool[`M`]`. In the second form, the type of the result is `unsigned<`u`>`, where `u` must be statically evaluable. One must have $u \geq 2^M$.

Because of incomplete value ranges, one may also want the result to be in a type `unsigned<`u`>` with $u < 2^M$. For instance, one may know that only the 13 first numbers can be encoded in a given 4-bit bitvector, and, therefore, one may want the result in `unsigned<13>`. This is easy to achieve using `assert<>`:

```
input Bits : bool[4];
output Value : unsigned<13>;
emit ?Value <= assert<13>(bin2u(?Bits))
```

There will be a runtime error if the bits do not satisfy the hypothesis, see Section **??**.

4.6.2 Onehot and Gray encoding

The handling of one-hot and Gray encodings is similar. The following expressions return bitvectors:

`u2onehot(`e_u`)`
`u2onehot(`e_u`, `b`)`
`u2gray(`e_u`)`
`u2gray(`e_u`, `b`)`

Let `unsigned<M>` be the type of e_u. For onehot, the result type is `bool[`M`]` in the unary case and `bool[`b`]` in the binary case, with the constraint $b \geq M$. For Gray, things are as for binary: the result type is `binsize(`$M-1$`)` in the unary case and `bool[`b`]` in the binary case, with the constraint $b \geq$ `binsize(`$M-1$`)`.

Conversely, the following expressions return unsigned numbers:

```
onehot2u(B)
onehot2u(B, u)
gray2u(B)
gray2u(B, u)
```

Let `bool[`M`]` be the type of `B`. For onehot, the result type is `unsigned<`M`>` in the unary case and `unsigned<`u`>` in the binary case, with the contraint $u \geq M$. One must have $M \geq 1$, and the argument bitvector must have exactly one bit with value 1, otherwise there is a run-time error described in Section **??**.

For Gray, the result type is `unsigned<`2^M`>` in the unary case and `unsigned<`u`>` in the binary case, with the constraint $u \geq 2^M$.

There is no direct re-encoding from an array to another array. To translate a signal array from binary to Gray, one must write

```
u2gray(bin2u(B))
```

4.6.3 Combining encodings

It is often useful to use different encodings in a single calculation. For instance, assume we want to compute 2^u for a natural number $u < M$. In Esterel, we can simply write 2∗∗u. However, this operation may not be synthesizable by backends. Another way to write it using bitvector intermediates is as follows:

```
module Pow :
generic constant  SZ : unsigned;
input u : unsigned<M>;
input x : unsigned<2**(M-1) + 1>;
emit ?x <= assert<2**(M-1)+1>(bin2u(u2onehot(?u)))
end module
```

The call to `u2onehot` builds a bitvector of length M with a 1 at index `?u`. Calling `bin2u` on this bitvector computes $2^{\texttt{?u}}$, but with type `unsigned<2**M>`. The call to `assert` safely shrinks this type to the type `<2**(M-1)+1>`, which is the best type for the result whose value can reach `2**(M-1)`.

4.6.4 From signed to bitvectors and back

For signed numbers, we consider only one encoding, 2's-complement binary. The functions that translate signed numbers to bitvectors is `s2bin`, unary or binary:

```
s2bin(e_x)
s2bin(e_x, b)
```

As usual, the highest-order bit (msb) of `s2bin(`e_x`)` is the sign bit. Assume that the type of e_x is `signed<`M`>`. Then, in the unary case, the result type is `bool[binsize(`M`)+1]`. For the binary case, the result type is `bool[`b`]`, with the constraint $b \geq$ `bool[binsize(`M`)+1]`.

The reverse conversion from bitvectors to signed numbers is performed by `bin2s`:

```
bin2s(B)
bin2s(B, s)
```

Let `bool[`M`]` be the type of B. Then the result type is `signed<[`M`]>` in the unary case and `signed<`s`>` in the binary case, with the constraint $s \geq 2^M$. As for unsigned numbers, one can use the `assert< >` function to narrow the range. For instance, if only numbers from -5 to 4 should be encoded by a bitvector B of width 5, one write

```
assert<5>(bin2s(B))
```

A run-time error will be generated if the decoded value is out of range, see Section **??**.

4.7 From enum to unsigned or signed and back

In this section, we consider the enum types CardSuit and CarManualGear of Section **??**:

```
type CardSuit = enum {
  Clubs = 1,  // explicit code 1
  Diamonds,   // implicit code 2
  Hearts = 4, // explicit code 4
  Spades      // implicit code 5
};

type CarManualGear = enum {
  Rear,       // implicit code -1
  Idle = +0, // explicit code 0
  First,      // implicit code +1
  Second,     // implicit code +2
  Third,      // implicit code +3
  Fourth,     // implicit code +4
  Fifth       // implicit code +5
};
```

The `enum2u` function can be used to convert a value of an unsigned enum type into its unsigned code. For example, `enum2u(Diamonds) = 2`. If *exp* is an expression of an unsigned enum type, then `enum2u(exp)` is of type `unsigned<M>`, where M is the unsigned size of the enum type.

The `u2enum<E>` function can be used to convert an unsigned value to the element of the unsigned enum type E whose code is that value. For example, `u2enum<CardSuit>(5) = Spades`. If *exp* is an expression of unsigned type, the type of `u2enum<E>(exp)` is E. The unsigned-to-enum conversion run-time error occurs if the value of *exp* is associated with no enum constant in E, see Section **??**.

The `enum2s` function can be used to convert a value of a signed enum type into its signed code. For example,`enum2s(Third) = +3`. If *exp* is an expression of a signed enum type, then `enum2u(exp)` is of type `signed<M>`, where M is the signed size of the enum type.

The `s2enum<E>` function can be used to convert a signed value to the element of the signed enum type E whose code is that value. For example, `s2enum<CarManualGear>(-1) = Rear`. If *exp* is an expression of signed type, the type of `s2enum<E>(exp)` is E. The signed-to-enum conversion run-time error occurs if the value of *exp* is associated with no enum constant in E, see Section **??**.

Chapter 5

Signals and Variables

Signals are the main objects dealt with by Esterel v7_60 programs. They are shared scoped objects, broadcasted within their scope in a way that ensures fully safe synchronization of parallel threads. More conventional *variables* are also available, but they are of restricted use since they cannot be freely shared between concurrent control threads. However, variables are useful as intermediates in computations and necessary as vehicles to communicate with the host language, being passed by reference in procedure calls (see Section **??**).

In this chapter, we first introduce the signal categories in Section **??**: interface signals, local signals, oracles, assertions, assumptions and coverage points. Then, in Section **??**, we study the main features of signals, the *status* and the *value*. Section **??** is devoted to *standard signals*, the most common signals that act instantaneously. Section **??** is devoted to *registered signals*, which act with a delay of one tick and are fundamental for Moore machines and pipelines. Section **??** presents *temporary signals*, which do not hold their value over time. Section **??** presents *signal arrays*.

The full signal declaration syntax will be presented in Chapter **??**, together with the difference between interface and module features of signals. This chapter mostly concentrates on behavioral aspects.

5.1 Interface and local signals, verification signals, and oracles

There are several categories of signals corresponding to different uses:

- *Interface signals* are declared within interface or module declarations. They establish communication between a module body and its environment. Their scope is a module body.

- *Local signals* are declared either in a module header or in a module body in a declaration statement block. The scope of a local signal declared in a module header is the module body; the scope of a local signal declared in a module body is the declaration statement block.

- *Assertions*, *assumptions*, and *coverage points* are special local signals designed for simulation or formal verification-based validation. Assertions, assumptions, and coverage points are generically called *verification signals*.

- *Oracles* are special local signals used for non-deterministic specifications. They are declared locally but input from the environment in a hidden way. *Oracles are incomplete and experimental in Esterel v7_60.*

This chapter focuses on the common features of all signals and only uses interface and local signals in examples.

Basic example

Here is a simple example to illustrate how interface and local signals are declared and what their scopes are:

```
module Stopwatch :
  type TimeType;
  input Second;
  output Time : TimeType;
  signal InternalTime : TimeType in
    ...
    signal Reset in ...
      ...
    end signal
  end signal
end module
```

The interface signals are `Second` and `Time`, with global module scope. The local signals are `InternalTime`, whose scope is the full module body, and `Reset`, with scope delimited by the `signal`—`end signal` keywords pair.

The `Second` signal is a *pure* signal that conveys a bit-level *present / absent* status information. Being an *input*, it is generated by the module's environment, which is the global execution environment for the main module and the the local execution context for a submodule. The `Time` *valued* signal is made of the pair of a bit-level status and a value of type `TimeType`. It is declared *output*, which means that it is generated by the module itself and output to the module's environment. There are also *inputoutput signals*, which are bound to master signals in both input and output mode, see Section **??**.

5.2 Signal status and value

We browse quickly through the main features of signals before describing the precise signal kinds. The first features we present are the main components of a signal: the *status* and *value*. Their combination leads to two different kinds of signals: *pure* and *valued*, the latter divided into two subkinds, *full*, and *valueonly*.

The next features concern temporal behavior: signals can be either *standard* , i.e. instantaneous, or *registered*. These features are orthogonal to the previous ones.

Valued signals can have two additional features, independent of each other and of the temporal ones: being *initialized* and being *combined*.

Furthermore, for better hardware synthesis, there are *temporary signals* that do not keep their value over time. They are a bit special and discussed in a specific section.

In the last part of this section, we discuss which features are interface features and which are module features.

5.2.1 Signal status

The status of a signal `S` is the basic tool for control path programming. It is a binary information usually referred to as *absent / present*, or *false / true*, and sometimes also called *0 /1*, or *low / high*, or *unset / set*, which is standard hardware terminology. The status of `S` is *absent* by default, and it is set *present* either by the environment for an input signal or by the program when an `emit` statement is executed, either directly in the module where the signal is declared or indirectly through a submodule connection. The status of `S` is tested for presence or awaited to be *present* by statements such as "`if S then ...`", "`await S`", "`every 3 S do ... end`", etc. In such statements, the value of the Boolean expression `S` evaluates to *false* if `S` is *absent* and to *true* if `S` is *present*. The status-handling statements are presented in full details in Chapter **??**. Here is a simple example of status-based sequential programming:

```
module Status ;
  input I;
  output O;
  await I;
  emit O
end module
```

The `await I` statement terminates at the first instant `I` occurs, first instant excluded. The semicolon '`;`' that follows the `await` statement is sequencing. Control is immediately passed to "`emit O`" that immediately emits `O`. See Chapter **??** for the full description of control propagation.

Dissymmetry of status with *absent* as default is key to Esterel control path programming: there is no need to execute any statement to keep `O` *absent*.

5.2.2 Signal Value

A valued signal is declared using a colon and a type name after the signal name, as in "`S : T`". The value `?S` of `S` is a data object of type `T`. If the type is an array type, the value is an array value that can be indexed; for instance, with `T` = `bool[8]`, the value `S` is a bitvector of width 8 that can be indexed to return a Boolean, as in `?S[3]`.

As for the status, the value is set either by the environment if `S` is an input or by `emit` statements executed by the module body or in a connected submodule. Here is a simple example:

```
module Value :
  input A : unsigned<[8]>;
  output X : unsigned<[9]>;
  every immediate I do
    emit ?X <= 2*?A
  end every
end module
```

Whenever it receives `A`, the `Value` module emits twice the value of `A` as the value of `X`.

5.2.3 Pure, full, and valueonly signals

The signals `I` and `O` of the above `Status` example are the simplest Esterel objects, called *pure signals*. They only have a status.

The signals `A` and `X` of the `Value` example are called *full signals*. Their declaration involves a value type after a colon '`:`'. They have both a status and a value.

The statement "`emit ?X <= 2*?A`" performs two simultaneous functions: it sets the status of `X` *present*, and it sets its value `?X` to that of the right-hand-side `2*?A`. Status and value are coordinated: the signal value changes only when the status is *present*. In hardware terminology, one would say that the status acts as a *value enable*. The status is tested for presence in control statements such as "`every immediate I`", while the value is used in data expressions such as `2*?A`. For instance, for `I` and `O` unsigned signals, the statement

```
every I do
  emit ?O <= 2*?I
end every
```

emits `O` with twice the value of `I` whenever this value is emitted by the rest of the program.

There are also *value-only* signals that have a value but no status. They are declared using a `value` keyword before the type. Here is an example:

```
module Converter :
  input Farenheit : value float;
  output Celsius : float;
  input Second;
  every Second do
    emit ?Celsius <= (?Farenheit-32.f) * 5.f / 9.f
  end every
end module
```

Here, the input value `?Farenheit` is assumed to be delivered by a thermometer that one can read at any time but that does not send value change information. This is why we use a statusless value-only signal. In `Converter`, we chose to output the `Celsius` value as a full signal. We could also output it as a valued signal, using the declaration

```
output Celsius : value float;
```

In that case, the user would not be warned of value changes on `Celsius`.

Full signals and value-only signals are collectively called *valued signals*. Therefore, a signal is either pure or valued.

For valued signals, one may allow simultaneous multiple emission of values. The values are then combined using a combination (or resolution) function. This will be detailed in Section **??**.

5.3 Standard signals

A standard signal is declared in the simplest way, by giving its name and possibly its type after a colon, as for "`input I`" or "`output O : Byte`". Standard signals status and value are instantaneously updated by signal emission. For valued signals, the value is persistent and it can be read at any time provided it is defined. The `pre` operators give access to the status and value at previous instant.

5.3.1 Standard signal emission and reception

When a standard signal is emitted, its status and value are made instantaneously available to other active statements. Here is an example:

```
module Standard :
  output O : unsigned;
  signal S : unsigned in
    pause;
    emit ?S <=1;
    pause;
    pause;
    emit ?S <= 2;
  ||
    await S;
    emit ?O <= ?S+1;
    await S;
    emit ?O <= ?S+2
  end signal
end module
```

At first instant, the first parallel branch initial `pause` statement waits for one tick while the second parallel branch `await` statement starts waiting for `S`. At second instant, the first branch emits `S` with value 1, which immediately triggers emission of `O` with value 2 by the second branch. The first branch pauses for two ticks, while the second one waits for another occurrence of `S`. At fourth instant, the first branch emits `S` with value 2 and the second branch emits `O` with value 4.

Status and value transmission from `S` to `O` through the "`await S`" statements and the `?S` expressions take no logical time, i.e. are performed combinationally in the same tick as the emission. So are the intermediate additions.

5.3.2 Value persistency and initialization

If a standard signal `S` is valued, its value `?S` is persistent. If it is not emitted nor received from the environment (for an input) in one instant, it remains that of the previous instant. For instance, in `Standard` above, reading the value `?S` at third instant would return 1.

By default, before the first emission, the value is *uninitialized* and reading it is a run-time error, see Section **??**. For instance, in the `Standard` module above, the value of `S` and `O` are undefined at first instant and reading them at that instant would be an error.

One can explicity initialize the signal value at declaration time using the `init` keyword, as for

```
output O : unsigned init 0;
```

In that case, the value is always defined and it can be read at all instants. The initial value is overwritten by the first signal emission or by the first reception from the environment for an input. Here is an example involving an initialized local signal:

```
module InitializedSignal :
  output O : unsigned;
  signal S : unsigned init 0 in
    pause;
    emit?S <= 1;
  ||
    sustain ?O <= ?S
  }
end module
```

Here, `O` is emitted with value 0 at first instant and value 1 from second instant onwards. Removing the `pause` statement would make the initial value overwritten at first instant and thus useless.

For arrays, the initial value can be either a full array literal or a simple literal of the base type, see Section **??**:

```
signal S : unsigned[5] init {0,1,2,3,4},
       T : bool[12] init '0 in
   ...
end signal
```

Notice that systematically holding the value over time involves storing it in some memory, which may be too expensive for hardware applications. Thus, we also provide temporary signals that do no store the value, see Section **??**.

5.3.3 The status pre(S) operator

For a pure or full standard signal `S`, one can read the status at previous instant using the `pre(S)` and `pre1(S)` Boolean expressions, For instance, one can detect a rising edge of `S` by computing "`S and not pre(S)`". The only difference between `pre(S)` and `pre1(S)` is the Boolean value at first instant of the lifetime of a signal, i.e. at the instant where the signal declaration is executed. At that instant, `pre(S)` is false while `pre1(S)` is true. Here is an example:

```
module Pre :
  input I, J;
  output X, Y;
  {
    sustain {
      X <= pre(I),
      Y <= pre1(J)
    }
  ||
    pause;
    signal S in
      emit S;
    ||
      sustain Z <= pre1(S)
    end signal
  }
end module
```

Here, `X` is emitted at any instant that follows an occurrence of `I`, `Y` is emitted at first instant and at any instant that follows an occurrence of `J`, and `Z` is emitted at second and third instant. Notice that the expression `pre1(S)` yields its initial value 1 only at second instant since this is when the lifetime of `S` starts because of the `pause` statement.

The `pre` operator extends to status expressions as explained in Section **??**. For instance, "`pre(S and not T)`" is the same as "`pre(S) and not pre1(T)`" (noticing that "`pre(not T)`" is equal to "`not(pre1(T))`" and not to "`not(pre(T)`"), because of the initial instant — a well-known retiming pitfall).

Status `pre` operators cannot be nested. If needed, nesting can be achieved using auxiliary signals, as in

```
signal preS, prepreS in
  sustain {
    preS <= pre(S),
    prepreS <= pre(preS)
  }
||
  ... await prepreS; ...
end signal
```

or auxiliary sliding window arrays, as in

```
signal PreS[N] in
  emit {
    PreS[N-1] <= S ,
    PreS[0..N-2] <= pre(PreS[1..N-1])
  }
||
  ... if PreS[i] then ...
end signal
```

See Section **??** and Section **??** for details on status arrays and slices.

5.3.4 The value pre(?S) operator

One can also read the value at previous instant using the `pre(?S)` data expression. The previous value is defined at the first instant that follows value definition. If the signal has an initial value, it is also the initial value of `pre(?S)`.

There is no extension of data `pre` to expressions: one cannot write `pre(?S+?T)`, but only "`pre(?S)+pre(?T)`".

Value pre operations cannot be nested. To implement sliding windows for values, one can use either intermediate signals or sliding windows as for status. Here is an example

```
signal PreValS[N] : unsigned in
  emit {
    ?PreValS[N-1] <= ?S ,
    ?PreValS[0..N-2] <= pre(?PreValS[1..N-1])
  }
||
  ... ?PreValS[i] + 1
end signal
```

See Section **??** and Section **??** for details on value arrays and slices.

5.3.5 Standard signal submodule connection

Standard signals can be sent to or emitted by submodules through signal connections, detailed in Section **??**. Consider the following example:

```
module Sub :
  input I : unsigned;
  output O : unsigned;
  sustain ?O <= ?I + 1
end module
```

```
main module Main :
  input A : unsigned;
  output X : unsigned;
  emit ?X <= 1;
  pause;
  run Sub [A / I, X / O]
end module
```

Here, `A` in `Main` is connected to the input `I` of `Sub`, while `X` is connected to the output `O` of `Sub`. The status and value of `A` are tranmitted to the input `I` of `Sub`, while the status and value of `O` in `Sub` are transmitted to `X`. In `Main`, the `X` signal is emitted in two different ways: at first instant, by the explicit "`emit ?X`" statement; after the first instant, i.e. when `Sub` runs, through the "`X / O`" connection with help of the "`A / I`" connection. Assume that `Main` receives `A` from its environment with value 2. Then, `A` is *present* in `Main` with that value. After the first instant, since `Sub` is running, the connection "`A / I`" instantaneously sets `I` *present* with value 2 within `Sub`. This provokes instantaneous emission of `O` with value 3. The connection "`X / O`" makes `X` *present* in `Main` with value 3, and this value is instantaneously returned to the environment.

Therefore, an `emit` in the submodule acts just as an `emit` in the main module through the signal connection.

5.4 Registered signals

Registered signals act with a delay of one tick for status and value transmission. Pure, full, and value-only signals can be registered. Local and output signals can be registered, but input signals cannot.

5.4.1 Registered signal declaration and emission

The register declaration uses a `reg` or `reg1` keyword after a colon and before the type. Registered signals are emitted using "`emit next`" instead of `emit`, and tested or value-read in the same way as standard signals. A registered signal is *present* at an instant if it has been emitted at the previous instant; its value is also shifted by one tick. Here is a registered version of `Standard` above:

```
module Registered :
  output O : unsigned;
  signal S : reg unsigned in
    pause;
    emit next ?S <=1;
    pause;
    pause;
    emit next ?S <= 2;
  ||
    await S;
    emit ?O <= ?S+1;
    await S;
    emit ?O <= ?S+2
  end signal
end module
```

As for `Standard`, emission of `S` is performed at second and fourth tick. However, since status and value transmission consume one tick, `S` is *present* only at third and fifth tick, with respective values 2 and 4. The output `O` is emitted at these ticks with these values.

For a `reg1` signal, the status is also set *present* at the first instant of the signal lifetime. Replacing `reg` by `reg1` in `Registered` would make `S` *present* at initial instant.

Notice that the "`emit next`" statement is instantaneous, just as the `emit` statement: the status and value are instantaneously posted for the next instant and control instantaneously proceeds in sequence.

5.4.2 Value persistency and initialization

The value `?R` of a registered signal `R` is persistent. By default, it is uninitialized until the instant that follow the first emission of the signal. Therefore, in `Registered` above, the value of `R` at first instant is uninitialized, and reading it would be a run-time error. This is also true for a valued registered signal declared `reg1`, since the '`1`' applies only to status and not to value.

As for a standard signal, the value of a registered can be initialized using an `init` keyword. Then, the value at first signal lifetime instant is the initial value, which implies that no uninitialized signal error can occur. Consider the following variant of `Registered`:

```
module RegisteredInit :
  output O : unsigned;
  signal R : reg unsigned init 0 in
    pause;
    emit next ?R <=1;
    pause;
    pause;
    emit next ?R <= 2;
  ||
    emit ?O <= ?R
    await R;
    emit ?O <= ?R+1;
    await R;
    emit ?O <= ?R+2
  end signal
end module
```

In `RegisteredInit`, `O` is emitted at first instant with value 0 by the second parallel branch, the subsequent behavior being identical to that of `Registered`.

5.4.3 The next(R) status operator

For a registered pure or full signal `R`, the `next(R)` operator tests whether an "`emit next R`" statement is executed in the instant. Here is an example:

```
module Registered2 :
  output X : reg, Y;
  emit {
    next X,
    Y <= next(X)
  }
end module
```

Here, `Y` is *present* at the first instant since "`emit next X`" is executed at that instant, while `X` is only *present* at the second instant since it is registered.

The expression `pre(R)` is disallowed for a registered signal `R`.

5.4.4 The next(?R) value operator

For a registered valued signal `R`, the `next(?R)` operator returns the posted value if an `emit next ?R <= ...` statement is executed in the instant; otherwise, it returns the current value of `?R`. Of course, in the latter case, there is a run-time error if that value is yet undefined. Here is an example:

```
module Registered3 :
  output X : reg unsigned, Y : unsigned;
  emit {
    next ?X <= 314,
    ?Y <= next(?X)
}
```

In `Registered3`, `Y` is *present* at first instant with value `314`, while `X` is undefined at first instant and *present* at second instant with the same value.

The expression `pre(?R)` is disallowed for a valued registered signal `R`.

5.4.5 Registered signals submodule connection

Registered signals can be connected to submodule inputs in the same way as local signals. Here is an example similar to that of Section **??**:

```
module Sub :
  input I : unsigned;
  output Rsub : reg unsigned;
  sustain next ?RSub <= ?I+1
end module

module MainReg :
  output X : unsigned;
  output R : reg unsigned;
  emit next ?R <= 2;
  pause;
  run Sub [R / I, X / RSub]
end module
```

Ar first instant, `Main` emits `R` with value 2 for the next instant. At second instant, `R` is made *present* with value 2, and `Sub` starts. Because of the "`R / I`" connection, `I` is made *present* in `Sub` with value 2, which provokes emission of `Rsub` with value 3 for the next instant. At third instant, because of the "`X / Rsub`" connection, `X` is made *present* with value 3 and output to the environment.

Altogether, there is no difference between standard and registered connections if one only thinks in terms of current status and value: the status and value considered are those in the instant, and the registers act only to locally delay status and value setting within `Main` and `Sub`.

Because registers only support emission for the next instant, it is not possible to connect a submodule output signal to a master module registered signal through an output connection.

However, in the future, it may be useful to introduce connections of the form "`next R / O`*" — advice wanted!*

5.4.6 Building data pipelines with registered signals

Registered signals are fundamental for Moore machine design and for hardware data pipelining. Since their current status and value depend only on past actions, they are Moore signals in the usual sense. For an example of data pipelining, consider the following calculation: given four unsigned value-only inputs `A`, `B`, `C`, and `D`, compute the value $(\mathtt{A} \times \mathtt{B}) + (\mathtt{C} \times \mathtt{D})$. The obvious idea is to write

```
module Calc :
  input {A, B, C, D} : value unsigned;
  output O : unsigned;
  sustain ?O <= (?A*?B)+(?C*D)
end module
```

But the data calculation may lead to a unacceptable critical path, and one may want to pipeline the design by inserting a register barrier between the multiplications and the addition. This is very easy by Esterel program transformation. First, we introduce two intermediate signals to cut the operation:

```
module Calc :
  input {A, B, C, D} : value  unsigned;
  output O : value unsigned;
  signal {AB, CD} : value unsigned in
    sustain {
      ?AB <= ?A*?B,
      ?CD <= ?C*?D,
      ?O <= ?AB + ?CD
    }
  end signal
end module
```

So far, we haven't changed anything in the behavior, since the intermediate standard signals `AB` and `CD` act in a purely combinational way. However, pipelining is now made easy . We simply declares the local signals `AB` and `CD` to be registered, annotating their emission with the `next` keyword, and we delay the first emission of `O` by one tick using the `pause` statement presented in Section **??**:

```
output O : value unsigned;
signal {AB, CD} : reg value unsigned in
  sustain {
    next ?AB <= ?A*?B,
    next ?CD <= ?C*?D
  }
||
  pause;
  sustain ?O <= ?AB + ?CD
end signal
```

Of course, `O` is emitted with a delay of one tick, which is the pipeline latency. Since `next` is used for all equations in the first `sustain` statement, we can factor it out at `sustain` level, as will be explained in Section **??**:

```
signal {AB, CD} : reg value unsigned in
  sustain next {
    ?AB <= ?A*?B,
    ?CD <= ?C*?D
  }
...
end signal
```

In the above design, `?O` is the result of an addition. In hardware implementation, this means that `?O` is the output of some combinational logic. To improve circuit delay, one often desires output values to be register driven instead of combinational logic driven, i.e. to compose Moore machines instead of Mealy machines. To achieve this, it suffices to declare `O` registered:

```
output O : value unsigned;
signal {AB, CD} : reg value unsigned in
  sustain next {
    ?AB <= ?A*?B,
    ?CD <= ?C*?D
  }
||
  pause;
  sustain next ?O <= ?AB + ?CD
end signal
```

Of course, latency is now 2: the first value of `O` is output at third instant.

5.5 Temporary signals

Temporary signals are restricted kinds of valued standard signals with non-persistent values. While the lifetime of the value of a standard or registered signal is the lifetime of its declaration statement, the lifetime of the value of a temporary signal is an instant. Of course, the value `pre` operator `pre(?S)` is not available for a temporary signal `S`, and a temporary signal cannot be registered.

Temporaries are used mostly for hardware synthesis, where one does not want to store values in memory unless strictly necessary.

5.5.1 Temporary signal declaration

A temporary signal is declared by the `temp` keyword before the type name:

```
output X : temp Byte;
```

A temporary signal can be value-only; it is then declared using the `temp` and `value` keywords in any order:

```
output Y : temp value Byte;
signal S : value temp Byte in ... end
```

A temporary signal is internally emitted just as a standard signal, using an `emit` statement. For a temporary input of a module `M`, there are two cases:

- if the signal is full (not value-only), its value is supposed to be received from the module environment only when the signal is *present*.

- If the signal is value-only, the value is assumed to be received from the environment at each instant in the lifetime of `M`.

Therefore, one can view a value-only temporary input as a temporary whose status is always *present*.

5.5.2 Temporary signals submodule connection

A temporary signal `S` can also be sent to a submodule or received from a submodule. Consider first the connection of a temporary `S` to the input `I` of a submodule `Sub`, as in "`run Sub[S / I]`":

- If `S` is full, at each tick where `S` is present, `I` is set *present* in `Sub` if it is pure or full, and the value of `S` is passed to `I` if `I` is valued.
- If `S` is value-only, then `I` must also be value-only, and the value of `S` is passed to `I` at each tick where `Sub` is executed.

The laws are dual for a temporary `S` connected to an output `O` of `Sub`, as in "`run Sub[S / O]`":

- If `S` is full, so must be `O`; then `S` is set *present* and `O`'s value is passed to `S` at each tick where `O` is *present* in `Sub`.
- If `O` is value-only, `O`'s value is passed to `S` at each tick where `Sub` is executed.

The rules will be made fully precise in Chapter **??**.

5.5.3 Temporary signal initialization

The value of a temporary signal can be initialized using the `init` keyword. Unlike for a standard signal for which initialization is performed only at lifetime start, initialization of a `temp` signal is performed anew at each instant[1].Therefore, the value of an initialized `temp` signal is always defined. Here is an example:

```
module Temp :
  output O : unsigned;
  signal S : temp unsigned init 0 in
    pause;
    emit ?S <= 1;
  ||
    sustain ?O <= ?S
  end signal
end module
```

In `Temp`, the value of `S` is initialized to 0 at each instant; since `S` is emitted at second instant with value 1, it takes that value at that instant, but for that instant only. Therefore, `O` is emitted at first instant with value 0, at second instant with value 1, and at each instant with value 0 from then on. The behavior would be the same with `S` declared "`value temp`".

Beware, because initialization is different, it may be non-innocuous to turn a standard signal into a temporary one just to improve circuit synthesis. One must check that the value is not accessed at a time where the signal is not emitted, since one would find the previous value for the standard signal and the initial value for its `temp` variant.

[1]The initial value could be more accurately called a *default* value. However, we did not want to add one more keyword just for this case.

5.5.4 Temporary signal value definition

If a temporary signal is initialized, its value is always defined and can always be read. Otherwise, the value can be undefined. There are two cases:

- For a normal non-value-only temporary signal, the value is defined exactly when the status is *present*.
- For a value-only temporary signal, the value is defined when the signal is *driven*. An input `value temp` signal is always driven by the environment if it is an input; it is driven by any `emit` statement executed within the module; finally, when connected to a submodule output, it is driven as long as the submodule is alive, see Section **??**.

Reading the value of an undriven temporary signal is a run-time error, see Section **??**.

5.6 Single vs. combined signals

By default, a valued signal `S` of any kind supports only one value emission at a time. We say that `S` is *single*. It is a run-time error to perform two simultaneous emissions, see Section **??**. Such an error can be detected at run-time, compile-time, or verification time, according to the programming environment.

Esterel also provides its user with *combined signals* that support multiple simultaneous emission. The values are combined using a specified combination (or resolution) operation. Any kind of valued signal (standard, registered, temporary, full, value-only) can be combined.

A combined signal is declared by adding the `combine` keyword followed by a data function name or by an operator name. The function or operator must be of type `(T,T):T` if the value type is `T`. It must be associative and commutative. If several values at emitted at a time for `S`, say $v_1, v_2, \ldots, v_n$, and if the function is called `f`, then the final value `?S` of `S` is $\mathtt{f}(v_1, \mathtt{f}(v_2, \ldots.\mathtt{f}(v_{n-1}, v_n)))$

Anticipating on signal arrays, here is an easy way to compute the sum of an array of byte with addition as the combination function [2]

```
module ArraySum :
input I[16] : unsigned<256>;
output O : unsigned<16*255+1> combine +;
for i < 16 do
  emit ?O <= ?I[i]
end for
end module
```

Notice the result type `unsigned<16*255+1>`. This is the tightest possible type for the sum, since each array component maximum value is 255. If the signal type is an array type, the combination function must apply on the non-array base type of the signal and combination is done in a pointwise way. Here is an example:

```
type Word = bool[32];
signal W : Word combine or;
```

For `W`, `or`-combination is performed independently for each `?W[i]`. Combining arrays values globally requires explicit programming.

[2]Warning: for hardware synthesis, this method may not be optimal in term of adder width, depending on the synthesis system.

5.7 Signal arrays

Signals of any kind can be organized into *signal arrays* or arrays of arrays. The dimensions are given right after the signal identifier. They must be statically evaluable expressions. Here are signal array declarations examples:

```
input InBus[Size];                // array of pure signals
output Matrix[N][N] : float;      // square matrix of floats
signal S[N]: bool[M] in ... end   // signal array of bitvectors
```

Any kind of signal (standard, registered, temporary, full, value-only) can be an array. All properties of elementary signals extend pointwise to arrays.

5.7.1 Signal array status

A pure or full signal array has one status per index. For instance, the `Inbus` status array above can be indexed as in `Inbus[1]` or sliced as in `Inbus[1..5]`. Arrays of arrays such as `S[3][8]` can be totally indexed, as in `S[1][7]`, partially indexed, as in `S[1]`, or sliced, as in `S[1][2..3]`, `S[1..4][0..5]`, or `S[1..4]`. In the above example, the status of `Matrix` is an array of arrays, and the status of `S` is a simple array. Notice that the value array dimension of `S` is ignored as far as the status is concerned.

For standard signals, the `pre` operator applies pointwise, and it can be applied to partially or totally indexed arrays. Therefore, one can write the expressions `pre(Matrix)`, which is of type `bool[N][N]`, `pre1(Matrix)`, which is the same but with initial status *present* at each position, `Pre(Matrix[1][2..4])`, which is of type `bool[3]`, and `Pre(Matrix[1][2])` which is of type `bool`. The same holds for `next` expressions for registered arrays.

Notice that indexing must be inside `pre` or `next`. One cannot write `pre(Matrix)[1][2]`.

5.7.2 Signal array value

A valued signal array `A` declared of a non-array data type `T` also has one value of type `T` per index. Therefore, the expression `?A` denotes a data array of the same dimension as the status. This array can be indexed and sliced in the same way. For instance, for `Matrix` above, `?Matrix` is an array of type `float[N][N]`, `?Matrix[1][2]` is a float, and `?Matrix[1..4][2]` is an array of type `float[4]`.

If the data type `T` is itself an array type, the signal and data dimensions are concatenated from left to right, signal dimensions first. Here is an example:

```
input Memory[1024] : bool[32];
```

Here, there is one status per memory 32-bits word. As far as data is concerned, the `?Memory` expression has type `bool[1024][32]`. In the same way as the status expression `Memory[100]` denotes the status of the 100th word, the data expression `?Memory[100]` denoted the value of the 100th word, which is of type `bool[32]`. Therefore, `?Memory[100][8]` denoted the bit of index 8 of the 100th word value.

Whether to put the dimensions on status or value is a matter of control granularity. There are three posssible choices for `Memory`:

```
input Memory : bool[1024][32];
input Memory [1024] : bool[32];
input Memory [1024][32] : bool;
```

With the first choice, there is only one `Memory` status bit for the whole memory. That bits witnesses any change in the memory. With the second choice, there are 1024 status bits, one per memory word. Only one status bit changes when a word changes. With the third choice, there is one status bit for each memory bit, which gives maximal but expensive control over memory change information.

Be careful when the signal is declared using an intermediate defined array type, as in the following example:

```
type Word = bool[32];
input Memory[1024] : Word;
```

The type of `Memory` is still `bool[1024][32]`, which means that we apply the following implicit operation on defined types:

```
?Memory  :  Word[1024]
         =  (bool[32])[1024]
         =  bool[1024][32]
```

To avoid any confusion, there is no way to write directly a potentially ambiguous type expression of the form `(bool[32])[1024]` in Esterel, and one should simply remember that indices are always written in the order of dimension declaration. Therefore, `?Memory[i]` is a `Word`, and `?Memory[i][j]` is the `j`-th bit of that word[3].

As for the status, the value `pre` and `next` operators can be applied to arrays, and they act pointwise. For instance, `pre(?Matrix[1..4][2..5]` has type `float[4][4]`, while `pre(?Memory[1])` has type `Word` = `bool[32]`. As for the status, indexing must be done inside the call of `pre` or `next`

5.7.3 Combined arrays

Any valued array can be combined, using a pointwise combination function. The combination is done position per position independently. There is no way to use a function that acts globally on data arrays. Here is an example:

```
signal CombineMemory [N] : bool[M] combine or;
```

Then, `or`-combination is performed individually for each position `?Memory[i][j]`.

5.8 Variables

Besides valued signals, Esterel v7_60 can handle data through more classical typed *variables*. Variables are in the same namespace as signals and all other names. They are only local to modules, therefore they cannot appear in interfaces.

[3]It turns out that the potential pitfall disappears when one writes the postfix form `array 1024 of array 32 of bool` instead of the prefix form `bool[1024][32]`, but we want to keep the simplest and most usual notation for Esterel.

5.8.1 Variable declaration and scope

Variables are declared using the "`var` *vardecl* `in` *stat* `end`" statement which defines their scope. They can be initialized using the '`:=`' assignment symbol followed by an expression, see Section **??**. They can have array types, but, unlike signals, they cannot be themselves arrays (there is no need to distinguish between variable arrays and variables of array type since variables have no status). Here are examples:

```
var IterationNumber : unsigned<[8]>,
    VarArray1 : unsigned<[8]>[5] := {5{0}},
    VarArray2 : bool[12] := '0 in
  ...
end
```

For arrays, the initialization value can be either a full array literal, as for `VarArray1` above, or a single literal of the array base type, as for `VarArray2` above. See Section **??** for details.

5.8.2 Persistent vs. temporary variables

By default, a variable is *persistent*, i.e. retains its value between instants. If it is initialized, a persistent variable takes its initial value when the `var` statement that declares it starts; otherwise, it remains uninitialized until its first value assignment by an assignment or procedure call statement.

As for signals, using the `temp` keyword, one can also declare *temporary* (or combinational) variables that do not keep their values over time and do not generate hardware registers. Here are examples

```
var X : temp Byte,
    Y : temp float  := 3.14f in
  ...
end var
```

As for a signal, an initialized `temp` variable is reinitialized afresh at each instant. When changing a variable from `temp` to normal, beware that initialization is performed only once instead of at each tick; this is normal since the variable value has become persistent over time.

Variables values are updated either by '`:=`' assignment statement, see Section **??**, or by procedure calls, see Section **??**. When it exists, the initial value is overwritten.

Variable values are read in data expressions by simply mentioning their name, without the '`?`' symbol used for signals. For instance, the expression "`?I+X`" adds the value of the variable `X` to the current value of the signal `I`. Reading the value of an undefined variable is a run-time error, see Section **??**. Notice that an initialized persistent or temporary variable is never undefined.

5.8.3 Variables cannot be shared

Unlike signal emission and reception, variable updating embodies no built-in write / read synchronization mechanism. Therefore, a variable can take several successive values in the same instant. Here is an example:

```
var V : unsigned := 1 in
  emit O1(V);
  V := V+1;
  emit O2(V)
end var
```

Here V successively takes the values 1 and 2 in the same instant, O1 is emitted with value 1, and O2 is emitted with value 2.

Because of this, we must forbid arbitrary sharing of variables to ensure behavior determinism. The following behavior must be forbidden:

```
var V : integer := 0 in
  V := 1;
  V := 2;
||
  emit O(V)
end var
```

Since both assignments and the emission should occur in the same instant, there is no consistent value to emit for O and the program must be rejected.

There are two ways of forbidding shared variables:

- Requiring that no sharing conflict ever occurs at run-time. One should then report a run-time error, see Section **??**.
- Statically enforcing that this will be true, by some compile-time algorithm.

Note: In the Esterel Studio we require a variable not to be shared by two concurrent threads: a variable read by one branch of a parallel cannot be read nor written by any other branch of the parallel. In other words, a variable written by a computation thread ought to be local to this thread. This clearly rejects the above statement. More subtle static analysis could be performed by other compilers.

5.9 Verification signals: assertions, assumptions, and coverage points

5.9.1 General

Esterel v7_60 provides users with *verification signals*, which are *assertions*, *assumptions*, and *coverage points*. Unlike other signals, verification signals are not used to achieve the design functionality but are intended for the verification of the design and of the environment constraints.

Verification signals are directly declared in special signal equations of `emit` and `sustain` statements. An assertion is introduced by the `assert` keyword, an assumption is introduced by the `assume` keyword, and a coverage point is introduced by the `cover` keyword. For example, here are verification signal equations for a FIFO design:

```
input Put, Get;
output Full, Empty;
...
sustain {
  assume NoFifoFullError  = not(Put and Full),
  assume NoFifoEmptyError = not(Get and Empty),
  assert FifoFullEmptyExclusive = Full # Empty,
  cover FullCovered  = Full,
  cover EmptyCovered = Empty
}
```

In this chapter, we focus on the semantics of verification signals. For syntax issues, please refer to Section **??** and Section **??**.

5.9.2 Assertions and assumptions

Assertions and assumptions both expresses properties. An assertion or an assumption is specified as a Boolean expression. When the Boolean expression evaluates to true, we say that the assertion or assumption *passes* or is *valid*; when the Boolean expression evaluates to false, we say that the assertion or assumption *fails* or is *violated*.

The difference between assertions and assumptions lies in their intended interpration. Assertions express properties about the design whereas assumptions express properties about the design environment.In simulation, there is no difference between assertions and assumptions, except that when an assertion fails the problem is in the design, whereas when an assumption fails the problem is in the design environment, i.e. the design input sequence (testbench). In formal verification however, there is a fundamental difference: assertions are verified but assumptions are assumed to be always valid, that is, only input sequences respecting the assumptions are analyzed.

5.9.3 Coverage points

A coverage point is used to cover a Boolean expression, that is, to monitor that the Boolean expression evaluates to true. The intention of a coverage point is to signal that a certain logical condition has happened or has been covered. Coverage points do not pass or fail like assertions and assumptions; instead, they are either *covered* or *not covered*. A coverage point is covered if its Boolean expression is evaluated and evaluates to true. In simulation the hits of coverage points can be counted. In formal-verification, a coverage point can be proven unreachable by showing that no input sequence (respecting all assumptions) can cover the coverage point, or be covered by generating an input sequence (respecting all assumptions) that cover the coverage point.

Chapter 6

Interfaces and Module Headers

This chapter presents the declaration of signals and ports in interfaces and modules. We first present the signal declaration syntax in Section **??**, detailing all the attributes a signal declaration can have. We establish a distinction between two kinds of attributes:

- *Interface attributes*, which concern the way in which one can connect to a module interface signal from outside its module.
- *Module attributes*, which are more behavior-related and local to the module that owns the signal.

Signal declaration in interfaces can only involve interface attributes, and the other attributes of a signal must be given by *refinement declarations* in modules.

We then present hierarchical declarations of signals and ports within interfaces in Section **??**. We present input and output relations in Section **??**. We end the chapter by presenting module headers and the refinement of interface signals in Section **??**

Local signal and ports declarations in module bodies are deferred to Chapter **??**, where we study the `signal` local signal declaration statement in Section **??**.

6.1 Signal declaration overview

6.1.1 Signal declaration syntax

A signal declaration must occur after one the `input`, `output`, `inputoutput` or `signal` keywords. The first three keywords characterize *interface signal declarations*; they can only occur in interfaces and module headers. The `signal` keyword starts *module signal declarations*; it can only occur in module headers and bodies.

A signal declaration consists of the name of a signal, optionally followed by a dimension list for a signal array, and optionally followed by a colon and a list of attributes. One can use a single keyword for a comma-separated list of independent signal declarations. Furthermore, when there is an attribute list, a declaration can declare a list of signals or signal arrays sharing the same attributes by enclosing this list within curly brackets. Here are examples:

```
input I;
output X[4],
       {Y, Z[8]} : unsigned<M>[3];
inputoutput IO;
```

```
signal S : reg value signed init -2 combine + in
  ...
end
```

Local signal declarations can also declare ports, see Section ?? for details. We concentrate on interface signals in this chapter.

6.1.2 Signal declaration attributes

The attributes are as follows:

- The `reg` attributes declares the signal to be registered. Without it, the signal is standard. The signal must be output or local.
- The `value` attribute declares a valued signal to be value-only. Without it, a valued signal is full.
- The `temp` attribute declares a valued signal to be temporary. Without it, a valued signal is persistent.
- The type attribute declares the type of a valued signal. Without it, the signal is pure (and all attributes except `reg` are disallowed).
- The "`combine f`" declares a valued signal to be combined, with `f` as the combination function. One can replace `f` by `+`, `*`, `and`, `or`, or `xor` provided the type is adequate. Without the `combine` attribute, a valued signal is single.
- The "`init` *exp*" attribute declares the signal to be initialized by the given expression. Without it, a valued signal is uninitialized.

The `reg` attribute is the only one allowed for pure signals. All the other attributes require the signal to be valued, i.e. a type to be declared. The `reg` and `temp` attributes are exclusive. For a valued signal declaration, the `reg`, `temp`, and `value` attribute must appear before the type, in any order, while the `combine` and `init` attributes must appear after the type, in any order.

Here are some examples of valid attribute declarations (in module headers where all attributes are allowed):

```
output R : reg;
input  {X1, X2} : unsigned;
output Y : reg unsigned<[12]> init 0;
input  Z : temp value unsigned init 3 combine +;
output T : reg value bool combine or init false;
  ...
```

And here are examples of invalid declarations:

```
input  S : reg;                % no reg for input
input  S : temp;               % missing type
output R : reg value init 0;   % missing type
output X : temp reg unsigned;  % incompatible attributes
output Y : unsigned temp;      % wrong order
```

6.1.3 Interface vs. module attributes

Signal declaration in interface units are restricted. They must start with the `input`, `output`, or `inputoutput` keyword, and they can only involve the type, `value`, and `temp` attributes. These attributes are called *interface attributes*, while `reg`, `init`, and `combine` are called *module attributes*.

Interface attributes are exactly what is needed to understand how to connect to the signal from the global environment or from another module. The `value` attribute tells that the interface signal has a value but no status, while the `temp` signals declares how the value is driven. On the other hand, module attributes declare internal behavioral properties of signals that should not be known from external users of this signal.

For instance, consider a registered output `R` of a module `Sub`, which can be directly declared registered in the `Sub` module header:

```
module Sub :
  output R : reg;
  ...
  emit next R;
  ...
end module
```

Consider now `Sub` as a submodule of a module `M`, with a signal connection of the form "`run Sub[X / R]`". The fact that `R` registered is important for the behavior of `R` within `Sub`, but it irrelevant for the user `X` of this signal in `M`. Logically speaking, `X` simply sees the current status of `R` as it is produced by `Sub`, and the same holds for the value if the signals are valued. Changing `R` in `Sub` into a standard signal with exactly the same output behavior should be totally equivalent for `M`, which therefore should not know about the `reg` feature in `Sub`. For instance, here are two modules that perform the same, one with a standard output and one with a registered output:

```
module Sub1 :
  output O;
  pause;
  every 2 tick do
    emit O
  end every
end module

module Sub2 :
  output O : reg;
  every 2 tick do
    emit next O
  end every
end module
```

Since they are behaviorally equivalent, the only thing `Main` should know about them is their common interface, which is simply

```
interface SubIntf :
  output O;
end interface
```

Then the same `run` statement with the same signal renaming will work for both `Sub1` and `Sub2`.

Similarly, the `combine` and `init` attributes of a signal output by a module should be unknown to its external users:

- For `combine`, they should only see the result of the combination.
- for `init`, they should see the initial value as the value at initial instant, not different from any other value.

For a submodule input, we also disallow `combine` in interface units; it could in principle be allowed by handling multiple connections to a single input, but we feel that this mechanism is much too fancy and requires an explicit protocol which can be easily programmed in Esterel v7. We allow `init`, but only within the submodule: an `init` declaration gives the signal an initial value for the body in a way invisible to the external producers of the signal, see Chapter ??. Therefore we disallow `init` in interface units for input signals, restricting its use to module units.

Altogether, we forbid the `combine` and `value` attributes in interface units. Specifying these attributes in a module after having imported an interface requires a *refinement declaration*, presented in the next section.

6.1.4 Interface signal refinement

Back to the `M` / `Sub1` / `Sub2` example above, splitting the interface is obvious for `Sub1`:

```
interface SubIntf :
  output O;
end interface

module M1 :
  extends Intf;
  pause;
  every 2 tick do
    emit O
  end every
end module
```

Splitting the interface for `M2` uses very same interface `Intf`, but requires the addition of a *refinement* declaration in the module header to recover the missing attributes:

```
module M2 :
  extends Intf;
  refine O : reg;
  every 2 tick do
    emit next O
  end every
end module
```

A signal refinement declaration can only occur in a module header or a local signal declaration, see Section ??. It consists of the name of the signal followed by the added attributes. The type and `value` attributes already declared cannot be mentioned nor changed in the `refine` declaration. The `temp` attribute can be refined into `mem` or `reg`. With `mem`, the signal is actually memorized in the module, as if the `temp` keyword was simply ignored. With `reg`, the signal is declared registered. In both cases, the `temp` attribute is canceled. The `mem` and `reg` refinements are fundamental to allocate signal memories in designs, see Section ??. Here are examples of correct refinements:

```
interface Intf :
  input A : temp unsigned<M>;
  output X;
  output Y : signed<N>;
  output Z : temp bool;
end interface

module Mod :
  extends Intf;
  refine A : init 0 combine +;
  refine X : reg;
  refine Y : mem init -1;
  refine Z : combine and;
  ...
end module
```

And here are incorrect refinements of the same interface:

```
module Mod :
  extends Intf;
  refine A : value init 0;   % value forbidden
  refine X : reg unsigned;   % type addition forbidden
  refine Y : signed<N+1>;    % type change forbidden
end module
```

Finally, one can group refinements using curly brackets :

```
refine {A, B} : reg init 0;
```

6.2 Interfaces and Ports

Interface units describe the input / output structure of modules. Besides data declaration and extension already described in Chapter ??, interface declarations consist in signal, port, and relation declarations, plus interface extension. *Ports* are groups of signals typed by interfaces. They make the interface structure fully hierarchical, in addition to its object-oriented character that follows from extensibility. *Relations* are simple combinational behavior assertions about input or output signals.

In this section, we first describe *simple interfaces*, which are the leaves of the hierarchy. We define the *mirror* of an interface. Then, we define *ports* that lead to general interfaces. We give details about what full or selective extension means for interfaces. Finally, we present input and output relations.

6.2.1 Simple interfaces

A *simple interface* involves declarations and extension of data and declarations of signals starting with one of the directionality keywords `input`, `output`, or `inputoutput`. The declared signals can be typed, and, in that case, they can bear the additional interface attributes `value` and `temp`. The signals must have distinct names that do not conflict with other names visible from their declaration point. Here is a simple example:

```
interface Intf1 :
  type Byte = unsigned<[8]>;
  input I;
  output O : Byte;
  inputoutput IO : value Byte;
end interface
```

A simple interface can also contain relations, whose study is deferred to Section **??**.

6.2.2 The mirror of a simple interface

The *mirror* "`mirror Intf`" of a simple interface `Intf` declares exactly the same signals with opposite directionalities: `input` is changed into `output`, `output` is changed into `input`, while `inputoutput` stays the same. Input relations and output relations are also swapped, as will be explained in Section **??**. Mirroring has no effect on data. Of course, mirroring an interface twice yields it back unchanged. Here is an interface that extends the mirror of `Intf1` above, followed by its own mirroring:

```
interface Intf2 :
  extends mirror Intf1;
  output X;
end interface

interface ArrayInft2 :
  extends mirror Intf2;
end interface
```

In `Intf2`, `I` and `X` are outputs, `O` is an input, and `IO` remains an inputoutput. In `Intf3`, `I` and `X` are inputs, `O` is an output, and `IO` remains an inputoutput. Notice that the declaration of `Intf3` uses the extension mechanism just to give a name to the mirror of `Intf2`.

6.2.3 Ports

General interfaces can declare *ports*, where a port is a named groups of signals itself typed by an interface. A port is declared by its name, a colon, and an interface name possibly preceded by the `mirror` keyword. Here is an example:

```
interface Intf1 :
  type Byte = unsigned<[8]>;
  input I;
  output O : Byte;
  inputoutput IO : value Byte;
end interface

interface Intf2 :
  input J;
  port Direct : Intf1;
  port Mirror : mirror Intf1;
end interface
```

Of course, for the hierarchy to be consistent, dependency of interfaces through ports must have no cycle and port building must develop from simple interfaces.

The pointed notation is used to address port fields. In `Intf2`, `Direct.I` and `Mirror.O` are input signals, while `Direct.O` and `Mirror.I` are output signals. The signals `Direct.IO` and `Mirror.IO` are both inputoutput signals. Using curly brackets to represent interface boundaries, we can picture `Intf2` as follows:

```
interface Intf2 :
  input J;
  port P : { input I;
             output O : Byte;
             inputoutput IO : value Byte; };
  port Mirror : { output I;
                  input O : Byte;
                  inputoutput IO : value Byte; };
end interface
```

Warning: this notation is just for explanation; it is not accepted in Esterel.

When mirroring a general interface, the port interfaces are recursively mirrored. In the "`mirror Intf2`" mirror interface, `Direct.I` and `Mirror.O` become output signals, while `Direct.O` and `Mirror.I` become input signals.

Signals and declared ports in an interface must have distinct names; however, there is no name conflict between components of a port and the objects declared in the interface where the port is declared. An input `I` added to `Intf2` would not conflict with `Direct.I` nor with `mirror.I`.

To ease the declaration of local signals in module bodies, interfaces can be extended and ports can be declared and refined in local signal declarations, see Section **??**. The `open` delcaration makes it possible to locally forget about the dot notation by setting `I` as a synonym for `P.I`, see Section **??**.

6.2.4 Port arrays

One can also declare arrays of ports in an interface:

```
interface Intf3 :
  constant Size : integer;
  input J;
  port DirectArray[Size]  : Intf1;
  port MirrorArray[Size]  : mirror Intf1;
end interface
```

Port fields addressing uses a mix of dot notation and array indexation, as in '`DirectArray[3].I`'. Dimensions add up for signals and values. Consider the declarations

```
interface ArIntf :
  input I[32];
  output O : bool[8];
end interface


interface ArPortIntf :
 port P[16] : ArIntf;
end interface
```

Then `P.I` has dimension `[16][32]`, while `?P.O` is of type `bool[16][8]`. However, as said before, indexing is mixfix: one writes `P[i].I[j]` and `?P[i].O[j]`, not `P.I[i][j]` and `?P.O[i][j]`.

6.3 General Interfaces

We now present general interfaces, obtained recursively from simple interfaces by extension and port definition. Data extension was described in Chapter ??. Signal extension can be full or selective, i.e. limited to input, output, or inputoutput signals.

6.3.1 Full interface extension and port definition

Full interface extension consists of the `extends` keyword, optionally followed by `mirror` and `observe`, and followed by the interface name. Full extension imports all the signals and ports declared in the base interfaces. When `mirror` is used alone, the signal directionalities are mirrored. When `observe` is used alone, the signals are all imported as inputs. When "`mirror observe`" is used, the signals are all imported as outputs. Interface extension can be combined with port definition in the declaration of an interface. Here is an example using `Intf1` and `Intf2` of Section ??:

```
interface Intf1 :
  type Byte = unsigned<[8]>;
  input I;
  output O : Byte;
  inputoutput IO : value Byte;
end interface

interface Intf2 :
  input J;
  port Direct : Intf1;
  port Mirror : mirror Intf1;
end interface

interface Intf3 :
  extends Intf2;
  port P : Intf2;
  port Q : mirror Intf2.
end interface

interface Intf4 :
  extends observe Intf2;
  port R : mirror observe Intf2;
end interface
```

Here, `Intf3` directly extends `Intf2` and has two additional ports involving `Intf2`. Besides, `Intf4` observes `Intf2` and has an additional port involving `Intf2`. Using the curly bracket notation, we can picture `Intf3` and `Intf4` as follows:

```
interface Intf3;
  input J;
  port Direct : { input I;
                  output O : Byte;
                  inputoutput IO : value Byte; };
  port Mirror : { output I;
                  input O : Byte;
                  inputoutput IO : value Byte; };
  port P : { input J;
             port Direct : { input I;
```

```
                              output O : Byte;
                              inputoutput IO : value Byte; };
                port Mirror : { output I;
                              input O : Byte;
                              inputoutput IO : value Byte; }; };



    port Q : { output J;
                port Direct : { output I;
                              input O : Byte;
                              inputoutput IO : value Byte; };
                port Mirror : { input I;
                              output O : Byte;
                              inputoutput IO : value Byte; }; };

  end interface



  interface Intf4;
    input J;
    port Direct : { input I;
                    input O : Byte;
                    input IO : value Byte; };
    port Mirror : { input I;
                    input O : Byte;
                    input IO : value Byte; };
    port R : { output J;
                port Direct : { output I;
                              output O : Byte;
                              output IO : value Byte; };
                port Mirror : { output I;
                              output O : Byte;
                              output IO : value Byte; }; };

  end interface
```

There is no name conflict in `Intf3` and `Intf4`, since there is no confusion between `J`, `P.J`, `Q.J`, etc. However, extending both `Intf2` and "`mirror Intf2`" in the same interface would generate a name conflict for all the components of `Intf2`.

6.3.2 Selective interface extension

Selective interface extension or port definition extends or defines a port with only the inputs or outputs of an interface, descending port interfaces recursively to gather these inputs or outputs. The `extends` keyword is followed by an optional `mirror`, by `input`, `output`, or `inputoutput`, and by the interface name.

For "`extends input Intf`", all the input signals of `Intf` are imported, the ports of `Intf` being imported with interface restricted to their input components. For "`extends mirror output Intf`", all the output signals and port components of `Intf` are imported mirrored, i.e. as inputs. Extension is symmetrical with `output` instead of `input`.

Building observers by selective extension

A common practical usage is to transform all outputs of an interface into inputs. This is useful to build *observers* of programs, i.e. additional modules that take all interface signals of a given module as input and compute some properties of the whole set of signals. Here is an example:

```
module M :
  input I;
  output O;
  ...
end module

module Observer :
  extends input M;
  extends mirror output M;
  sustain assert O_only_if_preI = O => pre(I)
end module
```

the first `extends` directly imports the input `I`. The second `extends` imports `O` as an input, since `M`'s interface is mirrored before extracting the inputs. The `Observer` module inputs `I` and `O` and checks their temporal dependency.

Selective extension examples

Let us exemplify further extensions with ports. With `Intf1`, and `Intf2` as above, consider the following selective extensions, which can be seen as restrictions of `Intf3`:

```
interface Intf3In :
  extends input Intf3;
end interface

interface Intf3Out :
  extends output Intf3;
end interface
```

These interfaces are stripped versions of `Intf3` reduced to input or output signals. They can be pictured as follows, using curly brackets as interface delimiters:

```
interface Intf3In :
  input J;
  port Direct : { input I;  };
  port Mirror : { input O : Byte; };
  port P : { input J;
             port Direct : { input I; };
             port Mirror : { input O : Byte }};
  port Q : { port Direct : { input O : Byte };
             port Mirror : { input I; }};
end interface

interface Intf3Out :
  port Direct : { output O : Byte;  };
  port Mirror : { output I; };
  port P : { port Direct : { output O : Byte; }
             port Mirror : { output I }};
  port Q : { output J;
             port Direct : { output I };
             port Mirror : { output O : Byte; }};
end interface
```

Finally, consider the following interface:

```
interface Intf4 :
  extends input Intf2;
  extends inputoutput Intf2;
  port P : output Intf2;
  port Q : mirror input Intf2;
end interface
```

Then, `Intf4` can be pictured as follows.

```
interface Intf4 :
  input J;
  inputoutput IO : value Byte;
  port Direct : { input I; };
  port Mirror : { output O : Byte; };
  port P : { port Direct : { output O : Byte; }
             port Mirror : { output I }};
  port Q : { port Direct : { output I; };
             port Mirror : { output O : Byte; }};
end interface
```

6.3.3 Data renaming for generic interface extension

If an interface has generic components, one can instantiate them at instantiation time using the same syntax as for data extension:

```
interface Intf :
  generic constant N : unsigned;
  input I[N];
  output O : unsigned[N];
end interface



interface Intf32 :
  extends Intf [constant 32 / N];
end interface
```

6.3.4 Interface extension does not share components

.

Signal extension differs from data extension on another point. There is no sharing between common base interfaces, cf. Section **??**. For instance

```
interface Intf:
  input I;
end interface

interface Intf1 :
  extends Intf;
end interface

interface BadIntf :
  extends Intf;
  extends Intf1;
end interface
```

Here, it is considered that `I` is declared twice and the program is rejected. Multiple inheritance for interfaces would be much more complicated than the same for data because of ports.

6.4 Relations

A *relation* is a possibly named simple instantaneous (combinational) predicate bearing either on statuses of input signals only for an *input relation* or on statuses of output signals only for an *output relation.* For reasons explained later on in this section, inputoutput signals cannot appear in relations and there can be no mixed relation involving both inputs and outputs.

The predicates involve the signal operators defined in Chapter ??, i.e. the classical `and`, `or`, `xor`, and `not` connectives plus multiplexor `mux`, implication '`=>`', equivalence '`<=>`', and exclusion '`#`'.

When taking the mirror of an interface, an input relation becomes an output relation and conversely.

6.4.1 Input relations

An *input relation* of an interface `Intf` is a possibly named predicate on input signal statuses. It expresses an assumption over the environment of a module: at any instant, the signals sent by the main or local environment of a module or port of interface `Intf` are assumed to satisfy all input relations. This can be checked in simulation or assumed in verification and optimization. In verification, a property may be valid only if the input relations are satisfied. In optimization, input relations can be used as *input don't cares* [?, ?] to optimize circuits further. Here are examples:

```
  input relation Minute => Second;
  input relation RadioButtons: A # B # C;
```

The first unnamed relation expresses a basic fact about time units. The second named relation expresses that the input button signals `A`, `B`, and `C` are exclusive, i.e. that at most one of them can be present at each instant.

Relation names are semantically unimportant, but they can be useful in programming environments, for example to report violated relations in simulations.

Port components and explicit array elements can appear in relations, but full arrays cannot. Here is an example of a relation involving ports and arrays:

```
  interface Intf1 :
    input I[4] : integer;
    output O;
  end interface
```

```
interface Intf2 :
  input J;
  port P : Intf1;
  port Q : mirror Intf2;
  input relation J => (P.I[2] or Q.O)
end interface
```

In practice, '#' exclusion input relations are very useful for optimization and verification, since they drastically restrict the input event space. For example, with n inputs declared exclusive, the number of input events is $n + 1$ (the empty event and all one-signal events) instead of 2^n. One can also use input relations to momentarily stick a signal at *present* or *absent*, which is often useful to debug programs. Here is a way to stick `A` *present* and `B` *absent*:

```
input relation A and not B;
```

6.4.2 Output relations

Dually, an output relation is a predicate on outputs which must hold for any state of a module and any input that satisfies the input relations:

```
output relation (O1 and OA[1]) => not O2;
```

Ports and arrays are handled as for input relations.

Unlike input relations, output relations are not assumed to hold by default for a module or port which extends the interface, since their truth results of the behavior of the module itself. They should be checked at simulation or formal verification time. They can also serve as output don't care conditions for logic synthesis, see [**?**].

Output relations of an interface `Intf` become input relations of the mirror interface "`mirror Intf`". Therefore, they are essential for optimization and verification of modules that extend "`mirror Intf`".

6.4.3 Relation inheritance

Relations are automatically imported by interface extension, and they are mirrored if the extension is mirrored: the mirror of an input relation is an output relation with the same expression and conversely.

There is a more subtle extension scheme: a relation declared in an interface `Intf` is automatically propagated to all ports and port arrays of interface `Intf` or "`mirror Intf`". Consider the following example;

```
interface Intf :
  input I, J;
  output X[2];
  input relation I => J;
  output relation X[0] # X[1];
end interface

interface Intf2 :
  port P : Intf;
  port Q[2] : mirror Intf;
end interface
```

Then the following leaf signal relations are automatically inferred for `Intf2`

```
input relation P.I => P.J;
output relation P.X[0] => P.X[1];
output relation Q[0].I => Q[0].J;
output relation Q[1].I => Q[1].J;
input relation Q[0].X[0] # Q[0].X[1];
input relation Q[1].X[0] # Q[1].X[1];
```

6.4.4 Why not allowing more general relations

Esterel v7 interface relations are voluntarily limited in expressive power. One could consider much more elaborate relations, involving simultaneously inputs and outputs and sequential `pre` operators, but it would not be clear where to stop and intuition might get lost.

First, by defining relations between input and outputs, one could fully specify the behavior of a module without the need to give it a body. Here is an and-gate example:

```
interface AndIntf :
  input I, J;
  output O;
  relation (I and J) <=> O;
end interface
```

However, notice that the mirror relation does not define a deterministic behavior and cannot be used as a module specification.

Second, using the `pre` operator, one could state that an input signal `I` alternates between *present* and *absent*:

```
input relation I xor pre(I);
```

But where should we stop? Should we also include arbitrary temporal logic predicates, including fairness ones? Our decision is to stick to simple but proven useful combinational input-only and output-only relations within the language. We leave the more elaborate mixed and sequential relations to the programming environment, where fancy temporal assumption can be used to define complex sequential environment or conditions to check, see for example reference [**?**].

6.5 Module headers

Module headers contains the declaration of all objects used by the module. They are similar to interface declarations, except that all attributes are allowed in `input`, `output`, and `inputoutput` signal declarations, and that refinement declarations can be added to specify module attributes for interface signals imported either by interface extension or by port declarations.

We have seen refinement examples in Section **??**. Here is another example:

```
interface Intf :
  input I[32] : integer;
  output O : integer;
end interface
```

```
module M :
  extends Intf;
  refine I mem init {32{0}};
  refine O : reg combine +;
  port P : mirror Intf;
  refine P.I : reg1;
  ...
end module
```

Notice that array dimensions do not appear in refinement of signals or port arrays.

When considering M as an interface, i.e. in an "`extends interface M`" declaration, only the directionalities and interface attributes of signals (type, `temp` and `value`) are retained, the other module attributes being discarded.

Last, local signals and ports can be declared in a module header with the `signal` keyword. The interest is to make them visible by observers, see Section **??**. The scope of local signal and ports declared in a module header is the module body. The syntax is the same as the local signal declaration statement in module body (see Section **??**), except that the declaration finishes with '`;`' instead of the "`in ... end signal`" bracketing the local declaration statement scope. Here is an example:

```
interface Intf :
  input I[32] : integer;
  output O : integer;
end interface

module N :
  signal extends Intf,
         refine I mem init {32{0}},
         refine O : reg combine +;
  signal port P : Intf, refine P.I : reg1;
  ...
end module
```

6.5.1 Memory assignment control by mem and reg refinements

In hardware designs, it is frequent to broadcast a valued signal generated by a producer module to a set of consumer modules, with the producer generating the value from a memory and the consumers viewing the value as temporary and not memorizing it. In this case, it is best to declare the signal as `temp` in a common interface, with a mirror for either the producer or the consumer, and to refine it either `mem` or `reg` in the producer. Here is an example where two signals are exchanged in this way between two modules:

```
interface Intf :
  input X : temp unsigned;
  output Y : temp unsigned;
end interface



module GenY :
// reads X as temp, generates Y from own memory
extends Intf;
  refine Y : mem;
```

```
    ...
    emit ?Y <= ...
    ...
  end module

  module GenX :
    // reads Y as temp, generates X from own memory
    extends mirror Intf;
    refine X : reg;
    ...
    emit next ?X <= ...
    ...
  end module
```

Here, `Y` keeps a defined value from its first emission by `GenY` on, while `X` keeps a defined value from the tick that follows its first emission by `GenX` on. ‘

Chapter 7

Expressions

Expression are built from literal, Boolean signal status tests, signal values, variables, literals, and enumeration values using operators and function calls. They are strongly typed, with limited operator overloading and implicit conversion to avoid ambiguities. They can be freely parenthesized. Expressions are either *simple expressions* of non-array type or *array expressions* of array type. An expression of type `bool` is called a *test*, an array expression of base type `bool` is called an *array test*.

Signals and variables can be *fully indexed* by simple expressions, yielding a simple result, or they can be *sliced* if they are arrays, returning an array of the same base type and reduced dimension. One can slice several dimensions, as in `X[3..5][6..10]`.

An expression is *statically evaluable* if it only involves constants operators, and predefined functions. A *dimension expression* is a statically evaluable simple expression of unsigned or signed type, with result bigger than 1 (dimension 0 is disallowed).

Assume the following declarations:

```
input S : unsigned;
input T;
output X[8][10];
input Y[10]
var V : unsigned in ... end
```

Here are well-typed expression examples illustrating the various kinds of objects:

```
S and T                       // status expression
S and ?S>0                    // status + value test
X[1][2] and S                 // full array indexation, status and
X[1] [or] Y                   // pointwise array expression
X[1][2..3] [xor] Y[9..10]     // slice expressions
V + ?S                        // addition of a variable and a signal value
mux(?S>0, V+1, bin2u(Y))      // mux between data expressions,
                              // bitvector-to-unsigned conversion
```

Remark: unlike in the previous version Esterel v7_10, expressions now freely mix signal statuses and values. A signal status is simply wiewed as a Boolean or as an array of Booleans. However, compilers can internally perform very different calculations on statuses and values because data and control optimization rely on different algorithms. For instance, to optimize control, the Esterel v7_60 optimizer for Esterel Studio performs status extraction, sequential reachability analysis, and status expression optimization, while leaving value expression basically unchanged. This has performance impacts for the generated code and on formal verification, but no impact on meaning.

7.1 Expression Leaves

Expression leaves are the terminal elements of expressions. There are literals, already described in Chapter **??**, signal status tests and `pre` tests, and signal values or `pre` values, and variable values.

7.1.1 Signal status tests

The status of a signal `S` is tested by the trivial expression '`S`', possibly indexed if the signal is an array. The base type of the result is `bool`.

Assume that `S` is a signal array such as `S[`M`][`N`]`. Then, one can return a Boolean by fully indexing `S` and a Boolean array by slicing `S`. A slice is specified by a pair `[`k`..`l`]` of statically evaluable unsigned or positive signed numbers k and l with $k \leq l$. A slice with $k = l$ such as `S[2..2]` defines a array of type `bool[1]`, not a `bool`. A *full slice* for a dimension `[`M`]` is written `[..]`. It spans the entire dimension, i.e. is equivalent to `[0..`$M-1$`]`. There can be slices on several dimensions, and full indexations and slices can be mixed. The dimension of a sliced array is obtained by concatenating the elementary slice dimensions from left to right. For instance, `X[2..5][3..12]` has dimension `[4][10]`.

An indexation is called *partial* if not all dimensions are indexed or sliced, in which case the missing slices are assumed to be full.

Here are legal indexations for `S[5][6]`:

```
       S[1][3]  :  bool
    S[1..3][2]  :  bool[3]
    S[1][2..4]  :  bool[3]
      S[..][2]  :  bool[5]
          S[1]  :  bool[6]  =  S[1][..]
 S[1..3][2..5]  :  bool[3][4]
       S[k..l]  :  bool[l-k+1][6]   k and l statically evaluable, k ≤ l
```

Slices are renumbered from 0 for furter indexations. For instance,

```
(S[3..5][2]) [1]  =  S[4][2]
```

Notice that the parentheses that isolate `S[3..5][2]` are mandatory, since the indexation `S[3..5][2][1]` would be considered a 3-dimensional indexation and therefore ill-typed.

7.1.2 Testing the previous status of a signal

The previous value of a signal is tested using the `pre` and `pre1` operators, with `S` indexed or sliced as before if it is an array. For `pre`, the initial value at signal life first instant is *true*, while it it *false* for `pre1`. The initial value applies elementwise for arrays.

Here are legal expressions for `S` simple and `T[5][6]` an array:

```
pre(S)
S and not pre(S)      // rising edge of S
not S and pre1(S)     // falling edge of S
pre1(T)               // array pre1(T)[5][6]
pre(T[2][5..7])       // slice of dimension [3]
```

Consider the second expression above, falling edge: because of the use of `pre1`, the edge is considered falling at index i, j if `T[`i`][`j`]` is absent at first instant. Using `pre` instead of `pre1` would make the edge never falling at first instant.

The `pre` and `pre1` operators can be extended to expressions containing only signal status terms, with the following rules:

$$\begin{aligned}
\texttt{pre(not E)} &= \texttt{not pre1(E)}\\
\texttt{pre1(not E)} &= \texttt{not pre(E)}\\
\texttt{pre(E1 or E2)} &= \texttt{pre(E1) or pre(E2)}\\
\texttt{pre(E1 and E2)} &= \texttt{pre(E1) and pre(E2)}\\
\texttt{pre(E1 xor E2)} &= \texttt{pre(E1) xor pre(E2)}\\
\texttt{pre(E1 \# E2 \# ... \# En)} &= \texttt{pre(E1) \# pre(E2) \# ... \# pre(En)}
\end{aligned}$$

Nesting `pre` is not possible. Therefore, `pre(pre(S))` is disallowed. One must use an auxiliary signal to compute it, as in the following example:

```
signal preS in
  sustain {
    preS <= pre(S),
    O <= pre(preS)
  }
end signal
```

To build a more general `pre` array, one usually builds a shift register. This is very easy in Esterel:

```
module ShiftReg :
generic constant M : unigned;
input S;
output PreArray[M];
sustain {
  PreArray[0] <= S,
  PreArray[1..M-1] <= pre(PreArray[0..M-2])
}
end module
```

7.1.3 Reading signal values

Given a valued signals `S`, the expression `?S` reads the value of the signal in the current instant. If `S` is a simple signal of basic type, the expression `?S` has the type of `S`. If `S` is a signal array or if it has an array type, then `?S` is a data array expression with dimension the full dimension of `S`, i.e. the concatenation of the signal dimensions and the data dimensions of `S`. Consider the following declarations:

```
input I[12] : bool;
type Word = bool[32];
output
O : Word;
signal S[3] : Word in ... end
```

Then ?I has type bool[12], ?O has type Word = bool[32], and ?S has type bool[3][32]. The distinction between status and value dimensions disappeard when reading the value of a signal.

Full indexation and slicing is exactly as for statuses, see Section ??. Here are slices and their types:

?I[2..4]	:	bool[3]
?O[12..23]	:	bool[12]
?S[..][12..23]	:	bool[3][12]
?S[1..2]	:	bool[2][32]
?S[1..2][12..23]	:	bool[2][12]

7.1.4 Reading signal previous values

The previous value of a signal S is read using the expression pre(?S). Value pre applied to a single signal and cannot be extended to expressions, unlike for status. For instance, pre(?S + ?T) is disallowed. As for status, valued pre cannot be nested.

If S is a signal array, one use indexing and slicing inside pre, as for pre(?S[2][3..4]). Indexation and slicing works exactly as for standard value read.

7.1.5 Reading variable values

The value of a variable V is read by the trivial expression V, indexed and sliced as for signal status and value if the type is an array type.

7.1.6 Port component access

For ports, one read the status or value of a leaf signal using the dot notations P.S and ?P.Q.S.

If the final signal S is an array, one can index and slice it as any other signal array. If the port is an array, one must index it fully, as in P[12].S or ?P[12][4].Q[k].S[5]. One can index or slice the final signal if it is an array, as in P[1].S[3..4], or ?P[12].Q[k].S[5..7]. but one cannot slice the port array itself: P[1..2].S[3] is disallowed.

The status and value pre operators can be applied to ports, as for pre(p.S[2]) or pre(?P[1].X).

7.2 Operators

Operators apply to basic objects and can be extended to arrays by putting them into brackets, as for [or] and [+]. Note that array extension is not automatic to avoid ambiguities.

operator	**symbol**	**type**	**assoc.**
negation	`not`	`bool`	right
unary minus	`-`	numerical	right
unary plus	`+`	numerical	right
power	`**`	numerical	right
multiplication	`*`	numerical	left
division	`/`	numerical	left
modulo	`mod`	unsigned	left
addition	`+`	numerical	left
subtraction	`-`	numerical	left
left shift	`<<`	bitvector	none
signed left shift	`<<<`	bitvector	none
right shift	`>>`	bitvector	none
signed right shift	`>>>`	bitvector	none
less than	`<`	numerical	none
less than or equal to	`<=`	numerical	none
greater than	`>`	numerical	none
greater than or equal to	`>=`	numerical	none
equality	`=`	basic, bitvector (non-array)	none
unequality	`<>`	basic, bitvector (non-array)	none
exclusive or	`xor`	`bool`	left
conjunction	`and`	`bool`	left
disjunction	`or`	`bool`	left
implication	`=>`	`bool`	right
equivalence	`<=>`	`bool`	left
exclusion	`#`	`bool`	truly n-ary

Figure 7.1: Data operators, ordered by decreasing precedence.

The operators and their precedence table are presented in Figure **??**. In this figure, 'left' means left-associative, 'right' means right-associative, 'none' means non-associative (i.e. one cannot write "`X <= Y <= Z`"), and "truly n-ary" means that the operator has no associative character and takes all arguments as a single list, as for "`X # Y # Z`". An integral type is defined as a signed or unsigned type, and a numerical type is an integral type, `float`, or `double`. When used on mixed types, integral operators promote unsigned to signed, and numerical operators promote unsigned to signed to float to double.

The operator `<=>` is another way of writing Boolean equality `=`, often more conventional and more readable.

.

7.2.1 Applying operators to arrays

All operators are extended to arrays in a pointwise way, provided they are enclosed in square brackets as for `[+]`. Extension is not fully implicit to avoid ambiguities. The argument array dimensions must match, and the resulting array has the same dimension.

function	**type**
`bin2u`	(bitvector) : unsigned
	(bitvector, unsigned) : unsigned
`binsize`	(unsigned) : unsigned
`extends`	(bitvector) : bitvector
`gray2u`	(bitvector) : unsigned
	(bitvector, unsigned) : unsigned
`lcat`	(bitvector, . . . , bitvector) : bitvector
`mcat`	(bitvector, . . . , bitvector) : bitvector
`mux`	$(\texttt{bool}, T, T) : T$
`onehot2u`	(bitvector) : unsigned
	(bitvector, unsigned) : unsigned
`reverse`	(bitvector) : bitvector
`sextends`	(bitvector) : bitvector
`u2bin`	(unsigned) : bitvector
	(unsigned, unsigned) : bitvector
`u2bool`	(bool) : unsigned¡2¿
`u2gray`	(unsigned) : bitvector
	(unsigned, unsigned) : bitvector
`u2onehot`	(unsigned) : bitvector
	(unsigned, unsigned) : bitvector

Figure 7.2: PredefinedFunctions.

Its base type is given by that of the operator when applied to single arguments of array argument base types.

Here is a way to add two arrays and to check whether each position is positive, returning an array of Booleans:

```
(X [+] Y) [>] 0
```

7.3 Function calls

Function calls are performed in the usual way by passing comma-separated arguments to the function. Figure **??** gives the list of predefined functions.

The `mux`(c, x, y) multiplexor function takes a first Boolean argument and is type-generic in its second and third arguments. It returns x if c is true, and y if c is false.

7.4 Array indexation

Array indexation or slicing cannot be applied to arbitrary expressions. Here are the cases where they are meaningful:

- Constant indexation or slicing, for instance `C[`i`]` where C is a constant of array type.
- Variable indexation for a variable of array type, for instance `V[1..3][`j`]]`.

- Signal status indexation for a signal array, for instance `S[..][1][2..3]`.
- Signal value indexation for a signal array or for a signal or signal array of array type, for instance `?S[2][1]`. Remember that signal and value dimensions are concatenated for signal arrays, see Section **??**.
- Encoding expressions, as for `u2bin(`u`)[`m`]`, `u2onehot(`u`)`[2..5], or `s2bin(`x`)[`k`]`.

For all other cases, one must use an auxiliary variable.

Chapter 8

Statements

In this chapter, we presents the executable statements that define the behavior of Esterel v7 programs.

8.1 Statement overview

Without redefining it here, we recall the basic Esterel statements behavior which is formally defined in [?].

Statements have two inter-dependent roles: propagating the control and emitting signals. As far as control is concerned, a statement which is executed in an instant has one of three exclusive elementary behaviors:

- *Termination:* the statement completes its execution and releases the control.
- *Pausing:* the statement holds the control at some point for the next instant, and it restarts from that point at next instant, unless killed or aborted.
- *Trap exit:* the statement executes an "`exit T`" statement which provokes immediate termination of the corresponding `trap` statement, killing all concurrent threads in the `trap` scope.

Basic control propagation statements perform elementary control actions, see Section **??**. Assignment and procedure call deal with variables, see Section **??**. Signal status and value emission is performed by the `emit` and `sustain` statements, which can also check for run-time assertions, assumptions, and coverage points, see Section **??**. Sequencing and looping provide basic control propagation, see Section **??** and Section **??**. Control is switched according to Boolean tests by `if` statements, see Section **??**.

The parallel statements `||` and `for-dopar` fork the control into synchronous threads, see Section **??**. At each instant, they terminate when all the threads are terminated or pause if at least one thread pauses, except if one of the threads exits a `trap`, in which case the outermost exit is propagated.

Temporal statements wait for delays to start, suspend, or kill other statements, see Section **??**.

The `trap` statement provides structured forward gotos, useful to exit from loops and parallel statements and for many other purposes, see Section **??**.

The `finalize` statements allows cleanup actions when a statement is terminated or killed, see Section **??**.

Finally, local signal declarations provide signal scoping, see Section **??**, open port declarations ease the use of ports, see Section **??**, and local variable declarations provide variable scoping, see Section **??**.

The language is fully orthogonal: all statements can be freely mixed and in an arbitrary way. One can sequence parallel statements or put sequences in parallel, one can subject any statement to an abortion, etc. There is no limit to statement nesting.

8.1.1 Syntactic matters

All statements except sequencing '`;`' and concurrency '`||`' use bracketing keywords: one writes "`abort ... end abort`", etc. Repetition of the initial keyword is optional, so that "`abort ... end`" is also correct. To resolve the remaining syntactic ambiguities, any statement can be explicitly bracketed using curly brackets '`{...}`'.

In the sequel, we do not increment indentation for the parallel bars in parallel statements. Therefore,

```
signal S in
  p
||
  q
end signal
```

is preferred to the more indented form

```
signal S in
    p
  ||
    q
end signal
```

8.2 Basic control statements

There are three basic pure control statements:

```
nothing
pause
halt
```

Their behavior is as follows:

- The `nothing` statement terminates instantaneously when started.
- The `pause` statement pauses for one instant. More precisely, it pauses when started, and it terminates in the next instant
- The `halt` statement pauses forever and never terminates. Beware, any code that follows a `halt` statement in sequence will be dead code.

Notice that `halt` is equivalent to "`loop pause end`", see Section **??**. Also, `pause` can be written "`await tick`", see Section **??**.

8.3 Variable handling statements

8.3.1 Assignment

An assignment "*lhs* `:=` *e*" assigns the value of a data expression *e* to a left-hand side variable cell *lhs*. For a *simple assignment*, the left-hand-side is a simple cell designator, i.e. either a non-array variable `V` or a fully indexed array variable such as `A[N+1][?S]`. For an *array assignment*, the *lhs* can be a full array `A` or a slice such as `A[1..2]` or `B[..][1..4][2..5]`. Then *e* must be an array expression of matching type and dimension, see Section **??**. Array assignment is performed cell by cell for the given variable indices. Elementary and array assignments are instantaneous. Here are examples:

```
V := V+1
A[3][3] := 3.14f
A[1..2][2..5] := B[1..2][2..5] [xor] X[3..4][8..11]
```

8.3.2 Procedure call

Procedure calls have the form

```
call P (e1, e2,..., en)
```

where `P` is a procedure name and the e_i are expressions. The arity and expressions types must match those of the declaration, see Section **??**. The expression constraints are as follows, distinguishing between simple non-array arguments and array arguments:

- For `in` simple arguments, the expressions are arbitrary provided their types match, and they are passed by value.
- For `out` or `inout` simple arguments, the expression must refer to a simple variable cell, as for the left-hand side of a simple assignment, see Section **??**. The cell is passed by reference.
- For `in` array arguments, the expression must be a single array identifier and the array is passed by value (warning: this can be expensive; however, an implementation can use passing by reference if it is guaranteed equivalent). One cannot pass slices nor partially indexed arrays (because it would be too difficult to generate the code for many host languages).
- For `out` or `inout` array arguments, the expression must be a simple array identifier. The array is then passed by reference. One cannot pass slices nor partially indexed arrays.

Consider the following procedure declaration:

```
procedure P (in unsigned,
             inout T;
             in T[5],
             out T[5]);
```

Here is a legal call with `X`, `Y`, and `Y` arrays of base type `T`:

```
call P (?I[2] + 1,  // expression for in argument
        X[1],       // variable cell for simple inout
        Y,          // array variable for in array
        Z);         // array variable for out array
```

8.4 Emit, sustain, assert, assume, and cover

Instantaneous signal emission is realized by the `emit` statement, while sustained emission is realized by the `sustain` statement. The `emit` statement immediately terminates, while the `sustain` statement never terminates and keeps emitting until preempted. Besides the initial keyword, `emit` and `sustain` share the same syntax.

The `emit` and `sustain` statements can also define *verification signals*, which represent a standard way of stating properties of programs or hypotheses about the environment one wants to rely on (see Section **??** for the semantics of verification signals).

8.4.1 Emission overview

The body of an `emit` or `sustain` statement is composed of *simple emissions*, which are equations dealing with the emission of one signal, *simple emissions with case lists*, where the equation right-hand-side can be decomposed into a case list, *concurrent emissions*, which group simple emissions in parallel, possibly within `for`-loops, and *conditional emissions* which provide the user with conditional `if-then-else`, `if-case`, and `switch-case` structures. Concurrent and conditional constructs can be freely merged to any depth.

Adding the `seq` keyword after `emit` or `sustain` makes the emission *sequenced*: the simple emissions are taken in order instead of concurrently. This makes it possible to define combinational carry structures.

8.4.2 Simple emission

A simple emission has one of the following four forms:

emit *lhs*
emit *lhs* **if** *t*
emit *lhs* `<=` *rhs*
emit *lhs* `<=` *rhs* **if** *t*

A sustained emission has exactly the same form, with `sustain` replacing `emit`.

In an emission, *lhs* is the *left-hand-side*, *rhs* is the optional *right-hand-side*, and *t* is the optional *test*. The test is always a simple Boolean expression (not an array expression). Signal emission, assertion, assumption or coverage point is said to be *conditional* if there is a *rhs* or an `if` test, *unconditional* otherwise.

There are three kinds of simple *lhs* : *pure emission*, *valued emission*, and *verification signal definition*.

- For pure emission, *lhs* is a status or next status designator such as `S`, "`next R[1]`", `P.S`, or "`next Q[2].R[3][?I]`" where `P` and `Q` are ports and `R` is registered. The `next` keyword must be used if and only if the signal or port component is registered. There are two pure emission subkinds:
  - For pure simple emission, *lhs* is fully indexed to designate a terminal non-array status component, and *rhs* is a Boolean expression.
  - For pure array emission, *lhs* designates a signal array or slice, and *rhs* is a Boolean array expression.
- For valued emission, *lhs* is a value or next value designator such as `?S`, "`next ?R[1]`", `?P.S`, or "`next ?Q[2].R[3][?I]`" where `P Q` are ports and `R` is registered. As for

status, the `next` keyword must be used if and only if the signal or port component is registered. The indices can correspond either to signal dimensions or to value type dimensions. There are two valued emission subkinds:

  – For valued simple emission, *lhs* designates a simple non-array value cell, and *rhs* must be a data expression of the same type.
  – For valued array emission, *lhs* is sliced or not fully indexed, and *rhs* must be a data array expression of the same type.

- For verification signal definition, *lhs* has the form "`assert` *A* `=`", "`assume` *A* `=`", or "`cover` *A* `=`". It defines a verification signal, i.e. an assertion, an assumption, or a coverage point. The verification signal has scope the module in which it is defined. No two verification signals in the same module can have the same name.

If it exists, the test *t* is executed first and the emission is abandoned if the test returns *false*. For a pure signal, the signal or signal component is emitted if the Boolean *rhs* evaluates to *true*. For a valued signal, the signal is emitted with value the result of the evaluation of *rhs*.

We details the difference cases in the sequel: signal emission from Section **??** to Section **??**, and verification signals in Section **??**. Grouping conventional signal emissions and verification signal definitions is very convenient in practice, since verification properties are most often about the status or values of signals.

8.4.3 Pure simple emission

A *pure simple emission* emits or sustains a pure signal. The *lhs* is a simple status designator, and the *rhs* is a simple Boolean expression. Here are examples:

```
emit S                              // pure simple signal
sustain next A[1]                   // pure array full indexation
emit A[?X][0] <= I and not J        // pure array full indexation
sustain P.X <= I and (?I > 0)       // pure simple port component
emit P[0].Y[0] if not I             // indexed component
                                    // of indexed port array
emit next S <= (?I > 0)  if I
sustain S <= pre(I) if (X = Y)
```

If there is no *rhs* and no test, the signal is emitted. If there is a *rhs* or a test clause but not both, the *rsh* or *t* predicate is evaluated and the signal is emitted only if the predicate is true. If there is both a *rhs* and a test, then the "`if` *t*" test is evaluated first. If *t* is false, the signal is not emitted; otherwise, the *rhs* expression is evaluated and the signal is emitted if *rhs* evaluates to true. Notice that the *rhs* and *t* predicate are equivalent if they do not appear together. This will not be true any more for array and valued emission.

For a `sustain` statement, evaluation is done in the same way at each tick, with fresh recomputation of *rhs* and *t*.

Notice that one can use an `if` clause to protect the evaluation of *e* from run-time errors. For instance, the following expression provokes a runtime error if `X` is absent and yet uninitialized:

```
emit S <= X and (?X > 0)
```

The subexpression `?X` is evaluated to evaluate *rhs*, which provokes the error. The following protected statement never provokes an error:;

```
emit S <= (?X > 0) if X
```

8.4.4 Pure array emission

A *pure array emission* emits or sustains a pure signal status array or array slice. The *lhs* is a status array designator, and the *rhs* is a Boolean array expression of the same dimension. Emission is performed as for a pure simple emission for each terminal array component, with matching *lhs* and *rhs* indices. Here are examples with `A[M][N]` a bidimensional signal array:

```
emit A                          // full array emission
emit A if (?I > 0)              // full array conditional emission
sustain next A[1]               // slice emission
emit A[..][3..5] <= X[1..M+1] [and] Y
                                // slicing, array expression
sustain P.A                     // port array component
emit P[0..2].A[3..5] if not I   // complex slice emission
```

As we shall see in Section **??**, a pure array emission can always be written as a pure simple emission enclosed in explicit `for–dopar` loops, see Section **??**. For instance, the fourth emission above is equivalent to

```
for i < M dopar
  for j < 3 dopar
    emit A[i][j+3] <= X[i+1][j] [and] Y[i][j]
  end for
end for
```

Or also, as we shall see in Section **??**, to

```
emit {
  for i < M do
    for j < 3 do
      A[i][j+3] <= X[i+1][j] [and] Y[i][j]
    end for
  end for
}
```

8.4.5 Valued simple emission

A *valued simple emission* emits a simple non-array value. The *lhs* is a simple signal value designator, the *rhs* is a data expression of the same type as the signal that defines the emitted value. Here are examples of valued signal emissions:

```
output X : float;
output Y[1] : unsigned<256>;
output Z : bool[8];
emit ?X <= 3.14f
sustain ?Y[1] = 5 if I
emit ?P[3].B[2] <= true if (?I > 0)
```

A simple value emission is performed as follows:

- If there is no `if` clause, *rhs* is evaluated and the signal is emitted with its value.
- If there is an "`if` *t*" test, the expression *t* is evaluated and there are two subcases:
  - If *t* is true, then *rhs* is evaluated and the signal is emitted with its value.
  - If *t* is false, then *rhs* is not evaluated and the signal is not emitted. Notice that this protects *rhs* from generating run-time errors.

Emitting a full signal also sets its status *present* (for next instant if the signal is registered).

For a single valued signal, if an `emit` or `sustain` statement is executed, it must be the only one in the instant; for a single or inputoutput single signal, no `emit` or `sustain` statement can be executed if the signal is received in the input event. For a combined signal, the emitted value is combined with those emitted by other `emit` or `sustain` statements executed in the instant using the signal's combination function. For a submodule input or inputoutput combined signal that is received from the caller module and locally emitted in the same instant, the received and emitted values are combined.

8.4.6 Valued array emission

A *valued array emission* emits a signal value array or array slice. The *lhs* is a signal array value designator, and the *rhs* is a data expression of the same type as the signal that defines the emitted value. Here are examples of valued signal emissions:

```
input I: bool[3][4];
input J[2][3] : bool
output O[3] : bool[4];
emit
emit ?O[1] <= 'xA
emit ?O[1..2][2..4] = J [and] ?J
```

Notice that the expression "`J [and] ?J`" returns an array of Booleans which are true if the corresponding component of `J` is *present* with value *true*.

8.4.7 Verification signal definitions

Assertions are characterized by the `assert` keyword that starts the left-hand-side. The general syntax is

```
assert Ident = e
assert Ident = e if t
```

where *e*, and *t* are Boolean expressions. For instance:

```
emit assert Exclusive = A[0] # A[1]
sustain assert PositiveIfPresent = ?X > 0 if X
```

Assumptions and coverage points share the same syntax, except that the `assert` keyword is replaced with respectively the `assume` and `cover` keyword. The scope of a verification signal name is the module. When evaluated, a verification signal evaluates the test *t* if it exists. If *t* is false, an assertion or assumption fails, a coverage point is not covered. Otherwise, the expression *e* is evaluated. If the result is *true*, an assertion or assumption passes, a coverage point is covered. If the result is *false*, an assertion or assumption fails, a coverage point is not covered.

One can abbreviate "`emit assert`" into `assert`, "`emit assume`" into `assert`, and "`emit cover`" into `cover` An "`emit assert`" statement is evaluated once in the instant, while a "`sustain assert`" is evaluated at each instant. Beware of a potential pitfall: because of this convention, `assert` alone is not sustained. Consider the following program fragment:

```
signal On, Off in
   ...
||
   assert OnOffExclusive = On # Off
end signal
```

Here, the assertion checks a very common property, exlusivity of `On` and `Off` signals. However, since `assert` is "`emit assert`", the assertion is tested *only once* when the `signal` declaration is entered. To check it as long as the `On` and `Off` signals are alive, one must write "`sustain assert`".

Note: having `assert` *as a synonym for "*`sustain assert`*" would lead to another pitfall:* `assert` *would never terminate, which is obviously also a bad default. Maybe we should only allow the explicit forms.*

8.4.8 Simple emissions with case lists in rhs

If is often useful to extend the right-hand-side into a case list. For any type of simple emission, this can be done using the '`|`' case separator:

```
emit {
  S <= A if X
     | B if Y
     | C,
  ?T <= ?A if A
      | ?B if B
}
```

To avoid syntactic conflicts with `if` statements, curly brakets are mandatory after `emit`. An `if` clause is mandatory for each case except the last one. The cases are examined in sequence, and the first true case determines the computed *rhs*.

8.4.9 Concurrent emission

A *concurrent emission* statement puts in parallel comma-separeted signal emissions and verification signal definitions in a block enclosed within curly brackets '`{ }`'. Emissions and verification definitions can also be replicated by a `for`–`do` or `for`–`dopar` statement with explicit indices declared as "`i <` e" as "`i in [`e_1`..`e_2`]`" where e, e_1, and e_2 are statically evaluable expressions with $e_1 < e_2$. Here are examples:

```
emit {
  S,
  ?X <= ?I[i][j],
  for i < N dopar
    for j in [5..7] dopar
      Y[i][j] <= (?A[i] + ?B[2*j]) if (A[i] and (?B[2*j] > 0)),
      Z[i][j] <= Y[i][j] and not pre(Y[i][j-1]),
      assert YZ_OK = Y[i][j] => X[i][j]
  end for,
  next ?Z[1..2][3] <= {2{0}} if  (?I > 0),
}
```

In the first `emit` above, notice that the assertion is replicated by the `for` loop. Here is a way to swap the values of `X` and `Y` from previous tick:

```
  sustain {
    X <= pre(Y),
    Y <= pre(X)
  }
```

The individual emissions are all performed concurrently. There is no order between them, and they can be shuffled arbitrarily. The following emissions are equivalent

```
  emit {
    X <= Y,
    Z <= X
  }


  emit {
    Z <= X,
    X <= Y
  }
```

The result is determined by concurrent evaluation controlled by *lhs* to *rhs* and test dependency, not by equation ordering.

To simplify long list handling, a trailing comma is permitted but not mandatory after the last emission in a concurrent emission statement or in a `dopar` emission list.

One can factor out the `next` keyword if all equations are for registered signals. For instance, one can write

```
  emit next {
    R1 <= I,
    R2 <= J if K
  }
```

instead of

```
  emit {
    next R1 <= I,
    next R2 <= J if K
  }
```

8.4.10 Conditional emissions

Emissions groups can be made globally conditional by using:

- A prefix `if` condition, which is similar to the `if-then-else` and `if-case` statements presented in Section **??**.
- A prefix `switch` expression, which is similar to the `switch` statement presented in Section **??**.

These constructs make it possible to factor out conditions for several simple emissions.

The conditional `if` emission has two main forms. The first form uses `then` and `else` clauses, each being optional:

```
emit {
  if S then
    X <= A,
    ?Y <= ?A +1
  else
    X <= B
  end if
}
```

```
emit {
  if S then
    X <= A,
    ?Y <= ?A +1
  end if
}
```

```
emit {
  if S else
    X <= B
  end if
}
```

If the `emit` body is composed of a single `if` clause, the curly brackets '`{}`' can be omitted.

The second form uses `case-do` clauses, with an optional `default` clause:

```
emit {
  if
    case S do
      X <= A,
      ?Y <= ?A +1
    case T do
      X <= B
    default do
      X <= C
  end if
}
```

```
emit {
  if
    case S do
      X <= A,
      ?Y <= ?A +1
    case T do
      X <= B
  end if
}
```

The cases are taken sequentially in order, the emissions corresponding to the first true case being the only ones executed.

For both forms, the body of a `then`, `else`, or `do` clause can be an arbirary emission body. Therefore, `if` constructs can be nested to any depth:

```
emit {
  if S then
    for i < N do
      if
        case T[i] do
          X[i] <= A,
          ?Y[i] <= i
        case U do
          ?Y[i] <= i+1
      end if
    end for
  end if
}
```

The conditional `switch` emission switches the control according to the value of the switch expression. The emissions corresponding to the case that matches the switch expression value are the only ones executed. An optional default clause is executed when no case matches the the switch expression value. The body of a `do` clause can be an arbirary emission body. Here is an example:

```
input Addr : value temp unsigned<[8]>;
output Error, {X, Y} : unsigned<[4]>, Ignore;
sustain {
  switch ?Addr
  case 0 do                 // illegal address
    Error
  case [0x10..0x1F] do      // X address space
    ?X <= trunc<[4]>(?Addr)
  case [0x20..0x2F]         // reserved for future usage
  case [0x30..0x3F] do      // Y address space
    ?Y <= trunc<[4]>(?Addr)
   default do               // ignored address
     Ignore
  end switch
}
```

8.4.11 Sequenced emission "emit seq"

Consider the following module:

```
module OrCarry :
  generic constant M : unsigned;
  input I[M];
  output O[M];
  emit {
    O[0] <= I[0],
    for i < M-1 do
      O[i+1] <= O[i] or I[I+1]
    end for
  }
end module
```

Here, `O` behaves as an or-carry chain: the output `O[`i`]` is *present* if `I[`j`]` is *present* for some $j \leq i$.

Generating C or HDL code for such a carry chain is easy provided one accepts to expand up to bit-level, i.e. to compute the `for` loop entirely at compile-time. Clearly, more clever compilers should not perform bit-level expansion that easily blows up, but they should be capable of generating host language `for` loops. For this, Esterel v7 provides its user with "`emit seq`" and "`sustain seq`" *sequenced emission* constructs. With "`emit seq`", one can write

```
emit seq {
  O[0] <= I[0],
  for i < M-1 doup
    O[i+1] <= O[i] or I[I+1]
  end for
}
```

In `for` loops, one must replace `do` by either `doup` or `dodown`. The idea is that the equations are evaluated one by one in the given order instead of being concurrent, and that the `for` loop indices increase or decrease as indicated by the `doup` or `dodown` keyword.

One can define mutually iterative equations as follows:

```
emit seq {
  X[0] <= I[0],
  Y[0] <= J[0],
  for i < N-1 doup
    X[i+1] <= I[i+1] or Y[i],
    Y[i+1] <= J[i+1] and X[i+1]
  end for
}
```

Notice that `X[i+1]` depends on `Y[i]` only while `Y[i+1]` depends on `X[i+1]`: the `X` equation is computed before the `Y` equation for each loop iteration. The following form would not work:

```
emit seq {
  X[0] <= I[0],
  Y[0] <= J[0],
  for i < N-1 doup
    X[i+1] <= I[i+1] or Y[i],
  end for
  for i < N-1 doup
    Y[i+1] <= J[i+1] and X[i+1]
  end for
}
```

Sequenced emission is also available for valued signals:

```
module IterativeSum :
  generic constant M : unsigned;
  input I : unsigned [M];
  output O : unsigned [M]
  emit seq {
    ?O[0] <= ?I[0],
    for i < M-1 do
      ?O[i+1] <= ?O[i] + ?I[I+1]
    end for
  }
end module
```

We impose the following restrictions on `emit seq`:

- Only array elements can be emitted.
- Input signals cannot be emitted.
- If an array is emitted in an “`emit seq`” statement, it cannot be emitted elsewhere; this either directly by another `emit` statement, indirectly by a submodule connection, see Chapter **??**, or indirectly by reincarnating the “`emit seq`” statement,because that would amount to having two “`emit seq`” statements, see Section **??**.

Semantically speaking, `seq`, `doup`, and `dodown` are irrelevant. They can be seen as implementation pragmas embodied in the language as first-class citizen since they are very useful. The semantics should be exactly the same as with a normal `emit` statement, and it is the compiler responsibility to check that the given order is indeed compatible with host language possibilities.

8.4.12 Which case or concurrent structure to choose?

The rich syntax available in `emit` and `sustain` bodies makes it possible to write the same thing in different ways. These ways actually correspond to different styles and tradeoffs.

Prefix vs. postfix if conditions

Prefix and postfix conditions have the same effect. It is identical to write

```
emit {
  if S then
    X <= A,
    Y <= B
  end if
}
```

or

```
emit {
  X <= A if S,
  Y <= B if S
}
```

The first form may be preferred because the condition is duplicated in the second form while factored out in the first form.

Rhs cases w.r.t. if-based cases

For the emission of one signal, case lists can be written in two ways. Here is an example

```
emit S <= A if X
        | B if Y
        | C
```

```
emit {
  if
    case X do
      S <= A
    case Y do
      S <= B
    default
      S <= C
  end if
}
```

In this case, the first form is obviously lighter. Assume now that some more complex conditions drive two signals:

```
emit {
  S <= A if X and ?X >0
     | B if Y
     | C
  ?T <= ?B if X and ?X >0
      | ?C+1 if Y
      | 3
}
```

Then, factoring out the condition can make code better and especially more maintainable since things are written only once:

```
emit {
  if
    case X and ?X >0 do
      S <= A,
      ?T <= ?B
    case Y do
      S <= B,
      ?Y <= ?C+1
    default
      S <= C,
      ?T <= 3
  end if
}
```

Which way to choose really depend on the application and user. We think one should try to minimize copy-paste of expressions and conditions, and there is no optimal way to do that.

Case lists vs. concurrent ifs

Case lists as above are very readable, but they can lead to inefficient implementation if the cases are always disjoint for reasons extraneous to the statement itself. For example, assume that the signals A, B, and C are always exclusive due to global program behavior. A case-based emission is written as follows:

```
emit {
  if
    case A do <emA>
    case B do <emB>
    case C do <emC>
  end if
}
```

where < *emA* >, < *emB* >, and < *emC* > are arbitrarily complex emissions. The following parallel emission can be more efficient:

```
emit {
  if A then <emA> end,
  if B then <emB> end,
  if C then <emC> end
}
```

Of course, the above case-based and parallel emissions are equivalent only if A, B, and C are exclusive. This can be checked by adding "`assert Excl = A # B # C`" in the `emit` statement body. Notice that there is no need to define a specific "parallel case statement": the combination of parallel emissions using '`,`' and of `if`-based tests does the job.

Conditional emissions w.r.t. emissions in conditionals

There is a subtle but important difference between placing conditionals within emissions or emissions within conditionals. Consider the following statement:

```
emit {
  if S then
    X <= A,
    Y <= B
  end if
};
p
```

Here, the `if` test occurs within the `emit` statement, whose execution is unconditional. Therefore, execution of *p* is also viewed as unconditional, and it does not depend on testing `S`. Using a conditional statement, the code would be written as follows:

```
if S then
  emit {
    X <= A,
    Y <= B
  }
end if;
p
```

In that case, the execution of *p* is dependent on the test for `S`, which introduces extra dependencies. In particular, if *p* is "`emit S`", then the second form generates a causality cycle while the first form is innocuous (see [?] for an in-depth analysis of dependency issues).

Therefore, using an `if` condition enclosed within an `emit` statement is much more lightweight than using a full conditional statement with `emit` in branches. The use of true conditional statemens should be restricted to real branching in the control flow.

Concurrent sustain vs. sustain sequences

Remember that `sustain` statements never terminate. A classical mistake is to write:

```
sustain X <= A and B;
sustain Y <= pre(I or J)
```

Since the first `sustain` never terminates, the second one is never started and is dead code. The right statement is a concurrent emission:

```
sustain {
  X <= A and B,
  Y <= pre(I or J)
}
```

8.5 Sequencing

Sequencing is performed by the '`;`' sequence operator:

p ; *q*

the first statement *p* is instantaneously started when the sequence is started, and it is executed up to completion or trap exit. If *p* terminates, *q* is immediately started and the sequence behaves as *q* from then on. If *p* exits some enclosing traps, the exits are immediately propagated and *q* is dead code since never started, see Section **??**. For instance, "`exit T; emit S`" does not emit `S`.

8.6 Looping

8.6.1 Basic loops

A basic loop has the following form:

```
loop
  p
end loop
```

The body *p* is instantaneously restarted afresh upon termination, this forever. If *p* exits some enclosing traps, the exit is propagated instantaneously and the loop stops. This is the only way to exit a loop from inside. Of course, a loop can also be killed by an external preemption statement, see Section **??** and Section **??**.

Here is a way to achieve C-like `break` and `continue` statements for a loop:

```
trap Break in
  loop
    trap Continue in
      ...
      exit Continue
      ...
      exit Break
      ...
    end trap
  end loop
end trap
```

This is slightly heavy, but has the advantage that break and continue positions of nested loops can be explicitely named. One always knows which level a break or continue refers to, and, for nested loops, one can break the outermost loop with a single `exit`.

8.6.2 Instantaneous loops

The body of a loop is not allowed to be able to terminate instantaneously when started. In full generality, this condition can be tested in very precise ways if an implementation has clever ways to detect *false paths*. Consider for instance the following loop:

```
loop
  if I then
    if not J then
      p
    end if
  else
    q
  end if;
end loop
```

where *p* and *q* are non-instantaneous statements. There is a potential instantaneous path in the loop body corresponding to the case where `I` and `J` are both present. However, if `I` and `J` are inputs declared incompatible by the input relation "`I # J`", the instantaneous path is a false one since it cannot be taken in any valid input configuration. The same holds if `I` and `J` cannot be simultaneous for any kind of dynamic reasons. However, since programs can involve arbitrary data, simultaneity of `I` and `J` is undecidable in general.

Implementations can be more stringent and reject programs with static potential false paths such as the one above. In this case, we say that they reject *statically instantaneous loops*. There is always a simple way to make a loop body non-instantaneous without altering its semantics: adding a `pause` statement in parallel. Here is the transformation of the previous example:

```
loop
  if I then
    if (not J) then
      p
    end if
  else
    q
  end if
||
  pause
end loop
```

8.6.3 Always loops

An `always` loop is used to transform an instantaneous statement into a permanent one. It is written as follows:

```
always
  p
end always
```

The semantics is simply that of "`loop` *p* `each tick`", but the statement is much more readable (and familiar to HDL users). Notice that `sustain` is simply "`always emit`". but again more readable.
Note: We used to recommend the `always` statement in conjunction with decision trees to factor out conditions in emission statements:

```
always
  if I and not pre(J) then
    emit {
      X <= A or B,
      Y <= A and B
    }
  else
    emit {
      X <= pre(X),
      Y <= A
    }
  end if
end always
```

This form is now subsumed by the even clearer use of `if-then-else` within `sustain` statements, see Section **??**:

```
sustain
  if I and not pre(J) then
    emit {
      X <= A or B,
      Y <= A and B
    }
  else
    emit {
      X <= pre(X),
      Y <= A
  end if
}
```

Note for Esterel Studio 5.3 users *there is currently no requirement for the body p of an* `always` *statement to be instantaneous. Because of the semantics of* `always` *which aborts p at each tick, a delay statements in p will never become active, and all the statements that depend on delays will be unreachable. Only unreachable code warnings are issued by the Esterel Studio v7_60 compiler in this case.*

8.6.4 Repeat loops

A `repeat` loop executes its body for a finite number of times. The body is not allowed to terminate instantaneously, static or dynamic detection of this property being as in Section **??**.

Simple repeat loop

The simplest form of `repeat` loop is:

```
repeat e times
  p
end repeat
```

The data expression e must be of unsigned type or must be a positive signed constant, then automatically converted into unsigned. It is evaluated only once at starting time, yielding a result m. The body is then executed m times. It is not executed if $m = 0$.

Repeat counter

One can also declare a counter in a `repeat` loop.:

```
repeat i := e times
  p
end repeat
```

The counter `i` goes down from e to 0, performing e iterations. As for simple `repeat`, e is computed only once, when the `repeat` statement is entered. If e is a statically evaluable constant with value m, the type of the counter variable is `unsigned<`$m + 1$`>`. Otherwise, the type of the counter is the type of the expression.

The counter value can be read within the loop body, but the counter cannot be assigned to: it is read-only.

Positive repeat

Although its body cannot terminate instantaneously, a `repeat` statement is considered as possibly statically instantaneous since it can be executed 0 times. Therefore, it cannot be put itself in a loop if not preceded or followed by a delay, this even if its own body is non-instantaneous. For example, the following statement can be rejected as a potentially instantaneous loop, independently of the body *p*:

```
loop
  repeat e times
    p
  end repeat;
  emit O
end loop
```

Esterel compilers are not required to perform static analysis and discover that *e* is always strictly positive, which is undecidable anyway. To solve this problem, the user may assert that the body will be executed at least once by adding the '`positive`' keyword:

```
loop
  positive repeat e times
     ...
  end repeat
end loop
```

```
loop
  positive repeat i := e times
     ...
  end repeat
end loop
```

In the "`positive repeat`" statement, the test for repetition is performed only after the first execution of the body. The body is not allowed to be able to terminate instantaneously, and the whole "`positive repeat`" statement inherits the same property. The body is executed once if *e* evaluates to 0.

8.7 The if statement

The `if` statement branches according to the instantaneous values of a Boolean expression. Each of the `then` and `else` branches can be omitted, but at least one of them must be specified. An omitted branch is implicitly `nothing`:

```
if S then p else q end if
if Second and Meter then p end
if S[2] then q end
if S and (?S >0) else q end if
if pre(P.S[2]) else q end
```

See Section ?? for the difference between the `if-then-else` statement and `if` clauses in `emit` and `sustain`.

8.7.1 Case tests

The `case` form tests several signal expressions in sequence:

```
if
  case Meter do
    emit ?Distance <= pre(?Distance) +1
  case Second do
    emit ?Speed <= ?Distance
end if
```

The tests are taken in order, and the first true expression starts immediately its `do` clause. If the `do` clause is omitted, the `if` statement simply terminates. If none of the expressions is true, the `if` statement terminates. One can add an `default` keyword to be executed in that case:

```
if
  case (Op[0] and Op[1]) do
    emit Load
  case (Op[0] and not Op[1]) do
    emit Store
  case (not Op[0] and Op[1]) // no-op
  default do
    exit WrongOpCode
end if
```

Notice that at most one `do` or `default` clause is executed in an `if-case` statement.

See Section **??** for the difference between the `if-then-else` statement and `if` clauses in `emit` and `sustain`.

8.7.2 The if static statement

The `if static` statement has the same semantic as the classic `if` statement but requires in addition the Boolean condition expression to be *statically evaluable*. An error is triggered if the condition expression cannot be evaluated. According to the condition value, the unreachable then or else branch is discarded at compile-time.

The `if static` statement makes it possible to write *configurable code*. For example, the following code snippet outlines a configurable DMA design. The DMA instantiates channels, which perform concurrent data transfers on a bus, and an arbiter, which arbitrates the bus access between channels. The arbitration strategy can be configured to be either round-robin or static priority. The arbiter is instantiated only if the design has more than one channel.

```
// declaration of generic parameters
data generic_parameters:
generic constant nb_channels : unsigned; // number of channels
generic constant use_round_robin : bool; // true if use round-robin
                                         // arbitration; false if use
                                         // static priority arbitration
end data

// configurable arbiter module: either round-robin or static priority
// strategy can be used
module arbiter:
```

```
extends generic_parameters;
if static use_round_robin then
  run round_robin_arbiter     // implements round-robin strategy
else
  run static_priority_arbiter // implements static priority strategy
end if
end module

// configurable dma module: number of channels and channel arbitration
// strategy can be configured
module generic_dma:
extends generic_parameters;
  if static nb_channels <> 1 then
    run arbiter // arbiter instantiated if two channels or more
  else
    ... // trivial code that always gives the channel access to the bus
  end if
||
  for i < nb_channels dopar
    run channel
  end for
end module

// dma configured with one channel
module one_channel_dma:
constant nb_channels : unsigned<> = 1; // use one channel
extends generic_dma;
run generic_dma
end module

// dma configured with four channels and round-robin arbitration
module four_channels_round_robin_dma:
constant nb_channels : unsigned<> = 4;  // use four channels
constant use_round_robin : bool = true; // use round-robin strategy
extends generic_dma;
run generic_dma
end module
```

Here, the `generic_dma` module can be configured thanks to the constants `nb_channels` and `use_round_robin`. The value of `use_round_robin` determines whether the arbiter implementing the round-robin strategy or the arbiter implementing the static priority strategy is instantiated. When configured with `nb_channels = 1`, no arbiter module is instantiated.

Furthermore, the `if static` statement makes possible *static run module recursion*, where the recursion condition is based on generic constants of the module. The `if static` statement is used to distinguish terminal and recursion cases according to the constants' values. In the recursion case, the generic constants are renamed in recursive `run` statements in order to move towards the terminal case. Owing to the `if static` statements, run module recursion can be fully performed at Esterel compile-time. For example, the following recursive module performs a n-ary boolean `or` operation using a binary tree structure. The recursion is based on the number of inputs: if there is only one, the module simply outputs its unique input; otherwise, the module recursively instantiates itself twice in dividing its inputs into two, and it performs the Boolean `or` operation on the two submodule outputs.

```
// the interface of the n-ary Boolean or operation
interface intf:
generic constant n : unsigned;
input  a[n];
output x;
end interface

// the module performs a n-ary Boolean or operation;
// the module recursively describes a binary tree
module or_binary_tree:
extends intf;
if static n = 1 then // terminal case
  sustain x <= a[0]
else                  // recursion case
  signal x0, x1 in
    run binary_tree [ constant (n/2) / n;   signal a[0..n/2-1]/a, x0/x ]
  ||
    run binary_tree [ constant (n-n/2) / n; signal a[n/2..n-1]/a, x1/x ]
  ||
    sustain x <= x0 or x1
  end signal
end if
end module

// the module performs a 64-bit or operation using a binary tree
module or_binary_tree_64:
constant n : unsigned = 64;
extends intf;
run binary_tree
end module
```

Like an `if` statement, the `if static` statement has a `case` form: the case expressions are evaluated at compile-time in sequence and only the first do clause whose case expression is true is kept. If no case expression is true, the default do clause is kept if there is one. For example:

```
constant N : unsigned;
...
if static
case N=0 do
  ... // code for N=0
case N<10
  ... // code for N<10
default do
  ... // code for N>=10
end if
```

8.8 The switch statement

The `switch` statement switches the control according to the value of an expression called the *switch expression*, which can be of type `enum`, `unsigned`, `signed` or bitvector. It starts with `switch` and ends with `end` or "`end switch`". Its body is made of cases starting with the `case` keyword, each involving a comma-separated list of case values or case ranges for unsigned or signed switch; the case definition is optionally followed by the `do` keyword and an arbitrary statement called the case statement. The case statement is triggered if the switch expression value is in its value list. An optional default case starting with the `default` keyword is activated if no explicit case is. If there is a case statement following the `do` keyword, this statement is started and determines the behavior of the `switch` statement from then on; if there is no case statement to trigger, the `switch` statement terminates. If no case is activated and no default is given, the `switch` statement simply terminates .

A case expression must be statically evaluable and it must be compatible with the switch expression type. If the switch expression is of enum type, a case expression must be either an enum value identifier or a generic constant identifier. Values of case expressions must not overlap.

Here are switch examples:

```
type T = enum {A,B,C};
input I : T;
output O;
switch ?I
  case A // "do nothing" is implicit
  case B,C do
    emit O
end switch
```

```
constant N : unsigned<20> = 1;
input I  : unsigned;
output O : unsigned;
switch ?I
  case 0 do
    nothing
  case N do
    emit ?O <= ?I
  case N+1 do
    emit ?O <= ?I+1;
    pause
  case [N+2..7] do
    pause;
    emit ?O <= ?I-1
  case [8..11], 13, [15..19] do
    pause;
    emit ?O <= ?I;
    pause
  default do
    emit ?O <= 0
end switch
```

In the last example, case values are given in increasing order. This is not compulsory.

8.9 Concurrency

There are two concurrency constructs: the explicit parallel statement '`||`' and the replication construct '`for...dopar`'.

8.9.1 The parallel operator

The parallel operator '`||`' puts statements in synchronous parallel. The signals emitted by any of its branches or by the rest of the program are instantaneously broadcasted to all branches in each instant.

A parallel can be binary, as in p `||` q, ternary, as in p `||` q `||` r, or of any arity. Syntactically, the sequencing operator '`;`' binds tighter than the parallel operator '`||`'. Therefore, p`;` q `||` r means `{`p`;` q`}` `||` r, which is different from p`;` `{`q `||` r`}` where the brackets are mandatory.

A parallel statement forks its incoming thread when it starts, starting instantaneously one thread per branch. All threads behave synchronously until termination or trap exit. The parallel terminates when all its branches have terminated, waiting for the last one if some branches terminate earlier. The parallel propagates a trap `T` as soon as one of its branches exits T, weakly aborting all its branches at that time. See Section **??** for the case where several traps are simultaneously exited.

There are restrictions for sharing variables among parallel branches, see Section **??**

8.9.2 The for-dopar replication statement

The `for-dopar` replication statement can be written in one of two forms:

```
for i < e dopar
  p
end for
```

or

```
for i in [e1..e2] dopar
  p
end for
```

where i is an integer variable called the *iterator* and where e, e_1, and e_2 are statically evaluable expressions.

In the first form, replication goes from 0 to $e - 1$ included and e is called the *replication count*. If e statically evaluates to m, The iterator is implicitely declared of type `unsigned<`$m + 1$`>`.

In the second form replication goes from e_1 to e_2 included, these two expressions being called the *replication bounds*. If e_1 and e_2 statically evaluate to m_1 and m_2, the iterator is of type `unsigned<`$m_2 - m_1 + 1$`>`.

In both cases, the value can be read in the body p, but it is read-only and cannot be assigned to.

Assume that there are m replications. Then the construct conceptually builds m parallel copies of p, which act concurrently with each other. The copies have independent control, communicate witch other, and need not terminate at the same time. The whole replication construct behaves exactly as this conceptual parallel statement: it terminates when all the copies of p have terminated; it exits a trap if one of its p copies exits a trap; if several copies of p exit different traps at the same time, only the outermost one matters, all other ones being discarded.

Beware: parallel replication is very different from `repeat` loop. In a loop, the body is repeated in sequence and it is not allowed to be instantaneous. In a `dopar` statement, the body is instantiated in parallel and there is no timing restriction for it. Instantaneous statements can be freely replicated.

8.9.3 Replication examples

The classical `ABO` automaton terminates and emits `O` when it has received `A` and `B`, either simultaneously or in succession, initial instant excluded. Its code is as follows:

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

Here is how to put `N` copies of `ABO` in parallel, feeding them with elements of input arrays and sending their outputs to an array, with global termination detection:

```
module NABO :
  generic constant N : integer;
  input A[N], B[N];
  output O[N], Done;
  for i < N dopar
    run ABO[signal A[i] / A, B[i] / B, O[i] / O]
  end for;
  emit Done
end module
```

Since a `dopar` statement terminates when all its branch are terminated, the `Done` signal is output exactly when all the inputs `A[i]` and `B[i]` have been received.

Here is an amazing way of computing the $N!$ factorial for any integer N using a `dopar` and a combined signal:

```
module SynchronousFactorial :
  constant N : integer;
  output Fact : unsigned combine *;
  for i < N dopar
    emit Fact(i+1);
  end for
end module
```

The concurrent emissions of `Fact(i+1)` are combined by multiplication, yielding the factorial value $N! = 1 \times 2 \times \ldots \times N$ for the `Fact` signal.

To illustrate how branches can synchronize in a fancy wasy, here is a computation of $(N!)^N$ in one tick

```
module SynchronousFactorialPower :
  constant N : integer;
  output Fact : unsigned combine *,
         PowFact : unsigned combine *;
  for i < N dopar
    emit Fact(i+1);
    emit PowFact(?Fact)
  end for
end module
```

Within the instant, the `N` concurrent emissions of `PowFact` have to wait for the value of `Fact` to be entirely computed before being able to act. When `?Fact` finally evaluates to $!N$, still in the instant, each emission of `PowFact` emits $!N$, yielding the final result `PowFact`$= (N!)^N$ by `*`-combination. The necessary waiting and synchronization mechanism is built-in the language semantics.

8.10 Delays

Delay expressions are used to denote occurrence of future events in temporal statements such as `abort`, `await`, `every`, etc.

8.10.1 Simple delays

A *simple delay expression* is a Boolean test:

```
await X
abort p when X and ?X>0
every X and Y[2] do p end
```

The delay expression then elapses at the first instant in the *strict* future where the test is true.

8.10.2 Immediate delays

An immediate delay consists of a Boolean test preceded by the `immediate` keyword:

```
await immediate X
abort p when immediate X and ?X>0
every immediate X and Y[2] do p end
```

The delay expression then denotes the first instant where the test is true, *current instant included.* For instance, in the above examples, the `await` statement immediately terminates if `X` is present, the body of the `abort` statement is not started and the `abort` immediately terminates if `X` is present with a positive value, and the `every` statement immediately starts its body if its condition is true.

8.10.3 Count delays

A *count delay* consists of an integer expression called the *count expression*, and a simple delay. There are two possible syntactif forms:

- The count is an arbitrary unsigned expression or a positive constant signed expression. It is followed by the `times` keyword and a Boolean test.
- The count is an unsigned constant, a simple identifier denoting an unsigned variable or constant, or a signed positive constant or a parenthesized expression of unsigned type. It is immediately followed by the simple delay which must be a single identifier, possibly indexed.

The second form makes it possible to forget about the `times` keyword in simple cases. Here are examples:

```
await 3 Second
await (f(?I)+1) Second
await 3 times Second
await 3 times Second or Meter
every f(?I) times (X and Y[1]) do ... end
```

Remember that the `times`keyword is mandatory if the test is not trivial

When the statement that bears the count delay starts, the integer expression is evaluated into a value called the *count.* Every future instant where the test is true, the count is decremented. When the count becomes less than 1, the delay elapses. Beware: this means that an initial count value strictly less than 1 is equivalent to 1. A statement with a count expression always takes time. This can be misleading if the count starts at 0 !

Notice that the count is evaluated *only once*, whence the delay is initially started, while the test is re-evaluated independently at each instant. Consider for instance

```
await ?S times S and ?S>0
```

When the await statement starts, the count expression '?S' is evaluated to define the count. It will not be re-evaluated during the delay. On the opposite, the expression '?S' in the if part of the delay is re-evaluated at each instant.

In Section **??**, we give the exact definition of a delay by macro-expansion into simpler statements.

8.11 Temporal statements

8.11.1 The await statement

The await statement is the simplest temporal statement. In its basic forms

```
await d
await immediate d
```

it simply waits for a delay. Here are examples:

```
await Second
await immediate Second
await immediate ?I > 0
await pre(Second)
await immediate pre(S[2]) and (?S[1] > 0)
await 2 Second
await 2 times P[1].S[1][2]
await X+1 times (?S > 2)
await f(Y) times (Second and not pre(Meter))
```

The delay is started when the await statement is started. The statement pauses until the delay elapses and terminates in that instant. An immediate await statement terminates instantaneously if the signal expression is true in the starting instant. Be careful: the sequence

```
await immediate Meter;
await immediate Meter
```

terminates instantaneously if Meter is present in the starting instant (this is why making immediate the default would be misleading).

A do clause can be used to start another statement when the delay elapses, with the following syntax:

```
await d do p end await
```

This is simply an abbreviation for "await *d* ; *p*" which makes the dependency of *p* on *d* more explicit. For instance:

```
await 2 times Second do
  emit Beep
end await
```

As for if, one can introduce a case list, where do clauses can be omitted:

```
await
  case d1 do p1
  case d2
  ...
  case dn do pn
end await
```

An absent `do` clause is equivalent to "`do nothing`". When the first delay elapses, the corresponding `do` clause is started and the whole `await` statement terminates when the clause terminates. If several delays elapse simultaneously, only the first one in the case order is considered. Consider for instance:

```
await
  case 2 Second do p
  case immediate Meter
  case Button do q
end await
```

The above statement immediately terminates if `Meter` occurs at start time. Otherwise, the first delay to elapse determines the subsequent behavior: *p* is started if "`2 Second`" elapses first, the `await` statement simply terminates if `Meter` occurs first, and *q* is started if `Button` occurs first. If `Meter` and `Button` occur simultaneously, then the `await` statement terminates and *q* is not started since the first delay takes priority.

Unlike for `if`, no `default` clause is allowed for an `await` statement, which is not supposed to terminate right away. One can use "`case tick`" to achieve the same effect.

8.11.2 The abort-when statements

An abortion statement kills its body when a delay elapses. For strong abortion, performed by `abort`, the body does not receive the control at abortion time. For weak abortion, performed by `weak abort`, the body receives the control for a last time at abortion time. Simple abortion syntax is as follows, where *d* is a delay:

```
abort p when d
weak abort p when d
```

For instance:

```
abort p when 2 Meter
abort p when 3 times Meter or Second
weak abort p when Meter
weak abort p when immediate Meter
```

For both constructs, the body *p* is run until termination or until the delay elapses. If *p* terminates before the delay elapses, so do the `abort` and `weak abort` statements. Otherwise, *p* is preempted when the delay elapses; in that instant, *p* is not executed with strong abortion, and it is executed for a last time with weak abortion (*p* has rights to its "last wills").

If the delay is immediate and elapses immediately at starting time, the body is not executed at all with strong abortion, and it is executed for one instant with weak abortion For instance, in

```
abort
  sustain O
when immediate I
```

the `abort` statement terminates immediately without emitting `O` if `I` is present at starting time. If `abort` is replaced by `weak abort`, the whole statement also terminates instantaneously but `O` is emitted once.

As for `await`, one can add a `do` clause to execute a statement *q* in case of delay elapsing:

```
[weak] abort
  p
when d do
  q
end abort
```

With both weak and strong abortion, q is executed if and only if p did not terminate strictly before delay elapsing. At abortion time, with strong abortion, p is not executed and q is immediately started. With weak abortion, the first instant of q is done in sequence after the last instant of p. This behavior is different from that of

```
[weak] abort
  p
when d;
q
```

where q is executed wheter p terminates or d elapses. As for `await`, one can introduce an ordered list of abortion cases:

```
[weak] abort
  p
when
  case d1 do p1
  case d2
  ...
  case dn do pn
end await
```

An absent `do` clause is equivalent to "`do nothing`". Here, p is immediately strongly or weakly aborted if an immediate delay elapses at start time. Otherwise, p is run for at least one instant. The elapsing of any of the delays strongly or weakly aborts p and the corresponding `do` clause is immediately started; the whole statement terminates when the clause terminates. If more than one of the delays elapses at abortion time, then the first one in the list takes priority as for the `await` statement. If p terminates before any of the delays elapses, then no `do` clause is executed and the whole construct terminates. Here is an example:

```
abort
  p
when
  case Alarm do r
  case 3 Second do q
  case immediate Meter
end abort
```

Nesting `abort` statements automatically builds outside-in priorities. In the statement

```
abort
  abort
    p
  when I do
    q
  end abort
when J
```

the signal `J` takes priority over `I` if they occur simultaneously, and *q* is not started in that case. This is no special rule, but just a consequence of the strong abortion semantics of `abort`.

Finally, notice that "`await S`" can be defined as "`abort halt when S`".

8.11.3 The abort-after statements

The `abort - after` statements use the termination of a statement as the abortion condition. It has the following form:

```
[weak] abort
   p
after
   q
end abort
```

where *p* and *q* are arbitrary statements. The statements are started in parallel, and the whole construct terminates when *q* terminates. In the `abort` strong form, *p* is strongly preempted (not executed) when *q* terminates. In the `weak abort` weak form, *p* is executed at the tick in which *q* terminates. This new statement is particularly useful to define equations valid during a statement, as in:

```
weak abort
   sustain RisingS = S and nor pre(S)
after
   ...
   await RisingS;
   ...
   await RisingS;
   ...
end abort
```

The weak form is equivalent to the following trap construct:

```
trap T in
   p ; halt
||
   q; exit T
end trap
```

The strong form is slightly more complex:

```
trap T in
  signal S in
    abort
      p;
      halt
    when immediate S
  ||
    q;
    emit S;
    exit T
  end signal
end trap
```

8.11.4 Temporal loops

Temporal loops are loops over strong abortion statements. The first form is

```
loop
  p
each d
```

where d is a non-immediate delay. At start time, the body p is started right away, and it is strongly aborted and immediately restarted afresh whenever the delay d elapses. If p terminates before d elapses, then one waits for the elapsing of d to restart p. The `loop-each` statement is simply an abbreviation for

```
loop
  abort
    p; halt
  when d
end loop
```

The delay cannot be immediate, otherwise the loop body would be instantaneous. The second temporal loop has the form

```
every d do
  p
end every
```

The difference is that d is initially waited for before starting the body p. The delay d can be immediate. In that case, in the starting instant, p starts immediately if the delay elapses immediately. The statement

```
every 3 Second do
  p
end every
```

abbreviates

```
await 3 Second;
loop
  p
each 3 Second
```

The statement

```
every immediate Centimeter do
  p
end
```

abbreviates

```
await immediate Centimeter;
loop
  p
each Centimeter
```

All temporal loops are infinite. The only way to terminate them is by exiting a trap, see Section **??** or by the elapsing of an enclosing abortion delay.

8.12 Traps

A trap defines an exit point for its body. The basic syntax is

```
trap T in
   p
end trap
```

The scope of T is the body *p* and scoping is lexical. A redeclaration of a trap hides the outer declaration.

The body *p* is immediately started when the `trap` statement starts. Its execution continues up to termination or trap exit, which is provoked by executing the "`exit T`" statement. If the body terminates, so does the `trap` statement. If the body exits the trap T, then the `trap` statement immediately terminates, weakly aborting *p*.

The "`weak abort`" statement can be defined using traps. The construct

```
weak abort
  p
when S
```

is an abbreviation for

```
trap T in
  p;
  exit T
||
  await S;
  exit T
end trap
```

8.12.1 Nested traps

When traps are nested, priority is inside-out. Consider for example

```
trap U in
  trap T in
    p
  end trap;
  q
end trap;
r
```

If *p* exits T, then *q* is immediately started. If *p* exits U, then *q* is discarded and *r* is immediately started. If *p* exits simultaneously T and U, for example by executing "`exit T || exit U`", then U takes priority and only *r* is executed. From the point of view of the "`trap T`" statement, T is discarded and U is propagated.

8.12.2 Trap Handlers

A handler can be used to handle a trap exit, with the following syntax:

```
trap T in
  p
handle T do
  q
end trap
```

If p terminates, so does the trap statement. If p exits `T`, then p is weakly aborted and q is immediately started in sequence.

8.12.3 Concurrent Traps

Several traps can be declared using a single `trap` keyword:

```
trap T, U, V in
  p
handle T do q
handle U do r
end trap
```

In this case, the traps are called *concurrent traps* and they must have distinct names. Concurrent traps are at the same priority level, and any of them can have a handler. If several traps are simultaneously exited, then the corresponding handlers are executed in parallel.

Here, q and r are executed in parallel if p exits `T` and `U` simultaneously. Since they are concurrent, the q and r handlers cannot share variables, see Section **??**. The `trap` statement simply terminates if p exits `V` that has no handler. Here is the translation of "`weak abort` p `when S do` q `end`" using concurrent traps:

```
trap Terminate, WeakAbort in
  p;
  exit Terminate
||
  await S;
  exit WeakAbort
handle WeakAbort do
  q
end trap
```

8.12.4 Valued Traps

Traps can be valued. Valued traps are useful to pass a value to the handler. The value is emitted within parentheses, as in "`exit T(3)`". The value is read as the result of the expression '`??S`', which is allowed only in the handler. Combined traps are allowed.

Trap value initialization and the expression `pre(?S)` are not available for traps; they would make no sense since a statement that exits a trap dies instantaneously.

```
trap Alarm : unsigned combine + in
  ... exit Alarm(3) ...
  ... exit Alarm(5) ...
handle Alarm do
  emit Report(??Alarm)
end trap
```

Concurrent traps can be valued and combined, but their value cannot be initialized:

```
trap T,
     U : integer,
     V : integer combine + in
  p
handle T do
  q
handle U and V do
  emit O(??U + ??V)
end trap
```

Here, the second handler starts if and only if `U` and `V` are exited in parallel, in which case `O` is emitted with values the sum of the values of `U` and `V`.

Note for discussion: ***valued traps are quite complex. Are they really necessary in Esterel v7? Their usage seems to be fairly rare.***

8.13 Suspension statements

Abortion or trap exit violently preempts a statement and kills it, in the same way as ˆC kills a process in Unix. Suspension has a milder action, simply freezing the statement for the instant. There are two suspension statements:

- For "`suspend` p `when` t", the body p is frozen and does not react when the test t is true. This is similar to ˆZ in Unix for the current instant only.
- For "`weak suspend` p `when` t", the body p acts for the instant but its state change is frozen. For hardware, this is the effect of clock-gating p's registers by t.

The name `weak suspend` comes from the similarity with `abort` and `weak abort`: at abortion time, the body is not executed with `abort` while it is executed with `weak abort`. The `weak suspend` statement was first introduced in [**?**]. It is fundamental for clock-gating and for multiclock simulation, see Chapter **??**. Since its behavior is less intuitive than that of `suspend`, we present it it great details with examples.

8.13.1 The suspend statement

The syntax of `suspend` is as follows:

```
suspend
  p
when t
```

where t is any test (Boolean expression). When the `suspend` statement starts, its body p is immediately started. If p terminates or exits a trap at that instant, so does the `suspend` statement. If p pauses, the behavior is chosen as follows at each subsequent instant:

- If t is true, then p remains in its current state without being executed. No signal is emitted and no state change is performed. The `suspend` statement pauses for the instant.
- If t is false, then p is executed for the instant. If p terminates or exits a trap, so does the `suspend` statement, and suspension is over. If p pauses, so does the `suspend` statement, and suspension is re-examined in the next instant.

For instance, the statement

```
suspend
  abort
    sustain O
  when J
when I
```

emits `O` in the first instant and in all subsequent instants where `I` is absent, until the first instant where `I` is absent and `J` present. Then the `suspend` statement terminates, with `O` is not emitted because of the `abort` statement.

The default `suspend` statement is *delayed*, in the sense that the signal expression is not tested for in the first instant. The *immediate* form performs that test:

```
suspend
  p
when immediate t
```

Here p is not started in the first instant if t is true. The immediate form can be macro-expanded as follows:

```
await immediate not t;
suspend
  p
when t
```

8.13.2 The weak suspend statement

When the suspension test of a "`suspend` p `when` t" statement is true, the body p is not activated and it keeps its state. With the "`weak suspend` p `when` t" statement, when t is true, the body p is activated in the instant, but no state change is performed for it unless p exits an enclosing trap. In other words, the combinational part acts, but the state is kept unchanged unless the statement commits suicide (or is killed by a concurrent statement, of course).

In hardware, weak suspension has exactly the effect of clock-gating by t the registers generated by p, with synchronous reset performed by trap exit. Of course, hardware implementations without clock-gating are possible. One can for example use disabling logic to keep the register state unchanged. Implementation is not studied here, and we concentrate on the behavior of the `weak suspend` statement, using hardware vocabulary since `weak suspend` is most useful for hardware.

As for `suspend`, there is a delayed and an immediate form. The delayed form is written

```
weak suspend
  p
when t
```

where p is any statement and t is a Boolean test.

When the `weak suspend` statement starts, the body p is immediately started. If p terminates instantaneously, so does the whole `weak suspend` statement. Otherwise, at any further instant, the test t is evaluated and the behavior is selected as follows:

- If t is false, p is evaluated, and its termination, pausing, or trap exits are propagated to the whole `weak suspend` statement. The state change of p is propagated if p pauses, otherwise weak suspension is over.

- If t is true, p is also evaluated. If p exits a trap, so does the `weak suspend` statement. Otherwise, state change and potential termination are ignored, and the whole `weak suspend` statement pauses. It will restart from exactly the same state in the next instant, unless preempted by an enclosing preemptive statement in the current instant.

Section **??** for details and examples on trap exit.

For the immediate form

```
weak suspend
  p
when immediate t
```

the test t is also performed at first instant. If t is true at first instant, p is executed for the instant but its termination and state change are ignored unless p exits an enclosing trap, in which case so does the whole statement. Otherwise, from next instant on, the statement behaves just as a `weak suspend` statement.

The immediate form can be obtained from the standard form by the following macro-expansion:

```
trap Done in
  loop
    trap Immediate in
      {
        weak suspend
          p;
        when t;
      ||
        if t then exit Immediate end
      };
d     exit Done;
    end trap // Immediate
    pause
  end loop
end trap
```

The loop ensures fresh restart if t is true at first instant. If t is false at first instant, the whole statement behaves as the inner `weak suspend` statement.

8.13.3 Suspension and weak suspension examples

Effect of suspension on control

Consider the following example

```
suspend
  pause; // A
  emit X;
  pause; // B
  emit Y
when Susp;
emit Z
```

where `S` is an input and `X`, `Y`, `Z` are outputs. Here is a sequence of reactions to inputs, with '`-`' denoting an empty input or output:

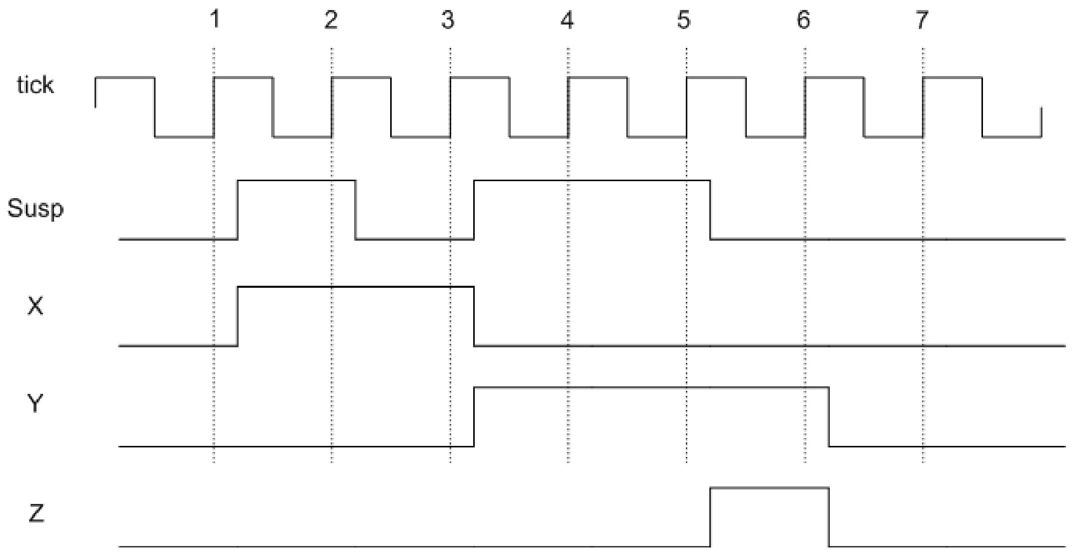


Figure 8.1: control suspension waveforms

```
1. -    → -   new state A
2. Susp → -   new state A
3. -    → X   new state B
4. Susp → -   new state B
5. Susp → -   new state B
6. -    → Y Z done
```

This behavior is pictured in timing diagram form in Figure ??. There is no output when `Susp` is present.

Let use replace `suspend` by `weak suspend`:

```
weak suspend
  pause; // A
  emit X;
  pause; // B
  emit Y
when Susp;
emit Z
```

Then, the outputs are generated at each instant since the body keeps being activated:

```
1. -    → -   new state A
2. Susp → X   new state A
3. -    → X   new state B
4. Susp → Y   new state B
5. Susp → Y   new state B
6. -    → Y Z done
```

See also the timing diagram form in Figure ??.

Consider steps 5 and 6. At step 5, execution resumes from pause labeled `B`. The "`emit Y`" statement is executed, `Y` is emitted, and the `weak suspend` body terminates. Since `Susp` is present, body termination is discarded, the `weak suspend` statement pauses, and there

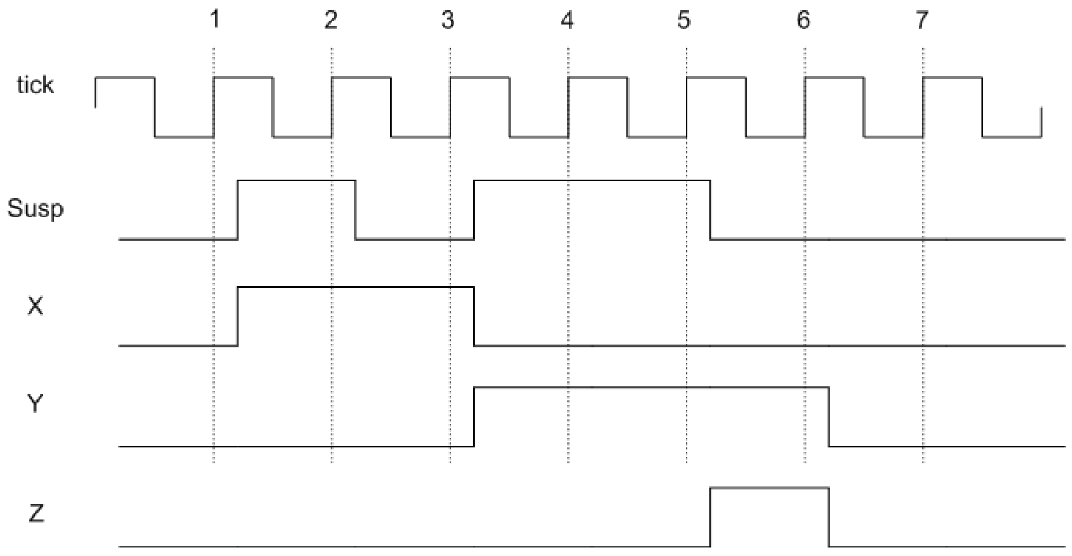


Figure 8.2: control weak suspension waveforms

is no control state change. Therefore, step 6 also starts from state `B`. In that step, "`emit Y`" is executed again, `Y` is emitted, and the `weak suspend` body terminates. This time, since `Susp` is absent, termination propagates to the whole `weak suspend` statement, and "`emit Z`" is executed.

Effect of suspension on data

To understand how suspension and weak suspension act on data, consider the following example:

```
module SuspendCount :
  input Susp;
  output Count : temp unsigned;
  suspend
    var C : unsigned := 0 in
      loop
        emit ?Count <= C;
        pause;
        C := C + 1
      end loop
    end var
  when Susp
end module
```

The program counts ticks in the following way:

- At first instant, the `C` variable is initialized to 0 and `Count` is emitted with that value.
- At any subsequent instant, if `Susp` is present, nothing happens and control stays paused at the `pause` statement; if `Susp` is absent, control resumes from the `pause`

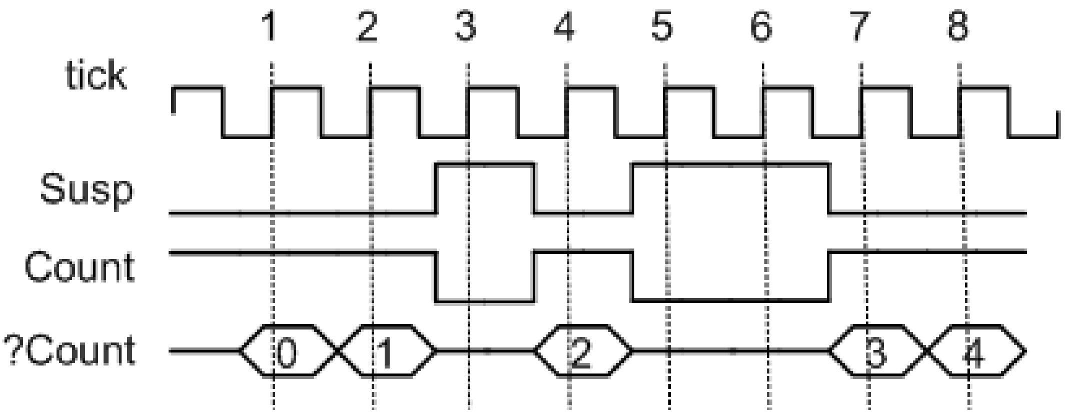


Figure 8.3: count suspension waveforms

statements. The assignment to `C` increments its value, the loop is looped, `Count` is emitted with the incremented `C` value, and control pauses again on the `pause` statement.

Here is an execution, with '`-`' denoting a blank input or output, also pictured as waveforms in Figure **??**:

```
1. -    → Count(0)
2. -    → Count(1)
3. Susp → -
4. -    → Count(2)
5. Susp → -
6. Susp → -
7. -    → Count(3)
8. -    → Count(4)
```

Replace `suspend` by `weak suspend`:

```
module WeakSuspendCount :
  input Susp;
  output Count : temp unsigned;
  weak suspend
    var C : unsigned := 0 in
      loop
        emit ?Count <= C;
        pause;
        C := C + 1
      end loop
    end var
  when Susp
end module
```

The behavior is exactly the same at first instant or when `Susp` is absent. It differs when `Susp` is present after first instant. Then, control propagates exactly as when `Susp` is absent, and `Count` is emitted with the incremented value of `C`. However, since `C` is within the scope of the `weak suspend` statement, the incremented value is not stored in `C`'s state

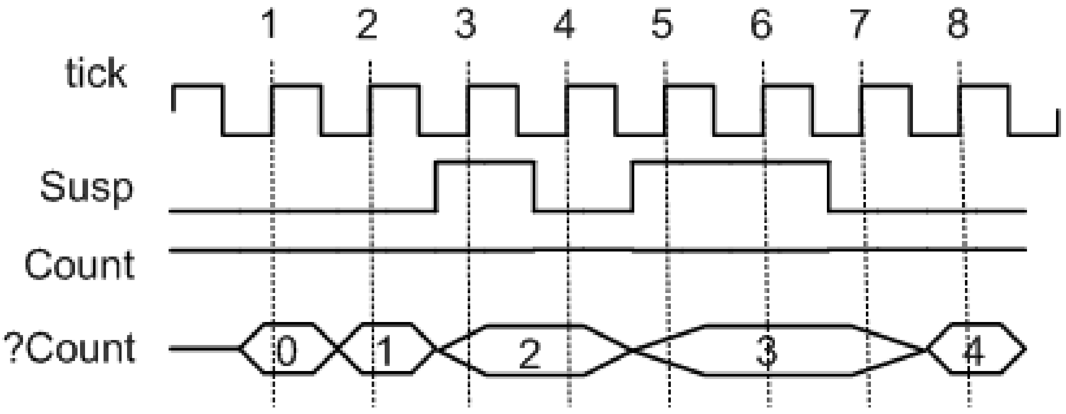


Figure 8.4: count weak suspension waveforms

memory, and the next execution will resume from the same value. In hardware terms, `C`'s register is disabled or clock-gated by `Susp`. Execution is as follows:

```
1. -    → Count(0)
2. -    → Count(1)
3. Susp → Count(2)
4. -    → Count(2)
5. Susp → Count(3)
6. Susp → Count(3)
7. -    → Count(3)
8. -    → Count(4)
```

Waveforms are shown in Figure **??**. Consider step 2: the value of `C` before the step is 1, set and stored by step 1. Since `Susp` is absent, `C` is incremented and its new value 1 is emitted as the value of `Count`. Consider step 3: control propagates as usual, incrementing `C` to 2 and emitting `Count` with this value. However, the new value 2 is not stored in the state but discarded; in hardware terms, the combinational computation results are discarded when the registers are clock-gated). Step 4 will be performed with `C` = 1. The next steps are similar.

Relative placement of signal and variable declarations and suspension

The relative placement of declarations and suspensions is crucial. In our previous example, things would be different if `C` was declared outside the `weak suspend` statement:

```
module BadWeakSuspendCount :
  input Susp;
  output Count : temp unsigned;
  var C : unsigned := 0 in
    weak suspend
      loop
        emit ?Count <= C;
        pause;
        C := C + 1
      end loop
    when Susp
  end var
end module
```

Then, `C`'s state (hardware register) is not subject to weak suspension (clock gating) and the new value of `S` is stored at each cycle, yielding the execution

```
1. -    → Count(0)
2. -    → Count(1)
3. Susp → Count(2)
4. -    → Count(3)
5. Susp → Count(4)
6. Susp → Count(5)
7. -    → Count(6)
8. -    → Count(7)
```

Since there is no suspended data state and only one control state bit (generated by the `every` statement), the `weak suspend` statement has no effect at all here.

Notice that there would be no behavior change for the same declaration/ suspension inversion with `suspend` instead of `weak suspend`, since the `Count` incrementation would not be performed either when `S` is present. The `suspend` and `weak suspend` statements are very different in this respect.

Notice finally that the intended behavior cannot be acheived without declaring an auxiliary variable (or signal) local to the `weak suspend` body. The following programs also performs a continuous incrementation instead of a suspended incrementation, since the value of `O` is declared at top-level and not subject to suspension:

```
module BadWeakSuspendCount :
  input Susp;
  output O : unsigned init 0;
  weak suspend
    loop
      pause;
      emit ?O <= pre(?O) + 1
    end loop
  when Susp
end module
```

Here also, the `weak suspend` statement is ineffective since it only acts on the unique control bit.

8.13.4 Interaction with traps

For `WeakSuspendCount` above, assume that we want to stop the count when value 3 is reached, irrespectively of the value of `Susp`. We can do this using an external trap:

```
module WeakSuspendCountWithTrap :
  input Susp;
  output Count : temp unsigned;
  output Done;
  trap DoneTrap in
    weak suspend
      var C : unsigned := 0 in
        loop
          emit ?Count <= C;
          if ?Count = 3 then exit DoneTrap end if;
          pause;
          C := C + 1;
        end loop
      end var
    when Susp
  end trap;
  emit Done
end module
```

With the same input sequence as before, the behavior is as follows:

```
1. -    → Count(0)
2. -    → Count(1)
3. Susp → Count(2)
4. -    → Count(2)
5. Susp → Count(3) Done
6. Susp → -
7. -    → -
8. -    → -
```

The “`exit Done`” statement reached at step 5 provokes immediate termination of the whole `weak suspend`.

Behavior is different with `trap` declared inside the `weak suspend` statement:

```
module WeakSuspendCountWithTrap2 :
  input Susp;
  output Count : temp unsigned;
  output Done;
  weak suspend
    trap Done in
      var C : unsigned := 0 in
        loop
          emit ?Count <= C;
          if (?Count = 3) then exit Done end if;
          pause;
          C := C + 1;
        end loop
      end var
    end trap
  when Susp;
  emit Done
end module
```

```
1. -    → Count(0)
2. -    → Count(1)
```

```
3. Susp → Count(2)
4. -    → Count(2)
5. Susp → Count(3)
6. Susp → Count(3)
7. -    → Count(3) Done
8. -    → -
```

Here, the inner trap is exited at steps 5, 6, and 7, provoking termination of the `weak suspend` body. however, at step 5 and 6, the body termination is discarded because of weak suspension, and `Done` is not emitted.

8.14 The finalize statement

The finalize statement has the following form:

```
finalize
  p
with
  q
end finalize
```

where the statement q is called the *finalizer*. The finalizer can contain only instantaneous statements: `emit`, assignment, `call`, possibly placed in sequence or in parallel.

When the finalize statement is started, it immediately starts p. If one of the following event occurs, the finalizer q is immediately executed and the whole finalize statement immediately terminates:

- p terminates, in which case q is executed in sequence after p;
- the `finalize` construct is strongly aborted by an an enclosing `abort` statement, in which case q is executed but not p (of course, unless the `abort` statement is immediate and immediately aborted, in which case the `finalize` statement is not started).
- the `finalize` construct is weakly aborted by an enclosing `weak abort` statement or by a concurrent exit of an enclosing trap, in which case the execution of q follows the last execution of p in the instant.

The finalizer is executed only once even if there are several reasons to execute it. Notice that the finalizer is executed even in the body of a strongly aborted statement, while strongly aborted normal statements are not executed. Finalizers allow the user to perform cleanup when a statement terminates or is aborted for any reason, or to broadcast a message as in the following example:

```
...
  finalize
     p
  with
    emit IamDead
  end
...
```

Consider the following example of a complex finalization context:

```
input I, J, K;
output X : integer,
       Y : integer;
abort // immediate I
  trap T in
    weak abort // J
      finalize
        emit ?X <= 1;
        await L;
        emit ?X <= 2
      with
        emit ?Y <= ?X + 1
      end
    when J
  ||
    await immediate K;
    exit T
  end trap
when immediate I
```

Here are some typical finalization cases:

- Termination: if `I`, `J`, and `K` do not occur, then `X(1)` is first emitted, followed at next tick by the simultaneous emissions of `X(2)` and `Y(3)` when `L` occurs.
- Strong abortion, with two subcases:
  - If `I` occurs at first instant, then the `finalize` statement is not executed at all and the finalizer is not called.
  - If `I` does not occur at first instant but occurs before `J`, `K`, and `L` or at the same time as them, then `X(1)` is emitted at first instant and `Y(2)` is emitted by the finalizer when `I` occurs.
- Weak abortion by `weak abort`: if `I` and `K` do not occur but `J` occurs after the first instant, then `X(1)` is emitted at first instant and there are two termination cases when `J` occurs:
  - if `J` occurs without `L`, then `Y(2)` is emitted.
  - if `J` and `L` occur simultaneously, then `X(2)` and `Y(3)` are emitted.
- Weak abortion by `trap–exit`: if `I` and `J` do not occur but `K` occurs at first instant or before `L`, then `X(1)` and `Y(2)` are emitted; if `K` and `L` occur simultaneously after the first instant, then `X(2)` and `Y(3)` are emitted.

In all cases, the whole finalize statement terminates.

When nested finalization occur, a cascade of finalizers can be executed. They are executed in inside-out order:

```
var X : integer := 1 in
  abort
    finalize
      finalize
        p
      with
        X := X+1
      end;
      X := X*2
    with
      X := X+3
    end
  when I
end var
```

If *p* terminates before `I` occurs, the innermost finalizer is executed, then the multiplication by 2, then the outermost finalizer, yielding `7`. If `I` occurs before *p* terminates, the innermost finalizer is executed before the outermost one and the multiplication is bypassed, yielding `X=5`.

Beware: `finalize` is a difficult statement to compile, and it can generate nasty cycles in cases a finalizer re-triggers itself in some way. Keep finalizers simple!

8.15 Local signal declaration

8.15.1 Local signal and port declaration and refinement

In Chapter **??**, we have seen the form of a local signal declaration "`signal` *decls* `in` *p* `end`" that declares a list of signals and ports with all attributes allowed for signals. Here is a example with signals only:

```
signal S,
       R : reg1 unsigned<[32]>,
       A[M][N] : bool[8] init '0,
       B[M] : temp value Byte init 0 combine + in,
       extends Intf,
       port P : Intf
  ...
end signal
```

An interface extension such as "`extends Intf`" locally declares all signals and ports defined at toplevel in `Intf`, ignoring their input / output directionalities. Here is an example:

```
interface Intf :
  input I : unsigned;
  output O;
end interface
```

```
module M :
  ...
  signal extends Intf in
    emit ?O <= 1;
    emit I;
    ...
  end signal
end  module
```

The "`extends Intf`" declaration is equivalent to the declaration sequence

```
signal I : unsigned, O in ... end
```

Mirroring `Intf` in the extension is allowed but ineffective since directionalities are ignored.

A local port declaration such as "`port P : Intf`" declares a local port `P` of interface `Intf`, where the interface input / output directionalities are ignored. Here, two signals `P.I` and `P.O` are declared. Mirroring `Intf` is accepted but has no effect. The port can be opened using the `open` statement of Section **??**.

For local extension and port declaration, since only interface attributes are declared in the interface, one can add `refine` declarations to specify the remaining module attributes needed for interface or port components. Here is an example, with `Intf` as in the previous example:

```
module M :
  ...
  signal S,
         extends Intf,
         refine O : reg,
         port  P : Intf,
         refine P.I : reg init 0 combine +,
                      // reg OK since I not input any more
         refine P.O : reg,
  in
    ...
  end signal
end module
```

Notice that `I` has been refined `reg`, which would not be allowed in a module header since `I` is declared input in the interface. In the `signal` context, the `input` declaration is ignored. Note that refinements can be grouped using curly brackets as for interface refinements. In the previous example, one could write:

```
signal extends Intf,
       port P : Intf,
       refine {O, P.O} : reg in
  ...
end signal
```

Local signals and ports can be declared in module headers (see Section **??**). In this case, the scope is the module body. The syntax is the same as the local signal and port declaration statement, except that the declaration finishes with '`;`'. Here is an example:

```
module N :
signal extends Intf,
       port P : Intf,
       refine {O, P.O} : reg;
   ...
end module
```

8.15.2 Local signals and suspension

Suspension interacts with taking the `pre` operators for a signal declared within the suspension body, as in

```
suspend
  signal S in
     ...
     if pre(S) then ...
     ...
  end signal
when I
```

The expression `pre(S)` refers to the status of `S` *in the previous instant where the signal declaration statement was activated*, not to the status of `S` in the previous absolute instant or tick. Instants where `I` suspends the `signal` statement do not count for `pre`. In some sense, the `suspend` statement steals the tick from its body. This is obvious when expanding the `pre` operators as explained in Section **??**:

```
suspend
  signal S, preS in
    loop
      if S then
        pause;
        emit preS
      else
        pause
      end if
    end loop
  ||
    ...
    if (preS) then ... end
    ...
  end signal
when I
```

The `pause` statements used to compute `preS` are indeed suspended by `I`.

On the opposite, the absolute `pre` is taken if the signal is declared outside the `suspend` statement:

```
signal S in
  suspend
    ... pre(S) ...
  when I
end signal
```

Here the expression `pre(S)` in the `suspend` body refers to the previous instant of the `signal` statement, independently of the presence or absence of `I`.

8.15.3 Local signal reincarnation

Because of instantaneous looping of loops, local signals can have several simultaneous instances that we call *reincarnations*. They pose no particular problem, but one has to be aware of their existence, in particular to understand causality issues. Here is an example:

```
loop
  signal S in
    if S then emit O1 else emit O2 end;
    pause;
    emit S
  end signal
end loop
```

In the first instant, the local signal `S` is declared. It is absent since there is no emitter for it. Therefore, the `else` branch of the `if` statement is taken and `O2` is emitted. In the second instant, the `pause` statement terminates and `S` is emitted and set present. The loop body terminates and it is rest arted afresh right away. The local signal declaration is immediately re-entered. It declares a *fresh incarnation* of `S`, distinct from the old one, whose status is lost since the declaration has been exited. The fresh incarnation is absent, unlike the old one. The `if` statement tests the fresh incarnation and only `O2` is emitted. Everything happens as if the loop body was duplicated:

```
loop
  signal S in
    if S else then emit O1 else emit O2 end;
    pause;
    emit S
  end signal;
  signal S in
    if S then emit O1 else emit O2 end;
    pause;
    emit S
  end signal;
end loop
```

In this obviously equivalent statement, the old and fresh incarnations are split into two syntactically distinct signals that happen to bear the same name `S` and the `if` statement is duplicated. In the original form, the single `S` generates two distinct dynamic incarnations, and the `if` statement dynamically tests the current incarnation of the signal.

The `pre` and `pre(?S)` operators always refer to the current incarnation. For example, in

```
loop
   signal S in
      if pre(S) then emit O1 else emit O2 end;
      pause;
      emit S
   end signal
end loop
```

the `O2` signal is continously emitted. The `S` emitted at the end of the loop body is not matched by `pre`, which matches the new incarnation, with `pre(S)` initially absent, as specified in Chapter **??** and Section **??**.

See [?] and [?] for thorough analyzis of reincarnation.

8.16 Open port declaration

Access to ports is normally done using the dot notation `P.I`, or `P[2].Q.I`, etc, see Section **??**. The `open` declaration makes it possible to make port field names directly visible, forgetting about the port name. The syntax is

open *port-name-list* **in** *p* **end open**

where *port-name-list* is a list of port names. For each port P in the list, the port field names N become synonym to the field $P.N$. A field name can hide a signal with the same name that already exists in the scope. Opening is not recursive. If a port `Q` is a field of `P` that itself has a field `S`, then "`open P`" declares `Q` as a synonym of `P.Q` but does not declare `S` as a synonym of `P.Q.S`. For this, one can open `P` and `Q` in sequence, typing "`open P, Q in` *p* `end`".

For a simple instance of `open` consider the following interface and module:

```
interface Intf :
  input I;
  output O;
end interface

module M :
  extends Intf;
  port P : Intf;
  input A;
  open P in
    emit O <= I and A
  end open
end module
```

Within the `open` declaration, `I` is a synonym to `P.I` and `O` is a synonym to `P.O`. Therefore, the `emit` statement is equivalent to

```
emit P.O <= P.I
```

If we add "`input I`" to the module header, then the `open` statement makes `I` an abbreviation of `P.I`, thus hiding the input `I` in the `open` scope.

An important usage of `open` is to call a submodule that extends an interface with argument a port with that interface. Without open, one must write such a call as follows:

```
interface Intf :
  input I;
  output O;
end interface

module Sub :
  extends Intf;
  ...
end module

module Master :
  port P : Intf;
  ...
  run Sub [P.I / I, P.O / O]
  ...
end module
```

which is really cumbersome if `Intf` has many fields. With `open`, one can use implicit binding:

```
open P in
  run Sub
end open
```

The implicit binding "`I / I`" then means "`P.I / I`". See for instance the memory example in Section **??**.

8.17 Local variable declaration

In Section **??**, we have presented local variable declarations. We recall that they start with `var` followed by a list of individual variable declarations. The variable scope is the variable declaration body, which is an executable statement that determines the behavior. Here is an example:

```
var V : unsigned,
    W : unsigned init 0,
    VarArray1 : unsigned[5] := {0,1,2,3,4},
    VarArray2 : temp bool[32] := '0 in
  ...
end var
```

Variables can be initialized at declaration time using the `:=` symbol. If the type is an array type, the initializer can be either an array literal or a simple literal of the array base type, see Section **??**. It the variable is declared `temp`, the initialization is performed at each cycle before any usage of the variable.

Chapter 9

The run module instantiation statement

A module can be instantiated within another module using the `run` executable statement. The instantiated module is called the *submodule*, generically called `Sub`, the other one being the *master module*, generically called `Master`. The basic conceptual principle is that `run` executes the code of `Sub` in `Master`, after instantiating all generic data parameters of `Sub` if there is any, and after directly or indirectly binding the signals of the master module to the signals of the submodule. Recursive `run` statements are allowed, under the condition that the recursion is *static*: the recursion must be based on generic constants of the submodule; terminal and non-terminal recursion cases must be tested using `if static` statements, see Section **??**.

The `run` statement is a normal statement that can be used anywhere a statement can, without restriction. A `run` statement starts the submodule body when it starts, terminates when its body terminates, and aborts its body when it is aborted. Therefore, a `run` statement can be put in sequence or in parallel with other `run` statements or with any other statement, it can be placed in any abortion context which will abort the submodule execution, etc.

9.1 Basic syntax

The `run` statement makes it possible to explicitly or implicitly pass data parameters to generic submodule data objects, and to pass reset signals to the submodule. Data parameters, signals, and reset parameters are passed in a bracketed list. Here is an example:

```
run Sub [type unsigned<8> / T1, bool / T2;
        constant 12 / N;
        signal X / I, Y / O;
        reset R ]
```

Since signal binding is the most frequent, the `signal` keyword can be omitted. Therefore, one can write

```
run Sub [type unsigned<8> / T1, bool / T2;
        constant 12 / N;
        X / I, Y / O;
        reset R ]
```

which is most useful when there are only signal renamings:

```
run Sub2 [X / I, Y / O]
```

For generic submodule data objects, there is a similar implicit substitution mode: if a generic parameter of `Sub` is not substituted, there must exist a data object of the same kind, name, and type in `Master`, which then replaces the generic parameter in the submodule interface and body.

There is a default mode for signal binding called *implicit mode:* when a signal argument is not explicitly bound in the `run` statement, it is assumed that a signal with the same name exists in `Master` and the binding is done to this master signal. Therefore, an absence of binding for an interface signal `S` is viewed as a binding "`S / S`".

There is also a *null binding* of the form '`/ S`' for a submodule interface signal or port `S`, see Section **??** for details.

All submodule generic data objects must be explicitly substituted or bound when running the submodule. A `run` statement does not make the data objects declared in the submodule visible.

9.1.1 Naming a submodule instance

The same submodule can be run several times in a master module. It is not necessary but often convenient to explicitly name the submodule instances, for instance to trace them back from the generated code or to display them in debuggers and browsers. This is done with the following syntax:

```
module Master :
  ...
  run Sub1 / Sub
  ...
  run Sub2 / Sub
  ...
end module
```

This form of instance naming applies to all all forms of `run` described below, with or without argument lists.

9.2 Argumentless run statement

The simplest form of `run` statement consists of the `run` keyword followed by the submodule name.

```
module Sub :
  generic type T;
  input I : T;
  output O;
  ...
end module

module Master :
  type T = unsigned<16>;
  type U;
  input I : T;
  signal O in
```

```
      ...
      run Sub
      ...
    end signal
  end module
```

Here, binding is done implicitly: the generic type paramemer `T` of `Sub` is bound to the actual type "`T = unsigned<16>`" of `Master`, the input signal `I` of `Sub` is bound to the input `I` of `Master`, and the output signal `O` of `Sub` is bound to the local signal `O` of `Master`. The precise behavior of data and signal bindings are presented in Section **??** and Section **??** below.

9.3 Data substitution

Each generic data object declared in the module header of `Sub` (directly or by extension) must be substituted at module instantiation time using the syntax already presented in Chapter **??**:

```
  run Sub [ type integer / T;
            constant 0 / Initial,
                     1 / Increment;
            function MyMult / Mult,
                     F / SubF;
            procedure P / SubP ]
```

Data substitutions lists consist of a data kind keyword followed by an arbitrary number of comma-separated substitution items. There can be any number of substitution lists, themselves separated by semicolons. As for generic data extension, a substitution "`X / Y`" is read "`X` is substituted to `Y`", or "`X` renames `Y`". The left argument `X` must be a predefined data object (constant, operator) or a data object declared in `Master`, which can itself be generic or defined generic. The right argument `Y` must be a generic data object of `Sub`. Both `X` and `Y` must be of the same kind (type, constant, function, or procedure). For constants, the types must agree after type renaming in `Sub`. For functions and procedures, the argument lists must match after type renaming in `Sub`.

All generic data objects of the submodule must be either explicitly or implicitly bound to data objects of the master module. Therefore, for any submodule generic parameter which is not explicitly bound, there must exist a data object ot the same name and kind in the master module, with matching type for constants, functions, and procedures. The master data object can itself be generic or defined generic.

Unlike for generic data extension presented in Section **??**, the data object declared in the submodule data part are not imported in the master module, Therefore, only generic data objects can be substituted in a `run` statement. Remember that data extension also permits renaming of defined generic data objects, in order to change their name when importing them. This facility is not needed here since there is no data import. Therefore, it is disallowed to rename generic defined data objects in a `run` statement.

9.4 Signal binding

The `signal` keyword that starts a signal binding list is optional and can be omitted. Signals are implicitly bound by name or explicitly bound using the same "`X / Y`" notation as for data, in which case we say that `Y` is *bound* to `X` or that `X` *renames* `Y`, or using the null binding notation "`/ Y`", in which case we say that `Y` is unbound.

Unless null-bound, all the interface signals of `Sub` must be implicitly or explicitly bound to signals of `Master` visible in the scope where the `run` statement is executed. If an interface signal `X` of `Sub` is not explicitly bound in the `run` statement, then *implicit binding by name* is used, the binding being assumed to be `X / X` where `X` must be defined in `Master`. There are several variants of signal bindings, described below.

9.5 Expression binding

Expression binding has the form

```
run Sub [ signal (e) / I ]
run Sub [ (e) / I ]
```

where `I` is an input of `Sub`, which must be either pure or value-only, and e is an expression valid in the the `run` context within `Master`. Binding an expression to a full signal or to an output signal of `Sub` is disallowed. Parentheses around e are mandatory to avoid ambiguity with other forms of binding. The expression is evaluated e at each instant when `Sub` is active. In particular, an access-to-uninitialized-signal-value run-time error occurs if e refers to a signal with undefined value at an instant when `Sub` is active.

9.5.1 Expression binding to a pure input

If `I` is a pure input signal or pure input array of `Sub`, then e must be a Boolean expression or Boolean array expression. For arrays, dimensions must match.

For example, "`run Sub [ (S.X[1] or T or ?T) / I]`". is a correct signal binding for `S` a port with a pure signal array field `X` and for `T` a signal of type `bool`.

At each instant where `Sub` is active, the expression binding sets `I` in `Sub` present if the the current value of e is *true* in `Master`.

9.5.2 Expression binding to a value-only input

If `I` is a value-only input signal or value-only input signal array of `Sub`, then the type and dimension of e must match the type and dimension of `?I`. At each instant where `Sub` is active, the expression binding sets the value of `I` in `Sub` to the the current value of e in `Master`. Here is an example:

```
module Master:
type T = bool[16];
map T {A[0..3], B[4..7], C[8..15]};
input I : T;
...
run Sub [(bin2u(?I.A))/J, (?I.B [or] ?I.C[0..3])/K]
...

module Sub:
input J : value unsigned<[4]>;
input K : value bool[4];
...
```

9.6 Input binding

Input binding has the form

```
run Sub [ E / I ]
```

where `I` is an input signal or signal array of `Sub` and `E` has one of the following forms:

- A simple signal, for instance `S`.
- A partially or totally indexed or sliced signal array identifier, for instance `S[1]`, `S[..][2][1..3]`.
- A port name, possibly indexed or sliced, followed by an expression of the same form, for instance `P.S` or `P[1..2].S[1]`.

In all the cases above, the master `S` can be either standard or registered. Of course, dimensions must match for arrays and slices. The submodule input can be declared `temp` independently of the characteristics of the master signal. It cannot be declared registered.

One can forget status or value attributes of the master signal in input binding. If the submodule input is pure, the master signal can be pure or full, in which case its value is forgotten. If the submodule input is value-only, the master signal can be value-only or full with the same type, its status being forgotten in the latter case. If the submodule input is full, then the master signal must also be full.

9.6.1 Full signal input binding

For a pure or full submodule input `I`, the status of `I` receives the status of the master argument. If `I` is an array, each component of the submodule input receives the corresponding component of the master array or slice.

Therefore, the status of the submodule input `I` signal is set *present* whenever the module is active and the status of the master signal or signal component is *present*. The status of `I` can also be set present by `Sub`'s body.

At `Sub` starting instant, `pre(I)` is *absent* and `pre1(I)` of `I` is *absent.*

Value transmission and initialization for full signal

Assume `I` is a full input signal of `Sub`. Then, the master signal or array component must also be a full signal. The value transmission laws follows from the general fact that value emission is controlled by the status for a full signal. The general law is as follows:

> *A full input is like a local signal of* `Sub` *for which the master module acts as an extra emitter.*

In the sequel, we detail the consequences of this general rule by an in-depth study of all the derived subcases.

Any emission of the master signal or signal component in the master module triggers an emission of `I` in the submodule with the master value. If `I` is combined in `Sub`, then the master value is combined with the values internally emitted in `Sub` if there are any. Value definition and initialization is as follows:

- If `I` is persistent (non-`temp`) and has no `init` attribute, its value `?I` is undefined until the first reception from the master module or the first local emission. The value `pre(?I)` is undefined until the first instant that follows that instant.
- If `I` is persistent and has an "`init` *E*" attribute, the values `?I` and `pre(?I)` are initialized to that of *E*. The initial value of `?I` is overwritten by the first reception or internal emission of `I`, which may occur at starting instant.
- If `I` is declared `temp` and has no `init` attribute, its value is defined only when the signal is *present*, i.e. received from `Master` or emitted in `Sub`. Remember that the expression `pre(?I)` is disallowed for a `temp` signal.
- If `I` is `temp` and initialized using a "`init` *E*" attribute, its value is always defined. At each instant, it is equal to that of *E* if `I` is neither received from `Master` nor internally emitted, and as the emitted value otherwise.

9.6.2 Value-only input binding

For a value-only input, the laws directly follow form the laws of full signals by considering the master status as always *present*. Therefore, the value of the master signal is imported at each instant into `Sub`. For `I` combined, the master value is combined with the values locally emitted in `Sub` when `Sub` is active. If `I` is not combined, then it cannot be locally emitted in `Sub`.

The value `pre(?I)` can be used for a persistent (non-`temp`) signal. It can be initialized using an "`init` *E*" attribute in the declaration of `I`. Notice that `init` then acts only on `pre(?I)` since `?I` is input from `Master` at `Sub` start time. Since this last point can be confusing, it is advisable for compilers to generate warnings stating that the initial value will never be used if `pre(?I)` is never used.

9.6.3 Input initialization examples

Single input initialization example

Consider submodule valued inputs of the form

```
module Sub :
  input I : integer;
  input Iinit : integer init 5;
  input IV : value integer;
  ...
end module
```

with a binding of the following form in `Master`:

```
run Sub [S / I, S / Iinit, S / IV]
```

We use a single master signal `S` to simplify the explanation; one could of course use three distinct master signals. Assume first that `S` is emitted in `Master` when `Sub` starts:

```
emit ?S <= 2;
run Sub [S / I, S / Iinit, S / IV]
```

In this case, the value 2 of `S` is passed from `S` to `I`, `Iinit`, and `IV`. The value of `pre(?Iinit)` at that start instant is 5, while the values of `pre(?I)` and `pre(?IV)` are undefined.

Consider now the case where `S` is not emitted in `Master` when `Sub` starts:

```
emit ?S <= 2;
pause;
run Sub [S / I, S / Iinit, S / IV]
```

In this case, at the second instant, `I` is left undefined, `Iinit` is initialized to its explicitly given initial value 5, and `IV` takes the value 2 of `S`. As before, the value of `pre(?Iinit)` at that instant is 5, while the values of `pre(?I)` and `pre(?IV)` are undefined.

Assume now that we remove the initial "`emit ?S <= 2`" statement in this last example and that `S` is not initialized in `Master` when `Sub` starts. Then, there is a undefined signal run-time error for `S` since one tries to read the value of `S` in order to initialize that of `IV`.

Combined inputs initialization examples

Consider now the case where the submodule inputs are combined:

```
module Sub :
  input I : integer combine +;
  input Iinit : integer init 5 combine +;
  input IV : value integer combine +;
  input Combine;
  if Combine then
    emit {
      ?I <= 6,
      ?Iinit <= 6,
      ?IV <= 6
    }
  end if;
  ...
end module
```

The behavior is exactly as before if `Combine` is not emitted by `Master` module when `Sub` starts. If `Combine` is emitted, Then, when `Sub` starts, initialization is disabled because of internal emission. There are two cases according to the status of `S`. Consider first the case where `S` is emitted.

```
emit {
  ?S <= 2,
  Combine
};
run Sub [S / I, S / Iinit, S / IV]
```

Then, combination of master and internal values is performed for all inputs, which all take value 8.

Consider next the case where `S` is not emitted when `Sub` starts:

```
emit ?S <= 2;
pause;
emit Combine;
run Sub [S / I, S / Iinit, S / IV]
```

The signal `S` and `Iinit` take value 6, while `IV` takes combined value 8 since a valued signal is always viewed as emitted by the environment.

9.7 Output binding

Output binding has the forms

```
run Sub [ E / O ]
run Sub [ next E / O ]
```

where `O` is output signal or signal array of `Sub` and E is a signal component, array, or slice of the same form as for input binding, see Section ??. With no `next` keyword, the binding is called an *immediate output binding* and E must be a non-registered signal component; with the `next` keyword, the binding is called a *delayed output binding* and E must be a registered signal component; Since output binding acts like a signal emission in an `emit` or `sustain` statement for the master signal (see output binding behavior below), the `next` keyword is mandatory in the syntax of an output binding for a registered master signal exactly like for the emission of a registered signal in an `emit` or `sustain` statement. The submodule output `O` can be standard or registered. Types must match for typed signals, and array dimensions must match for signal arrays.

One can forget signal attributes of `O` in the binding: if `O` is full, then E can be a pure or value-only signal or signal array. If E is pure, the value of `O` is discarded. If E is value-only, the status of `O` is discarded. One cannot add attributes from `O` to E: if `O` is pure, then E must be pure; if `O` is value-only, then E must be value-only.

9.7.1 Output binding behavior

Output binding behavior is simpler than input binding:

- As far as `Master` is concerned, whenever `Sub` is active, there are two cases:
  - If `O` is pure or full, presence of `O` in `Sub` triggers emission of E in `Master` provided `Sub` is alive. This emission of E can be combined with other emissions of E in `Master`.
  - If `O` is value-only and `Sub` is alive, then `O` triggers an emission of E in `Master` with the same value at each instant when the `run` statement is active. This emission is combined with other emissions of E in `Master` if E is combined.

  Signal emission acts the same as with an `emit` or `sustain` statement: for an immediate output binding, emission of E in `Master` impacts status and value of E in the current cycle; for a registered output binding, emission of E in `Master` impacts status and value of E in the next cycle, since E is registered.

- As far as `Sub`'s body is concerned, `O` is viewed exactly in the same way as a local signal. Its value is initialized by an `init` attribute (declared directly or by a `refine` declaration), or undefined until the first emission, See Section ??. Because of the second rule above, only a full signal can be left undefined. A value-only output signal must always have a defined value.

Notice that an emission of the submodule output is propagated only when the submodule is alive. Consider the following case:

```
module Sub :
  output R : reg;
```

```
    emit next R;
  end module

  module Main :
  output X;
  signal S in
    run Sub [S / R]
  ||
    await S;
    emit X
  end signal
  end module
```

Here, `Sub` is started at first instant, emits `R` for next instant, but immediately dies. Since `Sub` is dead at second instant, the registered emission of `R` is not propagated to `Main` and `S` and `X` are not emitted.

9.7.2 Output signal initial value transmission

Notice that the initial value of `O` in `Sub` is visible to `Master` in two cases detailed below

In the first case, `O` is declared `reg1`. In that case, `O` must be initialized by an `init` attribute. Since `O` is emitted at first instant in `Sub`, its initial value is passed to `Master`. Here is an example:

```
  module Sub :
    output O : reg1 unsigned init 3;
    ...
  end module

  module Main :
    output X : unsigned;
    run Sub [ X / O]
  end module
```

At first instant, `X` is output with value 3. Omitting the `init` attribute would provoke an undefined value run-time error. If, instead, `O` is declared `reg`, there is no value transmission at initial intant and no undefined value error if there is no `init` attribute.

In the second case of initial value transmission, `O` is declared value-only. Then, unless it is emitted in the first instant of `Sub`, `O` must have an `init` attribute, and its initial value is passed to `Master` at first instant.

9.8 Inputoutput binding

Inputoutput binding has the form

```
  run Sub [ S / IO ]
```

where `IO` is an inputoutput interface signal or array of `Sub` and `S` is a signal, signal array, or port component valid in the `run` context within `Master`, just as for input and output bindings. The inputoutput signal `IO` cannot be initialized in `Sub`. All the attributes of `IO` must match those of `S`: type, array dimension, `temp` or persistent character, combination function if any. Array dimensions must match. The `S` and `IO` signals must be both standard or both registered. If any of the signals is valued (resp. value-only), the other must be valued (resp. value-only) and with the same type (there is no forgetting for inputoutputs). Each component of `IO` is identified to the corresponding component of `S`.

In some sense, inputoutput binding acts as passing by reference in traditional software languages.

9.9 Port binding

Port binding has the form

```
run Sub [ Q / P ]
```

where `Q` is a port or port component of `Master` (i.e. `Q.R`, `Q[4].R[2]`, etc.), and `P` is a port interface signal or array of `Sub` having the same interface as `P`. Each signal component of `P` is bound to the corresponding signal component of `Q` according to the signal binding rules above.

9.10 Signal binding example

Here is an example of valid signal binding:

```
interface Intf :
  input A;
  output B;
end interface

module Sub:
  constant N : integer;
  input I, J, K[N], L;
  output O1, O2[5], O3;
  inputoutput IO;
  port P : Intf;
end module

module Master :
  input X [5];
  signal Y, Z[12], IO, port Q : Intf in
    ...
    run Sub [ constant 12 / N;
              (Y or Z[1]) / I,
              Y / J,
              Z / K,
              Z[1] / L,
              Y / O1,
              X / O2,
              Z[1] / O3,
              Q / P ]
    ...
  end signal
end module
```

Here, `(Y or Z[1]) / I` is an expression binding; `Y / J` and `Z / K` are input bindings, the latter binding arrays; `Z[1] / L` is an array to input binding; `Y / O1` and `X / O2` are output bindings, the latter binding arrays; `Z[1] / O3` is an output to array binding; `Q / P` is a port binding; finally, there is an implicit inputoutput binding `IO / IO`. Notice that master signals can appear several times in bindings, and even in both input and output bindings.

9.11 Null binding

Assume that `S` is an interface signal of `Sub` subject to a *null binding* "`/ S`" in a `run` statement:

```
run Sub [ / S, ...]
```

Then `S` is not bound to any master module signal and is made local to the instance of `Sub` generated by the `run` statement. If `S` is an input (resp. an output), no status or value is propagated from (resp. to) the master module.

Null binding is also available for ports. For an interface port `P` of `Sub`, "`/ P`" recursively means that all components of `P` are null-bound.

Warning: *the Esterel Studio v7_60 compiler currently puts a restriction on null binding: it is not available for valued inputs nor for inputoutput signals when* `Sub` *is compiled modularly into VHDL or Verilog. This is because only only null output binding is available for these languages. However, a pure input can be null-bound since this is equivalent to receiving 0 from* Master*. There is no restriction if* `Sub` *is inlined instead of being compiled modularly.*

9.12 Value passing rules

We give examples of value passing between a master module and a submodule.

9.12.1 Value passing examples

Consider the following submodule:

```
module Sub :
  output O : integer;
  output O1 : integer init 1;
  pause;
  emit {
    ?O <= 2,
    ?O1 <= 3
  }
end module
```

and the following signal binding:

```
run Sub [ X / O, Y / O1]
```

Assume first that `X` and `Y` are standard signals. Then, values are passed only when the output signals are emitted in `Sub`. Therefore, when `Sub` starts, no value is passed from `Sub` to the master module. The initialization of `O1` only applies within `Sub`; at next instant, value 2 is passed to `X` and value 3 is passed to `Y`. If the `pause` statement is removed, the values are passed at `Sub` start time, and the initialization of `O1` is discarded since the signal is emitted. Valus passing is the same if `X` and `Y` are value-only, the status of `O` and `O1` being ignored.

9.12.2 Passing values from registered submodule outputs

Consider the following submodule with registered valued outputs :

```
module Sub :
  output R : reg integer;
  output R1 : reg1 integer init 1;
  emit next {
    ?R <= 2,
    ?R1 <= 3
  };
  pause
end module
```

and a call for this submodule of the form

```
run Sub [X / R, Y / R1 ]
```

where `X` and `Y` are integer signal, full or value-only. The value of `R` is exported to `X` when `R` is *present*, one tick after being emitted. Since `R1` is declared `reg1` it is also *present* when `Sub` starts, passing value 1 to `Y`. Remember than the trailing `pause` in `Sub` is mandatory, otherwise the values of `R` and `R1` would not be exported since `Sub` would be dead at second instant.

9.13 Resetting submodules

One must specify a reset signal in a `run` statement that instantiates a module within a multiclock unit. There are two kinds of reset, *strong reset*, which acts at the beginning of the instant and is the default, and *weak reset*, which acts at the end of the instant (the terms weak and strong have the same meaning as for `abort`). In the hardware classical terminology, strong reset corresponds to "asynchronous reset", and weak reset corresponds to "synchronous reset". We do not use these ambiguous terms, since the semantics is synchronous is both cases.

9.13.1 Strong reset

Strong reset conceptually resets the module at the beginning of the instant and maintain it in its reset state. It is written as follows:

```
run Sub1 / Sub [reset R; clock C; ...]
```

The semantics is that of the following statement:

```
loop
  weak suspend
    run Sub1 [...];
  when immediate R or not C
each R
```

When `R` occurs, `Sub1` is strongly aborted, restarted, and weakly suspended.

9.13.2 Weak reset

Weak reset conceptually resets the module at the end of the instant when the clock is present. It is written as follows:

```
run Sub1 / Sub [clock C; weak reset R; ...]
```

The semantics is that of the following statement:

```
weak suspend
  loop
    weak abort
      run Sub1 [...]
    when immediate R;
    pause
  end loop
when immediate not C
```

When `C` is present, when `R` occurs, `Sub1` is weakly aborted, and it is restarted afresh at next instant.

9.13.3 Why passing reset signals

The interest of passing reset signals shows up in hardware synthesis: one can realize the above behavior in a way that is electrically much simpler and much more efficient than the Esterel expansion.

9.14 Transforming submodule assumptions and assertions

It is possible to transform submodule assumptions and assertions in a `run` or `mcrun` statement. Submodule assumptions can be transformed using an *assumption renaming*:

- The renaming '`assert/assume`' transforms `Sub` assumptions into assertions for `Master`.
- The renaming '`/assume`' discards `Sub` assumptions in `Master`.
- The renaming '`assume/assume`' leaves `Sub` assumptions unchanged for `Master`. It is implicit when no assumption renaming is specified in the `run` or `mcrun` statement.

Likewise, submodule assertions can be transformed using an *assertion renaming*:

- The renaming '`assume/assert`' transforms `Sub` assertions into assumptions for `Master`.
- The renaming '`/assert`' discards `Sub` assertions in `Master`.
- The renaming '`assert/assert`' leaves `Sub` assertions unchanged for `Master`. It is implicit when no assertion renaming is specified in the `run` or `mcrun` statement.

Here is an example based on an initiator-target protocol. Modules `TargetAssertions` and `InitiatorAssertions` are intended for the protocol verification respectively on the target side and master side. Assertions in `TargetAssertions` are actually assumptions in `InitiatorAssertions` and conversely, hence the use of assertion and assumption renamings.

```
interface TargetIntf:
input Valid;
output Ready;
input Data : value temp unsigned;
end interface

module TargetAssertions:
port T : observe TargetIntf;
open T in
  sustain {
    assert ReadyStable = Ready if pre(Ready and not Valid),
    assume ValidStable = Valid if pre(Valid and not Ready)
  }
end open
end module

module InitiatorAssertions:
port I : observe TargetIntf;
run TargetAssertions [ I/T; assert/assume, assume/assert ];
end module
```

9.15 Capture and connection of submodule interface signals

Warning: *this section is specific to the Esterel Studio v7_60 compiler. It details an implementation-related issue which is independent of the language definition proper.*

The rest of the chapter details a point which is central to understand how the Esterel v7_60 compiler handles signals and generates registers to compute their `pre` status and value: the connection of submodule interface signals to their master signal in a `run` statement.

Semantically speaking, only inputoutput signals in submodules loose their identity in the `run` statement since they are identified with their master signal. Input and output submodule signals keep their identity and are appropriately *connected* to their master signal using the rules presented above. This is essential when input signals are locally emitted or output signals are locally tested in the submodule, which makes perfect semantical sense and is often useful. Consider the following example:

```
module Sub1 :
input I;
output OSub;
   emit I
||
   if I then emit OSub end
end module


module M1 :
output OM, OSub;
signal S in
   if S then emit OM end
||
```

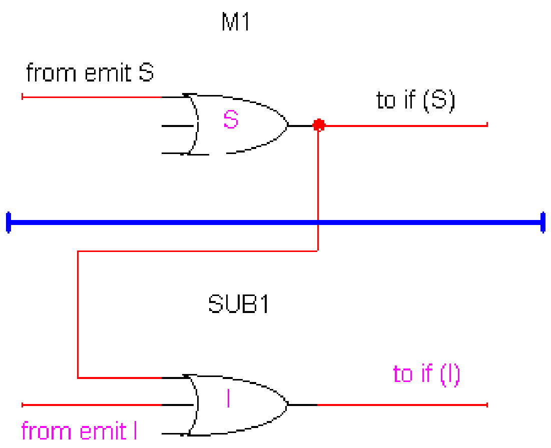


Figure 9.1: input connection

```
    run Sub1 [ S / I]
end signal
end module
```

Then, at first instant, `M1` outputs `OSub` but not `OM`: the input declaration in `Sub1` acts as a diod, preventing the emission of `I` in `Sub1` of percolating into `M1` and provoking emission of `S`. (notice that emission of `S` would indeed occur in `M1` if `I` was declared inputoutput in `Sub1`). Consider now the dual example with an output signal binding:

```
module Sub2 :
output O;
output OSub;
    if O then emit OSub end
end module

module M2 :
output OM, OSub;
signal S in
    emit S
||
    run Sub2 [ S / O]
end signal
end module
```

Here, `S` is emitted in `M2` but does not traverse the output connection barrier: `O` is not present in `Sub`. Therefore, `OSub` is not emitted. Here also, `S` in `M1` and `O` in `Sub1` have distinct statuses.

9.15.1 Implementing connection

We explain how pure signal connection is implemented in Esterel v7_60. The full handling of valued signals is similar but much more technical and will not be detailed here.

The status of a signal is implemented by an *or*-gate, to which emitters are connected and which is connected to `if` statements that test the signal. The wirings for modules `M1`

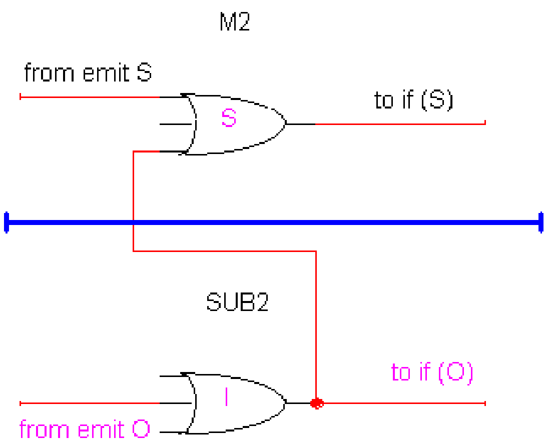


Figure 9.2: output connection

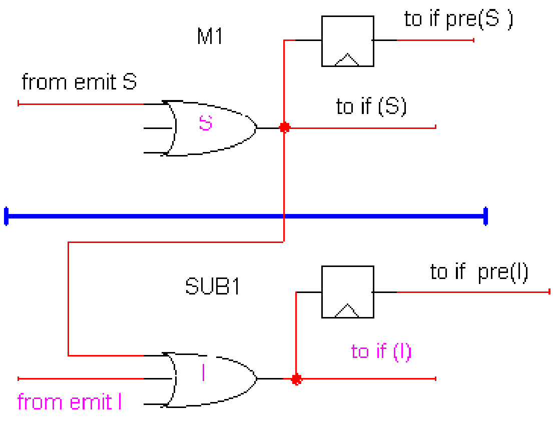


Figure 9.3: input connection

and `M2` above are respectively pictured in Figure **??** and Figure **??**.

9.15.2 The hidden cost of connection

Connection wiring is obviously correct and quite cheap as long as things remain combinational. But there is a cost issue with the `pre` operator and with persistent signal value handling. Assume that `I` is valued in `M1` above, and assume one computes `?S` and `pre(S)` in `M1` on the one hand and `?I` and `pre(I)` in `Sub` on the other hand. In the same way as `S` and `I` have distinct statuses, they must have distinct value cells and pre registers, as pictured in Figure **??**. This means a doubling of the number of registers in the circuit, which may be unacceptable for performance reasons.

9.15.3 Replacing connection by capture

The idea of *capture* wiring is to wire the circuit as if the submodule signal was an inputoutput, keeping only one status, one `pre` register, and one value cell for the master module and submodule. Capture is obviously much cheaper when correct since registers and value transfers are spared. As `M1` and `M2` above show, capture is incorrect in the general case. Therefore, it is essential to understand all the cases where capture is correct. For instance, in `M1`, capture becomes correct if emission of `I` in `Sub1` is removed, while in `M2` capture becomes correct if test for `O` is removed in `Sub2`.

Technically, it is simpler to detail all the cases where connection is mandatory and why this is the case. The Esterel v7 compiler automatically detects all the cases and warns when connection is performed if the `-eiclink:-connection_info` option is used.

Capture obviously makes sense only for signal-to-signal binding. It is irrelevant for expression binding.

Capture is not available in a multiclock context, see Chapter **??**.

Potential emission, testing, and reading of a signal

We say that a signal `S` is *potentially emitted* in a module in the following cases:

- There is an "`emit S`" or "`sustain S`" statement for the signal in the module.
- The module runs a submodule with `S` bound to an output or inputoutput signal of the submodule

We say that a signal `S` is *potentially tested* in a module in the following cases:

- There is an expression that reads the signal status `S`, the previous status `pre(S)`, or the next status `next(S)`.
- The module runs a submodule with `S` bound to a pure or full input or inputoutput signal of the submodule.

We say that a signal `S` is *potentially read* in a module in the following cases:

- The value `?S`, the previous value `pre(?S)`, or the next value `next(?S)` is read within the module.
- The module runs a submodule with `S` bound to a valued input or inputoutput signal of the submodule.

9.15.4 Input and output signal shared connection cases

A submodule input or output signal `X` in `Sub` cannot be captured by `S` in `M` and must be connected to it in the following cases:

1. The `run` statement is not toplevel.
2. `X` is declared with an initial value in `Sub`.
3. One of `X` and `S` is declared combined but not the other one or `X` and `S` are declared combined with different combination functions. Connection is required since values may differ in case of multiple emissions.
4. `S` is declared temporary in `M` but `X` is not temporary in `Sub`.

5. S is declared registered in M.

6. X is declared registered in Sub.

7. There is a suspend or weak suspend statement in M between the declaration of S and the run Sub statement. This case is more delicate. Consider the following program:

```
module Sub3 :
input I, J;
... pre(I)...pre(J)...
end module

module M3 :
input X, Susp;
suspend
  signal Y in
     ... pre(X)...pre(Y)
     ... run Sub3 [ X / I, Y / J]
when Susp
```

   Then, Y can be captured by J but X cannot be captured by I because *suspension acts on pre.* In M3, pre(I) is relative to the global tick, while, in Sub3, pre(X) is relative to the ticks where Sub3 is alive, i.e. the initial tick and those where Susp is absent. Therefore, pre(I) $\neq pre(X)$ and the signals cannot be identified There is no problem for the Y-J pair since both signals are subject to the same suspension.

9.15.5 Input signal specific connection cases

A submodule input signal I in Sub cannot be captured by S in M and must be connected to it in the following cases:

1. I is potentially emitted in the submodule. This is example M1 of Section ??.

2. I is declared full and S is declared with an initial value in M.

9.15.6 Output signal specific connection cases

A submodule output signal O cannot be captured by its master signal S and must be connected to it in the following cases:

1. O is declared full in Sub and S is declared pure or value-only in M.

2. O is potentially tested or read in Sub and S is potentially emitted in M outside the "run Sub" statement. This is example M2 / Sub2 of Section ??.

3. O is potentially tested or read in Sub, the "run Sub" statement is replicated, and there is a single master signal S for the replicated outputs.

4. O is potentially read in Sub and S has an init attribute.

5. S is emitted in M and O is defined by an "emit seq" or "sustain seq" statement in Sub.

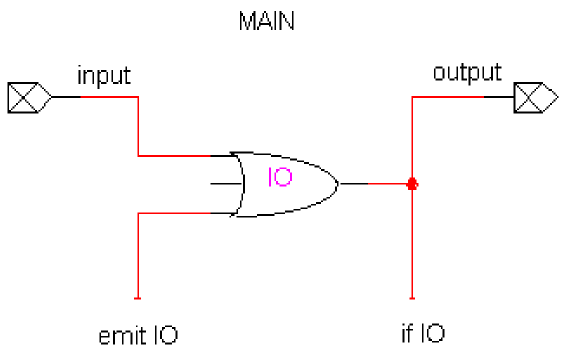


Figure 9.4: main module inputoutput

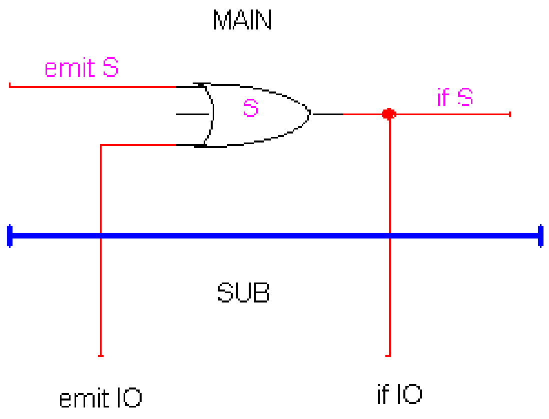


Figure 9.5: submodule inputoutput

6. `O` is declared value-only in `Sub` and `S` is potentially emitted in `M` outside the "`run Sub`" statement.

7. `O` is declared value-only in `Sub`, the "`run Sub`" statement is replicated, and there is a single master signal `S` for the replicated outputs.

9.15.7 Note on inputpoutput signals

In Esterel v7, we currently reject declaring an inputoutput signal `IO` in a main module `Main`, while we did accept it in Esterel v5. In v5, `IO` was output by `Main` at some tick either if it was input from the environment or if it was internally emitted within `Main`. This corresponds to the wiring pictured in Figure **??**. This behavior turned out not really useful in practice, and this is why we rejected it for Esterel v7. Furthermore, inputoutput signals do exist in HDLs and chip design, but with a quite different semantics and behavior. They are generally tri-state objects with implementation-dependent features, not straigthforward RTL objects.

An inputoutput `IO` in a submodule `Sub` bears a different intention: it is meant to be *captured* by some master signal `S` specified in the "`run Sub`" statement. This is fully consistent with Esterel semantics. No tri-state object is required for the implementation, which simply works as specified in Figure **??**. Modular Esterel v7 compiling can also be performed by declaring the appropriate status / data interface between the master module `M` and the submodule HDL function.

Chapter 10

Oracles

This chapter deals with oracles, which are special hidden input signals used to model non-deterministic behavior.

Warning: oracles are experimental in Esterel v7 20. They are only supported by verification, for experiment purposes. Simulation is not yet available for them.

Oracles are special signals assumed not to be on the user control. They are declared with a special signal declaration. They can be valued, but only the interface features `temp` and `value` can be used for them. The modules features `reg`, `init`, and `combine` are not available.

The status and value of an oracle are supposed to be freely determined by the environment in any execution. Oracles are meant to help modeling non-determinism. Here is how to write a choice between two statements p and q:

```
oracle Oracle in
  if Oracle then
    p
  else
    q
  end if
end oracle
```

Oracles are broadcast as any other signals. Here is a way to perform a coordinated choice in two branches of a parallel:

```
oracle Oracle in
  ...
  if Oracle then
    p1
  else
    q1
  end if
  ...
||
  ...
  if Oracle then
    p2
  else
    q2
  end if
```

```
    ...
  end oracle
```

If the two concurrent `if` statements are executed simultaneously, they both take their `then` branch or their `else` branch in a coordinated way since they test the same oracle.

Coordinated choice can be very useful in modeling non-deterministic applications and is usually not available in asynchronous concurrent formalisms.

Valued oracles return any value in the type:

```
  oracle Oracle : integer in
    emit ?O <= ?Oracle
  end oracle
```

The value of `O` can be any integer. The way the integer is chosen is left to the implementation.

Beware: one cannot choose a "random" integer, because such a thing cannot exist! Randomness makes sense only when a probability distribution is chosen, and no distribution exists for all integers. However, in makes perfect sense to choose randomly a bounded signed or unsigned integer.

It is also possible to explicitly emit an oracle using ususal `emit` or `sustain` statements, to explicitly gain control over the oracle if needed.

Chapter 11

Macro-expansion of Derived Constructs

11.1 Expansion of signal constructs

11.1.1 Expansion of pre

For a pure signal `S`, one can encode `pre(S)` using an auxiliary signal `preS`. Assume for instance that `S` is declared in a block such as

```
signal S in
  p
end signal
```

Then, call $p0$ the statement p where `pre(S)` is replaced by `preS`. The following program has the same behavior:

```
signal S, preS in
  trap Done in
    p0;
    exit Done
  ||
    loop
      if S then
        pause;
        emit preS
      else
        pause
      end if
    end loop
  end trap
end signal
```

For a valued signal, one uses an auxiliary variable to store the value between the instants:

```
signal S : T , preS : T in
  trap Done in
    p0;
    exit Done
  ||
    var X : T in
      loop
        if S then
          pause
        else
          X := ?S;
          pause;
          emit preS(X)
        end if
      end loop
    end var
  end trap
end signal
```

11.1.2 Expansion of next

One can code a registered signal `S` using a standard signal `nextS` and the `pre` operator:

- One replaces "`emit next S`" by "`emit nextS`" and "`emit next ?S <= exp`" by "`emit ?nextS <= exp`".
- One replaces "`if next(S)`" by "`if nextS`" and '`next(?S)`' by '`?nextS`'.
- One replaces "`if S`" by "`if pre(nextS)`" and '`?S`' by '`pre(?nextS)`'.
- For an output registered signal, one must also connect `pre(nextS)` instead of `S` to the output interface, see Chapter `Chapter:Run`.

11.2 Expansion of delays

Here is how delays described in Section **??** are expanded. Consider the following statement:

```
abort p when c times e
```

The expansion is as follows:

```
trap T in
  var C := c : integer in
    loop
      await e;
      C := C-1;
      if C <= 0 then
        exit T
      end if
    end loop
  end var
end trap
```

Chapter 12

Run-time errors

Run-time errors are errors that make the program work incorrectly. In software or hardware generated code, they can create erroneous behaviors. In software, they can also provoke crashes. Therefore, Esterel v7 implementations should make every effort to help the user detecting possible run-time errors before embedding the program. This can be done by using good simulation tools, static analyzers, or verification systems.

12.1 Data-related errors

12.1.1 Unsigned subtraction error

There is an unsigned subtraction run-time error if the second argument of an unsigned '`-`' operator is greater than the first one, see Section **??**.

12.1.2 Unsigned division error

There is an unsigned division run-time error if the second argument of an unsigned '`/`' operator evalulates to 0, see Section **??**.

12.1.3 Unsigned modulo error

There is an unsigned modulo run-time error if the second argument of an unsigned `mod` operator evalulates to 0, see Section **??**.

12.1.4 Unsigned out of range error

A run-time occurs when a variable or a signal is assigned a value outside the range of an unsigned type.

This error is raised by an explicit or implicit assertion about the required size. Explicit assertions are calls to the `assert<`M`>` function, see Section **??**. An implicit assertion is generated by an assignment to an unsigned variable, a call to a function or procedure with an unsigned argument, an emission of an unsigned signal, and an array indexation.

Note that run-time implicit assertions are necessary only if the type of the expression is bigger than the type of the objet the result is used for. Otherwise, type-checking ensures correctness. For instance, let `A[`M`]` be an array and `S` be a signal of type `unsigned<`N`>`. The indexation `A[?S]` is guaranteed correct at type-checking if $N \leq M$ and it requires a run-time implicit assertion otherwise.

12.1.5 Signed division error

There is a signed division run-time error if the second argument of a signed '`/`' operator evalulates to 0, see Section **??**.

12.1.6 Signed out of range error

A run-time occurs when the argument of an `assert_unsigned` function is not in the required unsigned range, see Section **??**, or when a variable or a signal of a signed type is assigned a value outside the range of the type. As for the unsigned case of Section **??**, this can occur because of an explicit call to `assert<`M`>`, see Section **??**, or because of an implicit assertion. An implicit assertion is generated by an assignment to a signed variable, a function or procedure call with a signed argument, or an emission of a signed signal.

As for unsigned, there is no need for an implicit run-time assertion if the operation can be shown safe at type-checking time.

12.1.7 Bitvector onehot2u conversion error

A run-time error occurs when a bitvector used as argument of `onehot2u` has either zero or more than one 1-component, as for `'b000` or `'b101`. In that case, the bitvector does not represent a 1-hot encoded unsigned number.

12.1.8 Unsigned-to-enum conversion error

A run-time error occurs when an unsigned number used as argument of `u2enum<E>` is the code of no enum constant in *E*. In that case, the number cannot be converted to an enum constant of *E*.

12.1.9 Signed-to-enum conversion error

A run-time error occurs when a signed number used as argument of `s2enum<E>` is the code of no enum constant in *E*. In that case, the number cannot be converted to an enum constant of *E*.

12.2 Signal-related errors

12.2.1 Access to uninitialized signal value error

For each incarnation of an uninitialized valued signal `S`, the following operations are run-time errors:

- If `S` is standard valued, reading the signal's value '`?S`' before the first instant where `S` is emitted.

- If `S` is registered valued, reading the next signal's value '`next(?S)`' before the first instant where a `next` emission of `S` is performed.

- If `S` is standard valued, reading the previous signal's value '`pre(?S)`' before the instant immediately following the one where `S` is emitted for the first time.

- If `S` is registered valued, reading the signal's value '`?S`' before the instant immediately following the one where a **`next`** emission of `S` is performed for the first time.

12.2.2 Bad access to temporary signal value error

It is a run-time error to read the value of a temporary signal if this signal has not been emitted nor received from the environment of from a submodule in the instant.

12.2.3 Multiple emission of single signal error

It is a run-time error to perform two emissions of the same incarnation of a single valued signal. The signal must be declared combined to support multiple emissions, see Section **??**.

12.2.4 Bad access to a signal array error

It is a run-time error to access a signal array position out of bounds. This error is a particular case of the unsigned out of range error of Section **??**.

12.3 Variable-related errors

Remember that a variable is assigned to by an assignment statement or a procedure call where the variable is passed by reference. For arrays, each position is considered independent of the other ones.

12.3.1 Access to uninitialized variable error

At each instant, it is a run-time error to read the value of an uninitialized variable before this variable has been assigned a value for the first time in the program execution.

12.3.2 Illegal access to temporary variable error

At each instant, it is a run-time error to read the value of a temporary variable before this variable has been assigned a value for the first time in the instant. This error cannot occur if the variable is initialized, since the initialization if performed at each instant.

12.3.3 Bad access to a variable array error

It is a run-time error to access a variable array position out of bounds. This error is a particular case of the unsigned out of range error of Section **??**.

12.3.4 Sharing a variable error

It is a run-time error to be in a situation where a variable can be read / write shared between two concurrent control threads: one thread can execute an action that reads the variables, the other threads can execute an action that writes the variable, and the actions are not dynamicaly ordered by a causality path, see Section **??**.

Notice that shared variables may be hard to detect. Implementations may put stronger static requirements to ensure non-sharing at compile-time. What is forbidden by the language definition is only to share variables at run-time.

Chapter 13

Introduction to multiclock Esterel design

This chapter informally presents multiclock design with Esterel. Multiclock design is strongly related to hardware issues. Therefore, in this chapter, we shall use a much more hardwarish vocabulary than in the previous chapters. Multiclock design is much trickier than single-clock design and has be much less studied in the previous Esterel literature. Therefore, we also present many more examples than before. We refer to the implicitly single-clocked Esterel language presented in the previous chapters as Classic Esterel.

In Section **??**, we first recall how one can generate single-clocked Register Transfer Level (RTL) circuits from Classic Esterel programs. We discuss the simple relation between timing closure issues and the synchrony hypothesis.

In Section **??**, we discuss the need for multiclock circuits in modern design and we present the Globally Asynchronous Locally Synchronous (GALS) model on which multiclock Esterel is based. We briefly discuss timing closure and metastability issues.

We informally present multiclock Esterel in Section **??** and discuss examples. We discuss how to simulate multiclock designs using `weak suspend` in classic Esterel, and how to synthesize multiclock circuits or single-cloked emulation from them.

13.1 RTL design

The simplest circuit design paradigm is single-clocked Register Transfer Level or RTL design. Circuits are composed of wires and gates. Gates can be combinational or sequential. Combinational gates continuously compute their output voltage in function of their input voltage. Sequential gates hold state and are called registers. They are driven by clocks, which are electronic devices that generates electrical rising and falling edges. Here, we only consider rising edges as meaningful clock events. A register changes state when a clock rising edge occur: it samples its input voltage and keeps it on its output until the next clock rising edge.

An essential requirement of RTL circuits is *timing closure*. For a state change to be meaningful, the input of a register must be electrically stable when a clock rising edge occurs. Similarly, a circuit primary output must be stable before being used by the circuit external environment. For a given circuit technology and layout, the time it takes to stabilize all register inputs and all primary outputs depends on the longest delay path in the logic gates and wires, which is called the critical path. The critical

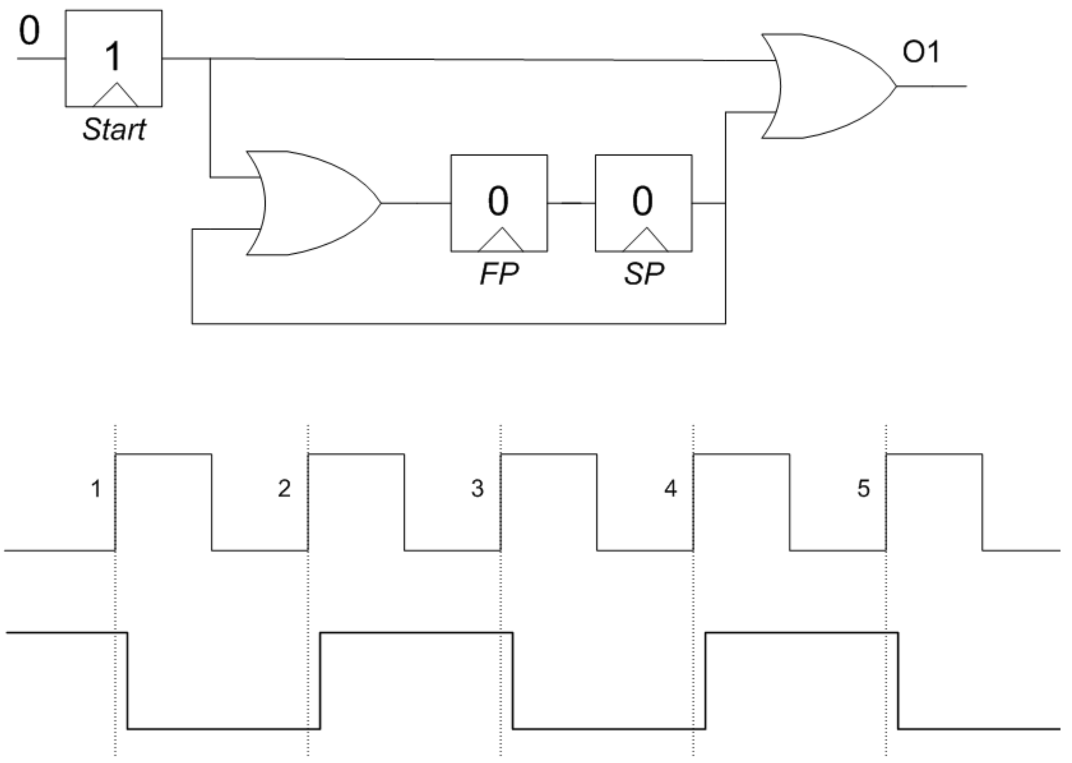


Figure 13.1: Circuit and waveform of M1

delay can be statically computed, and it determines the maximal clock frequency for the technology. Computing that frequency is performing the timing closure of the circuit. Below that frequency, there is no difference between the electrical behavior once stabilized and the logical behavior computed with delay 0 for all gates, wires, and state transitions. Therefore, the Classic Esterel zero-delay abstract model is adequately implemented by the actual circuit.

Notice that a clock needs not be regular in time: only its rising edge succession matters, provided that any two rising edges are not closer than required by critical path stabilization. For instance, irregular clocks result from shutting down clocks to save power when a circuit part is unused.

13.1.1 Translating Esterel programs into RTL circuits

The translation from Esterel programs to RTL circuits is explained in [?, ?]. We illustrate it with two examples that we shall later reuse in Section **??** to study communication in muticlock designs. Consider first the following inputless module `M1`:

```
module M1 :
  output O1;
  // Start
  loop
    emit O1;
    pause; // FP
    pause; // SP
  end loop
end module
```

The corresponding circuit is pictured in Figure **??**, together with a waveform denoting its abstracted electrical behavior. Circuit registers implement control states. They are named using the labels of the start condition or `pause` statement they implement[1]. Their reset value is the one written inside the register symbol. A value 1 in a control register means that control is currently paused at that point.

The logical behavior is the following sequence of tick executions:

```
tick  in    out
 1 :   - →  O1
 2 :   - →  -
 3 :   - →  O1
 4 :   - →  -
 5 :   - →  O1
 6 :   - →  -
 ...
```

where '`-`' means that all signals are absent.

Circuit execution mimics logical behavior. In Esterel semantics, at tick 1, `M1` starts from label *Start*, executes "`emit O1`", and pauses at `pause` statement *FP*. For the circuit, assume reset active level is high. Then, when reset falls, `O1` is driven high by the *Start* register initial value, until clock rising edge 1. At that edge, the *Start* register changes state to 0 while the *FP* register changes state to 1.

In Esterel semantics, at tick 2, control moves from `pause` labeled *FP* to `pause` labeled *SP*. In the circuit, at clock rising edge 2, register *FP* gets new state 0 while register *SP* gets new state 1. This drives `O1` to 1 until rising edge 3, i.e. during the electrical duration of tick 3. At rising edge 3, register *SP* gets new state 0 while register *SP* gets state 1. Being driven by *SP*, `O1` falls to 0 until rising edge 4 where *SP* gets state 1 again and drives `O1` to 1. Alternation between these two behaviors follows, which means that `O1` is kept high every other clock cycle.

In Figure **??**, waveforms depict the value evolution of signals over continuous time. We use a classical convention: a change in a signal is drawn a little afer the clock edge that provoked it. While logically equivalent to the 0-delay model, this "δ-delay model" simplifies waveform reading. A stabilized signal value at initial Esterel tick 1 can be read anywhere from the waveform start to value change 1, and, in particular, on the vertical dotted line of clock rising edge 1. A signal value at Esterel tick 2 is read anywhere from rising edge 1 excluded to rising edge 2 included. Generally speaking, signal values at Esterel tick n can be safely read on the vertical dotted line that starts from rising edge n, since all values are stable at that time. With a pure 0-delay waveform, a signal change

[1] In practice a better labeling could be `pause@FP`, as explained in Section **??**; we do not use it here for readability reasons.

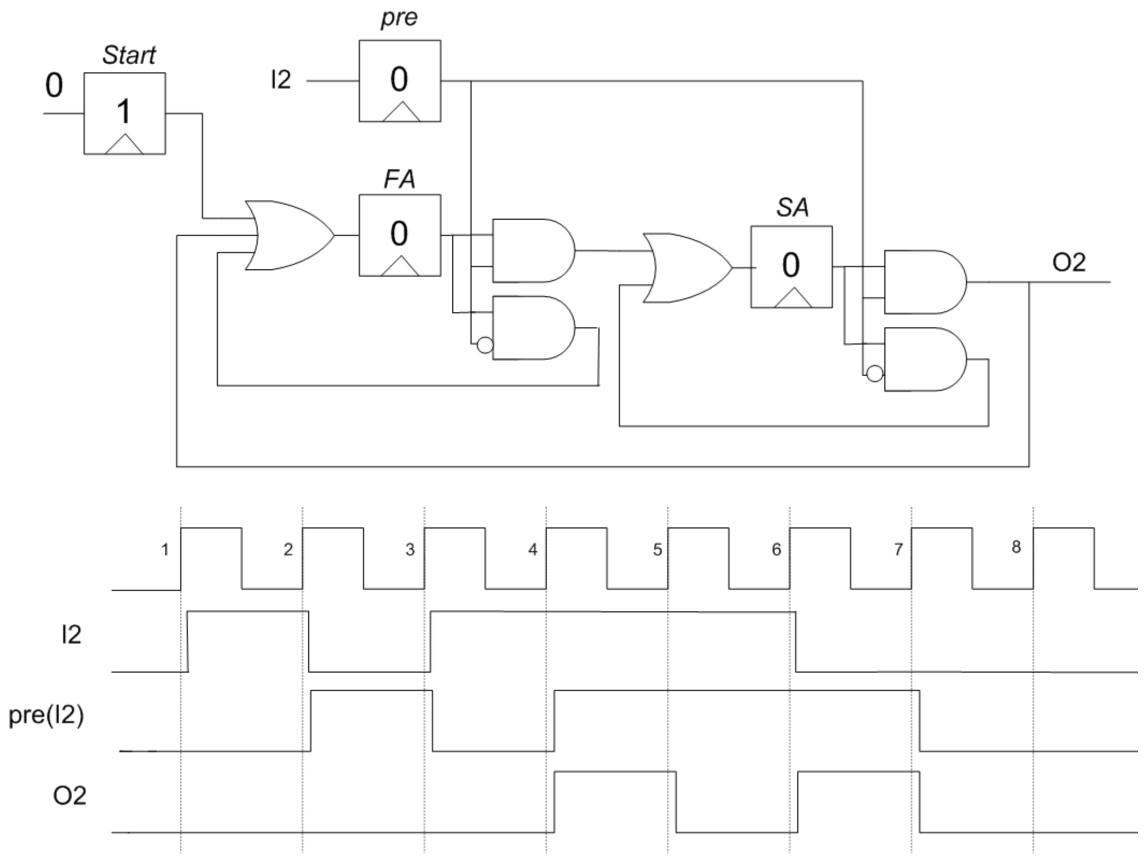


Figure 13.2: Circuit and waveform of M2

would be exactly synchronous with the clock edge, and it would be unclear whether the value to consider is the one right before or the one right after the clock edge.

Consider now the following module M2 with input I2:

```
module M2
  input I2;
  output O2;
  // Start
  loop
    await pre(I2); // FA
    await pre(I2); // SA
    emit O2
  end loop
end module
```

The corresponding circuit is pictured in Figure **??**. The pictured waveform corresponds to the following logical behavior:

```
1 : -   → -
2 : I2  → -
3 : -   → -   // pre(I2)
4 : I2  → -
5 : I2  → O2  // pre(I2)
6 : I2  → -   // pre(I2)
7 : -   → O2  // pre(I2)
```

```
  8 : -    → -
```

The `O2` output is set to 1 at every other edge where `pre(I2)` is 1. In the circuit, the *pre* register adds a signal state to the control states. It takes `I2` as input and returns `pre(I2)` as output. We says that the *pre* register *samples* `I2` at clock rising edge.

- *Tick 1, start to rising edge 1:* at start time, the *Start* register has value 1 and the other registers have value 0. Thus, `O2` has value 0.
- *Tick 2: rising edge 1 to 2,* At rising edge 1, the *FA* register changes state to 1, which corresponds to the fact that control in tick 2 is driven by the *FA* `pause` statement. The *Start* register state becomes 0, and the *SA* register state stays 0. The *pre* register state keeps value 0 since since `I2` is 0 at rising edge 1. The output `O2` is 0 since *SA* is 0.
- *Tick 3, rising edge 2 to 3:* At rising edge 2, since the current value of *pre* is 0, control stays at *FA* and control register states are unchanged. The *pre* register state becomes 1 since `I2` is 1. The `O2` output is still 0 since *SA* is 0.
- *Tick 4, rising edge 3 to 4:* At rising edge 3, since the upper and lower and-gate generated by the first `await` statement have respective value 1 and 0, *FA* becomes 0 and *SA* becomes 1. The *pre* register becomes 0 and drives the `O2` output to 0.
- *Tick 5, rising edge 4 to 5:* At rising edge 4, *FA* stays 0 and *SA* stays 1 because *pre* is 0. The *pre* register becomes 1. Since *SA* and `pre` are both 1, the `O2` output is driven to 1 during the clock period.
- *Tick 6, rising edge 5 to 6:* At rising edge 5, since *pre* is 1, the *FA* register becomes 1 and the *SA* register becomes 0, which corresponds to looping the loop. Thus, `O2` falls to 0. The *pre* register stays 1.
- *Tick 7, rising edge 6 to 7:* At rising edge 6, control moves from *FA* to *SA* because of the leftmost upper and-gate; the *pre* state stays 1 since `I2` is 1. The output `O2` is 1 since *SA* and *pre* are 1.
- *Tick 8, rising edge 7 to 8:* At rising edge 7, control moves back to *SA* since *FA* and *pre* are both 1. The *pre* register state becomes 0, which drives `O2` to 0.

In the waveform of Figure **??**, the `I2` input is supposed to be set exactly at state change and to be kept constant during the whole cycle. It is called an *early input* Because of sampling by the *pre* register, requiring an early input is actually not necessary: since sampling of `I2` occurs right at the clock cycle, only `I2`'s value at that point matters. Figure **??** shows that an indentical behavior would be obtained with a more aggressive `I2` waveform, which `I2` is late but polite enough to have stabilized to the right value just before clock rising edges occur. This intinsic tolerance of sampling w.r.t. late inputs will be fundamental for multiclock design.

13.2 The need for multiple clocks

Modern hardware designs usually have more than one clock and, in addition, can turn clocks on and off. There are several reasons for this. First, it is difficult to distribute a

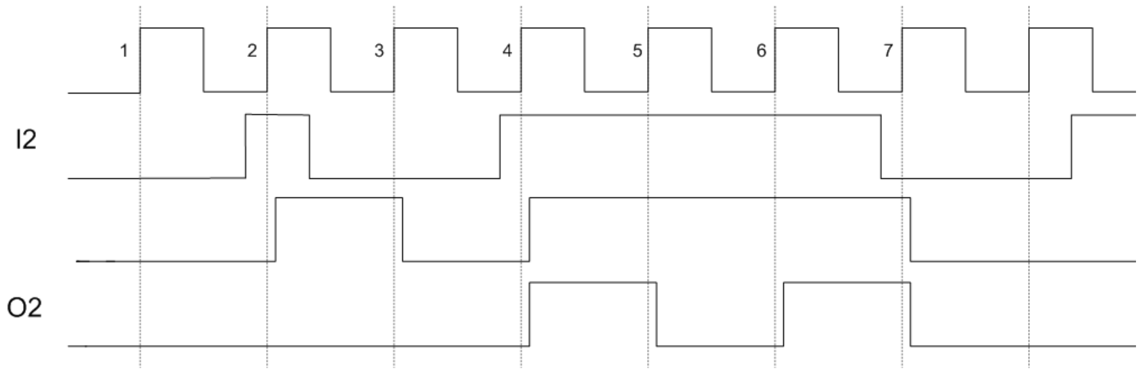


Figure 13.3: waveform of M2 with impolite input

single clock on a large chip, because of clock skew and power dissipation. Second, Systems on Chip (SoC) designs are usually composed of several IPs (Intellectual Property) modules of different nature, which must obey different clocking requirements. Third, not all IPs in a circuit are active at a given time, and it is essential to shut down the clock in inactive parts of circuits to save power. Finally, inputs and outputs of a circuit may require being clocked according to external protocols independently of how fast the circuit computes. For instance, a video filter may use an internal clock to compute images while being forced to obey a standardized video frame transmission protocol for I/O driven by an unrelated clock.

13.2.1 Metastability issues

Mutliple clocks can be generated either by a single PLL (Phase-Locked Loops) or by several distinct PLLs. In the second case, clocks are fully asynchronous, i.e. unrelated in time, and there is a danger of *metastability*. Assume that a signal driven by an emitter clock is the input of a register clocked by an asynchronous receiver clock. Then, if a change in the signal occurs too close from a receiver clock rising edge, the register can take an arbitrary non-Boolean metastable state for an arbitrary amount of time. In practice, noise makes the state randomly stabilize to 0 or 1 after some time, but the stabilized value may not be the intended one and stabilization time is not guaranteed. Therefore, metastability cannot be controlled in a deterministic way.

Notice that getting some bits wrong may be acceptable in some applications: a wrong bit in a large picture is usually not a problem, because there are many other physical reasons to generate wrong bits in a photo or video chain. But wrong bits can be unacceptable in more sensitive design, and in particular in control applications. Then, one must either avoid metastability or deal with it. This can be done in two fundamentally different ways:

- Relying over *absolute timing* to avoid metastability by guaranteeing that signals never change when clock rising edges occurs. This is only possible if there are guaranteed relationships between clocks and clock phase drifts, i.e. if the clock front positions remain comparable over time, and if propagation delays in the actual circuit are well-mastered. We call this design style *quasi-synchronous design* below. Back-end techniques to achieve quasi-synchronous design are outside of the scope of this paper.

- Relying over specific communication devices called *synchronizers* that cope with metastability, possibly associated with handshake and error-correction protocols.

Quasi-synchronous design is direct in multiclock Esterel. Synchronizers are programmable in Esterel, thus they do not need to be language primitives. There is a fairly wide variety of them, see for instance [**?**]. Here are some:

- Simple synchronizers use a register or a small chain of registers clocked by the receiver clock to reduce the probability of metastability for the last register; their output is electrically stable with very high probability but can be wrong.

- Multiple synchronizers transmit the same bits several times, serially or in parallel, and check consistency to detect errors due to metastability issues;

- Handshake synchronizers implement handshake feedback to acknowledge bit reception. They are often necessary if the clock frequencies are different enough.

- Dual-clock fifos are filled on a writer clock and read on a reader clock. They are classical tools to maintain throughput in streaming-based systems.

Synchronizers are mostly used for truly asynchronous clocks. However, they can also be used in cases where all clocks are generated by a single PLL and thus related in time, but where one does not want to predict exactly how relative clock rising edges are arranged because that depends on complex routing issues. In this case, one sticks to the worst case, assuming the clocks unrelated as for the multiple PLL case.

13.2.2 Some multiclock design terminology

Unfortunately, there is no well-established and clear terminology in current multi-clock design. Logic and electrical concerns are often mixed up. Since we mostly deal with the logical level in Esterel, we propose a simple terminology sufficient for our purpose. A more elaborate one can be found in [**?**].

Pure synchronous design

This is what the classic Esterel v7 language is about. There is a single clock. All register inputs are sampled synchronously by the rising edges of this clock, and, in the actual circuit, all signals are supposed to be stable when the rising edge occurs.

Clock-gated synchronous design

In synchronous designs, the clock can be gated to avoid sampling register inputs, this either to save power or for purely behavioral reasons. Conceptually, there is no model change w.r.t. pure synchronous design, only implementation improvements. The very same design can be implemented without clock gating, using logic to retain the register value instead of disabling the change. The Esterel `suspend` and `weak suspend` statements presented in Section **??** are the Esterel way to deal with clock gating.

Quasi-synchronous design

In *quasi-synchronous design*, a multiclock design deals with several clocks whose rising edges have precise relations over time. Generally speaking, they come from a single PLL by clock division or clock gating, but how they are built is inessential here. We distinguish between three subcategories listed below (they may be other):

- *Derived clocks:* given a clock C, a clock C' is derived from C if all rising edges of C' are rising edges of C, see Figure **??**. In practice, C' is essentially obtained from C by clock gating. Derived clocks are used to slow down clocks or to shut them down to save power.
- *Harmonic clocks:* two clocks are said to be harmonic if they have a fixed frequency and have simultaneous rising edges at some point in time. Then their coincidence point repeats regularly over time. Figure **??** shows the case of a ratio 3:4, where the first clock beats 3 times while the second clock beats 4 times. For harmonic clocks, the distance between any two clocks rising edges is precisely predictable. Harmonic clocks are common in designs where busses or memories have several frequencies values that are multiple of a common base frequency.
- *Phase-shifted clocks:* two clocks beat at the same frequency but one is shifted by some delay w.r.t. the other. In this way, the delay between rising edges is predictable.
- *Phase-shifted harmonic clocks* combine harmony and phase-shifting, as shown in Figure **??**. Timing relations become slightly more complex, but are still fully predictable.

Of course, other cases are possible. We speak of quasi-synchronous design for any case where timing constraints are mastered enough to avoid the need for resynchronizers to communicate between clock domains.

When the clocks are truly unrelated, i.e. generated by distinct analog devices, we say that they are fully asynchronous. Notice that designing with asynchronous clocks is different from what is usually called asynchronous designs. This name is used for something totally different, i.e. designs that uses no clock at all [**?**, **?**].

13.2.3 Globally Asynchronous Locally Synchronous (GALS) design

For multiclock design, the freedom for complex relative positioning of clock edges leads to much larger state spaces and combinatorial issues than for single-clock design. Therefore, multiclock design is intrinsically much more complex. To keep things manageable, one often adopts the *Globally Asynchronous Locally Synchronous* (GALS) design paradigm: the design is split into single-clocked purely synchronous islands that only communicate through wires and synchronizers. In this way, sequential computations are only performed by single-clocked modules and communication between clock domains is clearly identified. A signal that goes from a clock domain to another clock domain is called a *clock-domain crossing* signal or *cdc* signal.

13.3 Multiclock design in Esterel

For Multiclock Esterel, we adopt the GALS design paradigm together with the usual zero-delay model for combinational computation.

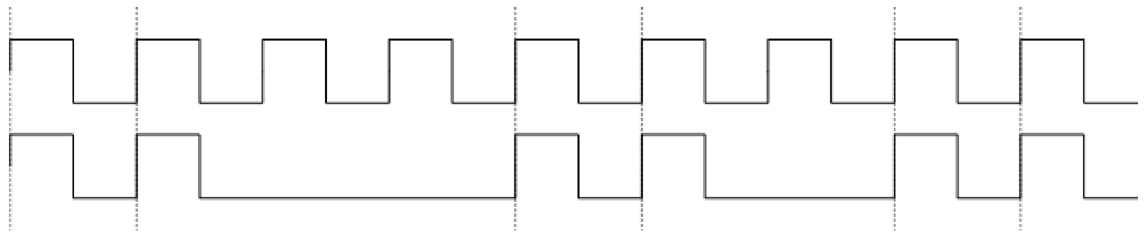


Figure 13.4: derived clock

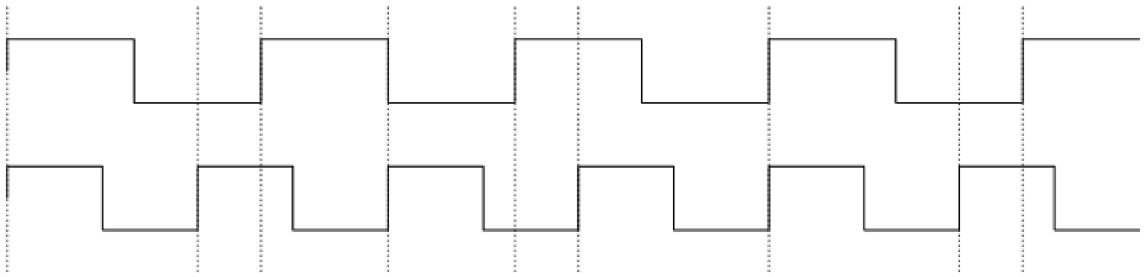


Figure 13.5: harmonic clocks

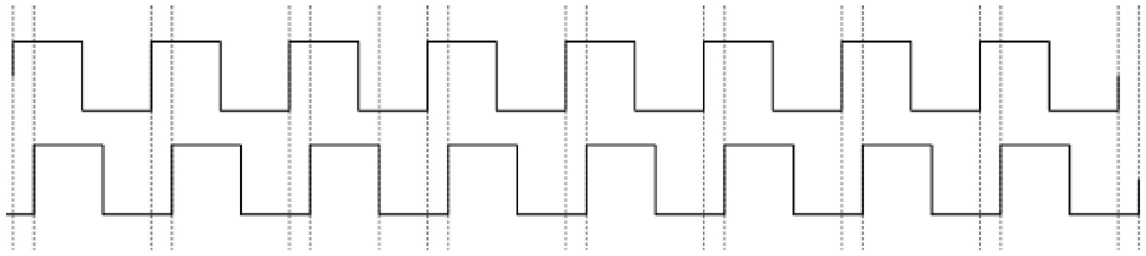


Figure 13.6: phase-shifted clocks

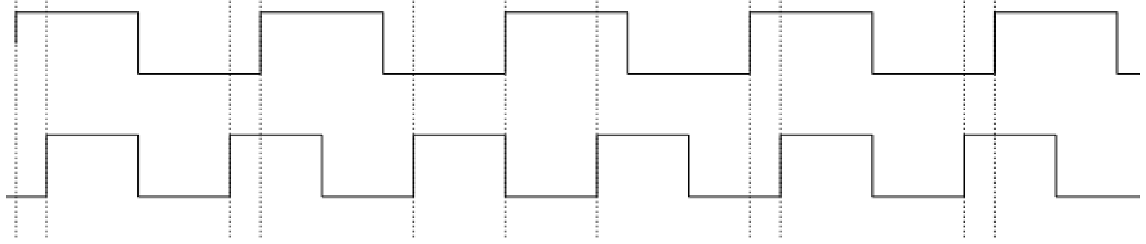


Figure 13.7: phase-shifted harmonic clocks (falling edge coincidence is meaningless since only rising edge smatter)

13.3.1 Overview

We introduce a new kind of signal called a *clock*, declared using the `clock` keyword. Clocks can only be used to clock registers in classic modules. No Boolean gates are available for them, and they cannot be declared `reg`. New clocks can be derived from existing ones by *clock gating* or *clock muxing*, using a specific `clock` definition statement.

We add a new multiclock unit declared using the `multiclock` keyword. A multiclock unit models a GALS system. It has a header similar to that of a module. In addition, it can declare input and output clocks. The multiclock body can declare local signals and clocks. It is composed of concurrent elements that can be as follows:

- A classic module clocked by an explicitly given clock, which can be an input clock or a derived clock.
- A combinational signal computation, which is assumed zero-delay and thus independent from any clock.
- Recursively, another multiclock unit instantiated with appropriate bindings for data clocks and signals.
- A clock gater used to derive an output or local clock from another clock and a signal, by masking source clock rising edges when the signal is false.
- A clock multiplexer that builds a clock from two other clocks using a signal to select between them.

Only multiclock units can deal with clocks, while only module units can perform computations, thus achieving the GALS separation of concerns.

Multiclock units can declare local signals and clocks. They are data-generic exactly in the same way as module units.

Here is an example of a multiclock design based on `M1` and `M2` presented in Section ??. Each module is driven by a different clock and their intputs and outputs are connected by the multiclock unit `MC`:

```
module M1 :
  output O1;
  loop
    emit O1;
    pause;
    pause;
  end loop
end module


module M2 :
  input I2;
  output O2;
  loop
    await pre(I2);
    await pre(I2);
    emit O2
  end loop
end module
```

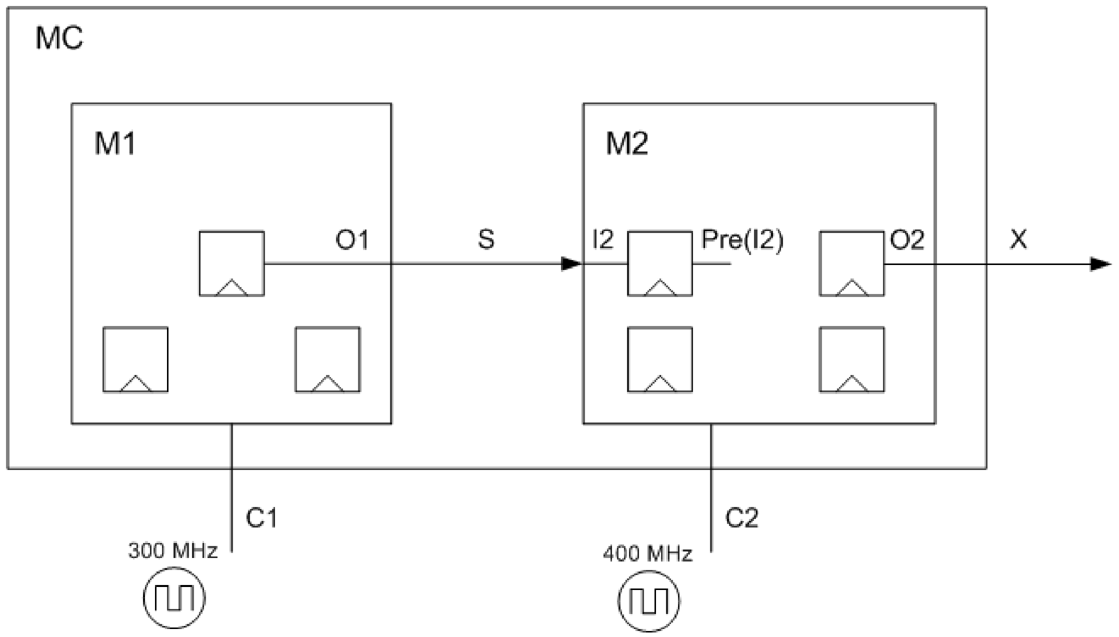


Figure 13.8: The MC multiclock structure

```
multiclock MC :
  input {C1, C2} : clock;
  output X;
  signal S in
    run M1 [clock C1; S / O1]
  ||
    run M2 [clock C2; S / I2, X / O2]
  end signal
end multiclock
```

Here, `S` is a clock-domain crossing signal since it goes from `M1` clocked by `C1` to `M2` clocked by `C2`.

The `MC` multiclock unit structure is pictured in Figure **??**. The local signal `S` is connected to the output `O1` of `M1` and to the input `I2` of `O2`. Connection is direct, which means that `O2`, `S`, and `I2` have exactly the same value over time.

Semantically speaking, multiclock units deal with signals by broadcasting them in zero delay, exactly as for modules. Although emission is zero-delay, we must deal with a richer time model to explain how signals are received by other modules. An emitted signal `S` is kept high for a full clock period of their master clock `C`, so that this signal can be sampled by modules functioning on other clocks at any time between two successive rising edges of `C`. In other words, we must promote the continuous time model used for waveforms in Section **??**.

Multiclock behavior of MC

Even for such a simple multiclock design, behavior needs to be studied with great care. While individual behaviors of `M1` and `M2` are quite simple, their multiclock combination is not.

In Figure ??, we picture the behavior of MC for the case of 3:4 harmonic clocks described in Section ??. In Figure ??, we picture the behavior of MC for the case of 3:4 phase-shifted harmonic clocks. The clocks do not have a half-cycle duty period and their falling edges shortly follow their rising edge; this is no issue since only the position of rising edges matters. In the waveforms, observe that O1 is emitted every other rising edge of C1, while O2 is emitted every other time I2 is true when sampled on O2 rising edges. An emitted signal is kept high for the whole clock cycle of the module that emits it.

Notice that a slight phase shift makes the initial single O2 event of the harmonic case disappear, all O2 events now going by pairs. With these two clocking schemes, timing closure is quite simple to achieve and there is no real risk of metastability: rising edges of C1 and C2 are either simultaneous, which is no problem, or separated by a well-determined minimal amount of time which must simply be bigger than the register output setup time[2].

With less constraints clocking schemes, there may be a risk of metastability for the pre(I2) register if O1 is changing during the rising edge of C2. In that case, one must use a more clever synchronizing scheme discussed below.

13.3.2 The importance of input sampling

In MC, the intermediate signals O1 = S = I2 and X = O2 are well-clocked: O1 is clocked by C1, i.e. remains low or high for full clock cycles of C1, while O2 is clocked by C2, i.e. remains low or high for full clock cycles of C2. For O2, this follows from the use of pre(I2) to sample I2 = O1 on rising edges of C2, which acheives reclocking of O1 from C1 to C2. In circuit terms, pre(I2) inserts a sampling register on the input line, as pictured in Figure ??. Notice that pre(I2) performs reclocking on C2 because it is written in the body of M2 which is run with clock C2. No explicit mention of the actual clock is needed for the pre operator which is always clocked by its module's implicit clock.

It is interesting to see what happens if I2 is directly used in M2 instead of the sampled pre(I2), as in the following code:

```
module M2_no_sampling :
  input I2;
  output O2;
  loop
    await I2;
    await I2;
    emit O2
  end loop
end module
```

The new behaviors are presented in Figure ??. The output X is not well-clocked any more and becomes much wilder: X = O2 is the conjunction of O1, clocked by C1, and of the output of the SA register, clocked by C2. The combinatorial complexity of the relative placements of C1 and C2 rising edges shows in full force, with a large difference between the harmonic and phase-shifted harmonic clocking cases. Futhermore, the behavior shown in Figure ?? is given in the logical zero-delay framework. Things are much worse in the real world because of electrical delays in the combinational logic. Using X as an output of another design would becomes really problematic.

[2] Ignoring more subtle timing issues that are out of the scope of this manual and well-handled by synthesis systems.

Using samplers is a good design principle that makes signals well-clocked and modules nicely composable. However, this principle is not enforced by the Esterel language.

13.3.3 Synchronizers

Input sampling is so important that it is usually not done by the receiver module M2 but by isolated specific modules called *synchronizers*. Here is the way in which one would really write MC in practice, here with a simple synchronizer made of one single register:

```
module M1 : // as before

module Synchronizer :
  input I;
  output O : reg;
  sustain next O <= I
end module

module M3 :
  input I2;  // assumes input already reclocked
  output O2;
  loop
    await I2;
    await I2;
    emit O2
  end loop
end module

multiclock MCsync :
  input {C1, C2} : clock;
  output X
  signal O1, I2 in
    run M1 [ clock C1 ]
  ||
    run Synchronizer [ clock C2 ; O1 / I, I2 / O ]
  ||
    run M3 [clock C2 ]
  end signal
end multiclock
```

The MCsync multiclock unit structure is pictured in Figure ??. The M3 circuit is exactly as theM2 circuit pictured in Figure ?? except that the register on the I2 input has been removed. The register is now placed in the synchronizer, which is equivalent but often preferred in practice. The new structure is pictured in Figure ??.

Notice that we use a registered output instead of a pre operator for Synchronizer. This is equivalent but more readable since registration is more explicit.

When clocks schemes are less constrained, one usually places two registers in a row in the synchronizer to decrease the probability of I2 being metastable:

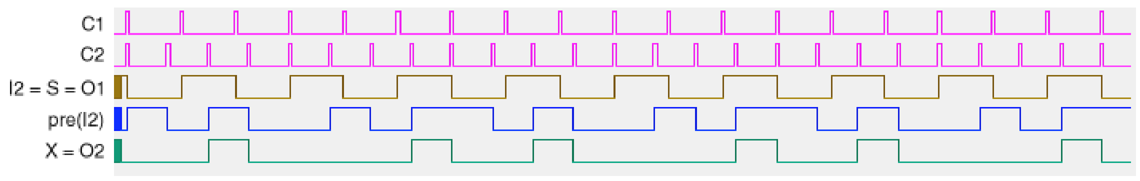


Figure 13.9: behavior of MC with harmonic clocks

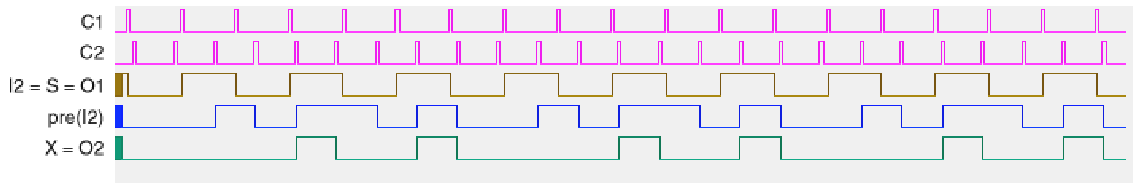


Figure 13.10: behavior of MC with phased-shifted harmonic clocks

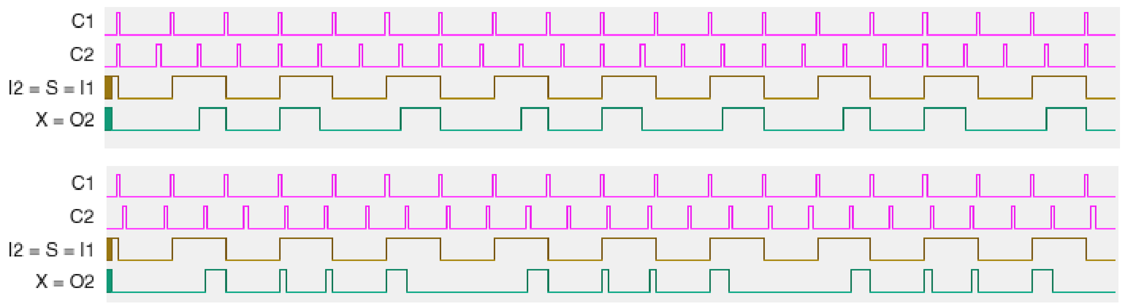


Figure 13.11: behavior of MC without input sampling

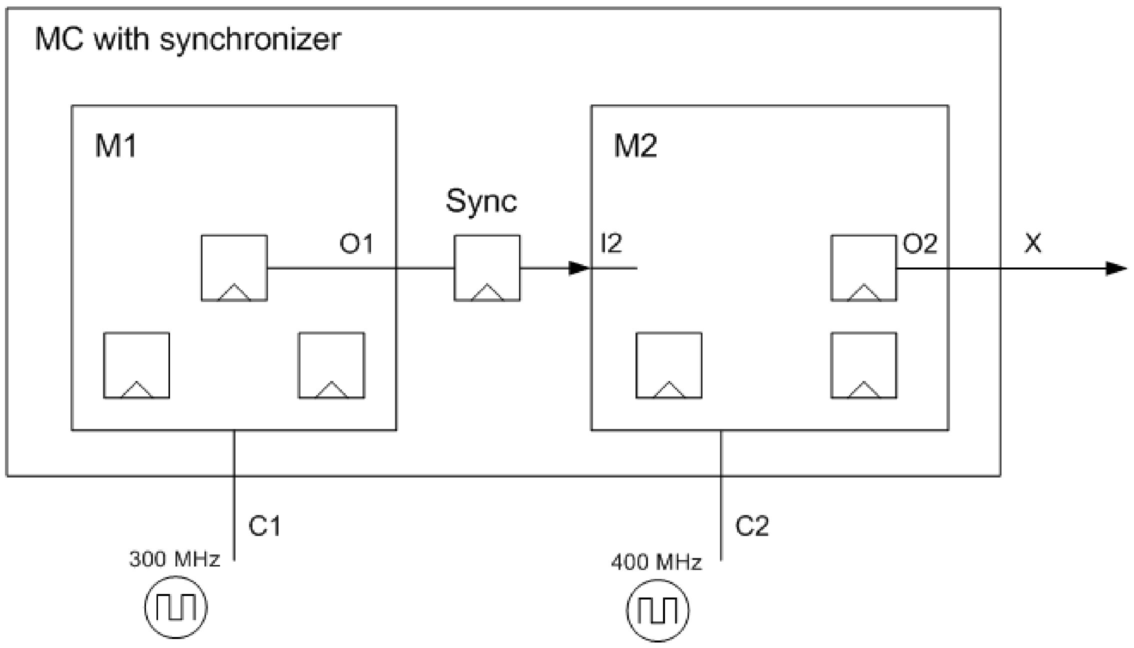


Figure 13.12: MCsync multiclock structure

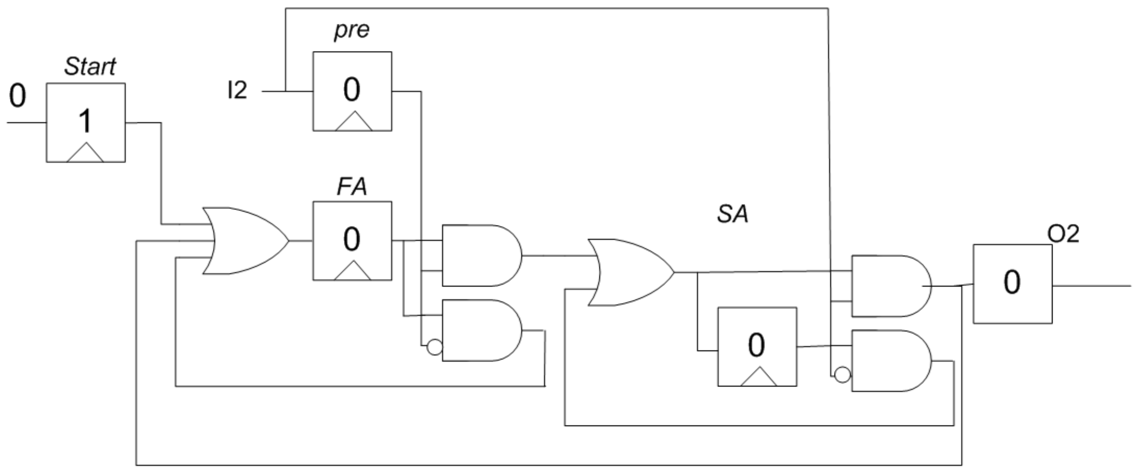


Figure 13.13: peripheral retiming of M2

```
module Synchronizer2 :
  input I;
  output O : reg;
  signal Aux : reg in
    sustain {
      next Aux <= I,
      next O <= Aux
    }
  end signal
end module
```

But this is still quite unsafe, and one may have to resort to more complex *handshake synchronizers*. This is discussed in depth in [?].

Electrically speaking, it is also better to only compose modules whose output signals are all true circuit register outputs to ensure that they are well-clocked and available as early as possible. This is not true for the `M2` circuit pictured in Figure ??. However, an easy *forward retiming* can transform the circuit into the equivalent circuit pictured in Figure ??, where `O2` has become a direct register output. See [?] for details on retiming.

13.3.4 Clock definition

A multiclock unit can derive new clocks from old ones using signals. This is performed by the clock definition mechanism, which has the form of a restricted `sustain` statement using the `clock` keyword. Left-hand-sides are clocks names. Righ-hand sides are clock names followed by `if` conditions that test a single pure signal, or `mux` clock expressions that select a clock according to the value of a single pure signal. Clock derivation is very useful to shut down parts of a design to save power:

```
module Compute :
  output MemoryNeeded;
  ...
end module


multiclock PowerSaver :
  input clock C;
  ...
  signal MemoryNeeded, MemoryClock : clock in
    run Compute [ clock C ] // computC.E. Leiserson and J.B. Saxe. Retiming synchro
es MemoryNeeded
  ||
    clock MemoryClock <= C if MemoryNeeded
  ||
    run Memory [ clock MemoryClock ]
  end signal
end multiclock
```

Complex tests for clock derivation must be placed in separate `sustain` statements, that must be combinational, i.e. involving no `pre` operators. Here is an example of local and output clock derivation using an auxiliary signal:

```
multiclock ClockDerivation :
  input {Cin1, Cin2} : clock;
  output Cout : clock
  input I, J;
  signal Cloc : clock, S, IandJ in
    run M1 [ clock Cin; I / In, S / Out]
  ||
    sustain IandJ <= I and J
  ||
    clock {
      Cout <= mux(IandJ, Cin1, Cin2)
      Cloc <= Cout if S
    }
  ||
    run M2 [clock Cloc]
end multiclock
```

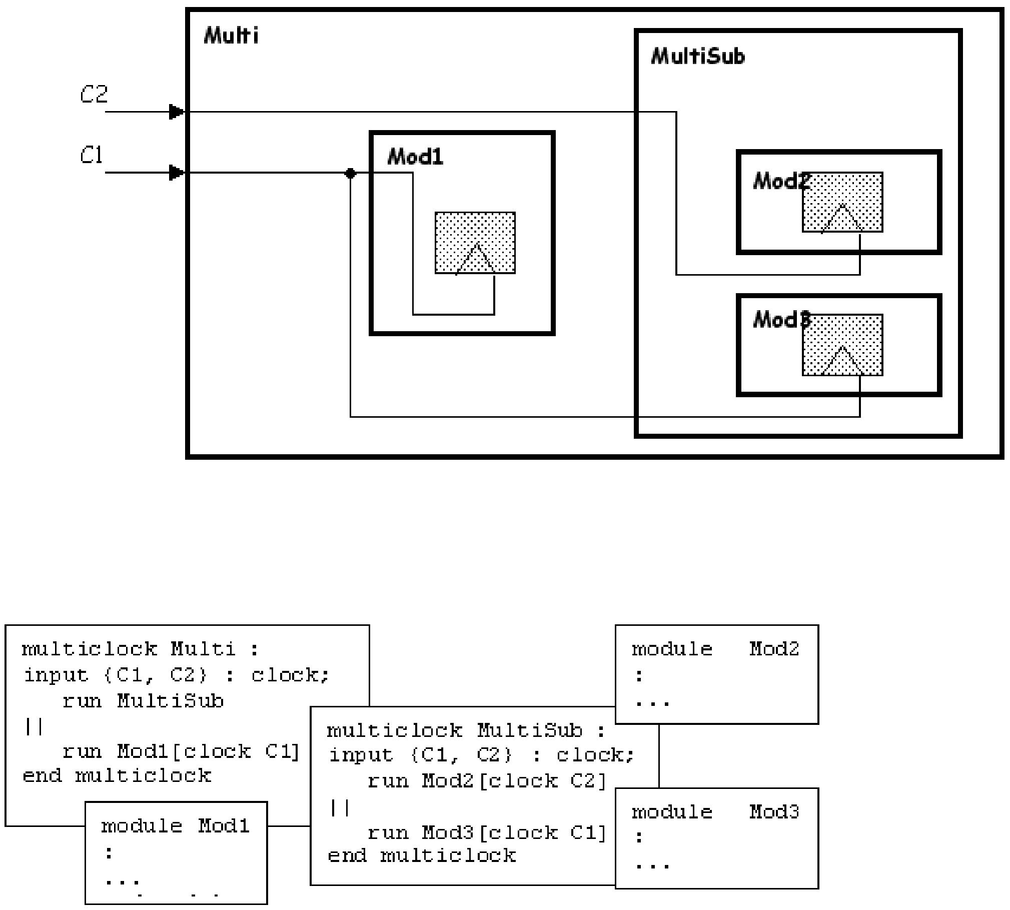
Semantically speaking, `if` and `mux` in `clock` definitions are logical operators specific to clocks. When generating hardware, they are implemented using special cells to ensure clean electrical behavior and avoid clock glitches.

In practice, for true hardware multiclock design, it is not recommended to use signals that are outputs of combinational logic as clock gaters or clock multiplexer drivers. It is much better to use clean signals that are primary inputs of the whole design or direct outputs of registers generated in one of the submodules. Timing closure and glitch avoidance is made easier. However, this is not enforced by the Esterel v7 language.

13.3.5 Hierarchical design

Hierarchical multiclock design is pictured in Figure ??, without mention of communication signals to make things simpler. The main multiclock unit `Multi` inputs two clocks `C1` and `C2`. It invokes another multiclock unit `MultiSub`, passing the clocks to it, and it runs a classic module `Mod1` with clock `C1`. The multiclock unit `MultiSub` runs a module `Mod2` driven by clock `C2` and a module `Mod3` driven by clock `C1`. All the registers of `Mod1` and `Mod3` are clocked by `C1`, while all the registers of `Mod2` are clocked by `C2`. The multiclock units `Multi` and `MultiSub` generate no other registers than those of their leaf modules.

Signal renaming is as for modules. Clock renaming is available for multiclock units:



```
multiclock Multi :
input {C1, C2} : clock;
   run MultiSub
||
   run Mod1[clock C1]
end multiclock
```

```
module Mod1
:
...
```

```
multiclock MultiSub :
input {C1, C2} : clock;
   run Mod2[clock C2]
||
   run Mod3[clock C1]
end multiclock
```

```
module   Mod2
:
...
```

```
module   Mod3
:
...
```

Figure 13.14: hierarchical multiclock design

```
multiclock Multi2 :
  input C1 : Clock;
  ...
  signal C2 : clock in
     run Multi [ C1 / Cin, C2 / Cout, ...]
    ...
  end signal
end multiclock
```

It is a simple name substitution operation, akin to signal renaming. Clock renamings appear within the signal renaming list. This is very different from running a module with a given clock, which determines how the module state changes are performed; this is why the `clock` keyword does not appear in the renaming. It is impossible to rename a clock for a classic module, which has no knowledge of the existence of clocks, and it is impossible to "'clock" a multiclock unit, which has no executable sequential actions by itself.

13.4 Semantics of multiclock designs in Classic Esterel

The new multiclock language requires no fancy new semantical tool for its semantics to be defined. A multiclock design can be easily translated into a single-clocked one, and that is enough to define its semantics.

13.4.1 Replacing clocking by weak suspension

The idea is to view clocks events as normal signal status presence events. For this, one introduces a new simulation Classic Esterel toplevel module associating input signals with input clocks. The tick of this module is used as a fictitious simulation tick, fast enough to observe all events in the multiclock design. Remember that the role of clocks is to tell when states should change. The same can be done with Classic Esterel signals using the `weak suspend` statement described in Section **??**. Here is how to translate the `MC` design of Section **??**:

```
module MC_sim :
  input C1_sig, C2_sig; //clock signals
  output X;
  signal S in
    weak suspend
      run M1 [S / O1]
    when immediate not C1_sig
  ||
    weak suspend
      run M2 [ S / I2, X / O2]
    when immediate not C2_sig
  end signal
end multiclock
```

It is easy to check that the behavior of `MC_sim` exactly mimics that of `MC`. This simple macro-expansion process is enough to define the semantics of multiclock designs. It is automatically performed by the Esterel compiler when needed.

13.4.2 The mcrun statement

To avoid the manual insertion of `weak suspend` statements, Esterel provides the user with a new `mcrun` statement that runs a multiclock unit within a module, passing signals to clocks. Using `mcrun`, the `MC_sim` simulation module of Section **??** is very simply written as follows:

```
module MC_sim :
  input C1_sig, C2_sig; // original clocks changed into signals
  output X;
  mcrun MC [ C1_sig / C1, C2_sig / C2 ]
end multiclock
```

Then `MC_sim` can be run in any Classic Esterel simulator.

Here is a single-clocked execution of `MC_sim` with harmonic clocks simulating the multiclock execution of Figure **??**:

```
 1: C1 C2  → - // O1=I2
 2: -      → - // pre(I2)
 3: -      → - // pre(I2)
 4: C2     → - // pre(I2)
 5: C1     → -
 6: -      → - // O1=I2
 7: C2     → - // O1=I2
 8: -      → X // O1 pre(I2)
 9: C1     → X // O1 pre(I2)
10: C2     → X // pre(I2)
11: -      → - //
12: -      → - //
13: C1 C2  → - //
14: -      → - // O1=I2
15: -      → - // O1=I2
16: C2     → - // O1=I2
17: C1     → - // O1=I2 pre(I2)
18: -      → - // pre(I2)
19: C2     → - // pre(I2)
20: -      → - //
21: C1     → - //
22: C2     → - // O1=I2
23: -      → X // O1=I2 pre(I2)
24: -      → X // O1=I2 pre(I2)
25: C1 C2  → X // O1=I2 pre(I2)
26: -      → -
```

This discrete-time single-clock simulation based on the fictitious simulation tick is equivalent to the continuous-time simulation shown in Figure **??**. The blank events in the above simulation scenario do not provoke output or state change. They could be suppressed, but the reading of clock relative frequencies would be much harder.

The relation between a signal and the clock it simulates is simple: There is a clock rising edge for the clock at any simulation tick where the signal is present. Thus, intuitively, the signal acts as a clock gater on the simulation clock to generate the clock of interest. In hardware, `mcrun` instantiates appropriate clock gaters.

13.4.3 Simulating clock generation

To study a particular clock pattern, it is often much simpler to generate the clocks in Esterel than to write the testbenches by hand. Here is an easy simulation of `MC` for the 3:4 phase-shifted harmonic case:

```
module MC_sim_3_4 :
  output X;
  signal C1_sig, C2_sig in
    loop emit C1_sig each 4 tick
  ||
    pause; // phase-shift
    loop emit C2_sig each 3 tick
  ||
    mcrun MC [ C1_sig / C1, C2_sig / C2 ]
  end signal
end module
```

13.4.4 Simulating metastability issues

Although metastability issues are not handled in Esterel, they can be simulated (at logical level only) for verification purposes. One can for example simulate potential wrong sampling by the first register of a bit synchronizer using an oracle:

```
module LossyBitSynchronizer :
  input I;
  output O;
  signal S in
    oracle Oracle in
      sustain {
        S <= pre(I) xor Oracle,
        O <= pre(S)
      }
    end oracle
  end signal
end module
```

Then, the Esterel verification engine will try all oracle configurations.

13.5 Hardware synthesis from multiclock designs

There are two ways to synthesize hardware from a multiclock Esterel design: true multiclock synthesis and simulated multiclock synthesis.

13.5.1 True multiclock synthesis

True multiclock synthesis generates a multiclock circuit from a multiclock Esterel design. Synthesis consists in individually synthesizing HDL code for each executable module, and synthesizing a global HDL wrapper to pass the clocks and link the signals as specified in the multiclock unit. No register is generated by this wrapper.

Clock gater and clock muxer library modules are needed for clock operations. Synchronizers may require a specific treatment: although they can be programmed as standard

Esterel modules that can be directly synthesized, it can be better for circuit performance to replace them by specific library gates that have better electrical characteristics than the synthesized version. This is easy using modular compilation of Esterel programs.

13.5.2 Simulated multiclock synthesis

The other way is to synthesize a single-clock design that mimics operation of the multiclock design. This is very useful for FPGA simulation of multiclock design. The idea is exactly the same as in Section **??**: using a simulation clock, one can directly implement the `MC_sim` simulation design in hardware or FPGA. Clock-mimicking signals can be generated in Esterel as in Section **??** or using any clock gating device available in the technology.

The advantage of simulated multiclock simulation is that one can simulate or execute the multiclock design on a single-clocked platform without changing a line of code.

13.5.3 Dealing with falling-edge clocks

All our circuit implementation examples were given with rising edge clocks. At Esterel logical level, what matters is only when clock events occur, and whether they are rising or falling edge events is simply a question of HDL implementation. When dealing with a falling-edge clock, simply think of its negation, which is a rising-edge clock that has clock events at the same time.

Dealing with designs that react to both rising edges and falling edges of the same clock is more problematic and we do not plan to model it for the time being.

Chapter 14

Esterel clocks and multiclock units

14.1 Clocks

14.1.1 Clock declarations

A clock is a signal declared by a specific declaration "`C : clock`", which can appear only in a multiclock unit. A clock can be declared `input` or `output` in the multiclock unit header or local in the multiclock unit body. All clocks are pure (i.e. not valued). Clock arrays and inputoutput clocks are not available. Clock declarations can be mixed with other signal declarations. Here are examples:

```
multiclock Multi :
  input I;                  // standard pure input
  input C : clock;          // input clock
  output {C1, C2} : clock;  // output clocks
  signal S : unsigned,      // standard valued local signal
         C3  : clock  in    // local clock
    ...
  end signal
end multiclock
```

Any output or local clock declared in a multiclock unit must be uniquely defined by a clock equation, see Section ??.

14.1.2 Clock usage

Clocks can only be used in multiclock units. They cannot occur in modules. Their usage is retricted to the following cases:

- In right-hand sides of clock equations, to derive other clocks. See Section ??.
- In module clocked `run` instantiations of the form "`run M [clock C; ... ]`". Such statements can only appear in multiclock unit bodies. They define the clock for the module's state change. See Section ??.
- In clock renamings in multiclock `run` instantiation, which can only appear in other multiclock unit bodies. See Section ??.

- In signal-to-clock bindings for "`mcrun Multi [ S / C ...]`" statements that run multiclock units inside modules. See Section ??.

Unlike pure signals, clocks cannot be used in signal expressions in right-hand-sides of signal or clock equations.

14.2 Multiclock units

A multiclock unit starts with `multiclock` and ends with "`end multiclock`". It is composed of a header and a body that bear many similarity with module headers and bodies.

14.2.1 Multiclock headers

A multiclock header declares data and interface signals and extend data and interface units exactly as for a module, except for restrictions mostly due to the absence of registers generated by a multiclock unit:

- All declared valued signals are implicitly declared `temp`.
- When interfaces are extended, all imported signals are made `temp`.
- Signal `mem`, `reg` and `init` declarations and refinements are not available.

In addition to data and signals, multiclock units can also declare clocks using clock declarations "`C : clock`", see Section ??. Clock declarations can be mixed with signal declarations.

Multiclock units can be data-generic in the same way as module units, and generic data instantiation is done exactly in the same way as for modules. See Section ?? for details.

Here is an example of a multiclock header:

```
data D :
  generic constant N : unsigned;
end data



interface Intf :
  input I : unsigned;
  output O : bool[4];
end interface


multiclock Multi :
  extends data D;           // multiclock unit becomes generic
  extends Intf;             // I and O are imported as temp
  input {C1, C2} : clock;   // declaration grouping is as usual
  output O,
         C3 : clock;        // mix of signal and clock declarations
  ...
end multiclock
```

14.2.2 Multiclock unit extension

As for a module, the declarations of a multiclock unit `Multi` can be imported in another module or multiclock unit using "`extends Multi`".

- If "`extends Multi`" appears in a multiclock unit `Multi2`, all declarations of `Multi` are imported in `Multi2`, including clock declarations.
- If "`extends Multi`" appears in an interface unit or a module unit, only the data and signals declared in `Multi` are imported, clock declarations being discarded.
- If "`extends Multi`" appears in a data unit, only the data components of `Multi` are imported, signals and clock declarations being discarded.
- To import only the data components of `Multi` in another unit, including in a multiclock unit, one can use "`extends data Multi`".
- To import only the data and signal components of `Multi` in another unit, including in a multiclock unit, one can use "`extends interface Multi`".

Notice that there is no "multiclock interface" unit. This is generally not a problem since clocks are far less numerous than signals. Because of the way multiclock unit extension is defined, multiclock units with body `nothing` can play the role of multiclock interface units.

14.2.3 Multiclock body structure

The body of a multiclock unit can be either `nothing`, in which case the unit has no behavior, or composed of the following elements:

- Local signal declarations, including local clock declarations.
- Combinational `sustain` statements, detailed in Section **??**.
- Clock equations, detailed in Section **??**.
- Module instantiations, detailed in Section **??**.
- Multiclock instantiations, detailed in Section **??**.
- Parallel compositions of the above elements.

Here an example of multiclock unit body:

```
signal C : clock in
  sustain X <= I or J
||
  clock C <= Cin if X
||
  signal S in
    run Mod [ clock C ]
  ||
    run Multi2
  end signal
end signal
```

14.2.4 Combinational computations in multiclock units

Combinational computations in multiclock units are simply `sustain` statements where `next` is disallowed in the left-hand-side and `pre` is disallowed in the right-hand-side expressions. Besides these restrictions, all the features of `sustain` statements are allowed, see Section **??**. Here are legal multiclock combinational computations:

```
   sustain X
||
   sustain ?Y <= 1 if I
||
   sustain {
     if
       case A do
         X <= I
       case B do
         ?T <= 1 if C
       default do
         ?Y <= 2
     end if
   }
```

Notice that we use `sustain`, which acts permanently as does hardware combinational logic, instead of `emit`, which would act once and terminate. There is no control flow in multiclock units, and only lasting operations are allowed.

As mentioned in Section **??**, clocks cannot appear in multiclock combinational computation expressions.

14.2.5 Clock equations

Clock equations define local or output clocks in function of other clocks and signals. Each output or local clock must be defined by exactly one clock equation. A clock equation or equation set starts with the `clock` keyword and contains a comma-separated list of clock definitions. There are three kinds of clock definitions:

- *Clock synonym:* a synonym definition “`C2 <= C1`” defines clock `C2` as a synonym for `C1`.

- *Clock downsampling:* a downsampling definition “`C2 <= C1 if S`” defines the clock `C2` as the subsampling of `C1` by the signal `S`. In the discrete multiclock semantics, `C2` has a clock event at any global tick where `C1` has a clock event and `S` is present. Notice that the `if` test is limited to a single pure signal. If a more complex expression is needed, declare a local signal and use a combinational definition for it. But, in hardware implementation, beware of timing closure issues for this signal.

- *Clock muxing:* a muxing definition “`C3 <= mux(S, C1, C2)`” defines the clock `C3` as the mux of of `C1` and `C2` according to presence of `S`. In the discrete multiclock semantics, `C3` has a clock event whenever `S` is present and `C1` has a clock event or `S` is absent and `C2` has a clock event. As for clock downsampling, the `mux` test is limited to a single pure signal.

Here are examples of clock equations:

```
    clock C2 <= C1 if I
  ||
    clock {
      C3 <= C1,
      C4 <= mux (J, C2, C3)
    }
```

For a clock downsampling or muxing, it is semantically safe for the signal `S` to change status exactly when the selected clock front occurs: the new status of `S` is then taken into account as always in the 0-delay model. However, such an operation would almost certainly lead to metastability if actually synthesized to hardware. In practice, controlling the change time of `S` w.r.t. the edges of the argument clocks is critical for correct electrical behavior.

14.2.6 Running modules in multiclock units

To achieve GALS behavior, a classic module unit can be instantiated within a multiclock unit body using the `run` statement. This statement is as for module instantiation in other modules, except that a clock must be explicitly passed using a "`clock C;`" specification in the substitution list. Moreover, a reset must be also explicitly passed using a "`reset R;`" or "`weak reset R;`" specification.

```
  run M [clock C;           // mandatory argument
         reset R;           // mandatory argument
         constant 2 / N;    // usual data substitution
         X / I, Y / O ]     // usual signal binding
```

The only place where a clock can occur is the `clock` specification. Refer to Chapter **??** for data and signal substitution, which is unchanged. Implicit substitution by equal names is of course available.

The behavior of `M` is run with clock C to clock `M`'s state. See Section **??** for monoclock expansion of the `run` statement with `weak suspend`. Signal binding is always done by connection, see Section **??**.

14.2.7 Running multiclock units in multiclock units

To achieve hierarchical multiclock design, a multiclock unit body can instantiate another multiclock unit using the `run` statement. In multiclock `run`, clocks can be bound to clocks in the same way as signals are bound to signals. Unlike for module `run`, no clock is passed as mandatory first argument since a multiclock unit has no sequential computation *per se*. Here is an example:

```
  run Multi [ constant 3 / N;      // usual data renaming
              C1 / C2,             // clock renaming
              X / I ]              // usual signal binding
```

Notice that clock binding can occur anywhere in the signal binding section since it is a normal signal binding.

14.2.8 Running multiclock units in modules

Finally, a multiclock unit can be instantiated in a module body using the `mcrun` statement, which has the same form as a `run` statement. All input and output clocks ot the instantiated multiclock unit must be bound to pure signals in the caller. The other data and signal objects are bound as usual. Here is an example:

```
multiclock Multi :
  constant N : unsigned;
  input CI : clock;
  output CO : clock;
  input I [N];
  output O;
  ...
end multiclock


module Mod :
  output SO;
  input I [3];
  output X;
  signal SI in
    loop emit SI each 2 tick
  ||
    mcrun Multi [ constant 3 / N;
                  signal SI / CI,  // SI generates events of CI
                         SO / CO,  // SO records events of CO
                         X / O ]
```

See Section **??** for the semantics of `mcrun` and for its used in multiclock design debugging.

Chapter 15

Observers

15.1 General

Observers give users a non-intrusive way of associating verification code with design code. The purpose of observers is to enhance modularity, clarity, and reuse of verification code. Using observers, it is possible to write and modify verification code without changing design code and, conversely, to change design code, for instance when fixing bugs, without impacting verification code.

The basic brick for building observers is an *observer binding*, which associates an observing (verification) module to an observed (design) module. Observer bindings are not restricted to observe main modules but can observe any submodule in the design hierarchy. An *observer unit* declares a set of observer bindings intended for observing the design. An observer unit can be seen as verification code, which is defined from the observer bindings, and which runs in parallel with design code. Like for modules, a *main* observer has to be chosen. The main observer specifies observing code to be compiled, in particular the assertions, assumptions, and coverage points that are declared in observing modules.

15.2 Observer bindings

15.2.1 General

An *observer binding* makes it possible to observe a design module or multiclock unit, called the *observed* unit, using a verification module or multiclock unit, called the *observing* unit. The syntax is as follows:

```
observe M with VMI / VM [ <renamings> ] ;
```

where:

- `M` is the observed module or multiclock unit.
- `VM` is the verification module or multiclock unit observing `M`. All interface signals of `VM` must be input.
- `VMI` is an optional *instance identifier* for the verification module instance. If not specified, it is implicitly `VM`.

- `<renamings>` are data renamings, signal renamings, assert and assume renamings, clock and reset definition, exactly like in a **run** statement, where `M` would be the master and `VM` would be the sub. Renaming signals are allowed to be local signals declared in the header of `M` (see Section **??**). Implicit renaming by name-capture is allowed as for a **run** statement.

Considering the classic `ABRO` design, here is an example of verification module and associated observer binding:

```
module AImpliesNotBVerifier:
input A, B;
sustain assert AImpliesNotB = (A => not B)
end module
...
observe ABRO with OImpliesNotRVerifier/AImpliesNotBVerifier [ R/A, O/B ];
```

In an observer binding, one can specify a particular instance for the module to be observed by giving the *absolute name* of the instance. The syntax is as follows:

```
observe M in MI with VMI / VM [ <renamings> ] ;
```

where `MI` designates the observed instance in the main instance tree.

The instance absolute name `MI` is a list of instance identifiers separated with the hierarchy delimiter '/' for example `MAIN/N1/M2`, where `MAIN` is the name of the main module or multiclock unit, `N1` is an instance name of a unit `N` in `MAIN`, and `M2` is an instance name of `M` in `N`. Since instance names are not required to be unique in Esterel, an absolute name may refer to several instances. In this case, the observer binding applies to all the instances.

An observer binding that uses an observed instance absolute name is called an *instance observer binding*. An observer binding that specifies no instance absolute name is called a *global observer binding*.

15.2.2 Semantics

We consider the observer binding:

```
observe M with VMI / VM [ <renamings> ] ;
```

Assuming that `M` is a module, we note `<header>` and `<body>` respectively its header and body:

```
module M:
  <header>
  <body>
end module
```

In the semantics, the effect of the observer binding is to modify the module `M` as follows:

```
module M:
  <header>
  weak abort
    run VMI / VM [ <renamings> ]
  after
    <body>
  end abort
end module
```

In the case of an instance observer binding, the expansion is semantically performed only for the specified instance of the observed module.

Notice that the observer binding does not impact the header of `M` but only the body, without changing the behavior of `M`: interface signals of `VM` are all inputs and the termination of `M` is preserved thanks to the `weak abort` statement. If `M` is a multiclock unit with header and body also noted `<header>` and `<body>`, then the body of `M` is modified as follows:

```
multiclock M:
<header>
  run VMI / VM [ <renamings> ]
||
  <body>
end multiclock
```

The `weak abort` statement is not used here, since `M` cannot terminate. If `M` is observed by several observer bindings, then `run` statements of observing modules are performed in parallel in the body of `M`. For example, if `M` is a module observed by `VM1I / VM1 [ <renamings> ]`, `VM2I / VM2 [ <renamings> ]` ... `VMNI / VMN [ <renamings> ]`, then the body of `M` is modified as follows:

```
module M:
<header>
weak abort
   run VM1I / VM1 [ <renamings> ]
||
   run VM2I / VM2 [ <renamings> ]
||
  ...
||
   run VMNI / VMN [ <renamings> ]
after
  <body>
end abort
end module
```

The observing unit in an observer binding is not allowed to be a multiclock unit if the observed unit is a classic module. The converse is allowed, provided that the clock and reset of the observing unit are defined.

15.3 Observer units

15.3.1 General

An *observer unit* is a set of observer bindings which are intended to specify together a consistent and full observer for the design. An observer unit can be viewed as verification code, specified by the bindings, running in parallel with design code. An observer unit is declared with the `observer` keyword followed by the observer name. Here is an example:

```
observer O:
observe MAIN with VMAIN;
observe M with VM;
observe N in MAIN/N1 with VN1;
observe N in MAIN/N2 with VN2;
end observer
```

15.3.2 Observer extension

An observer unit can import all the observer bindings declared by another observer with the `extends` keyword. Here is an example:

```
observer O:
observe MAIN with VMAIN;
observe M with VM;
...
end observer

observer O2:
extends O; // O2 imports all observer bindings from O
// this is equivalent to:
// observe MAIN with VMAIN;
// observe M with VM;
...
end observer
```

Assert and assume renamings are allowed in observer extension. They apply to the observer bindings imported from the extended observer unit. Here is an example:

```
observer O3:

extends O [ assert/assume ]; // transforms assumptions of O
                             // into assertions for O3
// this is equivalent to:
// observe MAIN with VMAIN [ assert/assume ];
// observe M with VM [ assert/assume ];

...
end observer
```

Diamond observer extension is not allowed. For example, the following is forbidden, because observer `O` would be extended twice:

```
observer O4:
extends O2; // O is extended through O2
extends O3; // error: O is extended again through O3
end observer
```

15.3.3 Main observer

A *main* observer unit has to be chosen. It specifies observing code to be compiled with the design that is specified by the main module. Observer units that are not directly or indirectly extended by the main observer are discarded. An observer unit can be declared as the main observer with the `main` keyword. Naturally, it is possible to select no main observer for compiling; in this case, all observer units are discarded.

15.4 FIFO observers example

We take the example of a simple FIFO, which extends the following control interface. Making abstraction of the rest of the FIFO design, we show how to use observers to associate verification code to FIFO code.

```
interface FifoCtrlIntf:
input  Put;   // put words into the FIFO
output Full;  // FIFO is full
input  Get;   // put words into the FIFO
output Empty; // FIFO is empty
end interface
```

It is obvious that signals `Full` and `Empty` must not be present at the same time. To verify the FIFO implementation, one can write the following verification module and the following observer:

```
module FullEmptyExclusiveChecker:
input Full, Empty;
sustain assert FullEmptyExclusive = Full # Empty;
end module

observer FullEmptyObserver:
observe Fifo with FullEmptyExclusiveChecker;
end observer
```

Now let us write verification code to check that the module that instantiates the FIFO makes a good usage of it, i.e. does not put words into the FIFO if it is full, and does not get words from the FIFO if it is empty:

```
module FifoUsageChecker:
extends observe FifoCtrlIntf;
sustain {
  assert NoFullError = not(Full and Put),
  assert NomptyError = not(Empty and Get)
}
end module

observer FifoUsageObserver:
observe Fifo with FifoUsageChecker;
end observer
```

Last, assume that the FIFO module declares a local signal `Size` in its header, which is the number of currently stored words:

```
signal Size : value reg unsigned<MaxSize+1> init 0;
```

where `MaxSize` is a constant declared in the FIFO module. For profiling purpose, we want to observe when the FIFO reaches a threshold of `MaxSize`, `MaxSize`/2 and `MaxSize`/4 stored words. Here is corresponding verification code:

```
module FifoSizeProfiler:
generic constant MaxSize : unsigned;
input Size : value temp unsigned<MaxSize+1>;
generic constant Threshold : unsigned<MaxSize+1>;
sustain cover ThresholdReached <= ?Size >= Threshold
```

```
end module

observer FifoSizeObserver:
observe Fifo with Profiler1/FifoSizeProfiler [constant MaxSize/Threshold];
observe Fifo with Profiler2/FifoSizeProfiler [constant (MaxSize/2)/Threshold];
observe Fifo with Profiler4/FifoSizeProfiler [constant (MaxSize/4)/Threshold];
end observer
```

15.5 Protocol observer example

This example shows how to use observers for modular verification based on the *assume-guarantee* principle. We consider a simple initiator-target protocol. The target interface is given below; the initiator interface is simply a mirror of it.

```
interface TargetIntf:
input Valid;
output Ready;
input Data : value temp unsigned;
end interface
```

The following modules are intended to verify protocol properties on target and initiator sides.

```
module TargetAssertions:
extends observe TargetIntf;
sustain {
  assert ReadyStable = Ready if pre(Ready and not Valid),
  assume ValidStable = Valid if pre(Valid and not Ready)
}
end module

module InitiatorAssertions:
extends TargetAssertions;
run TargetAssertions [ assert/assume; assume/assert ];
end module
```

Now consider the following design, which uses initiator and target ports at toplevel and submodule boundaries.

```
module Top:
port T : TargetIntf;
port I : mirror TargetIntf;
signal port P : TargetIntf in
  run M1 [ T/T1, P/I1 ]
||
  run M2 [ P/T2, I/I2 ]
end signal
end module

module M1:
port T1 : TargetIntf;
port I1 : mirror TargetIntf;
...
end module
```

```
module M2:
port T2 : TargetIntf;
port I2 : mirror TargetIntf;
...
end module
```

At toplevel boundary, the design has an initiator port `I` and a target port `P`. The main module `Top` instantiates two submodules `M1` and `M2`. Module `M1` is directly connected to the toplevel initiator port `I` and module `M2` is directly connected to the toplevel target port `P`. Modules `M1` and `M2` are connected together through a target-initiator protocol on a port `P`. We want to verify the protocol implementation in `M1`, `M2`, and `Top` in the following three steps. We will show how to design the main observer for each step.

1. Check in every submodule that protocol assertions are valid, based on submodule-level protocol assumptions.
2. Check that submodule-level protocol assumptions (used in step 1) are valid, based on toplevel protocol assumptions. Submodule-level protocol assertions (verified in step 1), are used as auxiliary assumptions in order to help the verification engine.
3. Check that toplevel protocol assertions are valid, based on toplevel protocol assumptions. Submodule protocol assertions are used as auxiliary assumptions like for step 2.

For step 1, we check that protocol assertions are valid in `M1` using the following observer as main:

```
observer M1Observer: // take as main for step 1
observe M1 with TargetAssertions [ T1/T ];
observe M1 with InitiatorAssertions [ I1/I ];
end observer
```

To complete step 1, we do the same for `M2` using the following observer:

```
observer M2Observer: // take as main for step 1
observe M2 with TargetAssertions [ T2/T ];
observe M2 with InitiatorAssertions [ I2/I ];
end observer
```

The following observer unit is used as main to perform the verification step 2. Notice how assert and assume renamings make it possible to re-use the precedently designed observer units `M1Observer` and `M2Observer`:

```
observer M1M2Observer: // take as main for step 2

// verify assumptions in M1
extends M1Observer [ assert/assume;   // verify assumptions
                     assume/assert ]; // use assertions as assumptions

// verify assumptions in M2
extends M2Observer [ assert/assume;   // verify assumptions
                     assume/assert ]; // use assertions as assumptions

// use Top assumptions
```

```
extends TopObserver [ /assert ]; // ignore Top assertions

end observer
```

where TopObserver is obviously defined as:

```
observer TopObserver:
observe Top with TargetAssertions;
observe Top with InitiatorAssertions;
end observer
```

Last, we use the following observer unit as main to perform the verification step 3. Once again, M1Observer and M2Observer are re-used with adequate assert and assume renamings.

```
observer EnhancedTopObserver: // take as main for step 3

// verify Top assertions
extends TopObserver;

// use M1 assertions as assumptions
extends M1Observer [ /assume;          // assumptions are useless here
                     assume/assert ]; // use assertions as assumptions

// use M2 assertions as assumptions
extends M2Observer [ /assume;          // assumptions are useless here
                     assume/assert ]; // use assertions as assumptions

end observer
```

Now it's done! For the sake of completeness, we give the main observer that can be used to perform the full verification of the design all at one time, as alternative of the 3-step modular verification that has just been presented.

```
observer MonolithicTopObserver:
// verify Top assertions
extends TopObserver;
// verify M1 assertions
extends M1Observer [ /assume ]; // discard M1 assumptions
                                // (could also be verified using assert/assume)
// verify M2 assertions
extends M2Observer [ /assume ]; // discard M2 assumptions
                                // (could also be verified using assert/assume)
end observer
```