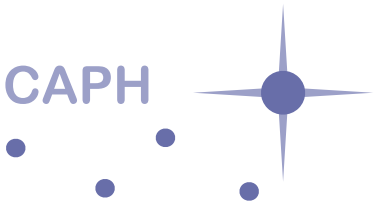
The CAPH Primer

J. Sérot

June 27, 2017



Introduction

This document is a short introduction to the CAPH programming language and associated toolset. It is divided in three parts.

Part 1 gives a short, informal introduction to the concepts and syntax of the language.

Part 2 introduces the CAPH integrated development environment (IDE). This IDE can be used to familiarize with the language and explore the basic functionalities such as displaying programs as data-flow graphs (DFGs) and simulating their behavior.

Part 3 goes a bit further and describes how to use CAPH in a command line based environment and to interface to existing third-party tools, such as C++ compilers and VHDL synthetizers.

Part I

The Caph language

Chapter 1

Dataflow programming

CAPH is based upon a strict dataflow model of computation : Applications are described as networks of computational units, called *actors*, exchanging *streams* of *tokens* through unidirectional, buffered channels. Data to be processed is simply "pushed" in the input ports of the network and results are collected at output ports. Execution occurs as tokens litteraly "flow" through channels, into and out of actors.

This model of computation is illustrated on a very simple example in Fig. 1.1. The dataflow network operates on unstructured streams of tokens carrying integer values. For each input token carrying value x, it produces a result token carrying value $(x + 1) \times (x - 1)$. For instance, if the input stream (provided to the `dup` actor, through the `i` input channel) is

```
1 2 3 4 ...
```

the the output stream (produced by the `mul` actor, on the `o` output channel) will be

```
0 3 8 15 ...
```

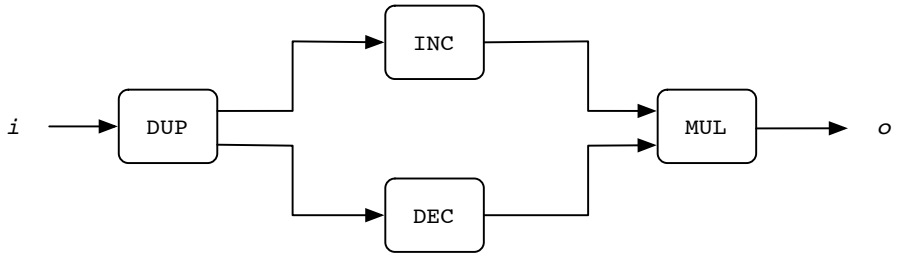


Figure 1.1: A very simple dataflow network

The network of Fig. 1.1 involves four simple actors. Actor `inc` (resp. `inc`) adds (resp. substracts) 1 to each element of its input stream, actor `mul` performs point-wise multiplication of two streams and actor `dup` duplicates its input stream.

To understand what really "happens" when it is "executed" we need to attach a *semantics* both to actors and to the channels connecting these actors.

The semantics of actors will be given as a set of *firing rules* describing exactly *when* an actor executes ("fires") and *what* happens then. In this first example, the firing rule is the same for each actor and it can be stated as : whenever a token is available on the channel connected to each input port then read ("consume") this token, compute the result(s) from the associated value(s) and write ("produce") the token(s) carrying this (these) result(s) on the channels connected to the output port(s)[1].

[1]We will see latter that CAPH allows more complex rules (and hence more sophisticated behaviors) to be expressed.

The semantics of channels is simple : they will be viewed as FIFOs (First In First Out) buffers. In the final implementation, the size of theses FIFOs will obviously be an important parameter. But for now, let us consider that they are essentially unbounded.

1.1 From sketch to code

There's a long way from the rather "informal" description of an application as given in Fig. 1.1 to a FPGA configuration performing the described functionnality on a stream of values.

The main steps in this path are illustrated in Fig. 1.2. These steps will be discussed in part 2 and 3 of this document. Let's focus for the moment in the initial step, which is writing the source code of the application using the CAPH language.

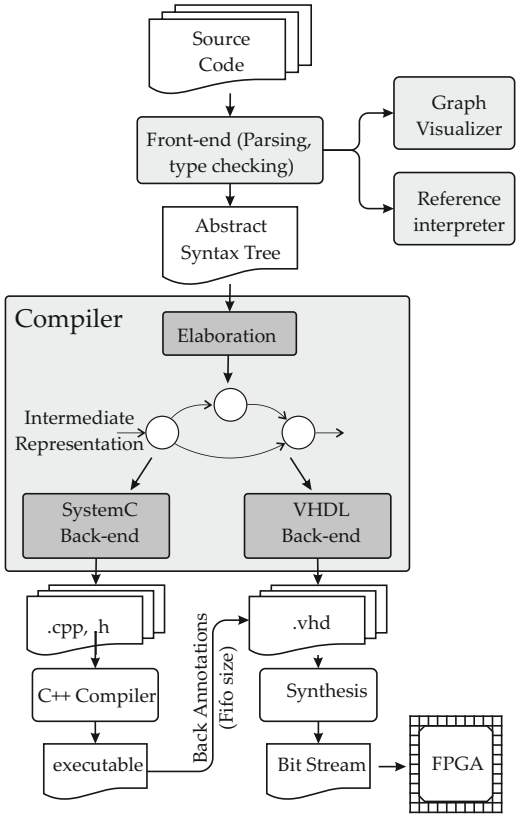


Figure 1.2: The CAPH design flow

1.2 Writing the source code

Let's write in CAPH the description of the application depicted in Fig. 1.1 in file `simple.cph`.

We start by declaring the input and the output of the network :

```
stream inp: unsigned<8> from "sample.txt";
stream outp: unsigned<8> to "result.txt";
```

The keyword `stream` introduces an I/O declaration. Each I/O has a name, a type and a description. Here, we declare

- `inp` to be a input, with type `unsigned<8>`, *i.e.* unsigned 8-bit integer, taking values from a a file named `sample.txt`[2].
- `outp` to be an output, also with type `unsigned<8>` putting values in a file named `result.txt`.

Concerning syntax, note that each declaration ends with a semi-colon.

The next step consists in describing the network of actors. Basically, this involves specifying which actors appear in this network and listing the connexions between these actor ("wiring" the network). In CAPH[3], this is done in a purely textual manner, by naming wires and viewing actors as functions from wires to wires.

In this particular case, we start with the following declaration :

```
net (x1,x2) = dup inp;
```

This declaration, introduced by the `net` keyword actually has two effects :

- first, it creates, in the network described by the program, a node named `dup`,
- second, it respectively *binds* the input of this node to the wire named *inp* (which, in this case is the input wire of the whole network) and its outputs to two wires named `x1` and `x2`.

We can now proceed (going "from left to right" in the graph of Fig. 1.1), with the following declarations :

```
net y1 = inc x1;
net y2 = dec x2;
```

The first declaration insert a nodes name `inc`, binding its input to the previously defined `x1` wire (*i.e.* the first output of the `dup` node) and its output to a new wire named `y1`. The second one does a similar thing with node `dec` and wires `x2` and `y2` respectively.

A last declaration inserts the `mul` node, connecting its inputs to the output of the `inc` (resp. `dec`) node (by means of wires `y1` and `y2`) and its output to the global output `outp` :

```
net outp = mul (y1,y2);
```

Note that the last three declarations could have been combined into a single one by writing :

```
net outp = mul (inc x1, dec x2);
```

Which style is better – with or without explicit naming of intermediate wires – is essentially a matter of taste since both will lead to exactly the same network.

Together, the set of `stream` and `net` declarations introduced above completely determines the *static* stucture of the actor network[4].

We now have to define the *dynamic* behavior of the actors appearing in this network.

Let's start with the `inc` actor. Its behavior is specified by the following declaration :

```
1 actor inc
2   in (i: unsigned<8>)
3  out (o: unsigned<8>)
4 rules
5 | i:x -> o:x+1;
```

This declaration is composed of two parts : The first part (the *interface*, lines 1–3) gives the name of the actor and lists its inputs and outputs (giving a name and a type to each of them), The second part (lines 4–5) specifies the behavior of the actor, by listing all the associated firing rules. Here, there's only one rule[5] and it can be read as follows : whenever there's a token, carrying a value x, available on input `i` then read (consumes) this token and write a token carrying value $x + 1$ on output `o`.

The definition of the `dec` actor is very similar :

[2] As said above, this file will be used for simulation.
[3] And for reasons which are advocated in the reference manual [1].
[4] In other words, its *topology*
[5] Each rule starts with a leading `|`.

```
actor dec
  in (i: unsigned<8>)
 out (o: unsigned<8>)
rules
| i:x -> o:x-1;
```

The `mul` actor has two inputs and a single input. This is reflected in its interface and in the format of the firing rule :

```
actor mul
  in (i1: unsigned<8>, i2:unsigned<8>)
 out (o: unsigned<8>)
rules
| (i1:x, i2:y) -> o:x*y;
```

The interpretation of the firing rule for the `mul` actor is an obvious generalisation of the one given for the two previous actors : the actor fires whenever a token (carrying values x and y respectively) is available on `inputs`, `i1` and `i2`. Concerning the syntax, note the use of brackets on the left-hand side of the rule, which is mandatory here.

The `dup` actor has a single input, but two outputs. This, again, is reflected in its interface and the format of the firing rule :

```
actor dup
  in (i: unsigned<8>)
 out (o1: unsigned<8>, o2:unsigned<8>)
rules
| i:x -> (o1:x, o2:x);
```

For this token, a token, carrying the same value (x) will be produced on both outputs (`o1` and `o2`) whenever the actor fires.

The full text of the program is given in Listing 1.1. Note that, contrary to the presentation order we have used above, the declarations of actors actually have to appear first (this is because these declarations will be used by the `net` declarations). Comments are introduced by the `--` character sequence[6].

Listing 1.1: Complete CAPH source code for the application depicted in Fig. 1.1

```
-- Actor declarations

actor inc
  in (i: unsigned<8>)
 out (o: unsigned<8>)
rules
| i:x -> o:x+1;

actor dec
  in (i: unsigned<8>)
 out (o: unsigned<8>)
rules
| i:x -> o:x-1;

actor mul
  in (i1: unsigned<8>, i2:unsigned<8>)
 out (o: unsigned<8>)
rules
| (i1:x, i2:y) -> o:x*y;
```

[6]Comments are single-line, like in Java.

```
actor dup
  in (i: unsigned<8>)
 out (o1: unsigned<8>, o2:unsigned<8>)
rules
| i:x -> (o1:x, o2:x);

-- I/O declarations

stream inp: unsigned<8> from "sample.txt";
stream outp: unsigned<8> to "result.txt";

-- Network declarations

net (x1,x2) = dup inp;
net y1 = inc x1;
net y2 = dec x2;
net outp = mul (y1,y2);
```

Chapter 2

Dealing with images

In this chapter we will show how to use CAPH to implement a very simple image processing application. This will be the opportunity to introduce the core concepts used for dealing with images – mainly their representation as *structured streams* of pixels – and to describe the tools used for manipulating them at the simulation level.

2.1 Representation of images

In chapter 1, the dataflow network used as an example, was operating on a raw, unstructured stream of data. In contrast, images are *structured* streams of pixels. In particular, in most of applications, we need a way to encode the dimensions of a given image (so that we can tell, for example, if a stream of 64 pixels actually represents an image with 8 lines of 8 pixels, an image with 4 lines of 16 pixels or even four successive images with 4 lines of 4 pixels).

For this, the idea is to insert, in the stream of pixels, *control* tokens expliciting the underlying structure of the data and to distinguish control tokens from *data* tokens (carrying pixel values) by attaching a *tag* to each token. In practice, for an application having to process images, the input of will be a sequential stream of tokens, where each single token is either

- the tag `SoI` (start of image),
- the tag `EoI` (end of image),
- the tag `SoL` (start of line),
- the tag `EoL` (end of line),
- a pixel value, with tag `Pixel`.

With this scheme, the 4×4 image depicted in Fig. 2.1, for example, will be represented ("encoded") by the following stream of tokens:

```
SoI SoL Pixel(10) Pixel(30) Pixel(55) Pixel(90) EoL SoL Pixel(33) Pixel(53)
Pixel(60) Pixel(12) EoL SoL Pixel(99) Pixel(56) Pixel(23) Pixel(11) EoL SoL
Pixel(11) Pixel(82) Pixel(46) Pixel(11) EoL  EoI
```

In fact, only two distinct control tokens are needed :

- a token `SoS` (*start of structure*), signaling the start of an image or the start of a line within a image,
- a token `EoS` (*end of structure*), signaling the end of an image or the end of a line within a image.

As a result, and using the following abbreviations :

- `<` for `SoS`,

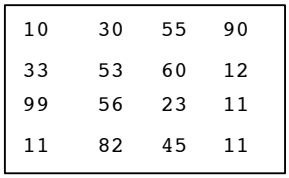


Figure 2.1: A 4×4 image

- `>` for `Eos`,
- `v` for `Pixel(v)`,

the image depicted in Fig. 2.1, can be represented by the following stream of tokens :

```
< < 10 30 55 90 > < 33 53 60 12 > < 99 56 23 11 > < 11 82 46 11 > >
```

2.2 Processing images

By using the pattern-matching mechanism introduced in Chap. 1 it is very easy to describe the behavior of actors operating on structured streams of values.

Consider, for example, an actor performing image negation on images made of 8-bit unsigned pixels, *i.e.* each pixel having value v is transformed to a pixel having value $255 - v$.

Such an actor is decribed in Listing 2.1.

First, note that the type of the input and output for this actor (`i` and `o`, lines 2–3) is *not*

```
unsigned<8>
```

but

```
unsigned<8> dc
```

The type `dc` (abbreviation for *data or control*) is here used for representing structured values[1]. A value having type `t dc`, where `t` is a scalar (unstructured) type, is either

- the control value `SoS` (which can be abbreviated as `'<`),
- the control value `EoS` (which can be abbreviated as `'>`),
- a data value `v`, of type `t` (which can be abbreviated as `'v`).

The actor rules use the pattern matching mechanism to inspect the tag of the input value and to produce the appropriate value on output :

- if the input token is a control token (`'<` or `'>`, lines 5 or 6), write the same token on output (this means that the *structure* of the image is unchanged),
- if the input token is data token (pixel), carrying value `x` (line 7), write a data token carrying value `255-x` on output.

[1] CAPH uses so-called *algebraic data types* (aka *variant types*) for this. Internally, the `dc` type constructor is defined as :
`type $t dc = SoS | EoS | Data of $t`
where `SoS`, `EoS` and `Data` are the *value constructors* associated to tags and $\$t$ denotes a *type variable*. The notations `'<`, `'>` and `'v` are then abbreviations for `SoS`, `EoS` and `Data v` respectively.

Listing 2.1: An actor computing image negatives in CAPH

```
actor inv
  in (i: unsigned<8> dc)
 out (o: unsigned<8> dc)
rules
| i:'< -> o:'<;
| i:'> -> o:'>;
| i:'x -> o:'255-x;
```

Note 1. The code in Listing 2.1 uses the abbreviated syntax for denoting values with type `unsigned<8> dc`. It is also possible to use the un-abbreviated syntax, as shown in Listing. 2.2.

Listing 2.2: An actor computing image negatives in CAPH (alternate syntax)

```
actor inv
  in (i: unsigned<8> dc)
 out (o: unsigned<8> dc)
rules
| i:SoS -> o:SoS;
| i:EoS -> o:EoS;
| i:Data(x) -> o:Data(255-x);
```

Note 2. The CAPH type system ensures that tagged and untagged values are used consistently in programs. It we write, for example, the last rule of actor `inv` as :

```
| i:x -> o:255-x;
```

we get the following error message from the compiler :

```
File "inv.cph", line 7, characters 11-16:
>| i:x -> o:255-x;
>...........^^^^^
An error occured when typing this expression: types unsigned<#a> and unsigned<8> dc cannot be unified.
```

What the type checker detects here is that the output `o`, supposed to have type `unsigned<8> dc` (*i.e.* to be assigned a tagged value) is actually assigned a value of type `unsigned<n>`[2] (*i.e.* the untagged value $255 - x$).

In chapter 8, we will describe the implementation, simulation and synthesis of an application making use of the `inv` actor.

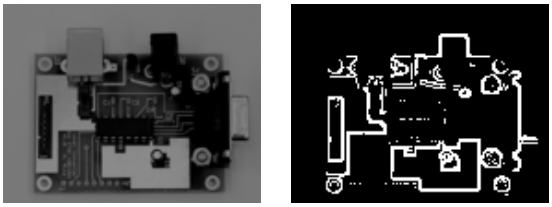
[2]The notation `#a` designates a *size variable*. It basically means "any, unknown, size n".

Chapter 3

Image processing

In this last chapter, we describe the use of the CAPH language to implement a more "realistic" image processing application.

This application performs edge extraction on images using the well-known Sobel filter.
An example of input and output image is given Fig. 3.1.



Input image Result image

Figure 3.1: Edge extraction with the Sobel operator

For each pixel $P_{i,j}$ of the input image, the magnitude of the local gradient is computed using approximations of the horizontal and vertical derivatives G_x and G_y, and the resulting value is compared to a fixed threshold for producing a binary image (with edge pixels encoded as 1 and background pixels as 0). To simplify, the magnitude of the gradient, $G = \sqrt{G_x^2 + G_y^2}$, will be here approximated as $|G_x| + |G_y|/2^n$, where n is a scaling factor.

Considering we have three actors, `grad`, `asum`, `thr`, computing respectively the gradient components, the half sum their absolute values and the binarisation of an image, the dataflow formulation of the corresponding is given in Listing 3.1. Figure 3.2 gives the corresponding dataflow network[1]

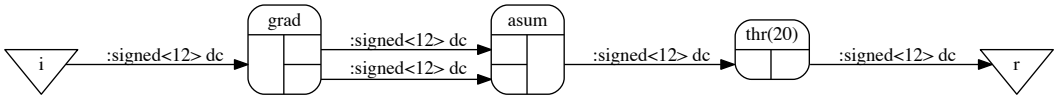


Figure 3.2: The graphical representation of the program given in Listing. 3.5

Listing 3.1: A Sobel edge extraction application in Caph (top level description)

```
1 actor grad in (i:signed<s> dc) out(o1:signed<s> dc, o2:signed<s> dc) ...
2 actor asum in ( i1:signed<s> dc, i2:signed<s> dc) out( o:signed<s> dc)  ...
```

[1]The binarisation threshold has here been arbitrarily set to 20.

```
3  actor thr (k:signed<s>) in ( i:signed<s> dc) out( o:unsigned<s> dc) ...
4
5  stream i:signed<12> dc from "pcb.txt";
6  stream r:signed<12> dc to "result.txt";
7
8  net (gx,gy) = grad i;          -- gradient x and y components
9  net gm = asum (gx, gy);        -- gradient magnitude (approx)
10 net r = thr 20 gm;
```

The `thr` actor is described in Listing 3.2. This actor applies a binarisation threshold to an image, returning a image of 1-bit unsigned pixels. The binarisation threshold is specified as a parameter (`t`, line 1), whose value will be set when instanciating the actor at the network level (see line 10 in Listing. 3.1). Binarisation is performed by simply comparing each pixel value to the threshold (last rule, line 7).

Listing 3.2: The `thr` actor in Caph

```
1  actor thr (k:signed<s>)
2    in ( i:signed<s> dc)
3    out( o:unsigned<1> dc)
4  rules i -> o
5  | '< -> '<
6  | '> -> '>
7  | 'p -> if p>k then '1 else '0
8  ;
```

The `asum` actor is described in Listing 3.3. This actor takes the computes an approximation of the gradient magnitude by summing the absolute value of its two components and dividing it by 2. The computation of the absolute value is performed using the global function `fabs`, which is declared before the actor. The division by 2 is implemented using the bit-shift builtin operator `>>`.

Listing 3.3: The `asum` actor in Caph

```
1  function fabs x = if x < 0 then -x else x : signed<s> -> signed<s>;
2
3  actor asum
4    in ( i1:signed<s> dc, i2:signed<s> dc)
5    out( o:signed<s> dc)
6  rules (i1,i2) -> o
7  | ('<,'<) -> '<
8  | ('>,'>) -> '>
9  | ('p,'q) -> '(fabs(p)+fabs(q))>>1
10 ;
```

The computation of the gradient components could be carried out by writing out dedicated actors, in the vein of those described in Sec. 2.5 of the reference manual. We will here adopt a more straightforward approach and rely on the predefined convolution actors provided in the standard CAPH library. The gradient x and y component can be computed by convolving the input image with the 2D kernels showed in Fig. 3.3.

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Figure 3.3: Convolution kernels for computing the gradient x and y components

For this, the file `convol.cph` provides the actor `conv233`. This actor accepts three parameters :

- a convolution kernel, given as a 2D array $k[0..2][0..2]$
- a scaling factor n,

- a padding value v.

Given a $M \times N$ input image x, represented as a structured stream

$$<<\ x_{1,1}\ x_{1,2}\ ...\ x_{1,N}\ ><\ x_{2,1}\ x_{2,2}\ ...\ x_{2,N}\ >\ ...\ <\ x_{M,1}\ x_{M,2}\ ...\ x_{M,N}\ >>$$

the `conv233` actor computes an output image y, with the same representation and dimensions

$$<<\ y_{1,1}\ y_{1,2}\ ...\ y_{1,N}\ ><\ y_{2,1}\ y_{2,2}\ ...\ y_{2,N}\ >\ ...\ <\ y_{M,1}\ y_{M,2}\ ...\ y_{M,N}\ >>$$

where

$$y_{i,j} = \begin{cases} v & \text{if } 1 \leq 2 \text{ or } 1 \leq j \leq 2 \\ \sum\limits_{\substack{0 \leq i' \leq 2 \\ 0 \leq j' \leq 2\}}} k_{2-i',2-j'}.x_{i-i',j-j'} \quad /2^n & \text{if } 2 \leq i \leq M \text{ and } 2 \leq j \leq N \end{cases} \tag{3.1}$$

The corresponding formulation in CAPH is given in Listing 3.4. The convolution kernels G_x and G_y are specified as 2S arrays. The padding value v (for the first two lines and columns) has here been set to 0 and the scaling factor is also 0.

Listing 3.4: Computation of the gradient components using the `conv233` actor of the standard CAPH library

```
net gx = conv233 ([[1,0,-1], [2,0,-2], [1,0,-1]], 0, 0) i;  -- grad x component
net gy = conv233 ([[1,2,1], [0,0,0], [-1,-2,-1]], 0, 0) i;  -- grad y component
```

The complete source code of the application is given in Listing 3.5[2]. The corresponding dataflow graph is given in Fig. 3.4. To simplify further assessment, the name of the input file and the binarisation threshold are here specified indirectly by using references to *macros*. The corresponding values will be set when invoking the compiler (with the `-D` option), thus offering a way of changing them without having to edit the source file (see Sec. 9.9 of the manual).

Listing 3.5: Complete source code of the Sobel-based edge extraction

```
#include "convol.cph"

function fabs x = if x < 0 then -x else x : signed<s> -> signed<s>;

actor asum
  in ( i1:signed<s> dc, i2:signed<s> dc)
  out( o:signed<s> dc)
rules (i1,i2) -> o
| ('<,'<) -> '<
| ('>,'>) -> '>
| ('p,'q) -> '(fabs(p)+fabs(q))>>1
;

actor thr (k:signed<s>)
  in ( i:signed<s> dc)
  out( o:unsigned<1> dc)
rules i -> o
| '< -> '<
| '> -> '>
| 'p -> if p>k then '1 else '0
;

stream i:signed<12> dc from %ifile;
stream r:unsigned<1> dc to "result.txt";

```

[2]It can also be found in directory `examples/working/primer/sobel` of the distribution.

```
26 net gx = conv233 ([[1,0,-1],
27                    [2,0,-2],
28                    [1,0,-1]],0,0) i;     -- grad x component
29 net gy = conv233 ([[1,2,1],
30                    [0,0,0],
31                    [-1,-2,-1]],0,0) i;   -- grad y component
32 net gm = asum (gx, gy);                  -- grad amplitude (approx)
33 net r = thr %threshold gm;
```

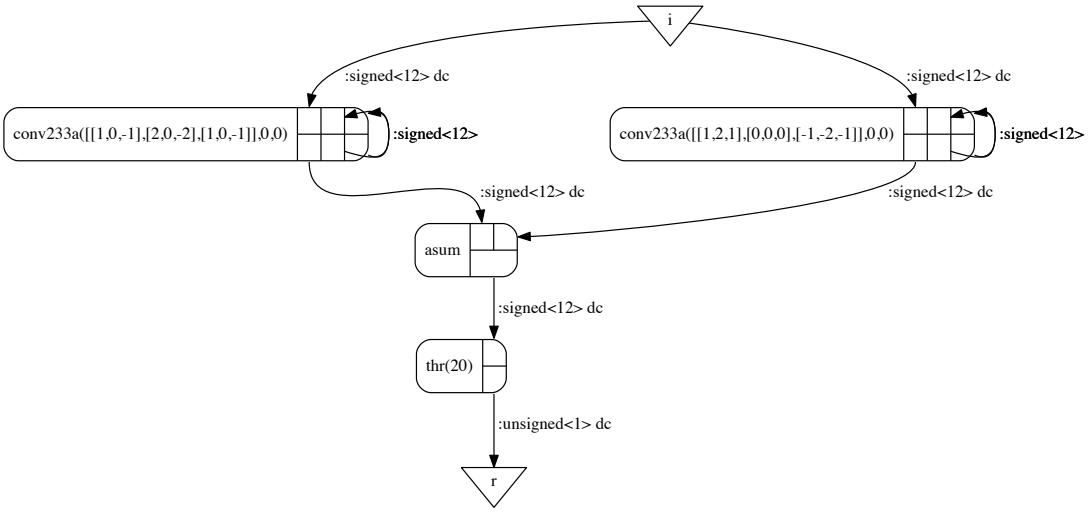


Figure 3.4: The graphical representation of the program given in Listing. 3.5

As for the application of the previous chapter, we will describe the implementation, simulation and synthesis of this application in part 3 of this document.

Part II

The Caph IDE

Introduction

This part describes the CAPH IDE. This IDE basically provides a Graphical user Interface (GUI) to the `caphc` compiler.

The CAPH IDE allows

- writing, reading and editing of CAPH programs,
- grouping all files associated to a CAPH program into *projects*,
- generating and viewing graphical representations of these programs,
- running simulations of these programs,
- generating SystemC and VHDL code.

Note. This document describes the Windows version of the CAPH IDE. The IDE can also be built and used on Unix-based systems (Linux, MacOS).

Chapter 4

Basic usage

We will illustrate how to write, compile and simulate with the CAPH IDE with a very simple CAPH program, even simpler than that used in Part 1. This program is reproduced in Listing 4.1. It involves a single actor, named `scale`, which multiplies by `k` each value read on its input port `i` and writes the result on its output port `o`. This actor is instanciated once, with `k=2`, and will read inputs from file `sample.txt` and write outputs to file `result.txt`.

Listing 4.1: A very simple program for testing the CAPH IDE

```
actor scale (k: unsigned<8>)
  in (i:unsigned<8>)
  out (o:unsigned<8>)
rules i -> o
| x -> k*x
;

stream inp:unsigned<8> from "sample.txt";
stream outp:unsigned<8> to "result.txt";

net outp = scale 2 inp;
```

First, **launch the CAPH application** by clicking on its icon in the installation directory or directly from the Windows *Start* menu.

The application main window is shown in Fig. 4.1. The main elements are (with corresponding areas labeled in red in Fig. 4.1) :

1. a menubar
2. four buttons for file manipulation; from left to right
   - create a new file,
   - open an existing file,
   - save a file,
   - save all files.
3. five buttons to invoke the compiler for
   - generating the dataflow graph representation of the current program and visualize it (button GRAPH),
   - simulating the current program and visualize it (button SIMU),
   - generating SystemC code from the current program (button SYSTEMC),
   - generating VHDL code from the current program (button VHDL),

- generating XDF representation of the current program (button XDF).

4. a tree view of the current project,
5. a tab for viewing and editing input source files,
6. a tab for viewing output files,
7. a log area, displaying issued command and outputs from the compiler.

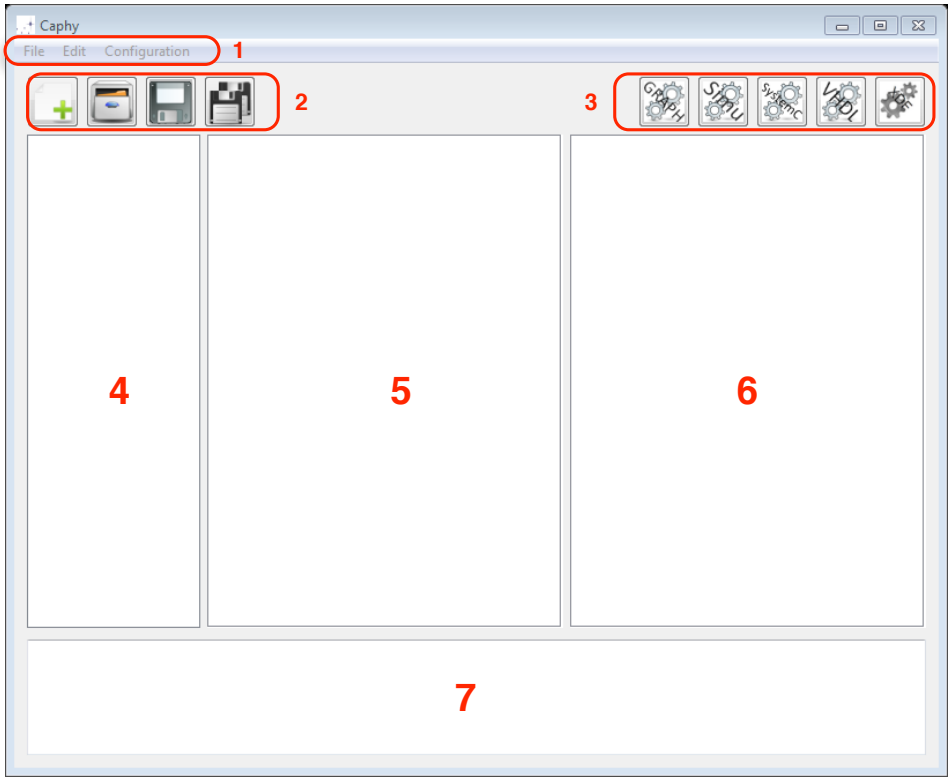


Figure 4.1: CAPH IDE main window

Invoke the [Configuration:Compiler and Tools] menu item and check that the specified paths are right (see Fig. 4.2). They should respectively point to

- the location of the `caphc` compiler (`<install>/bin/caphc`, where `<install>` is the CAPH installation directory, as specified during the installation process),
- the location of the program to invoke for viewing `.dot` graph files,
- the location of the program to invoke for viewing `.pgm` image files.

If the specified paths are not correct[1], adjust them and click Ok.

Create a new source file by clicking on the New file button (upper left) or invoking the corresponding item of the File menu. A new tab will appear, named `new` in the input files tab area. In this text tab, type[2] the program reproduced in Listing 4.1, as illustrated in Fig. 4.3.

Save the program by clicking on the Save file button or invoking the corresponding item of the File menu. Be sure to use the `.cph` filename suffix. Here we have saved it under name `main.cph`

To **generate the graph**, clicking the Graph button (upper right). This will

[1]This may be the case, for example, if you have changed the program to view graphs and/or images since CAPH was installed.
[2]Or copy-paste

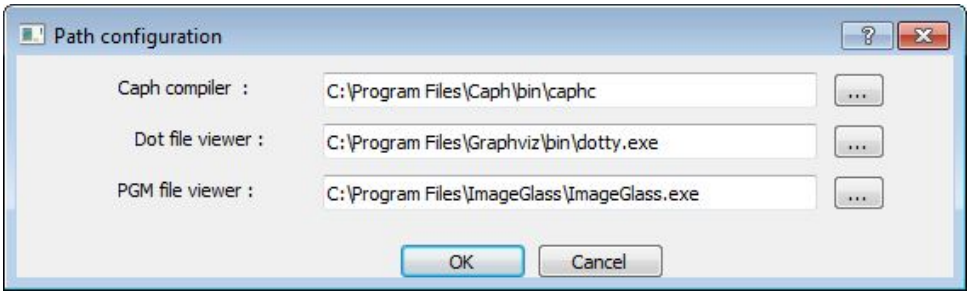


Figure 4.2: Path configuration window

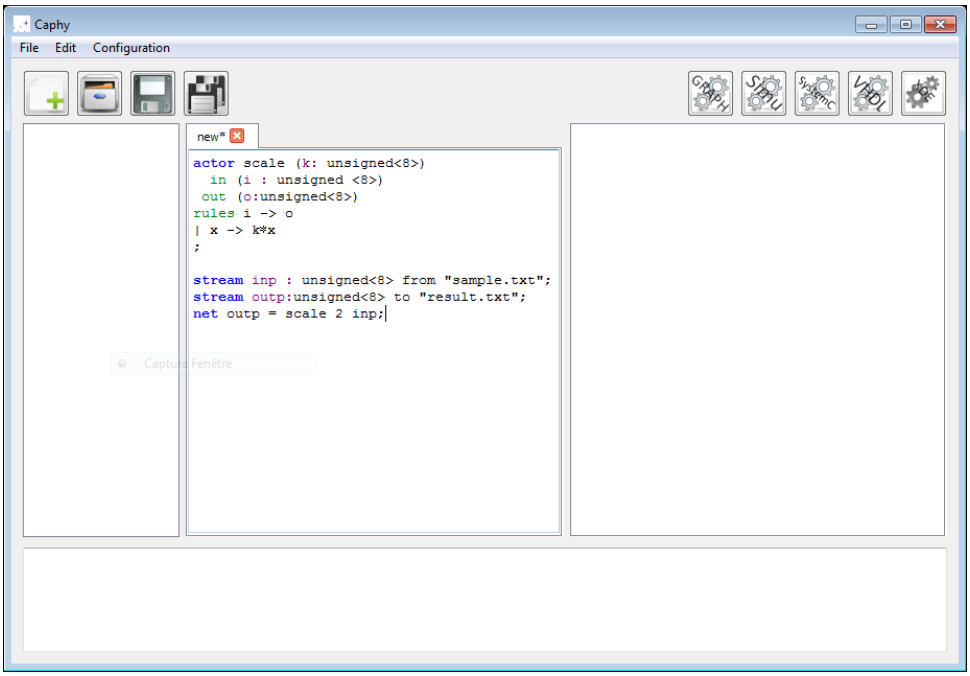


Figure 4.3: Entering program

- invoke the CAPH compiler with the adequate option(s),
- generate the `.dot` result file (in the same directory as the source file),
- view this result by invoking the graph visualisation program specified in [Configuration : Compiler and Tools] window.

The result is displayed in Fig. 4.4.

For **simulating the program**, we first need to create the file `sample.txt` containing the input tokens. Click on the New File button and type, for example, the following line in the newly created file tab :

```
1 2 3 4
```

Save the file under name `sample.txt` in the directory containing the caph source file (see Fig. 4.5).

Go back to the CAPH source file by selecting the corresponding tab[3] and invoke the compiler by clicking on the SIMU button. This will run the program, generate results in the file `result.txt`[4] and display the latter in a separate tab, as shown in Fig. 4.6.

[3] Simulation will not work otherwise !

[4] As specified by the `stream ... from` line in the program.

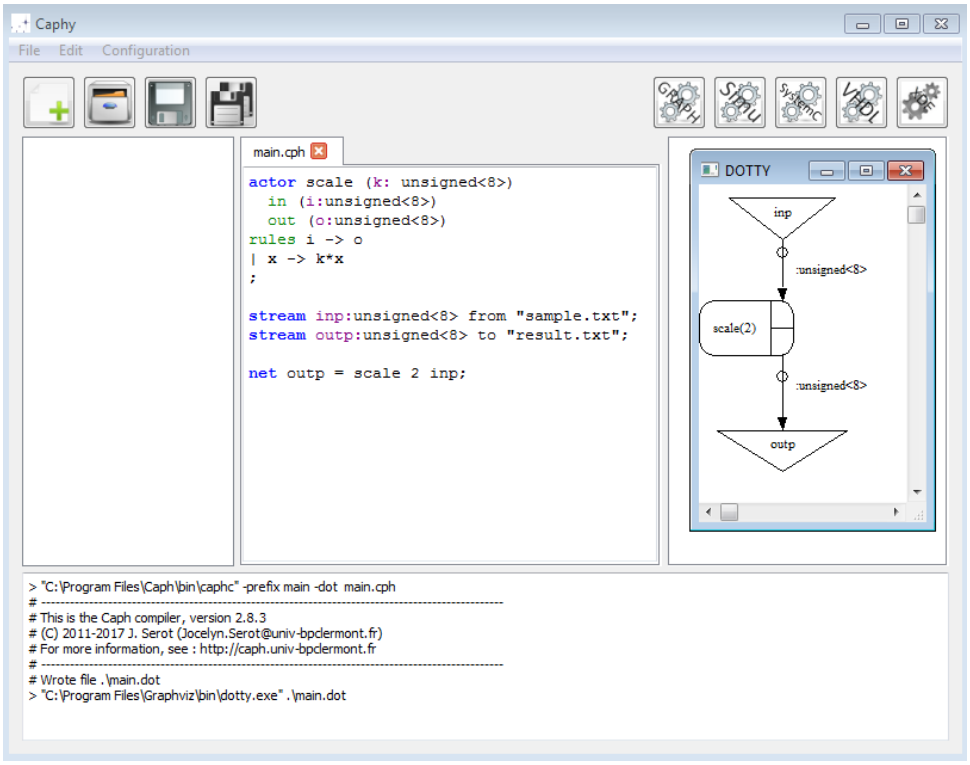


Figure 4.4: Viewing the dataflow graph of the program

For **generating the SystemC, VHDL or XDF** representation of the program, follow the procedure described for generating the graph representation :

1. select the tab containing the source program
2. click on the SystemC (resp. VHDL, resp. XDF) button

The result files will be generated in the same directory and displayed as separate tabs on the right, as illustrated in figures. 4.7 and 4.8 respectively.

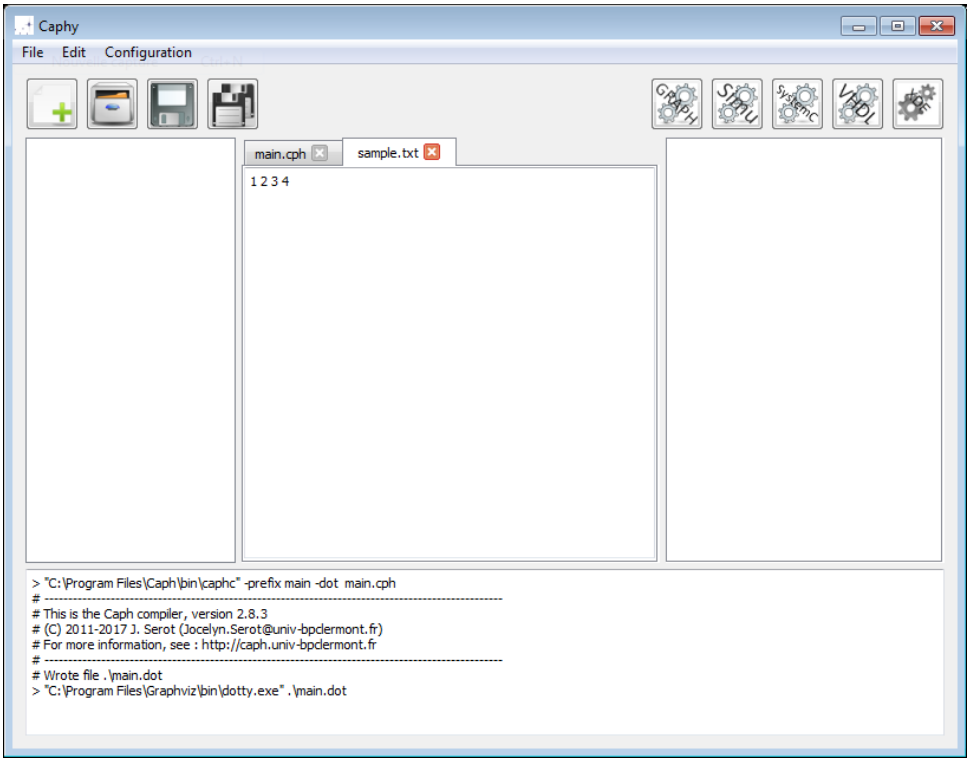


Figure 4.5: Writing the input data file for simulation

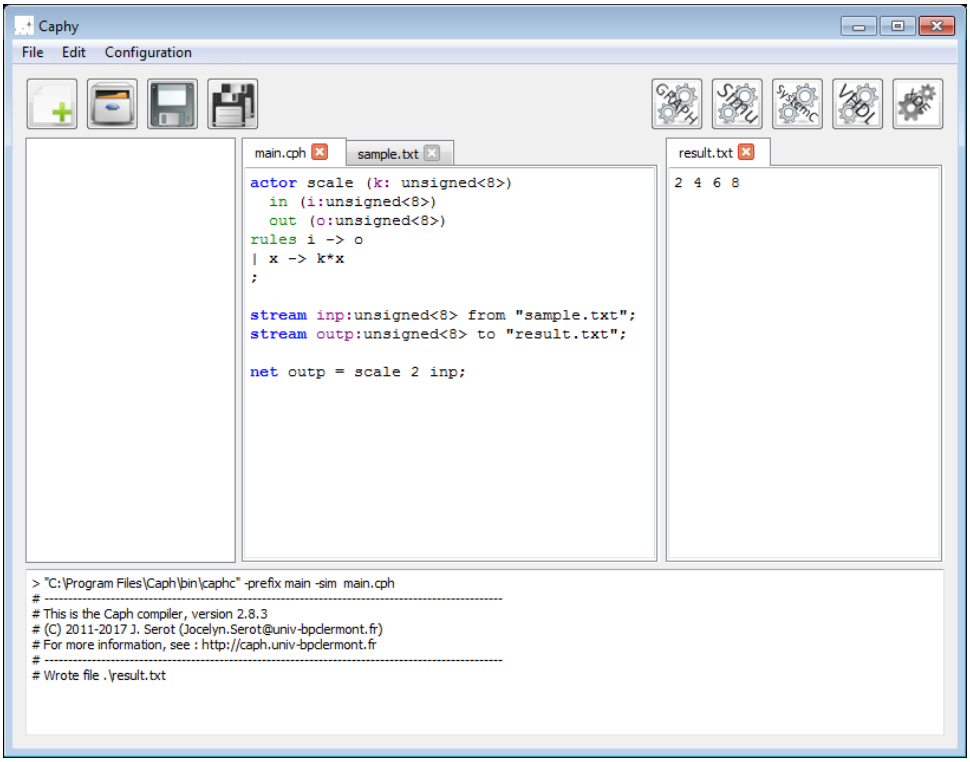


Figure 4.6: Viewing simulation results

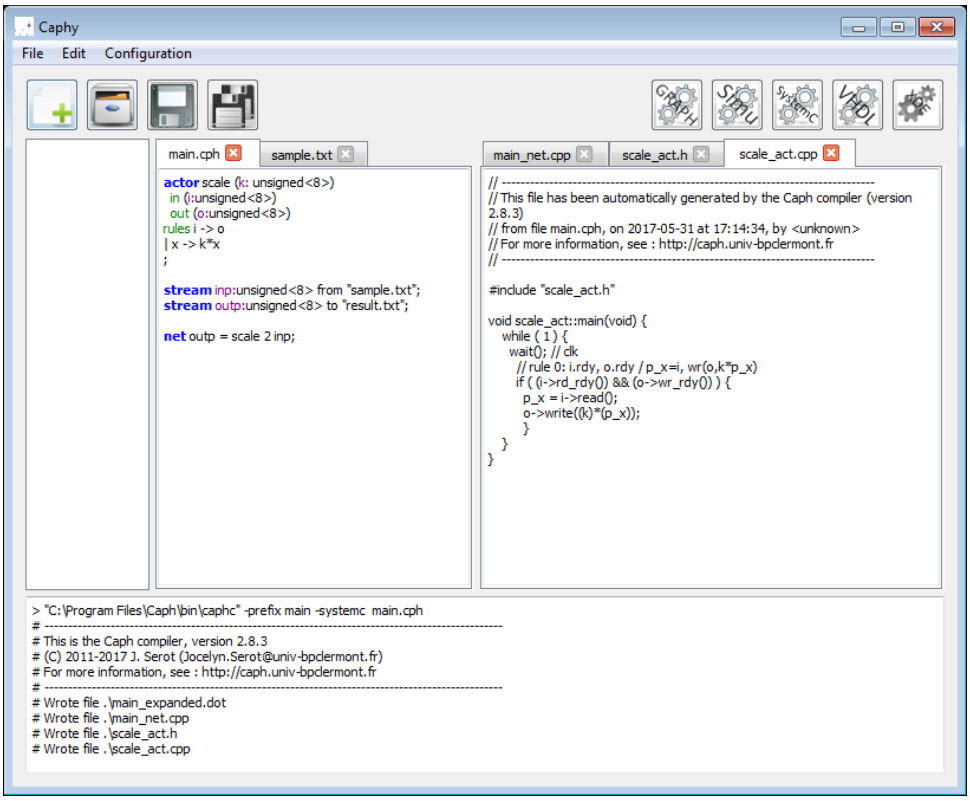


Figure 4.7: After generating SystemC code

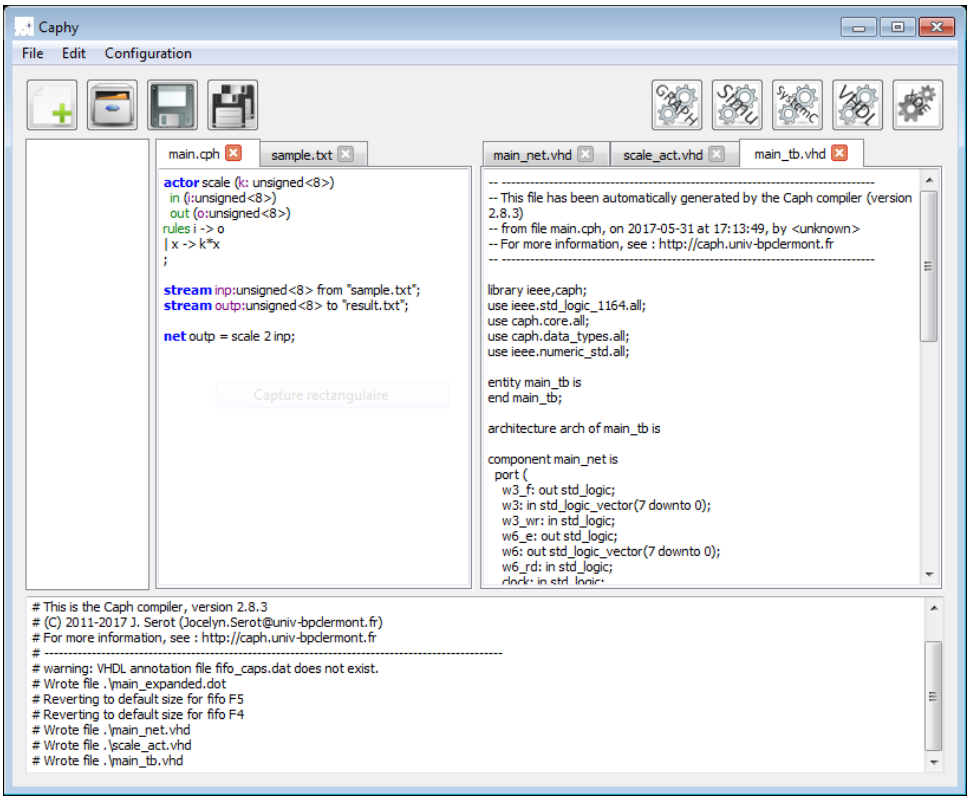


Figure 4.8: After generating VHDL code

Chapter 5

Working with projects

CAPH IDE provides a simple way of organizing files related to a given application within an entity called a *project*. Technically, a *project* is nothing but a directory gathering all files related to an application. This includes CAPH source files, input data files for simulation, files saving compiler options and a collection of subdirectories containing the files produced by the compiler in graph, simulation, SystemC or VHDL mode. Having a separate directory for each mode makes interfacing to external tools – C++ compiler, VHDL simulators and synthetizers in particular – easier.

In this chapter we will describe first how to create new projects and second how to use existing projects.

5.1 Creating a project

For simplicity, the created project will include a single source file, similar to that used in chapter 4.

. **Create a new project** by invoking the corresponding item in the File menu. In the displayed dialog (Fig. 5.1) give a a name to the project and specify a directory to host it. For example, if the name is `myproj` and the root directory `C:\Users\Bob\Desktop`, then all the files related to the project will be stored in directory `C:\Users\Bob\Desktop\myproj`. If the projet needs additionnal, pre-existing source or data files, add them in the corresponding text box or using the provided button. These files will be automatically copied in the project directory. No additionnal file is needed here.

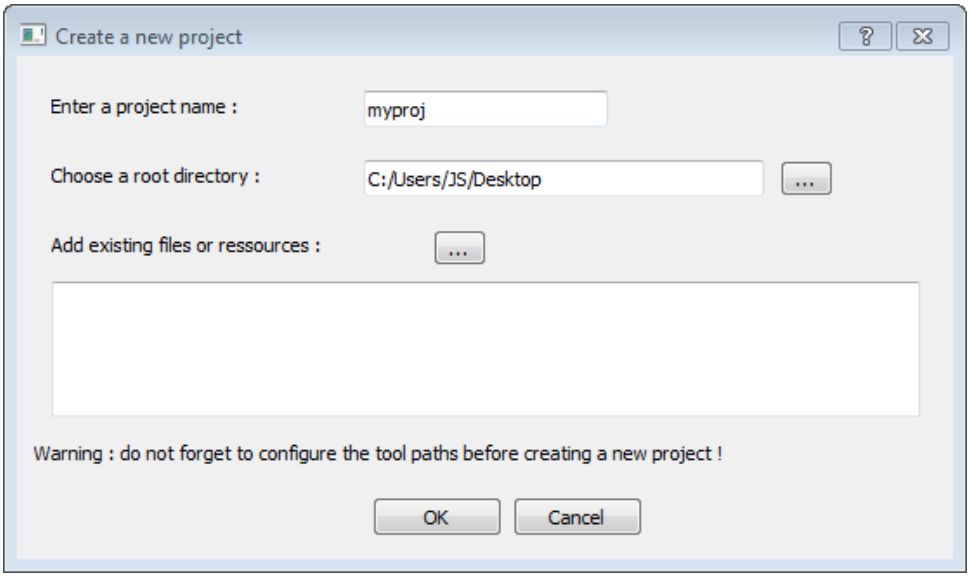


Figure 5.1: The dialog shown when creating a new project

When a project `myproj` is created, a "main" source file is created with name `main.cph` in the project directory and a file tab for editing is file is created (see Fig. 5.2). Type the CAPH source code of your program here[1] and save it (see Fig. 5.3).

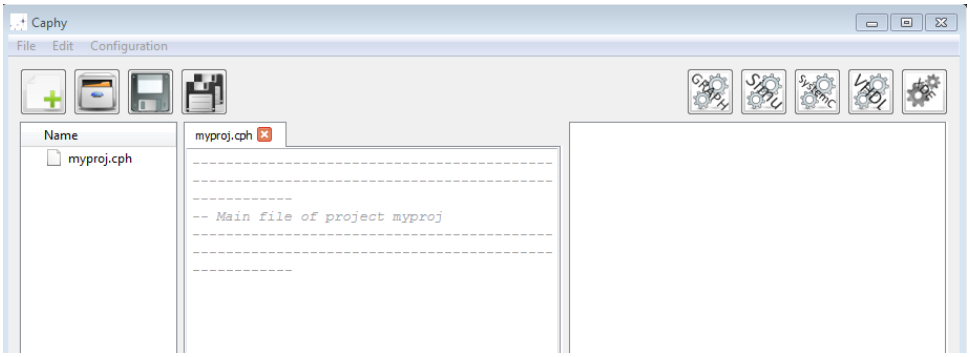


Figure 5.2: Ready to edit the project main source file

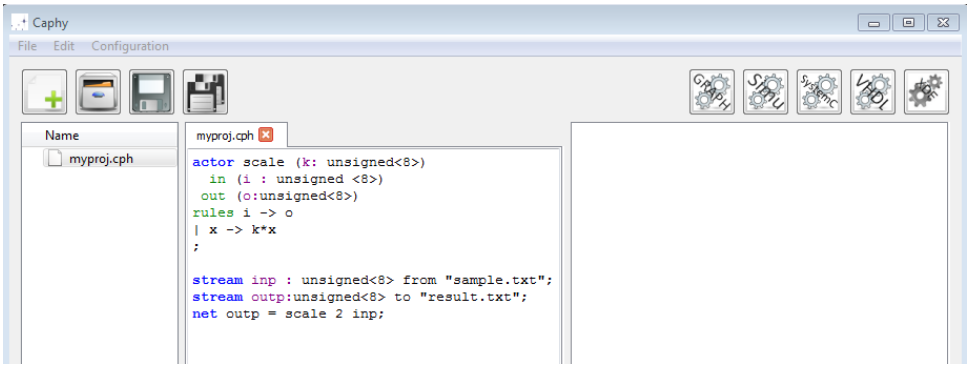


Figure 5.3: Main source file completed

From now, each compile action will

- implicitely operate on the project main source file,
- generate results in a specific directory (`dot` for graph, `simu` for simulation, `systemc`, `vhdl` and `xdf`).

The project tree representation (on the left) will be automatically updated to reflect the effect of each compile action. Navigation within this tree is of course allowed and double clicking on an element will open the corresponding file in a distinct tab (if not already opened).

For example, Fig. 5.4 display the GUI after clicking the Graph and the SystemC compile buttons. The complete list of generated files for each step can be viewed by clicking on the respective subdirectory in the tree view on the left.

5.2 Opening an existing project

To open an existing projec, invoke the Open Project item of the File menu and specify the name of the project description file (ending with the `.cphpro` extension), located in the project directory.

For example, Fig. 5.5 shows the IDE just about to open the project located in `primer/simple` directory which can be found in the examples provided with the CAPH distribution[2]. This project corresponds to the program described in Part 1 of this document.

[1] If you already have the source code, you can of course copy it and paste it.
[2] These examples have here been installed in `Documents/CaphExamples`.

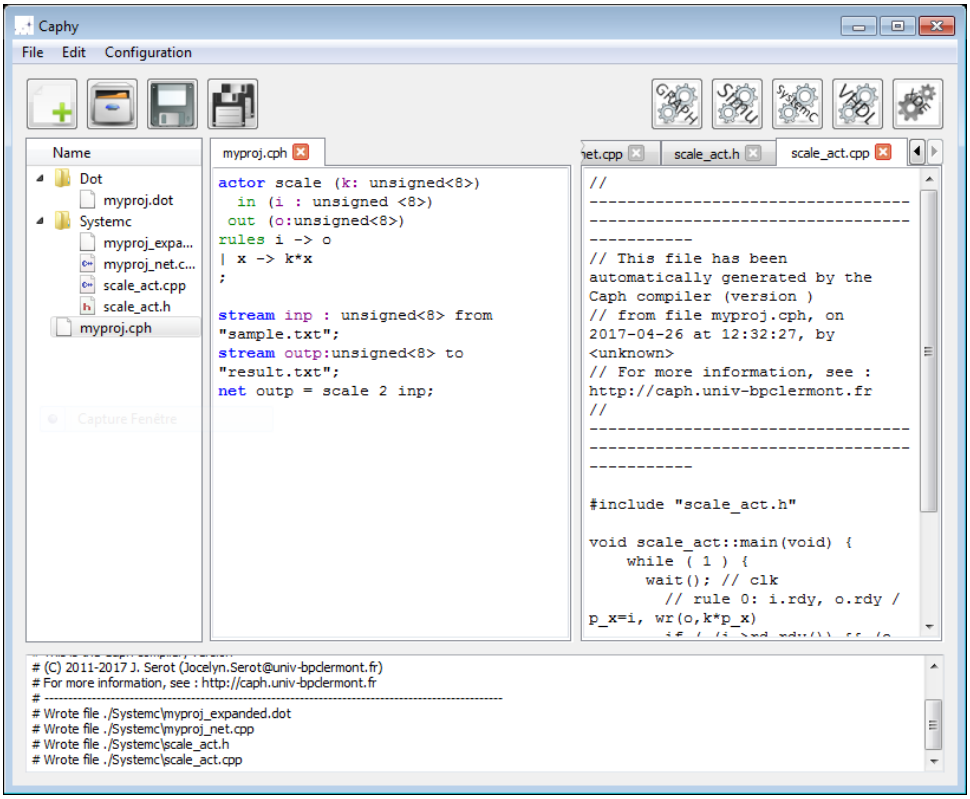


Figure 5.4: After clicking the Graph and SystemC buttons in project mode

Fig. 5.6 shows the IDE just after opening this project.

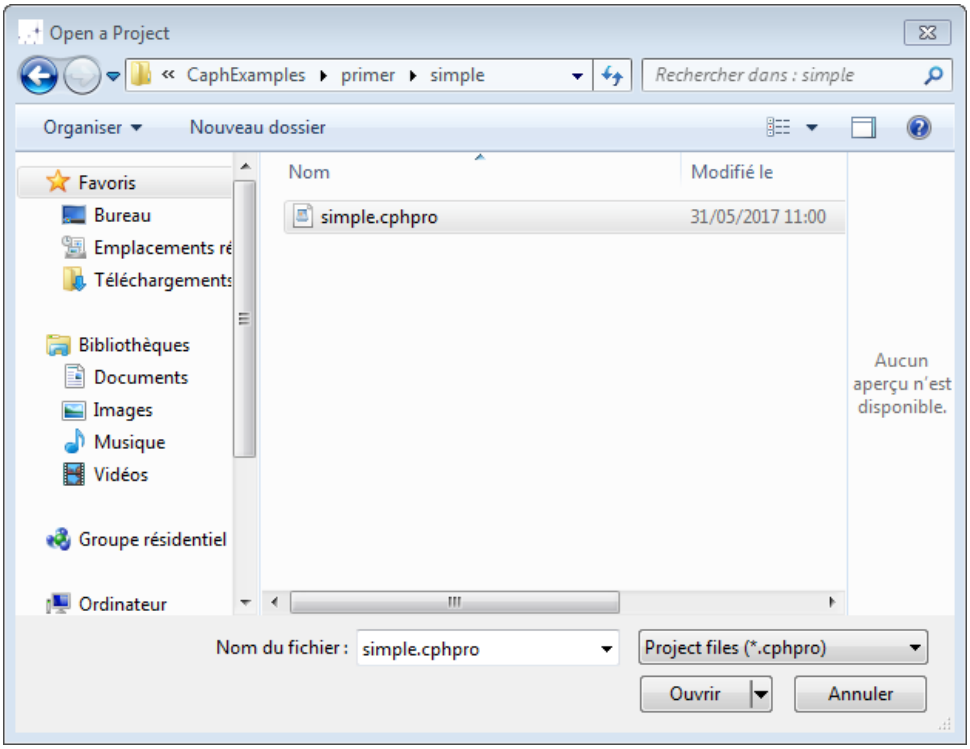


Figure 5.5: About to open the `Primer` project

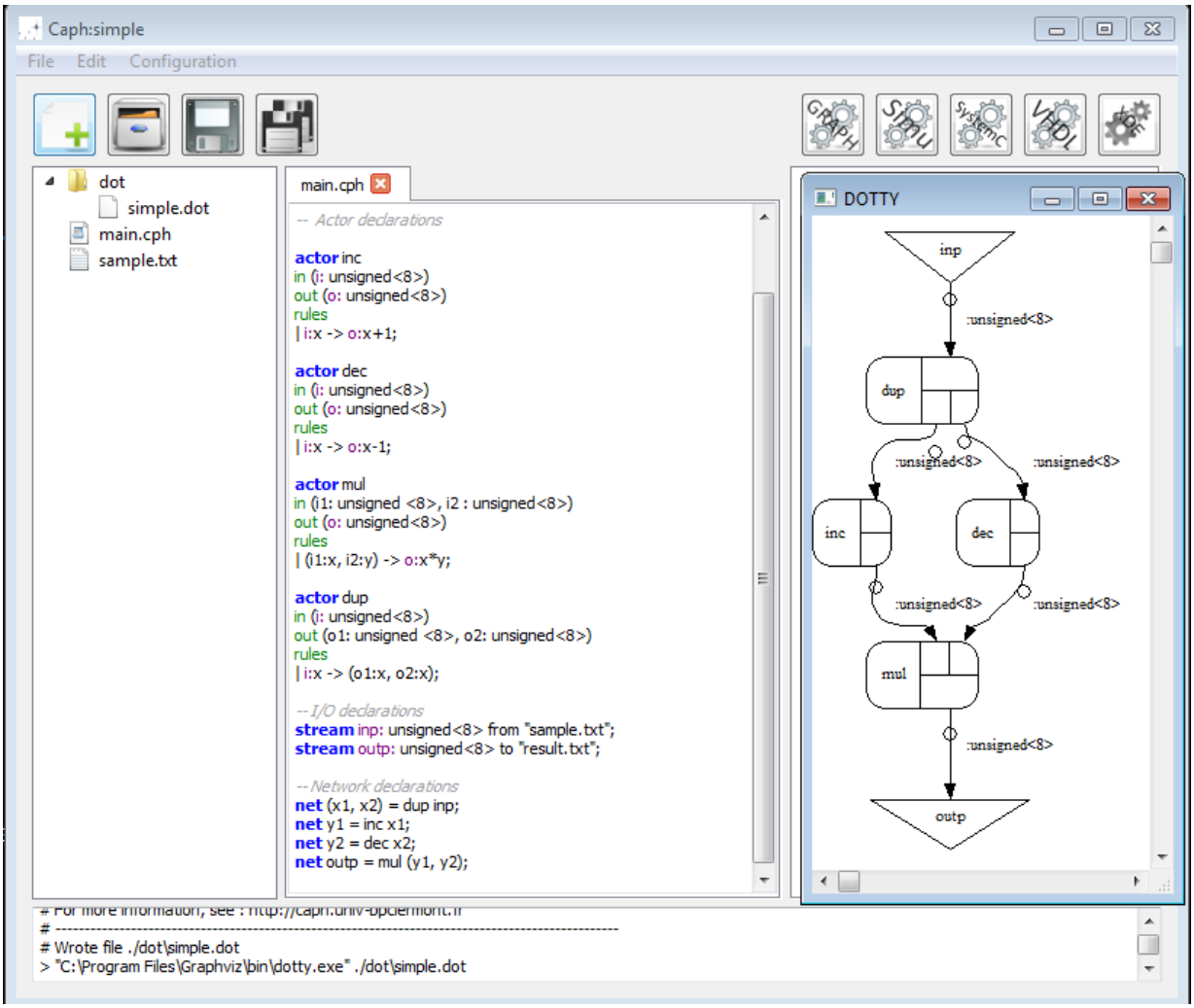


Figure 5.6: The `Primer` project opened

Chapter 6

Compilation options

The caph compiler comes with a fairly large number of options (see Sec. 12 of the language reference manual). Most of these options can be set and inspected by invoking Compilater options item of the Configuration menu. Options are organized by grouped by tabs, as illustrated in Fig. 6.1, in which the tab related to SystemC has been selected.

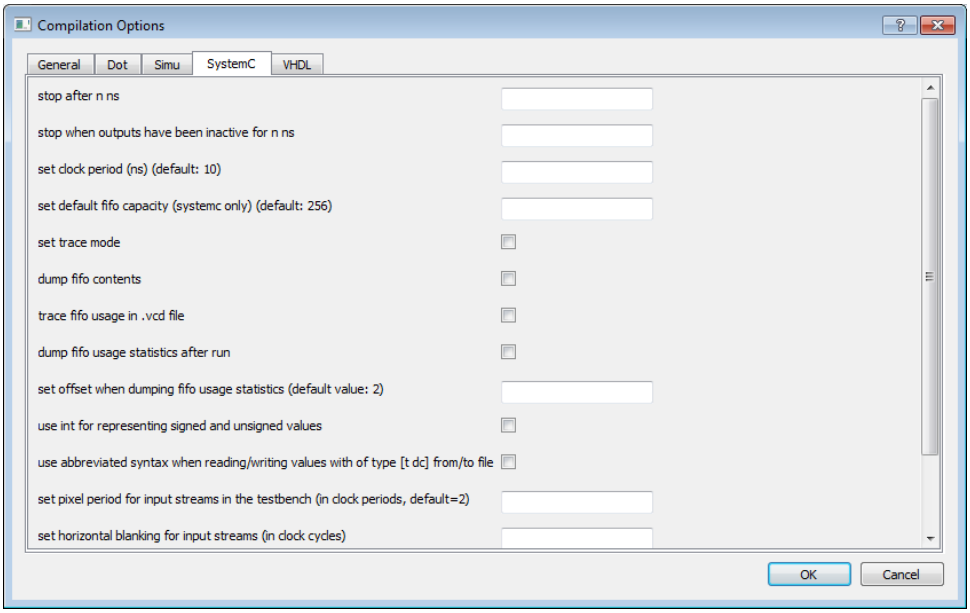


Figure 6.1: The options setting dialog

An important point is that, when working in project mode, each modification to compilation options is recorded and saved in a dedicated file in the project directory. This file, when present, is automatically read when a project is opened. This way, compilation options are remembered between sessions.

Part III

Makefile-based design with Caph

Introduction

This part describes how to use CAPH in a command line based environment, using `Makefile`s. On Unix-like platforms like Linux or MacOS, this is typically accomplished by running the corresponding tools from within a command shell. On Windows, this can be done using Unix emulation systems like MinGW [8] or Cygwin [9]. As stated in the general introduction, although this approach may appear a bit more complicated than the former at first sight but it provides a way of integrating existing third-party tools, such as C++ compilers and VHDL synthetizers, in a fully automatized design flow.

This part assumes a basic familiarity with command line interfaces, shell programming and `make`-based compilation flows. Aside, a knowledge in digital design (and of the VHDL language) will help to appreciate the final products of the CAPH toolset. Sections describing the synthesis of VHDL code on FPGA requires a previous knowledge of the ALTERA Quartus II environment.

The following typographic conventions are followed :

- source code is written in gray-shaded boxes, like this :

```
-- CAPH source code will appear here
```

- `makefile`s are written in pink-shaded boxes, like this :

```
-- Makefiles will appear like this
```

- shell input (on the command line) is written like this (the character `#` is the shell prompt) :

```
# command
```

- shell output is written like this :

```
shell output
```

Chapter 7

Using the caphc compiler

In this chapter, we will show how to invoke to `caphc` compiler from the command line in order to

- generate and view the dataflow graph corresponding to a program,
- simulate this program,
- generate SystemC and VHDL code.

The program used as example will be the one introduced in Part 1 and given in Listing 1.1. We assume that the corresponding source code has been placed in a file named `simple.cph`.

7.1 Configuring

Add a variable named `CAPH`, pointing to the root of your local CAPH installation, to your environment. For example (with a Bash shell) :

```
# CAPH=/usr/local/caph; export CAPH
```

Add `$CAPH/bin` to your `$PATH` environment, so that CAPH commands can be found :

```
# PATH=$CAPH/bin:$PATH; export $PATH
```

7.2 Viewing the dataflow graph

From the directory containing the source file, type, from a shell, the following command :

```
# caphc -dot simple.cph
```

Executing this command yields the following output

```
-------------------------------------------------------------------------------------------
This is the Caph compiler, version 2.8.3
(C) 2011-2017 J. Serot (Jocelyn.Serot@univ-bpclermont.fr)
For more information, see : http://caph.univ-bpclermont.fr
-------------------------------------------------------------------------------------------
Wrote file ./simple.dot
```

and produces the graphical representation of the program in file named `simple.dot`. This file is in the DOT format and can be visualized with the `graphviz` suite of tools [2]. Under MacOS, launch the `Graphviz` application and open the corresponding file[1]. Under Windows, use the `dotty` application. The resulting graph is shown in Fig. 7.1. The four involved actors can be readily recongnized. Wires are labeled with the types of

[1] Alternatively, from a terminal, type `open -a Graphviz simple.dot`.

the conveyed values (the type of intermediate wires is automatically inferred by the compiler). Input and output wires are drawn as triangles. Several options of the compiler allow the aspect of this graphical representation and the amount of displayed informations to be adjusted.

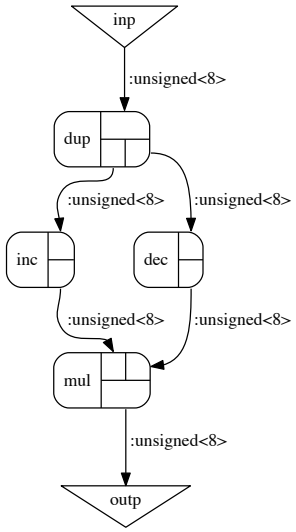


Figure 7.1: The graphical representation of the program given in Listing. 1.1 computed by the CAPH compiler front-end

7.3 Simulating the program

There are actually two ways of simulating programs : either directly from the source code, using the reference interpreter of the language[2], or by using the SystemC backend.

In both cases, input(s) and output(s) will be read (resp. written) to text files[3]. Input text files can be simply written by hand or generated from other data representations (images, in particulary) using ad-hoc conversion programs provided in the CAPH distribution (see Sec. 9.5 of the reference manual).

In our case, and in accordance to the `stream` declarations written in file `simple.cph`, the input file will be named `sample.txt` and the output file `result.txt`. The file `sample.txt` simply contains the sequence of input tokens (unsigned integers in this particular case, as shown in Listing 7.1 :

Listing 7.1: The input file `sample.txt` used for simulating the program of Listing 1.1

```
1 2 3 4 5 6 7 8
```

7.4 Simulation using the interpreter

Simulation is launched by invoking the compiler with the `-sim` option :

```
# caphc -sim simple.cph
```

[2]As demonstrated in Part 2, with the IDE

[3]Of course, for the final application, running the on target hardware platform, there will have to be a builtin mechanism for producing the stream(s) of input tokens and consuming the stream(s) of output tokens. In practice, this mechanism will take the form of a couple of dedicated VHDL processes reading and writing values from/to the I/O devices attached to the hardware platform (video cameras, digital display, host pc interface, ...).

Executing this command yields the following output

```
------------------------------------------------------------
This is the Caph compiler, version 2.8.3
...
Wrote file ./result.txt
------------------------------------------------------------
```

The contents of the file `result.txt` is given in Listing 7.2.

Listing 7.2: The output file `result.txt` generated when simulating the program of Listing 1.1 with the input file of Listing 7.1

```
0  3  8  15  24  35  48  63
```

7.5 Simulation using the SystemC backend

This actually requires three steps : first generating the SystemC code representing the application, second compiling this code to produce an executable and finally running this executable.

The whole process is greatly simplified by using the `caphmake` utility program included in the distribution[4]. This program automatically generates Makefile descriptions from *project* descriptions, describing the application-specific parameters. A detailed presentation of `caphmake` can be found in Sec. 9.10 of the Reference Manual. We will here only illustrate its basic usage.

Listing 7.3 shows a very simple project file for compiling and running the SystemC code derived from the `simple.cph` program. The `SC_OPTS` macro gives the options to pass the SystemC backend of the `caphc` compiler. Here the option `-sc_stop_time` specifies the duration of the simulation in ns[5].

Listing 7.3: File `simple.proj` for compiling and running SystemC code

```
SC_OPTS = -sc_stop_time 200
```

After writing `simple.proj`, just invoke `caphmake` with the name of the main source file :

```
# caphmake -main simple
```

This will write a file named `Makefile` in the current directory.

Now use this top-level Makefile to generate the SystemC-specific makefile :

```
# make systemc.makefile
```

This will write a file named `Makefile.systemc` in the current directory.

We now can generate the SystemC code by simply typing

```
# make systemc.code
```

yielding the following output

```
1 make -f Makefile.systemc code CAPH=/usr/local/caph
2 /usr/local/caph/bin/caphc -I /usr/local/caph/lib/caph -systemc -sc_stop_time 200 simple.cph
3 ------------------------------------------------------------------------------------------------
4 This is the Caph compiler, version 2.8.3
5 (C) 2011-2017 J. Serot (Jocelyn.Serot@univ-bpclermont.fr)
6 For more information, see : http://caph.univ-bpclermont.fr
7 ------------------------------------------------------------------------------------------------
8 Wrote file ./simple_expanded.dot
9 Wrote file ./simple_net.cpp
```

[4]Since version 2.8.1.

[5]The complete list of options is given in the language reference manual.

```
10 Wrote file ./dup_act.h
11 Wrote file ./dup_act.cpp
12 Wrote file ./mul_act.h
13 Wrote file ./mul_act.cpp
14 Wrote file ./dec_act.h
15 Wrote file ./dec_act.cpp
16 Wrote file ./inc_act.h
17 Wrote file ./inc_act.cpp
```

Line 2 shows the invocation of the CAPH compiler with the SystemC backend. Lines 8–17 show the different files generated by the CAPH compiler. The file `simple_expanded.dot` is a variant of the file `simple.dot` discussed in Sec. 7.2[6]. The file `simple_net.cpp` contains the top-level network description. The files `dup_act.h` and `dup_act.cpp` (resp. `mul_act.h` and `mul_act.cpp`, `dec_act.h` and `dec_act.cpp`, `inc_act.h` and `inc_act.cpp`) contain the interface and the implementation of the `dup` (resp. `mul`, `dec` and `inc`) actor.

The generated code can be compiled by simply typing

```
# make systemc.exe
```

yielding the following output, which shows the compilation of this code using the classical SystemC flow (in our case, `gcc`, with link to the `systemc` library[7]).

```
1  make -f Makefile.systemc exe CAPH=/usr/local/caph
2  (cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc -I/usr/local/systemc-2.3.1/include
3    -Wno-deprecated -Wno-parentheses-equality -D_CPP11 -c `basename inc_act.cpp`)
4  (cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc -I/usr/local/systemc-2.3.1/include
5    -Wno-deprecated -Wno-parentheses-equality -D_CPP11 -c `basename dec_act.cpp`)
6  (cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc -I/usr/local/systemc-2.3.1/include
7    -Wno-deprecated -Wno-parentheses-equality -D_CPP11 -c `basename mul_act.cpp`)
8  (cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc -I/usr/local/systemc-2.3.1/include
9    -Wno-deprecated -Wno-parentheses-equality -D_CPP11 -c `basename dup_act.cpp`)
10 (cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc -I/usr/local/systemc-2.3.1/include
11   -Wno-deprecated -Wno-parentheses-equality -D_CPP11 -c `basename simple_net.cpp`)
12 (cd .; g++ -L/usr/local/systemc-2.3.1/lib-macosx64 inc_act.o dec_act.o mul_act.o dup_act.o simple_net.o
13    -o simple_sc -lsystemc 2>&1 | c++filt)
```

Finally, simulation is launched by running the compiled executable

```
# make systemc.run
```

yielding the following output

```
1  make -f Makefile.systemc run CAPH=/usr/local/caph
2  ./simple_sc
3
4         SystemC 2.3.1-Accellera --- Aug 9 2015 15:42:56
5         Copyright (c) 1996-2014 by all Contributors,
6         ALL RIGHTS RESERVED
7  Simulation stopped at t=200 ns
8  Wrote file result.txt
```

The generated file `result.txt` contains the results of the simulation (which is exactly the same as the one obtained when running the source level simulator).

The three different steps (code generation, compilation, execution) can be run with a simple command by simply typing `make systemc.run` directly after invoking `caphmake`.

[6] This variant, only required by the SystemC and VHDL backend, has explicit FIFO and flow-splitting nodes.

[7] Appropriate definitions are provided in the file `$CAPH/lib/etc/Makefile.core` and can be adjusted according to your local SystemC installation.

7.6 Generating and simulating VHDL code

The CAPH compiler can produce a complete RT-level VHDL representation of the application which can be simulated and, latter synthetised using vendor specific tools (such as ALTERA Quartus or XILINX ISE). This section focuses on simulation (synthesis will be covered in Sec. 7.7).

Classicaly, simulation of VHDL code is performed using dedicated simulators included in the vendor toolsets (for example, the ALTERA Quartus toolset includes the `modelsim` simulator). We describe here another approach, using a freely available VDHL compiler and simulator called GHDL [3]. GHDL can be invoked directly from the command line and hence can be easily integrated in a makefile-based design-flow.

As for the SystemC backend, the **first step** is to define, in the project description file (`.proj`), all the application-specific options. In our case, the only thing to do is add a line dedicated to the VHDL backend in the file described in Listing 7.3, as shown in Listing 7.4.

Listing 7.4: File `simple.proj` for compiling and running SystemC and VHDL code

```
SC_OPTS = -sc_stop_time 200
GHDL_RUN_OPTS = --stop-time=200ns
```

The added line (line 2) specifies to options to be passed to the GHDL simulator.

After that, the process is very similar to that described in the previous section for the SystemC backend :

- invoke `make vhdl.makefile` to build the VHDL-specific Makefile,
- invoke `make vhdl.code` to generate the VHDL code,
- invoke `make vhdl.exe` to build the executable,
- invoke `make vhdlrunexe` to run the simulation.

As before, the three last steps can be obtained by simply typing

```
# make vhdl.run
```

yielding, in our case, the following output

```
1  make -f Makefile.vhdl run CAPH=/usr/local/caph
2  /usr/local/caph/bin/caphc -I /usr/local/caph/lib/caph -vhdl simple.cph
3  ------------------------------------------------------------------------------------------
4  This is the Caph compiler, version 2.8.3
5  (C) 2011-2017 J. Serot (Jocelyn.Serot@univ-bpclermont.fr)
6  For more information, see : http://caph.univ-bpclermont.fr
7  ------------------------------------------------------------------------------------------
8  Wrote file ./simple_expanded.dot
9  Reverting to default size for fifo F12
10 Reverting to default size for fifo F11
11 Reverting to default size for fifo F10
12 Reverting to default size for fifo F9
13 Reverting to default size for fifo F8
14 Reverting to default size for fifo F7
15 Wrote file ./simple_net.vhd
16 Wrote file ./dup_act.vhd
17 Wrote file ./mul_act.vhd
18 Wrote file ./dec_act.vhd
19 Wrote file ./inc_act.vhd
20 Wrote file ./simple_tb.vhd
21 warning: VHDL annotation file fifo_caps.dat does not exist.
22 (cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename simple_tb.vhd`)
23 (cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename inc_act.vhd`)
24 (cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename dec_act.vhd`)
```

```
25 (cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename mul_act.vhd`)
26 (cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename dup_act.vhd`)
27 (cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename simple_net.vhd`)
28 (cd .; ghdl -e -P/usr/local/caph/lib/vhdl simple_tb)
29 /usr/local/caph/bin/txt2bin uint 8 sample.txt > sample.bin
30 ghdl -r -P/usr/local/caph/lib/vhdl simple_tb --stop-time=200ns --vcd=simple_tb.vcd
31 ./simple_tb:info: simulation stopped by --stop-time
```

Line 2 shows the invocation of the CAPH compiler with the VHDL backend. The produced files are listed on lines 15–20. The file `simple_net.vhd` contains the top-level network description, The files `{dup_act.vhd`, `mul_act.vhd`, `dec_act.vhd` and `inc_act.vhd` contain the RTL description of the actors involved in this network. The file `simple_tb.vhd` contains a testbench for performing the simulation at the RTL level. Simulation itself is performed as shown in line 30.

Line 29 shows the invocation of the `txt2bin` utility program to generate the file `sample.bin` to be used as input for simulation. The reason for this is that the input data files read by the VHDL code use a special, text-encoded binary format. The `txt2bin` program is used to convert the input simulation file `sample.txt` to this format[8].

Simulation results are produced in file `result.bin`. This file is encoded using the same binary format and can be decoded using the `bin2txt` utility program. For this, it suffices to invoke

```
# make vhdl.show
```

This yields the following result, which shows the expected values in file `result.txt`

```
1 make -f Makefile.vhdl show CAPH=/usr/local/caph
2 /usr/local/caph/bin/bin2txt uint 8 result.bin > result.txt
3 result.txt: 0 3 8 15 24 35 48 63
```

The "functional" style of simulation illustrated above is in general sufficient for assessing the code. It is however possible to get a more "time oriented" view by using the `--vcd` option of the GHDL compiler. This option must be added to the macro `GHDL_RUN_OPTS` in the project file, as shown in Listing 7.5.

Listing 7.5: File `simple.proj` for compiling and running SystemC and VHDL code (2nd version)

```
SC_OPTS = -sc_stop_time 200
GHDL_RUN_OPTS = --stop-time=200ns --vcd=simple_tb.vcd
```

This instructs the GHDL simulator to dump a detailed *log* of the simulation in VCD format [5]. This log file can examined using various waveform visualisation programs. Fig. 7.2, for example, shows an excerpt of the log file as visualized by the `gtkwave` application [4]. Visualisation has been here limited to signals connected to the instance of the `inc` actor. One immediately spots the `clock` and `reset` signals[9]. The signal `i_empty` goes to 0 when a data is available on the FIFO connected to input `i` of the actor. Reading from the FIFO is then triggered by setting signal `i_rd` to 1. Symetrically, the signal `o_full` is 0 when place is available on the FIFO connected to output `o` of the actor. Writing to this FIFO is then triggered by setting signal `i_wr` to 1.

7.7 Synthetizing the VHDL code

By synthesis we mean the tranformation of the RT-level code generated by the CAPH compiler into a FPGA configuration. Contrary to simulation, this operation depends on the physical target device and requires the toolset from the corresponding vendor. We do not address the issue of physical I/O interfacing – *i.e.* we only describe the synthesis of the "core" functionality described by the CAPH network (integration of CAPH-generated code into a full-fledged hardware platform is can be carried out with the GpStudio IDE by example [7]).

[8]The extra arguments to the program, `uint 8` in this case, are infered from the type of the corresponding input stream. A complete description of the binary format and the associated converter programs is provided in the reference manual.

[9]The clock period is set by default to 10 ns. There's an option of the compiler to change it (see [1].

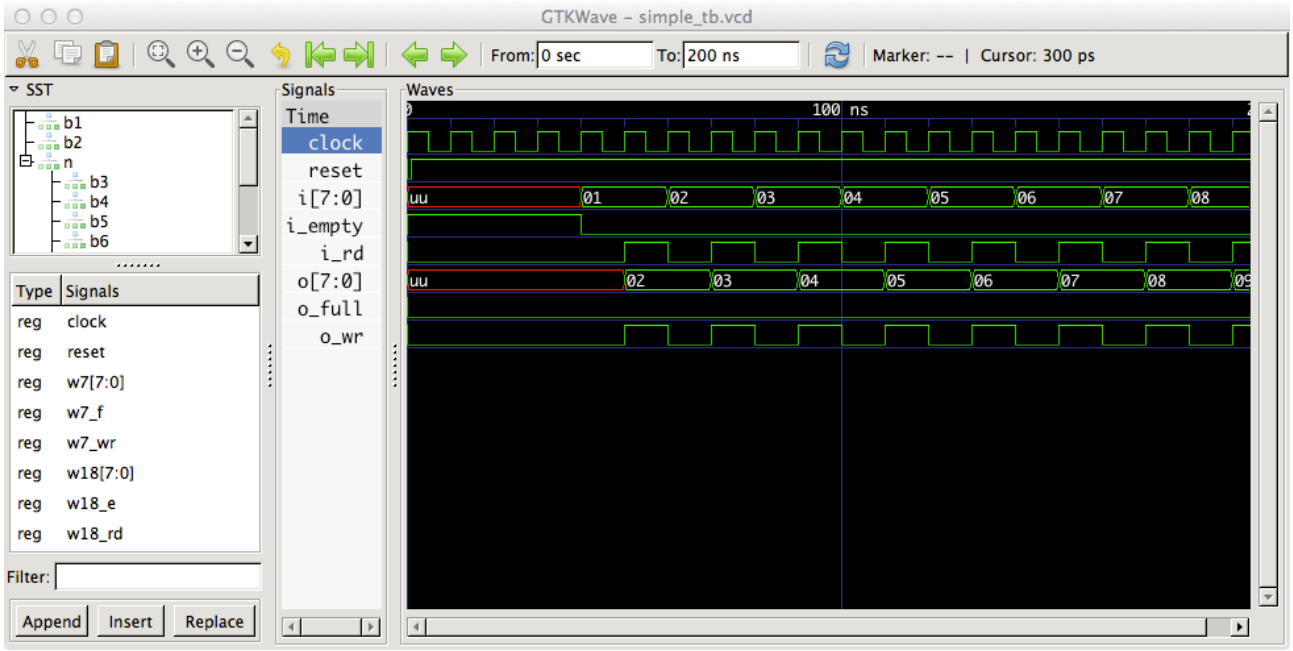


Figure 7.2: Some VHDL simulation results as viewed by `gtkwave`

In this section, we will illustrate the process with the Quartus II suite of tools from ALTERA, using the `simple` application[10].

Figs.7.3 to 7.6 illustrates the creation of the relevant project under the Quartus II (version 13.1) environment[11].

Fig. 7.3 shows the main Quartus window just after launching. In this window, select File in the top menu bar and then the New Project Wizard item.

A window named after this item pops up. Fill the requested text fields as illustrated in Fig. 7.4. In our case, we have copied all the VHDL files generated by the CAPH compiler in a separate directory named `Z:/vhdl/caph/simple`[12]. The name of the project and the name of the top-level design entity must be set to `simple_net`. Clicking on the Next butten then brings the window shown in Fig. 7.5.

In this window, using the ... and Add buttons, you have to specify the list of all the VHDL files included in the projet. In our case, two groups of files are added : the five files generated by the CAPH compiler : `dup_act.vhd`, `mul_act.vhd`, `dec_act.vhd`, `inc_act.vhd` and `simple_tb.vhd`; and two predefined files taken from the CAPH VHDL library : `../lib/caph.vhd` and `../lib/fifo_fb.vhd` (the former contains a set of types and functions related to the CAPH language, the latter the implementation of a generic FIFO). When completed, click again on the Next button.

This brings up the window shown in Fig. 7.6, in which you select the target device. In our case, a simple Cyclone III is chosen. Clicking then on the Finish button brings back to main window.

On the Project Navigator subwindow (top left), select Hierarchy to show the design hierarchy. Selecting an entity will then print the corresponding source file on the right subwindow, as illustrated in Fig. 7.7.

Synthesis is launched by selecting the Start Compilation item in the Processing menu (or simply by clicking the small right-oriented purple triangle in the toolbar). Depending on your machine this may take from a few seconds to a few minutes. In our case, the result is shown in Fig. 7.8. Here, it can be noted that only a very small fraction of the available hardware resources is used.

Fig. 7.9 shows the RT-level view of the design after synthesis[13]. This is obtained by invoking the Netlist

[10]This is only for pedagogical reasons since this application is obviously not a very useful one. Chap.8 and 9 will show how to implement more "realistic" applications, performing image processing.

[11]We make the assumption here that the reader has a minimum familiarity with this environment. Several good tutorials can be found online, in particular on the ALTERA website.

[12]The option `-vhdl_target_dir` of the compiler can be used for that purpose.

[13]Before physical mapping. It is also possible to get a post-mapping view.

viewer item in the Tools menu.

Figure 7.3: The Quartus II environment, just after launching

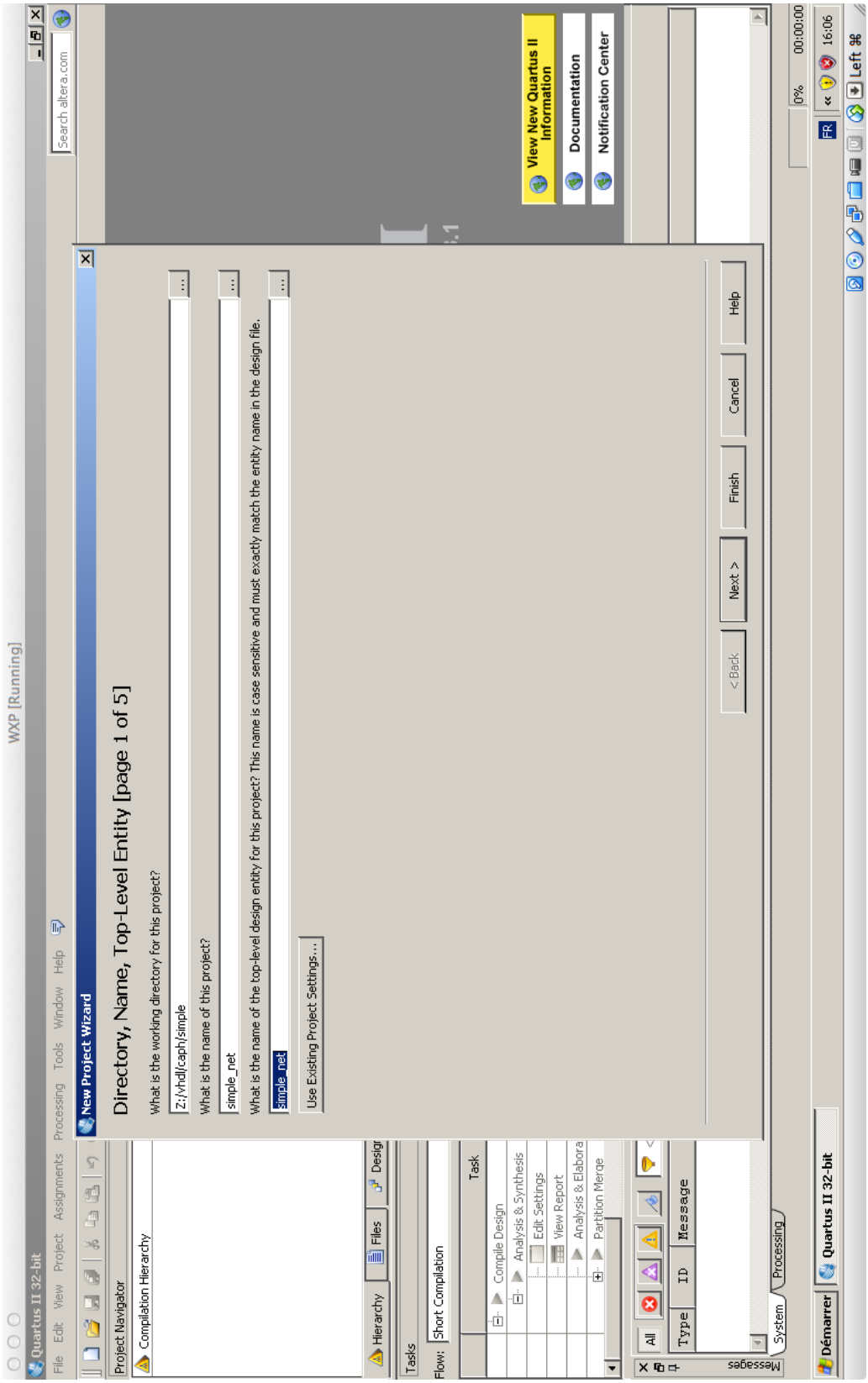


Figure 7.4: Setting the projet - directory and top entity selection

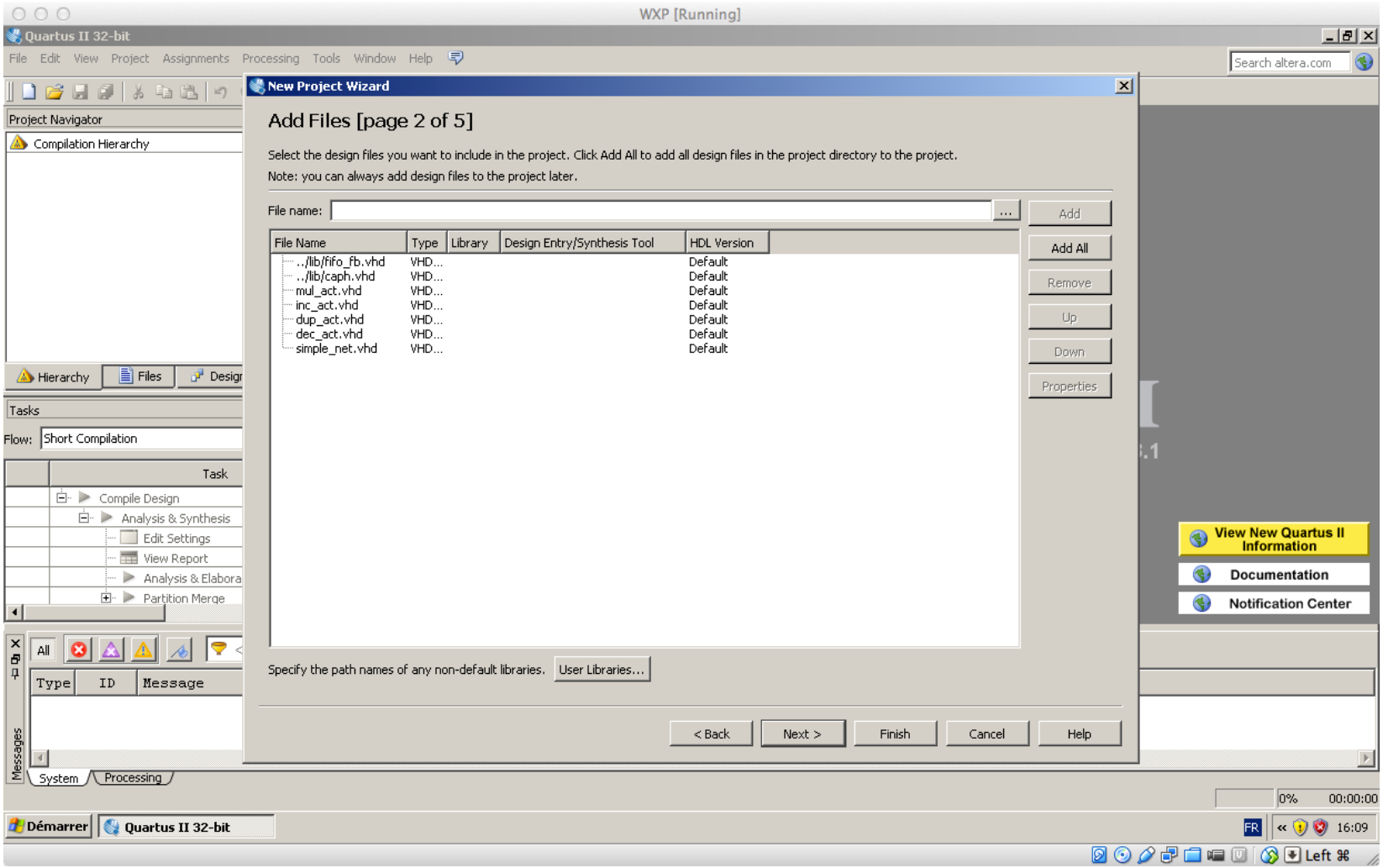


Figure 7.5: Setting the source files of the project

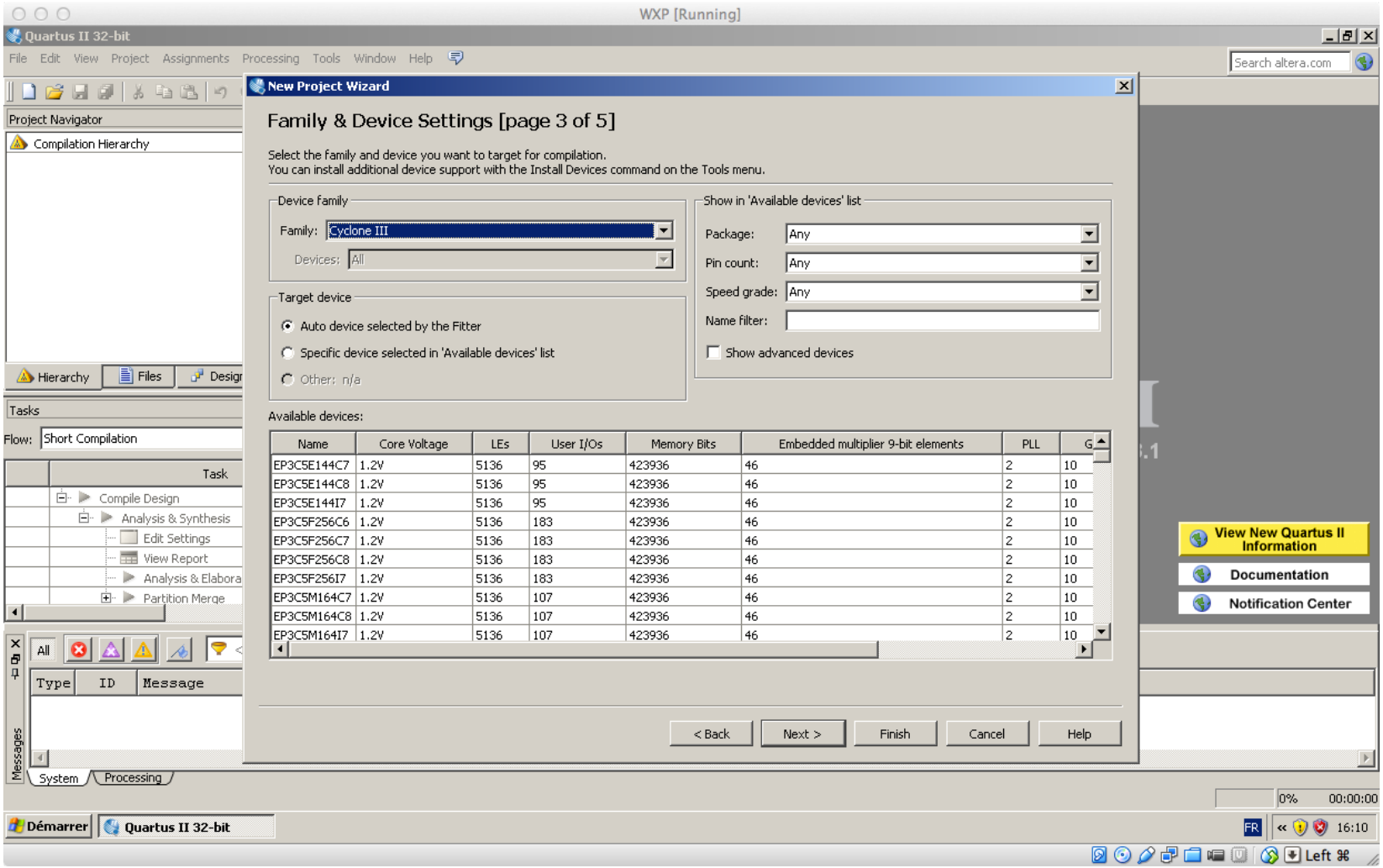


Figure 7.6: Setting the target device

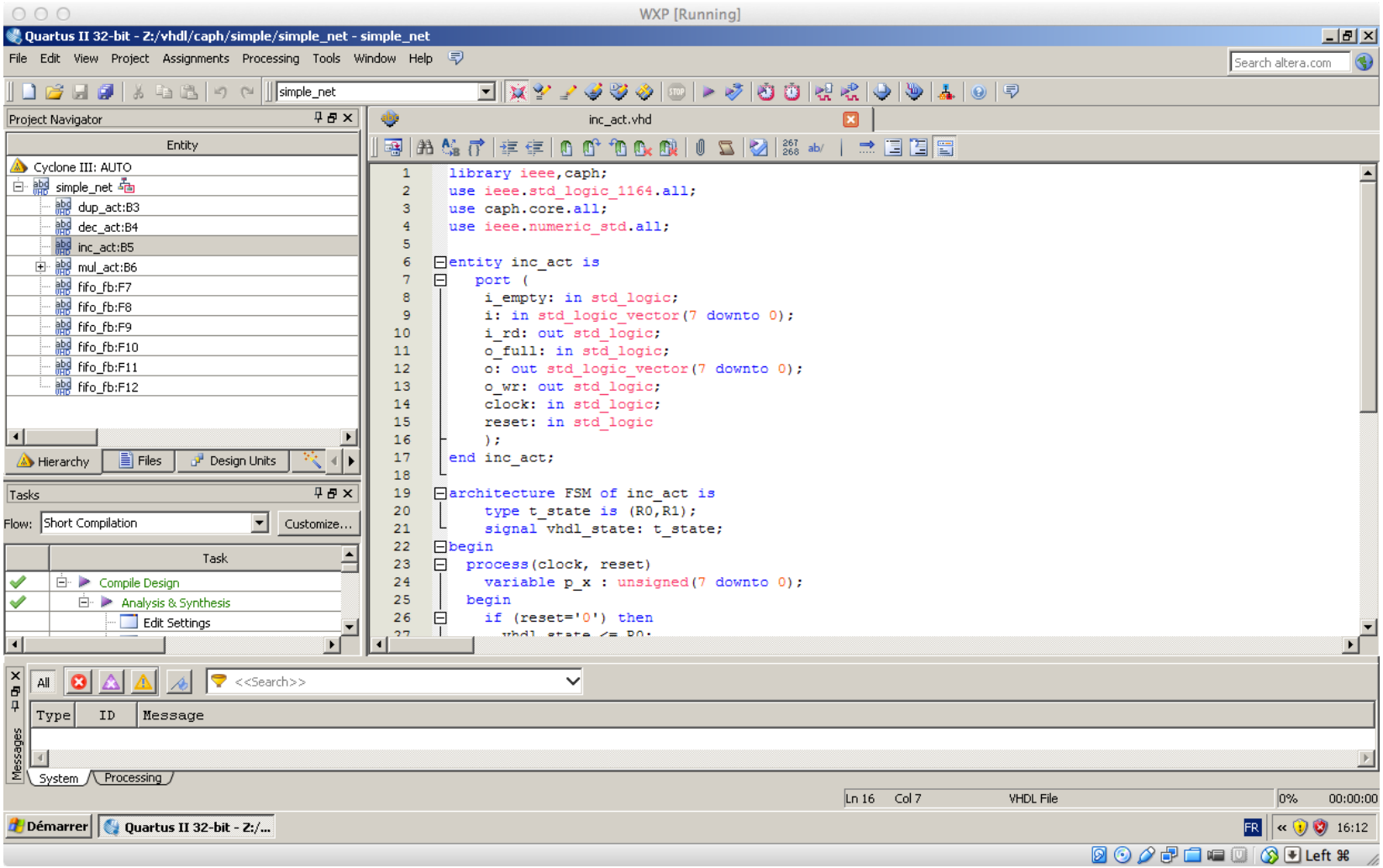


Figure 7.7: Displaying design hierarchy and source files

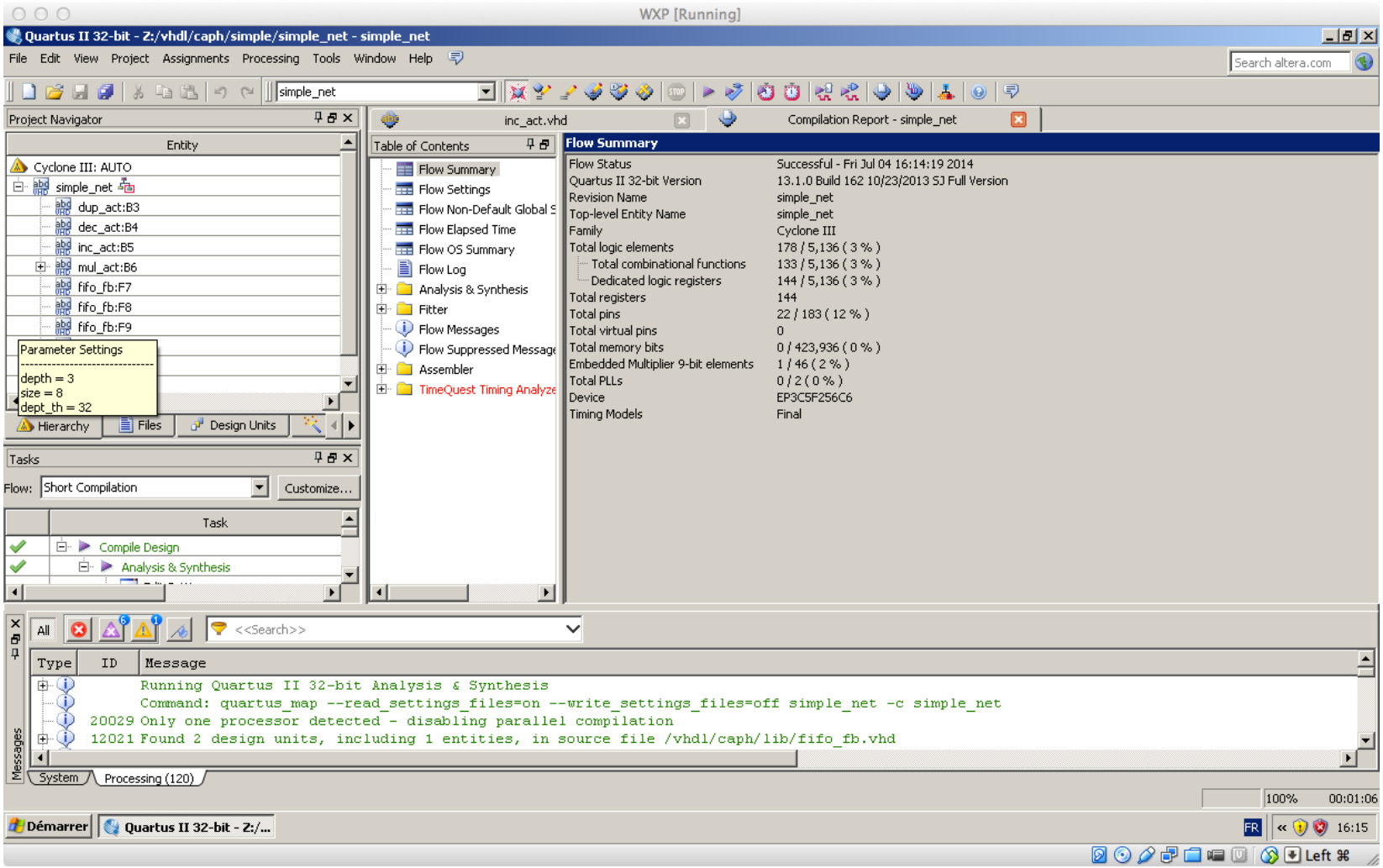


Figure 7.8: Synthesis results

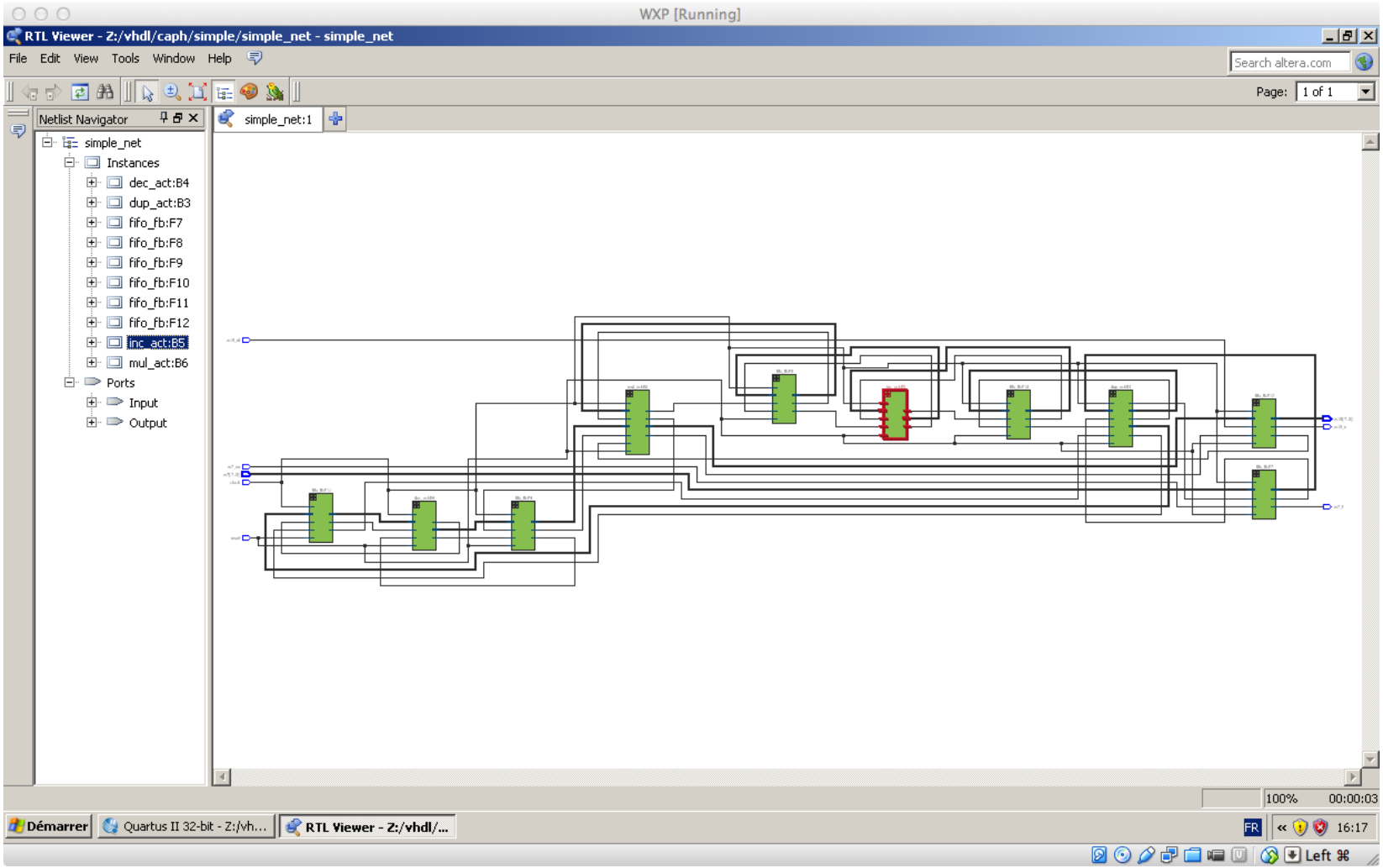


Figure 7.9: Post-synthesis, RT-level view

Chapter 8

Dealing with images

This chapter describes the implementation, simulation and synthesis, using the command-line interface, of the application based upon the concepts introduced in Chapter 2.

The code of this application, using the `inv` actor introduced in Chapter 2, is given in Listing 8.1. There's only `net` declaration, instantiating the `inv` actor. The first line (`#include "dc.cph"` is mandatory for making use of the `dc` type. The input image is to be read in file `lena128.pgm` and the result to be written in file `result.pgm`[1].

Listing 8.1: Complete CAPH source code for an application computing negative images

```
#include "dc.cph"

actor inv ()
  in (i:unsigned<8> dc)
  out (o:unsigned<8> dc)
rules
| i:'< -> o:'<
| i:'> -> o:'>
| i:'x -> o:'255-x
;

stream inp:unsigned<8> dc from "lena128.pgm";
stream outp:unsigned<8> dc to "result.pgm";

net outp = inv inp;
```

This program can be found in the `examples/primer/invimg` directory in the CAPH distribution.

The corresponding project file (also to be found in the examples directory) is shown in Listing 8.2. The option `-abbrev-dc-ctors`, at lines 1 and 2, tells the simulators (interpreter and SystemC-based, respectively) to read and write input and output files using the abbreviated syntax for control tokens.

Listing 8.2: File `invimg.proj` for the `invimg` program of Listing 8.1

```
SIM_OPTS = -abbrev_dc_ctors
SC_OPTS = -sc_stop_time 1000000 -sc_abbrev_dc_ctors
GHDL_RUN_OPTS = --stop-time=400000ns
```

Let's build the top-level Makefile by typing

```
# caphmake
```

Then, the dataflow graphical representation of the program is easily obtained by invoking

[1]The PGM (Portable Graymap Format) is a portable format for representing gray level images introduced in the NetPBM projet [6]. CAPH use the P2 (ASCII) sub-format.

```
# make dot.show
```

The representation is shown in Fig. 8.1

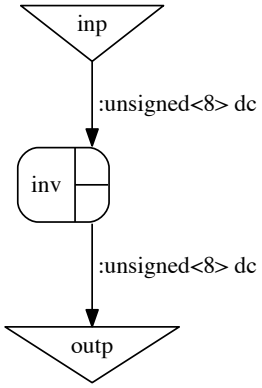


Figure 8.1: The graphical representation of the program given in Listing. 8.1

8.1 Simulation

The simulator cannot directly read and write images encoded with the PGM format. For this reason, the CAPH distribution comes with a pair of utility programs, `pgm2txt` and `txt2pgm`, to convert a PGM [6] file into a structured text file format and *vice-versa* in which pixels and start/end of line/frame are encoded using the `dc` type introduced in Chapter 2.1. A detailed description of these tools can be found in the reference manual. They programs can called directly from the command line before and after launching the simulation (to convert from and to the PGM format respectively), but this step can automatized further by writing a dedicated auxilliary files called a `.procs` file. In our case, the contents of this file (also to be found in the `primer/invimg` directory) is reproduced in Listing 8.3. The first line instructs the compiler to produce the file `lena128.txt` containing the input image in structured text format, ready for simulation, from the input image file `lena128.pgm`[2]. The second line instructs the compiler to produce the file `result.pgm` containing the result image in PGM format[3].

Listing 8.3: File `invimg.procs` for the `invimg` program of Listing 8.1

```
PRE_PROC = pgm2txt -abbrev lena128.pgm lena128.txt
POST_PROC = txt2pgm -abbrev 255 result.txt result.pgm
```

Simulation is then performed simply by invoking

```
# make sim.makefile
# make sim
```

This yields the following output

```
make -f Makefile.sim run CAPH=/usr/local/caph
/usr/local/caph/bin/pgm2txt -abbrev lena128.pgm lena128.txt
```

[2]The effect of the `-abbrev` option is to use the abbreviated format (`<`, `>`) for d(enoting control and data tokens. Without it, these tokens will be written as `SoS`, `EoS` and `Data` respectively.

[3]As for `pgm2txt`, the `-abbrev` option indicates that the input text file uses the abbreviated format for tokens. The numerical argument (255, here) gives the maximum value to be written in the PGM file header.

```
/usr/local/caph/bin/caphc -sim -I /usr/local/caph/lib/caph -I /usr/local/caph/lib/caph -abbrev_dc_ctors main.cph
-------------------------------------------------------------------------------------------------
This is the Caph compiler, version 2.8.3
(C) 2011-2017 J. Serot (Jocelyn.Serot@univ-bpclermont.fr)
For more information, see : http://caph.univ-bpclermont.fr
-------------------------------------------------------------------------------------------------
Wrote file ./result.txt
```

Viewing the result image is obtained by typing

```
# make sim.show
```

This invokes the `txt2pgm` utility and launch the PGM image viewing program which has been specified when installing CAPH. In our case, the results are show below and in Fig. 8.2-b.

```
make -f Makefile.sim show CAPH=/usr/local/caph
/usr/local/caph/bin/txt2pgm -abbrev 255 result.txt result.pgm
open -a Toyviewer result.pgm
```

a b

Figure 8.2: Input and output images after simulation for the program given in Listing. 8.1

8.2 Simulation using the SystemC backend

As in the previous chapter, this is done by simply typing

```
# make systemc.makefile
# make systemc.run
```

Executing this command yields the following output

```
make -f Makefile.systemc run CAPH=/usr/local/caph
/usr/local/caph/bin/caphc -I /usr/local/caph/lib/caph -systemc -I /usr/local/caph/lib/caph -sc_stop_time 1000000 -sc_a
main.cph
...
Wrote file ./invimg_expanded.dot
Wrote file ./invimg_net.cpp
Wrote file ./invimg_globals.h
Wrote file ./invimg_globals.cpp
Wrote file ./inv_act.h
Wrote file ./inv_act.cpp
(cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc ... -c `basename inv_act.cpp`)
(cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc ... -c `basename invimg_globals.cpp`)
(cd .; g++ -std=c++11 -I. -I/usr/local/caph/lib/systemc ... -c `basename invimg_net.cpp`)
```

```
(cd .; g++ -L/usr/local/systemc-2.3.1/lib-macosx64 inv_act.o invimg_globals.o invimg_net.o
   -o invimg_sc -lsystemc 2>&1 | c++filt)
./invimg_sc
       SystemC 2.3.1-Accellera --- Aug 9 2015 15:42:56
       Copyright (c) 1996-2014 by all Contributors,
       ALL RIGHTS RESERVED
Simulation stopped at t=1 ms
```

Viewing the file `result.txt` is then handled exactly like above, by invoking

```
# make systemc.show
```

8.3 Generating and simulating VHDL code

The process, again, is similar. Simply type

```
# make vhdl.makefile
# make vhdl.run
```

Executing this command yields the following output

```
make -f Makefile.vhdl run CAPH=/usr/local/caph
/usr/local/caph/bin/caphc -I /usr/local/caph/lib/caph -vhdl -I /usr/local/caph/lib/caph main.cph
...
Wrote file ./invimg_expanded.dot
Reverting to default size for fifo F5
Reverting to default size for fifo F4
Wrote file ./invimg_net.vhd
Wrote file ./invimg_types.vhd
Wrote file ./inv_act.vhd
Wrote file ./invimg_tb.vhd
warning: VHDL annotation file fifo_caps.dat does not exist.
(cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename invimg_types.vhd`)
(cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename invimg_tb.vhd`)
(cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename inv_act.vhd`)
(cd .; ghdl -a -P/usr/local/caph/lib/vhdl `basename invimg_net.vhd`)
(cd .; ghdl -e -P/usr/local/caph/lib/vhdl invimg_tb)
/usr/local/caph/bin/pgm2bin 8 lena128.pgm lena128.bin
ghdl -r -P/usr/local/caph/lib/vhdl invimg_tb --stop-time=400000ns
./invimg_tb:info: simulation stopped by --stop-time
```

Viewing the file `result.txt` is then handled exactly like above, by invoking

```
# make vhdl.show
```

The only difference here with the steps described in the previous section concerns the generation of the input file(s) and the conversion of the output file(s) to/from the custom `bin` format used by the VHDL simulator. The utility programs to use are now `pgm2bin` and `bin2pgm` respectively[4]. The corresponding calls to these utility programs are automatically inserted in the VHDL-specific Makefile generated by `caphmake`.

[4]These programs are also included in the CAPH distribution.

Chapter 9

Image processing

This chapter describes the implementation, simulation and synthesis, using the command-line interface, of the *Sobel* application introduced in Chapter 3.

The code of this application has been given in Listing. 3.5. The related project can be found in the directory `examples/primer/sobel` of the distribution.

9.1 Simulation using the interpreter

The simulation process is completely similar to the one described in Chapter 8. The project description file is reproduced in Listing 9.1.

Listing 9.1: Project file for the program of in Listing. 3.5

```
DOT_OPTS = -D ifile=pcb.pgm -D threshold=80 -suppress_cast_warnings
SIM_OPTS = -D ifile=pcb.pgm -D threshold=80 -suppress_cast_warnings -abbrev_dc_ctors
    -warn_channels -dump_channel_stats
SC_OPTS = -D ifile=pcb.pgm -D threshold=80 -suppress_cast_warnings -sc_abbrev_dc_ctors
    -sc_stop_when_idle 1000 -sc_dump_fifo_stats
VHDL_OPTS = -D ifile=pcb.pgm -D threshold=80 -suppress_cast_warnings -vhdl_annot_file
    sobel_fifo_stats.dat
GHDL_RUN_OPTS = --stop-time=160000ns
```

The `-D` option is used is give values to the symbols named `%ifile` and `%threshold` in the source code.

The option `-suppress_cast_warnings` is used to omit warning messages which are emitted when compiling the `fabs` function and which, in this context, can be safely ignored.

The `-warn_channels` option is set in order to detect channel overflows and the `-dump_channel_stats` is set in order to check channel usage after run.

The `.procs` file for this application, given in Listing 9.2, is similar to that given in the previous chapter. It simply tells how to convert from and to PGM format for the input and output images.

Listing 9.2: File `sobel.procs` for the `sobel` program of Listing 3.5

```
PRE_PROC = pgm2txt -abbrev pcb.pgm pcb.txt
POST_PROC = txt2pgm -abbrev 255 sim/result.txt sim/result.pgm
```

Simulation is performed, as usual with the following sequence of commands :

```
# caphmake
# make sim.makefile
# make sim.run
```

It produces the following result[1] :

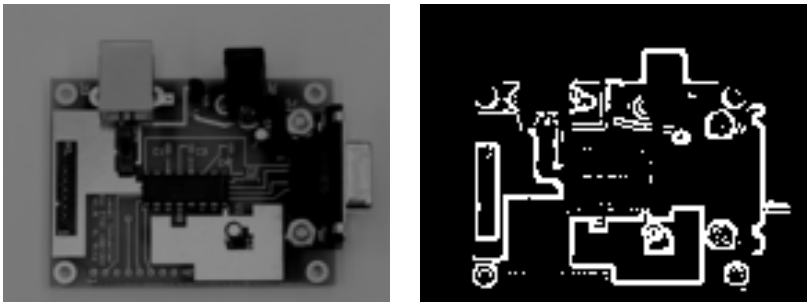
[1] Warning, this can take a few seconds

```
make -f Makefile.sim run CAPH=/usr/local/caph
/usr/local/caph/bin/pgm2txt -abbrev pcb.pgm pcb.txt
/usr/local/caph/bin/caphc -sim -I /usr/local/caph/lib/caph -I /usr/local/caph/lib/caph -D ifile=pcb.pgm -D threshold=8
...
Wrote file ./result.txt
W3: occ=0/256 max=2
W2: occ=140/256 max=140
W1: occ=140/256 max=140
W6: occ=0/256 max=2
W5: occ=140/256 max=140
W4: occ=140/256 max=140
W8: occ=0/256 max=2
W7: occ=0/256 max=2
W9: occ=0/256 max=2
W10: occ=0/256 max=2
```

Displaying the output image (Fig. 9.1-b) is obtained by invoking

```
# make sim.show
```

The maximum occupation reported for channels `W1`, `W2`, `W5` and `W4` is worth to be noted. The corresponding channels are used by the `conv233` actor to memorize the two previous lines when computing the convolution[2]. The maximum occupation value corresponds here to the width in pixels of the input image (140). No overflow occured because the default depth of channels in simulation is 256. Should we have used a larger image (ex: 512×512), it would have been necessary to adjust this depth with the `-chan_cap` option.



a b

Figure 9.1: Input and output images after simulation for the program given in Listing. 3.5

9.2 Simulation using the SystemC backend

SystemC simulation is performed exactly as detailed is Sec 8.2 (`make systemc.makefile; make systemc.run`) with some specific options, as shown in Listing 9.1. The `-sc_stop_when_idle` option is used to automatically stop the simulation after a given period of inactivity (1000 ns here, *i.e.* 100 clock cycles[3]). The `-sc_dump_fifo_stats` option is used to get a precise report on FIFO occupation in order to tune the VHDL backend. The resulting file, `sobel_fifo_stats.dat` is reproduced in Listing 9.4. A visual inspection of the result image shows that it identical to the one obtained using the interpreter.

Listing 9.3: Application-specific `Makefile` for simulating wth SystemC the application given in Listing. 3.5

```
SC_OPTS = -I $(CAPHLIB) -sc_abbrev_dc_ctors -sc_stop_when_idle 1000
    -suppress_cast_warnings -sc_dump_fifo_stats -D ifile=pcb.txt -D threshold=80
```

[2]These channels are those "looping around" the `conv233` actors in Fig. 3.4.

[3]The default clock period is 10 ns when using the SystemC backend. This can be adjusted with the `sc_clock_period` option.

Listing 9.4: File `sobel_fifo_stats.dat` produced by the SystemC backend for the application given in Listing. 3.5

```
w11 fifo_size = 3
w3 fifo_size = 3
w2 fifo_size = 142
w1 fifo_size = 142
w6 fifo_size = 3
w5 fifo_size = 142
w4 fifo_size = 142
w8 fifo_size = 3
w7 fifo_size = 3
w9 fifo_size = 3
w10 fifo_size = 3
```

9.3 Simulation using the VHDL backend

Again, this is very similar to what has been described in the previous chapter. The relevant line in the project file concerns the `VHDL_OPTS` macro. The `-vhdl_annot_file` option is crucial here. It gives the name of the annotation file generated by the previous SystemC execution (`sobel_fifo_stats.dat` here) to ensure correct sizing of the FIFOs in the final VHDL design (by default, FIFOs have a depth of only 4). Concerning the `GHDL_RUN_OPTS` macro, the value specified for the `--stop-time` option has here been derived from the final time reported by the execution of the SystemC code (154340 ns). Simulation is a bit longer than with SystemC (about ten seconds) and produce the same result image.

9.4 VHDL synthesis

Synthesis results for the application described by the `main_net.vhd` toplevel file on a Cyclone III FPGA with Quartus II are as follows :

- total logic elements : 828/119088 ($< 1\%$) (combinational function : 682, dedicated logic registers : 512)
- total memory bits : 6864/3981312 ($< 1\%$)
- IO pins : 23
- maximum clock frequency : 63.7 MHz

9.5 Centered *vs.* shifted convolution

As evidenced by Eq. (3.1), the `conv233` actor used in the previous sections implements a so-called *shifted* convolution : the output image is actually "shifted" one line down and one pixel right relatively to the input image. This can be easily explained by the fact that, since this actor operates on-the-fly on the input data streams, it can only use pixels which are "behind" the current pixel. This is illustrared in Fig. 9.2-a, in which the current pixel is y_{ij} and the "past" pixels are those shaded in gray. In this context, the "computation pattern" of Eq. (3.1) is represented by Fig. 9.2-b. More generally, with this formulation, for a $M \times N$ convolution, the output image would be shifted $M-1$ lines down and $N-1$ pixels right.

In certain situations, this "shifting" effect is not desirable and one would prefer a more classical definition of the convolution, in which the convolution kernel is "centered" around the current pixel, as illustrated in Fig. 9.3. The CAPH standard library therefore provides "centered" versions of 1D and 2D convolutions for several kernel dimensions. The program `mkconv`, described in App F of the reference manual, also has an option to generate centered convolution for any (odd) kernel dimensions.

In our case, the only modification is to replace the `conv233` actor in Listing 3.5 by its centered counterpart `cconv233`. This modification is denoted in Listing 9.5 (in which only modified lines have been reproduced).

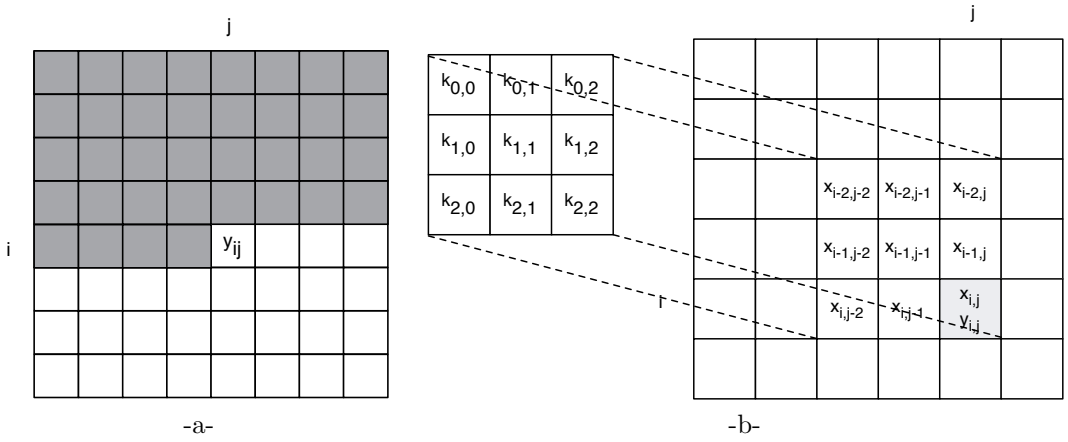


Figure 9.2: Shifted convolution

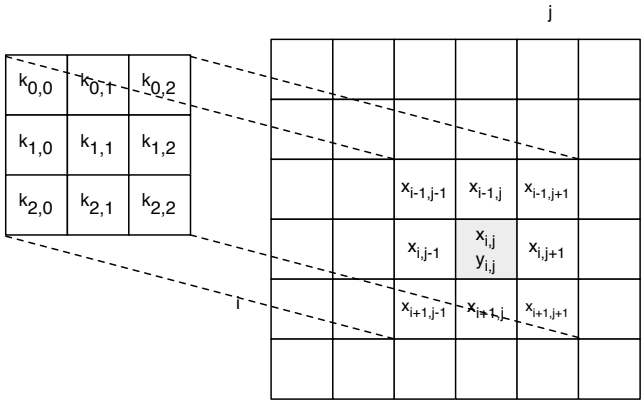


Figure 9.3: Centered convolution

Listing 9.5: Modification of listing 3.5 to use *centered* convolution

```
...
net gx = cconv233 ([[1,0,-1], [2,0,-2], [1,0,-1]], 0, 0) i;
net gy = cconv233 ([[1,2,1], [0,0,0], [-1,-2,-1]], 0, 0) i;
...
```

Simulation results with the interpreter are unchanged, except for the result image, which of course is no longer shifted and the channel occupation report, as shown in Listing 9.6.

Listing 9.6: FIFO occupation reported by the interpreter for the application using centered convolution actors

```
W3: occ=0/256 max=107
W2: occ=0/256 max=143
W1: occ=0/256 max=143
W6: occ=0/256 max=107
W5: occ=0/256 max=143
W4: occ=0/256 max=143
W8: occ=0/256 max=2
W7: occ=0/256 max=2
W9: occ=0/256 max=2
W10: occ=0/256 max=2
```

Note that, compared with the results obtained with the shifted convolution actors, the occupation of channels `W3` and `W6` can now grow to 107 places. A visualisation of the application dataflow graph (with the `-dot` and `-dot_show_indexes` options) shows that these channels are those connecting the `i` input to the `cconv233` actors. The reasons for this is that *centered* convolution actors, contrary to *shifted* convolution actors, requires a "flushing" phase at the end of each line of the image and the end of each image. This phase is needed to empty the FIFOs which are used to memorize previous lines and pixels. During this phase, no input can be read and if any are available, they accumulates on the FIFOs connected to the actor inputs.

The same behavior can be observed with the SystemC simulation : the file `main_fifo_stats.dat` obtained with option `-sc_dump_fifo_stats` reports a maximum occupation of 108 for the two FIFOs connecting input `i` to the actors `cconv233`[4]. As explained in Sec. 9.5.5 of the reference manual, this "accumulation" effect can be eliminated by inserting *blanking* clock cycles at the end of each line and each image. If one pixel is injected per clock period, the amount of horizontal (resp. blanking) for a $M \times N$ convolution should be equal to N (resp $L \times (M-1)/2$), where L is the width of the input images (number of pixel per column). In our case, this gives respective values of 5 and 140. This is achieved by modifying the SystemC-related options in the project file as illustrated in Listing 9.7. Note that we also had to increase the "idle time" used to detect the end of the simulation because of the inserted blanking cycles.

Listing 9.7: Modified project file for SystemC simulation (with centered convolution actors and blanking)

```
...
SC_OPTS = -D ifile=pcb.txt -D threshold=80 -sc_abbrev_dc_ctors -sc_stop_when_idle 2000
    -suppress_cast_warnings -sc_dump_fifo_stats -sc_istream_hblank 4
    -sc_istream_vblank 140
...
```

Blanking can also – and actually should if simulation is expected to reflect "real" behavior on the target hardware, as explained in Sec 9.5.5 of the reference manual – be simulated at the VHDL level. For this, the `-vhdl_istream_blanking` must be passed to the CAPH compiler and the option `-hblank` (resp. `vblank`) passed to `txt2bin` program. This is here achieved by modifying the project file as illustrated in Listing 9.8.

Listing 9.8: Modified project file for VHDL simulation (with centered convolution actors and blanking)

```
...
VHDL_OPTS = -D ifile=pcb.pgm -D threshold=80 -suppress_cast_warnings -vhdl_annot_file
    main_fifo_stats.dat -vhdl_istream_blanking
```

[4]The difference of 1 with the value obtained with the interpreter is not significant here.

```
...
```

Bibliography

[1] J. Sérot. CAPH Reference Manual. Available online at `caph.univ-bpclermont.fr`

[2] The `Graphviz` Graph Visualization Software. Available online at `www.graphviz.org`

[3] The GHDL VHDL Simulator. Available online at `gna.org/projects/ghdl`

[4] The GTKWave Software. Available online at `gtkwave.sourceforge.net`

[5] The Value Change Dump file format. `en.wikipedia.org/wiki/Value_change_dump`

[6] The NetPBM grayscale file format. `netpbm.sourceforge.net/doc/pgm.html`

[7] GpStudio : a Toolchain for FPGA-based smart camera. `gpstudio.univ-bpclermont.fr`

[8] Minimalist GNU for Windows `www.mingw.org`

[9] Linux for Windows `www.cygwin.com`

Contents