M1 Internship : Compiling Whiley to FPGAs

Baptiste Pauget

August 26, 2017

Introduction

Improving code performances lies more and more in the relevant use of computational helpers [NSD08]. Computing platforms like OpenCL or CUDA that enable to relocate resource consuming calculations to graphics processing unit (GPU) are now ordinary. Theses frameworks are rather low-level and require either a thorough learning or the use of higher level libraries that reduce their flexibility. A orthogonal concern of programming language is static checking. As the rise of strongly typed languages (Rust, Scala, Haskell, ...) reveals it, extended static checking is well-liked since it improves greatly the development process, lowering the debug work [FLL+13]. The Whiley language goes a step further, by providing an embedded support for specification, giving a chance to adjust the scope of compilation-time checks. Targeting Field Programmable Gate Arrays (FPGAs) for Whiley compilation tries to gather these two goals.

Acknowledgements

I would like to thank D. J. Pearce and A. Potanin for welcoming me in the Programming Languages Research group of Victoria University of Wellington and helping me during this internship.

Thanks a lot to T. Bourke, M. Pouzet and I. Delais without whom I could not have come to Wellington.

Contents

1 Working with FPGAs

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured by the user without any knowledge in electronics. Their use nevertheless requires a specific work-flow that needs to be understood to write relevant FPGA configurations.

1.1 A specific hardware

FPGAs try to gather the power of Application Specific Integrated Circuits (ASICs) with the flexibility of general computation devices (CPUs), by giving a way to reconfigure hardware without any physical operations.

1.1.1 Usage

FPGA's behaviour is defined by loading a hardware configuration at its start-up. Inputs and outputs of the device can then be used in a wider system as if they were separated by the requested circuit. This flexibility makes them appropriated for

- **Prototyping ASICs** : ASICs are still faster than FPGAs but they cannot be reconfigured once they are built. FPGAs allow their development process to be easier and cheaper.
- **Computation helpers** : Performances of specific applications (e.g., images processing, etc) can be enhanced with hardware compiled functions (e.g., Fast Fourier Transform), that are intensively used (FPGAs are often part of supercomputers).
- **Highly parallel applications** : because of their high parallelism potential, FPGAs are useful in applications that process lots of data simultaneously, like network interfaces (where large bandwidth is needed).

1.1.2 Structural organisation

FPGAs are organised as an array of interconnected identical computation cells, fortified with specific circuits. The coarse structure of FPGAs is illustrated in Figure 1 and contains the following components (with their number in Spartan XC3S500E devices) :

- **Configurable Logic Blocks (CLB)** (1,164) : The atomic units of FPGAs. They are divided in four slices containing
  - **2 Look-Up Tables** (LUT) : Implementation of logic computation in small LUTs (4 to 1 bits) that enable to compute any boolean function (They are RAM memory based)
  - **2 Storage Elements** : Can be used as flip-flops (registers) or latches
- **Input/Outputs Blocks** (232) : They provide an interface to the outside, preserving the internal impedances used in the FPGA.

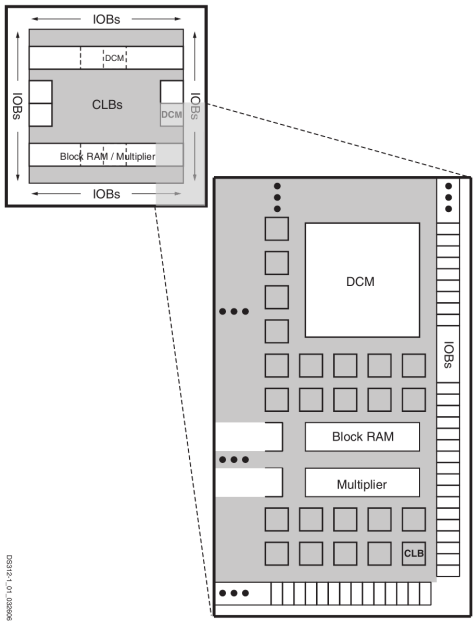


Figure 1: Spartan-3A Architecture [Xil11]

- **RAM Blocks** (20 : 368,640 bits) : These 18Kbits blocks can be used as RAM/ROM, with one or two I/O ports, supporting several memory configurations. (reading or writing up to 36 bits on each port).
- **Multipliers Blocks** (20) : They avoid an overuse of CLBs for multiplicative operations.
- **Digital Clock Manager Blocks** : They can duplicate, multiply, divide and delay clock signals

Some additional specific logic is provided to improve performances for complex operations (carry/arithmetic logic, dedicated multiplexers, ...).

1.2 Hardware Description Languages - VHDL

The lowest level description of FPGA's configurations is net-lists, which represent components and connexions between them. Higher-level descriptions were developed including circuit diagrams and hardware description languages (HDLs). The most used ones are Verilog and VHDL. Because of available documentation and a seemingly easier learning curve than Verilog, VHDL was chosen as the targeted language of the compiler developed during this project. A brief introduction to the main aspects of this language follows[MT13].

1.2.1 Overview

Contrary to Verilog, VHDL (VHSIC[1] Hardware Description Language) is a typed HDL: wires or bench of wires are represented with typed *signals* and the correctness of statements is checked during compilation.

As in numerous programming languages, sources are structured in packages and libraries. The main compilation unit is the **entity**, that defines an interface (input/output signals) and comes with one or more **architectures** that describe the inner circuit.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity foo is
  port (
    foo_in  : in  signed(31 downto 0);
    foo_out : out signed(31 downto 0)
  );
end entity foo;

architecture Behavioural of foo is
  -- Declarations
begin
 -- Circuit description
end architecture Behavioural;
```

1.2.2 Describing the circuit

VHDL is a data-flow language: a signal represents the state of some data at a precise place of the computation and architecture statements depict operations that link these signals. Their order is thus insignificant and signals can be assigned at most once.

Concurrent statements: the combinatorial description

A combinatorial circuit does not contain any loop. This ensure that it will stabilize if the inputs keep their values a long enough time. The following concurrent statements are converted to combinatorial circuits during the compilation of VHDL toward an FPGA configuration file.

- **Signal assignments** (`<=`) that link the result of an expression to a signal. Expressions are defined using signals and bitwise operations or typed operations whose circuit are described in imported libraries (addition, multiplication, ...).
- **Conditional signal assignments** (`when`/`select`) that construct multiplexers choosing between several signals by examining the values of other ones.
- **Component instantiations** that instantiate external entities, linking their interface with internal signals. They correspond to software programming languages function calls except in the fact that circuits are duplicated at each instantiation.
- **Process statements** that introduce an other kind of description of circuit discussed below.

Sequential statements : the behavioural description

Process statements delineated a description that is closer to software programming languages: its inner statements are executed sequentially and higher level constructions such as variables or `if` statements are available. Variables strongly differ from signals in that they are only accessible inside the `process` statement, and they behave as software programming languages variables, holding the last assigned value. More work is done by the compiler to convert these statements to a (non necessary combinatorial) circuit that will behave as described. It results in a larger configuration and a longer compilation time.

[1] VHSIC stands for Very High Speed Integrated Circuit

1.2.3 Execution model

Parallelism

Concurrent statements are independently re-evaluated when an event occurs on one of their signals they are sensitive to. This sensitivity list is implicitly set to the read signals of the expression except for `process` statements that defines it explicitly. The completion of expression's evaluation triggers a event on the signals that are written by it.

Concurrent statements are atomic computation units : every outputs are set at the same time. This is particularly important for `process` statements : read signals hold the value they had at the start of the evaluation, and the ones that are written are all updated at the same time at the end of the process. A signal that is written several times through the process will only keep the last value.

Considerations on time

The evaluation of expressions is completed when circuit's output impedances are stabilized, after a change in its inputs. The laws of physics prevent it to be immediate, and the require time is known as *delta-delay*. These delays should not interfere with the design conception, but may prevent result to comply with frequency constraints.

Introducing time in a design is achieved with external clocks. Usual boards that contain FPGAs include several clocks connected with predefined input ports. Using specific clock manager blocks, these signals can be duplicated and their frequency changed. To ensure the configuration will behave the desired way, clock's frequencies must be low enough to ensure that the period is greater than the *delta-delay* of the clocked circuit.

1.2.4 Registers

Storing values between updating orders is a very useful feature that is usually achieved using *latches* or *flip-flops*. Figure 2 shows the circuit of a set-reset latch. This circuits are intended to be used with at most one active input and are non-combinatorial since they contain feed-backs : their outputs depend indirectly of themselves. Such feed-backs are called *positive* when the circuit can become stabilized and *latches* or *flip-flops* are of this kind.

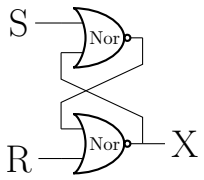


Figure 2: SR-latch

Registers are signals that update their value only when an event occurs on other one. They are mainly used with a clock signal as trigger. They are inferred by the synthesizer when an incomplete if statements is encountered in a process statement: if a signal is set in only some branches of a disjunction, a register will be generated. To avoid unwanted effects, `if` statements have to be carefully used.

```
Reg: process(clock) is
begin
  if (rising_edge(clock))
     then
    reg <= newValue;
  end if;
end process Reg;
```
Register generation

1.3 The compilation process

The compilation of VHDL code to an FPGA configuration file is a complex and time consuming process. Sources are first converted to net-lists (the synthesis) that are used to generate the configuration.

1.3.1 Compilation step

The compilation of net-lists to a configuration file strongly depends on the targeted device and is handled by commercial tools provided by FPGAs vendors. The major steps of the this process are:

1. **Mapping** : Conversion of net-list operations to LUTs and specific logic blocks that are available on the device.
2. **Place and route** : Positioning the LUTs and specific blocks on the FPGA, regarding the constraints (output pins, clocks), and drawing the connections between theses blocks.

3. **Time analysis** : Analysis of the circuit to determine the maximum delay needed for it to get stabilized on a change of its inputs, to adjust clocks.

4. **Configuration generation** : Generating a configuration file that can be loaded on the FPGA.

1.3.2 Simulations

Because of the expensive time cost of synthesis (about 1 minute 30 seconds for simple VHDL sources with Xilinx's tool) and the poor debugging aids on FPGAs, simulators provide a way to enhance the development process. Simulations can be run at several steps in the compilation process. They enable one to investigate configuration's behaviour in an almost interactive way, giving access to inner signal impedances, and their evolution through time.

Simulations are supported inside VHDL, with special statements (like `wait for 2ms`), operations or data types (floats) that are not supported by the synthesizer but that make the analysis easier. Values of inputs signals of the studied entity are planned with a separated VHDL file.

1.4 Existing high level compilers

To bridge the gap between hardware and software programming and make the use of FPGAs accessible for non-specialists, high-level compilers have been developed that compile software programming languages to HDL descriptions [BRS13] or net-lists [IS95].

1.4.1 Targeted execution environments

Most of high-level compilation tools target highly heterogeneous systems, aiming at providing a unique language to describe both software and hardware parts. These execution environments can be composed of multiple CPUs, GPUs, FPGAs or other specific hardware devices.

Theses systems are often built around a main processor that organises the computation on helpers. OpenCL provides unified abstractions to program highly parallel applications, and have been adapted to target FPGAs [CNK+12].

1.4.2 Xilinx's Vivado tool

The Vivado Design Suite gathers tools to develop software applications with hardware parts. A C to HDL compiler is included that enables hardware to be described in C. The supported language is relatively wide and includes general loops that are converted to complex state machines. Because these hardware circuits are intended to be part of a larger system, they are constructed with additional interface signals for linking components together (`start`, `done`, `ready` and `idle` signals).

1.4.3 Limitation

In several high level compiler studies, the supported language was restricted to a very small and basic core of the original language. Process statements were used to convert easily basic C code to VHDL [PTP94] but such solutions only bridge the (huge) syntactical gap between C and VHDL, providing no more abstractions that VHDL ones.

Most of chosen source languages were deeply imperative languages (C, Java) that increases a lot the conceptual gap between hardware descriptions and them. Using more functional languages could lead to more accurate high-level descriptions.

High level compilers also tried to add as few elements as possible in the source language. The compiler then adds a lot of VHDL elements and this gives few ways to control precisely the produced design. Modifying the source language by providing meaningful syntax constructions could make it a relevant high-level hardware description language, more than a software programming language with a tool that obscurely converts it to a hardware description.

2 The Whiley language and the compiler framework

The Whiley language is designed to provide extended static checking. The strong type system is coupled with support for specification that can be verified by a theorem prover.

The following presentation of the Whiley language focuses on the aspects that were studied in this work, that is especially unrelated to the specification and checking parts of the language. An exhaustive documentation can be found here [Pea14].

2.1 Overview

With a Python like syntax (with C-like comments), Whiley is both

- Oriented-object : Records and open records enable to define values with attributes and methods.
- With first class functions : functions (pure) or methods (can have side effects) can be used as values.

Specification is introduced through the `requires`, `ensures` and `where` keywords. Here is a typical Whiley program that illustrates how to build a specified function.

```
1  function firstIndexOf(int[] t, int v) -> (int|null i)     // No pre-conditions
2  ensures i is null || (t[i]==v && all  {k in 0 .. i | t[k]!=v}) // Post-
3  ensures i is int || all {k in 0 .. |t| | t[k]!=v}:              //  conditions
4      int k = 0
5      while k < |t|
6      where 0<=k && k<=|t|                  // Loop
7      where all {j in 0 ..k | t[j]!=v}:     //    invariants
8          if t[k] == v:
9              return k
10         k = k + 1
11     return null
```

The theorem proven will verify that every post-conditions are true on return values. To help it analysing loops, loops invariants can be specified. Invariants must be true at after one loop iteration, assuming they where true at the previous iteration.

2.2 Type system

Structural types

The Whiley language uses a structural type system : types are determined by their declaration instead of their identifiers. Two nominal types with different names or syntactical definitions can thus have the same values : types `int|null` and `null|int` represent both the same set of values.

In such a system, types can usefully be seen as a subset of the set of correct values $\mathcal{V}$ of the language. Type operations can then be interpreted as they equivalent on sets. The sub-typing operator, that tells if a type is a sub-type of an other one, is define as the set inclusion.

New types are constructed using :

- **Primitive types** : (non-exhaustive list)
  `any` $= \mathcal{V}$,
  `null` $= \{null\}$, `bool` $= \{true, false\}$,
  `byte` $= \{0,1\}^8$, `int` $= \mathbb{Z}$,
  `void` $= \emptyset$
- **Constructed types** : Records, Arrays, References, Functions or Methods.
- **Type operators** : Negation, intersection and union.

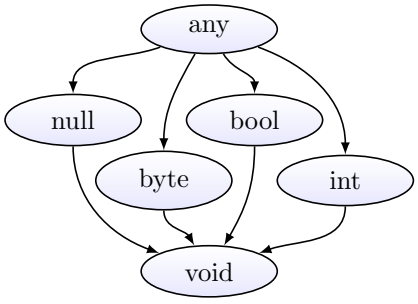


Figure 3: Lattice of primitive types

Constrained type

The syntax `define nat as (int x) where x >= 0` enables to define types using previously declared ones and adding to them some constraints on the possible values. Their use with values that do not meet constraints will result in a compile time error.

Effective types

To improve convenience, union of types that share the same structure (records, arrays, ...) exposes this common structure too. For example, the union of records that all have a field `f` is an effective record containing a field `f` of type the union of type for this field in the records.

2.3 Flow-typing

The Whiley language provides flow-typing that enables a variable to hold different types through the code. Type checking is introduced with the `is` operator. When used in a `if` statement, it refines the type of the checked variable on each branch, with enable to use them without cast.

```
function toBool(int|bool c) -> bool:
  if c is int:      // int|bool & int
    return c != 0 //    c : int
  else:             // int|bool & !int
    return c      //    c : bool
```

The above code shows how variable's type is constructed on each branches of the `if` statement, using negation and intersection operators. The `return` line comments show the simplified type of `c`.

2.4 The Compiler Collection

The Compiler Collection is a set of projects that enable to compile Whiley sources. They are first converted to Whiley Intermediate Language (Wyil) by the front-end compiler (wyc). Wyil bytecodes are then given to back-end compilers.

Wyc handles lexing, parsing and type checking of Whiley sources. It also fully resolves names, giving each variable of a function a unique identifier. Figure 4 shows some of existing backends.

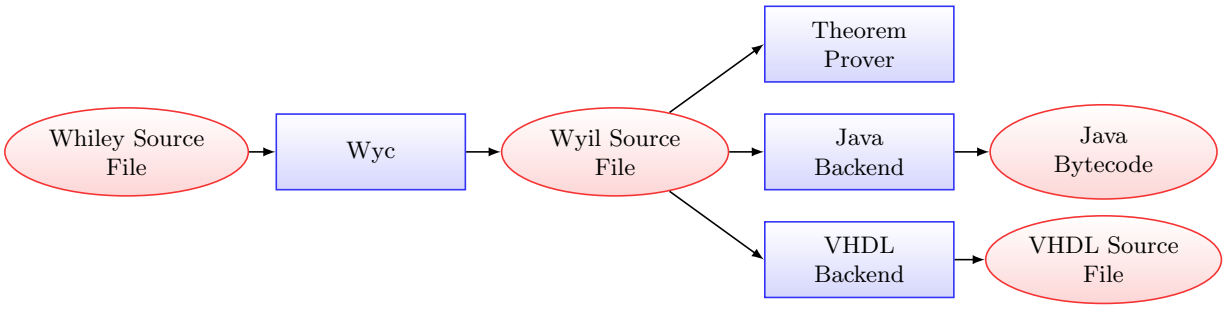


Figure 4: The Compiler Collection

This structure greatly simplifies the work of back-end compilers by gathering common part of the compilation process (lexing, parsing, typing) and giving back-end compilers only well formed and typed programs.

3 The WhileyToVHDL Compiler

As part of this study, a new back-end compiler was developed : the WhileyToVHDL compiler that compiles Wyil bytecode to VHDL source files. It was constructed without any modifications of the langage but keeping in mind that extensions could be proposed.

3.1 Overview

Compilation process

Because of the source and targeted languages, the compilation process has several specific traits :

- It is only a back-end compiler : no need to handle type checking nor verification, the input sources are supposed to be well formed and correct.
- The conversion of a sequential code to a hardware description (intrinsically parallel) requires the use of a specific transformation pipeline.

For each function, the compilation is achieved using the following steps :

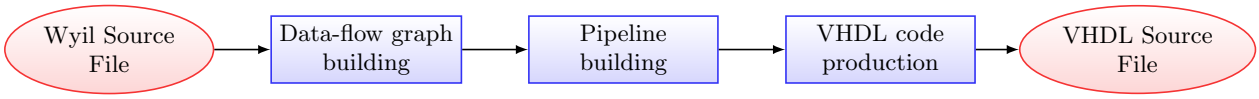


Figure 5: The compilation pipeline

Remark : Wyil files do not contain only function declarations. Nominal types can also be declared. They are pre-treated and stored for future use (See Types compilation).

Compilation units

The structure of Whiley files is preserved in the following way :

- Each file is compiled to a unique VHDL file
- Each function is compiled to a unique VHDL entity

Restrictions on the language

Only a small subset of the Whiley language is supported by the WhileyToVHDL compiler. The limitations arise from two major reasons :

- **FPGAs' limitations** : Some features (floating point numbers) cannot be efficiently introduced.
- **The amount of time necessary to add some concepts** : Arrays, partial recursion support, ...

Data-flow graph

This graph describes how data are modified through a series of statements. Every path on this graph shows a possible transformation chain of an entry. This representation exposes the fine-grained parallelism that was hidden by the sequential description of the computation [CN03, TGP07, PBD+08].

To ensure a maximum use of this parallelism, the data-flow graph is built with VHDL values as atomic components. It is thus necessary to :

- Convert the Whiley typed variables to VHDL ones, choosing a representation for each type.
- Build the data-flow graph from the functions body with the types constructed above.

VHDL generation

Because VHDL is a data-flow language, the compilation of data-flow graphs to VHDL sources is mainly a straight-forward translation of vertexes to VHDL concurrent statements. Process statements are only used to convert register vertexes. Signals are created at strategic locations, to avoid duplication of expressions.

3.2 Types compilation

The choice of representation for Whiley variables relies on the conversion of their type to a construction using VHDL types.

3.2.1 Purposes, difficulties and limitations

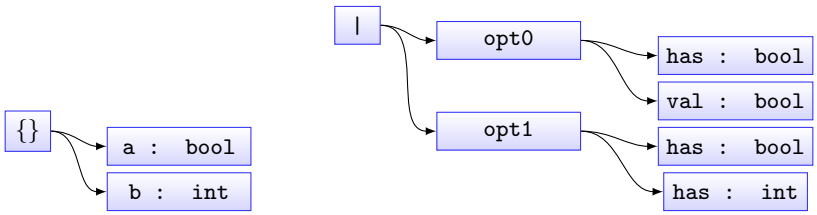
Contrary to a classical CPU execution environment where variables are compiled to memory addresses or registers, VHDL signals represent a bunch of wires. This simple fact has several severe consequences :

- **Types must be finite** : When a value of type T is expected, a only finite number of bits (wires) will be allocated for it and they must be enough to store every possible value of this type. Whiley unbounded `int` are thus impossible to handle, and neither is `any` or recursive types.
- **Type operations must be resolved** : Storing a value of type $T_1|T_2$ cannot be achieved with a pointer on a value of type T_1 or T_2 and a check of the effective type at runtime. Instead, the representation of both types has to be contained in the representation of the union type, as well as a way to distinguish the effective type of the value.
- **Type representations should optimized** : Because of the space limitation on targeted devices, the most compact representation as possible should be used. Storing the type {int a}|{int b} with two distinct values of type int is not desirable, as only one of them will be useful.

The compilation of (finite) arrays was not studied. A simple work around would lie in representing arrays with records with array's length fields of array's value type.

Type representation

When feasible, a Whiley type will be converted to a set of VHDL types that handles every possible value of this type. To make the processing easier and the produced code clearer, the different components of a type representation are organized in a tree. It also enables to easily create signals with unique names. The complete discussion about the choices of theses representation can be found below.



(a) `{bool a, int b}` (b) `bool|int`

Figure 6: Example of simple type representations

Remark : The case of Whiley's `int` type is resolved by fixing the size of integer (32 bits signed integers). To preserve Whiley semantic, an analysis of constraints of integer types could replace this solution, that would enable to adjust precisely the size of integers[Pea15].

Type resolution

Because some type constructions cannot be easily represented with VHDL types (intersections, negations), types must be simplified when feasible to equivalent forms free of unsupported operations.

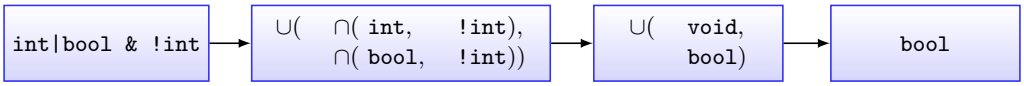
Such complex types arise when using flow typing : the types of variables on each branch are defined with intersections and negations. Because the front-end compiler simplify only basic cases, a deeper analysis was built to ensure the widest support of flow typing.

3.2.2 Type analysis

The simplification of type expressions is based on the elimination of redundant components in unions or intersections. Given a way to compare types, the simplification of type expressions can be achieved using the following facts :

- A component of a union that is included in an other one is useless and removed.
- A component of an intersection that contains an other one is useless and removed.
- If two components of an intersection are disjointed, the resulting type is `void`.
- Union or intersection with only one option are converted to this unique component.

A canonical form is used to reveal possible simplifications. The analyse of above flow typing example variable `c` in the `else` branch would work the following way :



In the following study,

- $\mathcal{V}$ is the set of acceptable Whiley typed values (closed by Cartesian product)
- $\mathcal{L}$ ={any, bool, byte, int, null, void} is the set of primitive types
- Records are represented with Cartesian products: $\{T1\ a,\ T2\ b\} \equiv T1 \times T2$, regardless field's names that are not relevant here.

The canonical form

We denote with *simple types* ($\mathcal{S}$) the set of primitive types and record types whose components are all simple types. Types are said *in canonical form* if they are defined as :

$$\bigcup_i \bigcap_j T_{i,j}, \quad T_{i,j} \in \mathcal{S} \text{ or } T_{i,k} = \mathcal{V} \setminus S,\ S \in \mathcal{S}$$

Type expressions are transformed using the classical set equations :

$$\begin{aligned} \overline{\bigcup_k T_k} &= \bigcap_k \overline{T_k} \\ \overline{\bigcap_k T_k} &= \bigcup_k \overline{T_k} \\ \bigcap_{i\in[\![1,n]\!]} \bigcup_{j\in J_i} T_{i,j} &= \bigcup_{I\in J_1\times\ldots\times J_n} \bigcap_i T_{i,I_i} \end{aligned}$$

The case of records (Cartesian products) needs more care :

$$T_1 \times (\overline{T_2}) = (T_1 \times \mathcal{V}) \cap \left(\overline{T_1 \times T_2}\right)$$

int	int|bool	{int|bool a, !byte b}
∪(∩(int))	∪(∩(int), ∩(bool))	∪(∩({int a, any b}, !{int a, byte b}), ∩({bool a, any b}, !{bool a, byte b}))

Figure 7: Some canonical form examples

Because types `any` (*resp.* `void`) is the neutral element for the intersection (*resp.* union) and redundant components are possible, the canonical form is not unique.

Type comparison

The implemented comparison between T_1 and T_2 is achieved trying to check the following properties:

$$\bullet\ \mathcal{P}_1 : T_1 \subset T_2 \qquad \bullet\ \mathcal{P}_2 : T_2 \subset T_1 \qquad \bullet\ \mathcal{P}_3 : T_1 \cap T_2 = \emptyset$$

Theses properties can independently be effective or not. (If the three properties are true, both types are necessarily empty)

A result of comparison is the set of properties that are known to be true. Figure 8 gives a name to the different results and shows their implication. Testing a subtype relation is done by checking if $\mathcal{P}_1$ or $\mathcal{P}_2$ is known to be true in the result of the comparison. Comparison must try to keep as much information as possible by giving the more precise result.

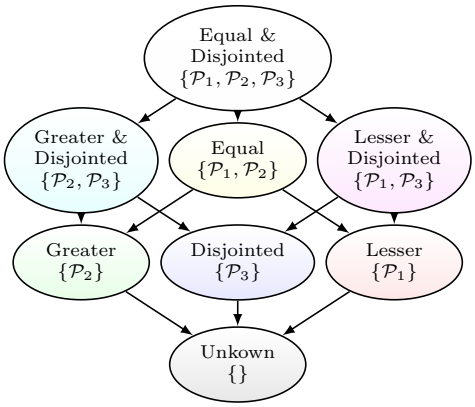


Figure 8: Lattice of comparison results

The comparison of two types is computed in the following way :

- If both of the types are primitive types, the result is well defined (Equal if they are the same, included if one of them is `any` and disjointed in the remaining cases except if one of them is `void` where a more precise result is available).
- If one of them is a constructed type, its components are compared to the other and the result is build using some merging rules on components comparison's results.

To keep more information during comparison, a fourth dimension could be added, corresponding to the property $\mathcal{P}_4 : T_1 \cup T_2 = \mathcal{V}$.

Remark : An other algorithm was proposed by D. J. Pearce [Pea13] to build the subtype operator, based on only computing the property $T_1 \cap T_2 = \emptyset$ $(\mathcal{P}_3)$. A similar canonical form is used to ensure this operator can be constructed. Other comparison results can be then implemented using it :

$$T_1 \subset T_2 \Leftrightarrow T_1 \cap (\mathcal{V} \setminus T_2) = \emptyset$$

3.2.3 Type representation

Once the types have been simplified, if they do not contain any intersection, negation or infinite types, they are converted to a tree which leaves are primitive types.

Primitive types

The leaves of this tree are VHDL types that can represent a Whiley type in an obvious manner. They are drawn in red on the following diagrams.

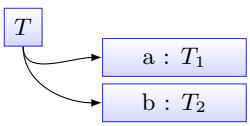
- `int` are represented with `signed(31 downto 0)`
- `bool` are represented with `boolean`
- `byte` are represented with `std_logic_vector(7 downto 0)`

Complex types

To build the other types, several kind of inner nodes are provided. Some of them do not correspond to any Whiley construction, but enable an easy compilation of type related operations :

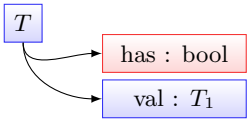
- **Record nodes** : Each field is stored in a sub tree labelled with its name.

  Type T : $\{T_1$ a, T_2 b$\}$



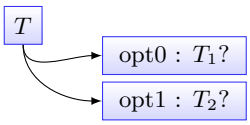
- **Option nodes** : Add a boolean flag to a type, intended to indicate if the value is meaningful or not. Theses nodes are used by other inner nodes.

  Type T : T_1?



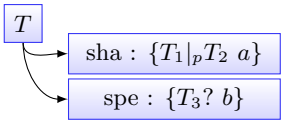
- **Primitive union nodes** : If every option's components is a primitive type, a tree with one option sub tree per component is built.

  Type $T : T_1|_pT_2$



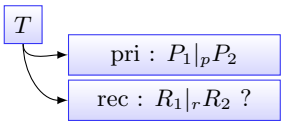
- **Record union nodes** : Gather identically named fields and store specific ones with an option sub tree.

  Type $T : \{T_1\ a\}|_r\{T_2\ a,\ T_3\ b\}$



- **General union nodes** : Store a general union, separating records options from primitive ones.

  Type $T : P_1|P_2|R_1|R_2$



Some primitive types are not straightforwardly convertible to VHDL values :

- The type `null` is converted to an empty record : the only value of type `Null` is null, so no bits are needed to store it. When part of a union (`int|null`) the option's boolean is holding the information.
- The type `void` is not convertible because it contains no values.
- Simple and record unions unions were introduced to easily handle Whiley's effective types.

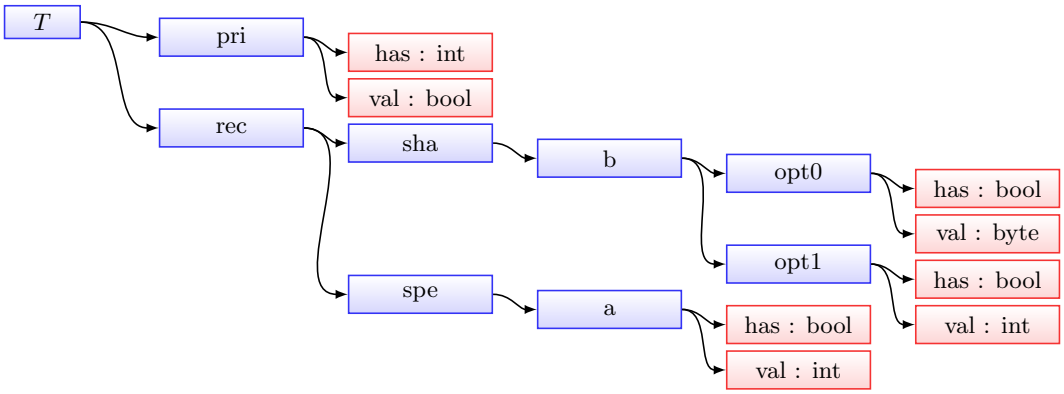


Figure 9: Representation of type T : {int a, byte b}|{int b}|int

Remark : Recursive types such as trees are not convertible as they may describe values of unknown and unbound size.

3.3 Building the data-flow graph

The building of the data-flow graph transforms the function body to a graph exposing the paths data can follow during the execution.

3.3.1 Vertexes

Because Whiley types are converted to a tree of VHLD types, each Whiley variable is associated with a tree with vertexes as leaves that represent its current state. This graph is made of two types of vertexes:

- **Symbolic vertexes** : represent constants or inputs/outputs (of the function or the called functions). They have at most one source.
- **Operation vertexes** : represent operations, with their operands as sources. Complex operations are compiled to equivalent set of basic operations (see the `is` operator).

When meaningful, vertexes are typed with a VHDL type, Only these kind of vertexes can then be compiled toward VHDL statements.

3.3.2 Compiling the function body

To compile a function body, a map of Whiley variables to the current tree of vertexes representing them is stored and initialized with input nodes for each argument. Body's statements are then successively processed to construct the graph. Some worth-mentioning statements are discussed below.

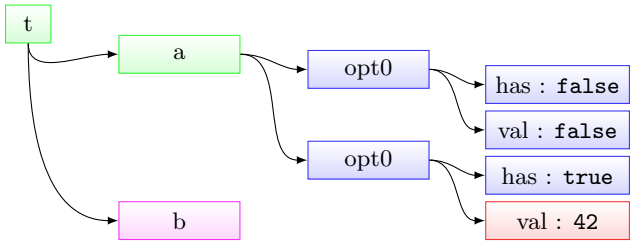
Variable assignments

The compilation of variable assignments modifies the tree of vertexes that represents the assigned variable. It requires three steps and Figure 10 illustrates the different nodes that are created.

```
type T is {bool|int a, byte b}

function answer(T t) -> T:
    t.a = 42
    return t
```

(a) Assignment statement



(b) Tree respresenting t after assignment

Figure 10: Example of assignment

1. **Building of the right hand side** : A tree representing the new value is built by compiling the expression. (`42` is represented with a simple integer vertex).
2. **Adjusting types** : Some additional signals can be created to make the type of the expression fit the assign variable's one (`int` value must be adapted to be of `t.a`'s `int|bool` type).
3. **Modifying trees** : Building the global tree of the modified variable, keeping some old sub-trees and building new inner nodes (`t.b` was not modified and should be kept).

`If` statements and returns

Contrary to a usual compilation strategy that use conditional jump on relevant part of the program, both true and false branch, as well as the condition are compiled separately, using pre-statement variable's representations. Modified variables are then gathered using a multiplexer.

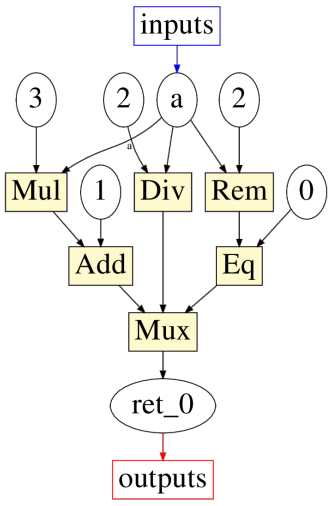
Extra care has to be taken with the management of returns when several statements are used in different blocks: because only one bunch of output signals must be constructed, they are treated as if returns were stored in a variable that is returned at the end.

```
function s(int a) -> int:
    if a%2 == 0:
        return a/2
    return 3*a+1
```

(a) Multiples return

```
function s(int a) -> int:
    if a%2 == 0:
        a = a/2
    else:
        a = 3*a+1
    return a
```

(b) Unique return



(c) Resulting data-flow graph

Figure 11: If statement and multiple returns

Function invocations

Function calls are compiled to a special untyped vertex that contains necessary details about the function. It has as many sources as the function's arguments and its only targets are special symbolic vertexes that represent the return values of the function. During the production of VHDL code, the vertex is converted to a component instantiation.

3.3.3 Flow typing

Flow typing provides a very convenient way to deal with constructed types and its support relies mainly on the operator `is` and alias declarations.

Operator `is`

Runtime type checking is mainly useful when dealing with union typed values. It is compiled using the additional flags that were added in option nodes : the expression `x is int` where `x` has type `int|bool` is converted to a simple reading of `int` option's `has` flag. More complex type checking operations require to combine flags with boolean operations (*e.g.* when checking toward a record).

Aliases

In the branches of an if statement with a type checking, the variables checked are access through aliases, that can be seen as a lens refining the type of original variables. It is translated by extracting or building the sub tree that correspond to the precise type, keeping the same leaves (vertexes).

3.4 Building a pipeline

The construction of the data-flow graph organizes the computation spatially, showing operations that can be carried out side by side. The computation model also needs to be analysed through time.

3.4.1 Objectives

FPGAs and compilation to hardware are mainly targeted to improve performances. A common way to ensure a maximum use of silicon lies in building a pipeline : the process is separated in several steps that can be executed simultaneously. The computation of a new value can thus be started after the completion of only one step instead of waiting for the whole calculation to finish.

The orchestration of the pipeline can be made with a clock that governs the sending of a step's outputs to the next one's inputs. The frequency of the clock will be determined by the longest step of the pipeline thus dividing the process in several simple steps can improve throughput, at the cost of a little increase of latency (Figure 12).

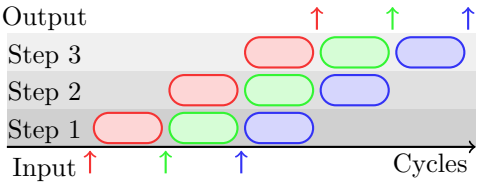


Figure 12: Simple three-steps pipeline

The introduction of a clock also makes interfacing with other processors easier. This is particularly relevant in the case of FPGAs that are often used in heterogeneous architectures.

The pipeline is built at the function level, which means that inputs and outputs of data-flow graphs are synchronized : inputs must be provided at the same time and outputs will be ready simultaneously, instantly (with only a *delta-delay*) or after a given number of clock cycles.

3.4.2 Introducing registers

The introduction of pipeline stages lies in the addition of registers through the computation process that will draw boundaries between the different parts of the calculation. Inferring the good places where to split the computation appears a complex job and the programmer should have a way to specify it to keep a precise control on the produce hardware configuration.

The `skip` statement

To study the impact and feasibility of this feature without modifying the language, the unused `skip` statement was reserved to indicate the introduction of stages on the computation of the current block of code.

```
function inc(bool a,
         int b) -> int:
    if a:
        b = b + 1
        skip
    return b
```

When a `skip` statement is encountered, vertexes representing variables are transformed by adding to them a register. The compilation continues for ther remaining block's statements using these new vertexes. At the end of the block, unmodified variables are restored to their previous state.

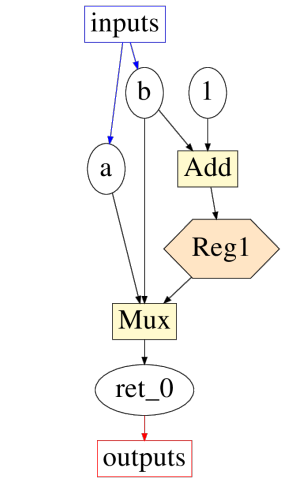


Figure 13: Resulting data-flow graph

Modifying the language

From the perspective of adapting the language to this specific compilation target, two ways of introducing registers could be considered :

- **A block boundary** : A specific keyword `split` or `stage` could be introduced to notify the splitting of a block into two parts.
- **A variable boundary** : Because splitting the whole block into two parts can be irrelevant (if one of the variables is known to be already the result of a several cycle computation), registers should be addable to precise variables. It can currently be achieved with the block boundary using a named block that modifies only the desired variable, but this work-around befuddles the resulting code.

These modifications could be gathered with a unique keyword that can be used either alone or with variables.

3.4.3 Synchronizing sources

During the building of data-flow graphs, registers are only added where `skip` statements were encountered. The resulting graph may contain inconsistencies between sources as shown in Figure 13, where one mux source is ready after the others.

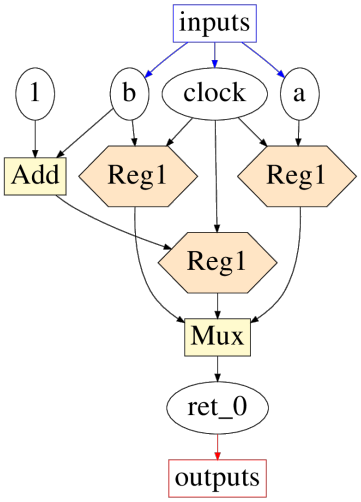


Figure 14: Synchronized data-flow graph

To ensure that the wanted result is computed, two constructions can be considered :

- Keeping the same inputs as long as they may still be needed
- Adding registers to delay early sources.

The second option enables the pipeline to operate at its higher rate (one input per clock tick). Figure 14 shows the synchronized graph of the above example. It can however delay unnecessarily the result in the case of `mux` where the unselected option is ready after the selected one.

3.5 Adding loops

Loops are commonly used in software programming languages. However, HDL such as VHDL only allow loops with statically known number of iterations that can be unfold in the resulting design.

3.5.1 General considerations on loops

A loop is a syntactical construction that enables a set of instructions (the body) to be successively executed a (possibly zero) number of time that depend on the value of a specific expression (the condition). Several facts need to be kept in mind to introduce their support.

- Contrary to other statements, the length of the calculation may not be statically known, because the number of iteration is often dynamically determined.
- The same piece of code/circuit must be used several times in the computation of a single entry, owing to the fact that circuits/programs are spatially finite.
- This multiple use of the same circuit prevents to build a pipeline as it was previously described. The synchronization of vertexes' sources must be adapted to this new kind of unknown latency.

Loops where considered is the most general case, with no guarantee on their execution. When additional information is known, more relevant constructions can be set-up. A complete example (including simulation) is given in appendix.

Remark : Using unbound loops does not seem to be an appropriate design for hardware description. However, it may sometimes be relevant and adding their support made a comprehensive time analysis unavoidable, which improved greatly the processing of code without loops.

3.5.2 Introducing loops in the data-flow graph

The conversion of `while` statements to data-flow graph vertexes is worth mentioning. A special `while` vertex is provided, whose sources are the variables that are used in the loop and which targets are new symbolic vertexes that represent the updated value of variables modified by the loop.

Additionally, this vertex contains two data-flow graphs : one for the body of the loop and an other for the computation of the condition.

3.5.3 Adding state signals

During time analysis, body and condition data-flow graphs of `while` vertexes are merged in the global graph, surrounded with additional logic.

In agreement with the computation model of VHDL, every part of the design will be evaluated at each cycle, whereas the computation needs several ones. A way to distinguish at runtime outputs that are meaningful from others (intermediate or irrelevant values) if thus needed. The addition of state signals gives a way to represent the progress of the computation.

A simple approach uses only two state signals : a *Start* input signal that is activated when entries are sent, and a *Done* output one that indicates when circuit's outputs are ready.

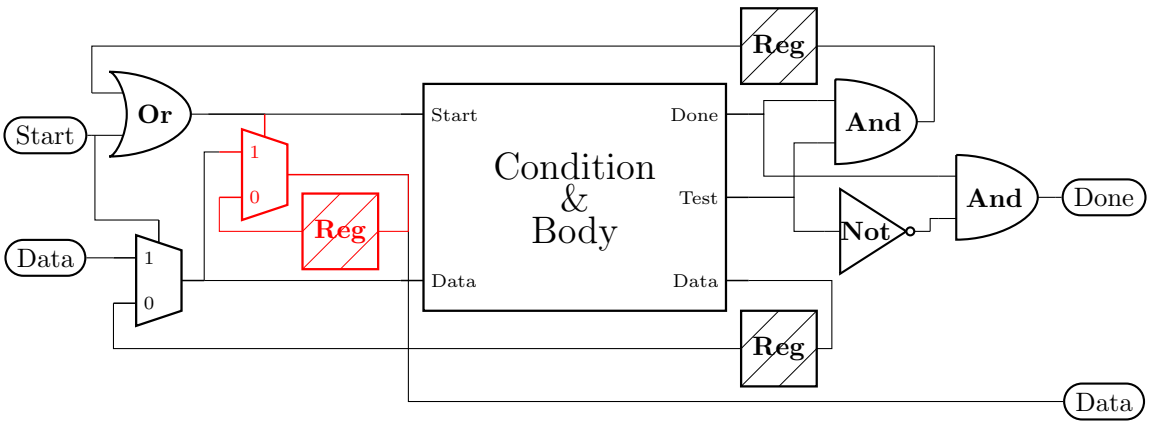


Figure 15: Additional logic around loops (American circuit diagram style[2])

[2]Node shapes convention can be found here

Figure 15 details the additional logic built around loop's body and condition in a simple case where they are supposed to be ready at the same time. This diagram shows that an extra iteration is computed, that is dropped if not necessary. The results of the previous iteration (or the inputs) are stored in a buffer waiting for the condition value to be ready.

3.5.4 Synchronizing sources

The synchronization of data that can be ready an arbitrary time after the beginning of calculation necessitates the use of buffers. The red part of Figure 15 circuit shows a simple buffer that updates its value regarding a storing signal.

A first approach to sources synchronization consists in adding a *delay* information (number of cycles needed or special unknown value) to vertexes that handles the length of computation from inputs to it. Buffers are then added between a vertex and its sources if one of them has an unknown delays (early sources will "wait" in buffers until they are all ready). This leads to an over used of such components (for example if sources come from the same while construction) that should be avoided.

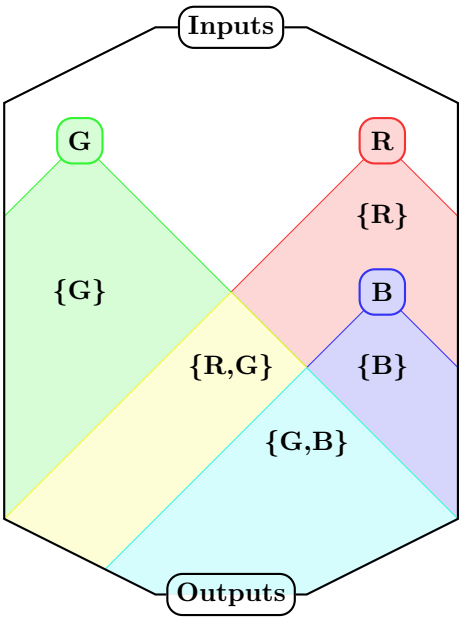


Figure 16: Abstract representation of a data-flow graph with 3 inner seeds and their zone of influence

A more relevant study of delays can save a lot of these useless buffers : *calculation seeds* are added to represent circuit places where the result can be ready at an arbitrary time after the calculation of sources and vertexes are decorated with the set of *seeds* from which they are delayed by a known number of cycles. These seeds are either a special input seed or inner seeds that correspond to vertexes with an unknown latency (while nodes or functions call with an unknown latency). The data-flow graph is thus divided into parts sharing the same set of seeds, and buffers are only needed when sources are in different zones.

3.5.5 Improving concurrency

The introduction of loops greatly reduces the possible concurrency : a new computation cannot be started as long as a loop is still running because it could be still unfinished when new data will come to it. Some enhancements can nevertheless be achieved.

Improving loop construction

Synchronizing body and condition of loops is unnecessary : the computation can stop as soon as the result of the condition is ready and false. However, it necessary to wait for the body to complete before starting to evaluate the next condition's value. Figure 17 shows how the evaluation of a loop can be achieved in a optimal number of cycles (*Sync* part describes storing registers used to delay previous iteration values).

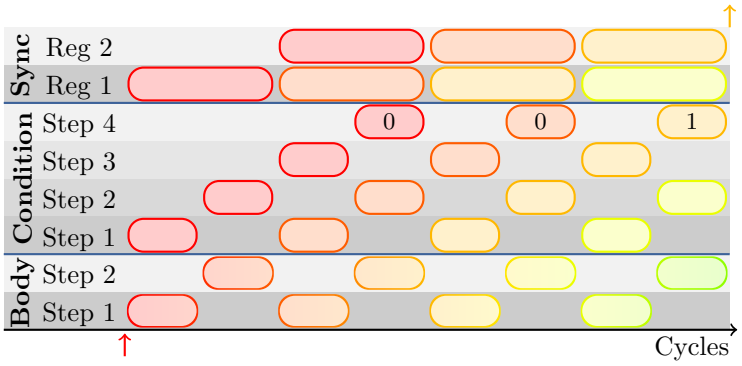


Figure 17: Optimized loop construction in the case of a 2-cycles long body and a 4-cycles long condition

Defining concurrency

Without further knowledge, a decent concurrency/space trade-off consists in setting the maximum number of values that can be processed at the same time. Fixed-size buffers are then needed before loops to ensure the intermediate results to be stored as long as it is still computing the previous value.

3.6 Future work

This study has set up some bases of compilation toward FPGAs, but several subjects could be further studied :

- **Targeting net-list** : To avoid as much as possible the dependency toward proprietary tools, net-lists could be produced instead of VHDL.
- **Type representation optimization** : A deeper analysis of types could lead to more compact representation of typed values, with less redundancy.
- **Data-flow graph optimization** : Several simplification of data-flow graphs could improve the generated code briefness and performances.
- **Using constraint informations** : Analysing constraints on types would enable to adjust more sharply the representation to the possible values.
- **Adding bounds on loops** : Further information on loop execution (bounds on the number of iterations) that should be checkable by the theorem prover would lead to a very interesting analysis on the building of an optimized pipeline.
- **Inlining relevant function** : In order to avoid synchronization of inputs and outputs, the data-flow graph of certain called functions data-flow graph could be inserted in the caller's one.
- **Improving the language support** : introducing array, terminal recursion (as it is space-bounded, with loop construction), references, *etc* could be worth considering.

Conclusion

Hardware configurations cannot be designed the way software programs are: FPGAs deeply differ from CPUs execution environments on space limitations and execution model.

Using high level software programming languages to describe hardware may still be relevant on the condition of finding a reliable way to convert a list of sequential instructions to a parallel description. The data-flow graph enables to analyse automatically a first kind of parallelism (fine-grained one) but a wider one (the pipeline) must be set up to fully take advantage of this specific hardware. Because of the numerous options that can be considered, it needs to be controllable by the programmer with special constructions or statements that are meaningless from a software programming point of view.

High-level abstractions (types, flow typing) and some functional traits of language are appropriated for hardware compilation and can be converted with simple constructions. They enable to code easily and to benefit from Whiley's features (extended type checking, specification).

The construction around loops highlight a major limitation in using imperative languages for hardware descriptions. Even if the built work-around shows a option to support general loops, the very weak control over the built state machine should be carefully use (the compiler could issue warnings about them), a better solution could consist in providing high-level language objects that exhibit their internal construction so that additional signal can be used in the sources (*e.g.* to explicitly synchronize two loops or to control precisely concurrency).

References

[BRS13] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[CN03] João MP Cardoso and Horácio C Neto. Compilation for fpga-based reconfigurable hardware. *IEEE Design & Test of Computers*, 20(2):65–75, 2003.

[CNK+12] Tomasz S Czajkowski, David Neto, Michael Kinsner, Utku Aydonat, Jason Wong, Dmitry Denisenko, Peter Yiannacouras, John Freeman, Deshanand P Singh, and Stephen D Brown. Opencl for fpgas: Prototyping a compiler. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[FLL+13] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Pldi 2002: Extended static checking for java. *ACM Sigplan Notices*, 48(4S):22–33, 2013.

[IS95] Christian Iseli and Eduardo Sanchez. A c++ compiler for fpga custom execution units synthesis. In *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 173–179. IEEE, 1995.

[MT13] Bryan Mealy and Fabrizio Tappero. Free range vhdl. *URL: http://www. freerangefactory. org/dl/free_range_vhdl. pdf*, 2013.

[NSD08] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, 2008.

[PBD+08] Andrew Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, Prasanna Sundararajan, and Susan Eggers. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 173–178. IEEE, 2008.

[Pea13] David J Pearce. Sound and complete flow typing with unions, intersections and negations. In *VMCAI*, pages 335–354. Springer, 2013.

[Pea14] David J. Pearce. *The Whiley Language Specification*, 2014.

[Pea15] David J Pearce. Integer range analysis for whiley on embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on*, pages 26–33. IEEE, 2015.

[PTP94] Matthew F Parkinson, Paul M Taylor, and Sri Parameswaran. C to vhdl converter in a codesign environment. In *VHDL International Users Forum. Spring Conference, 1994. Proceedings of*, pages 100–109. IEEE, 1994.

[TGP07] Justin L Tripp, Maya B Gokhale, and Kristopher D Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3), 2007.

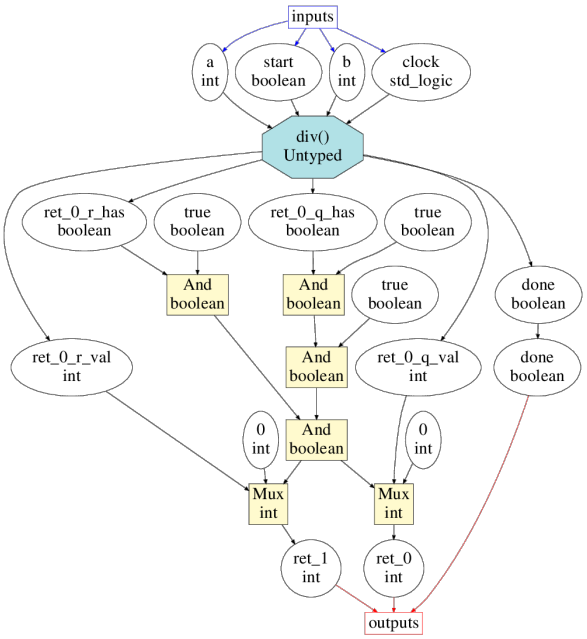
[Xil11] Xilinx. *Spartan-3 Generation FPGA User Guide*, June 2011.

Annex : A working example

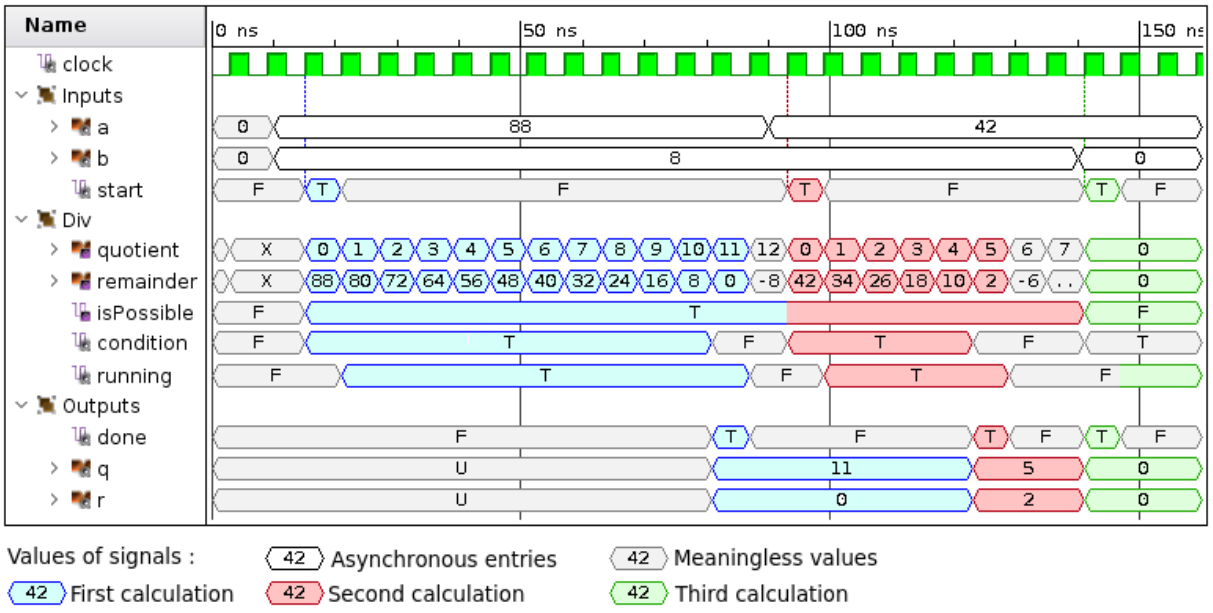
A naive implementation of Euclidean division of positive integers was chosen to enlighten the main features of the compiler.

```
1  type Result is {int q, int r}
2
3  function div(int a, int b)
4          -> Result|null:
5      if b==0:
6          return null
7      Result d = {q:0,r:a}
8      while d.r >= b:
9          d.q = d.q + 1
10         d.r = d.r - b
11     return d
12
13 function main(int a, int b)
14         -> (int q,int r):
15     Result|null d = div(a,b)
16     if d is Result:
17         return d.q,d.r
18     return 0,0
```

(a) Division implementation



(b) Synchronized data-flow graph of `main` function



(c) Screen-shot of the simulator. (Colours were modified to improve readability) Computations start at the first clock rising edge after the change of a entry

Figure 18: Example of assignment

The `div` function leads to a complex data-flow graph. The one of `main` function highlights how function invocation is represented and how flow typing operation `is` operator is converted (`and` nodes).

The generated VHDL code is introduced in a testing project that defines the values of inputs through time. Its simulation shows that expected values a actually computed and that it requires a variable time (delay between *start* and *done* `True` values).