Towards Compilation of an Imperative Language for FPGAs

Baptiste Pauget
Département d'informatique, École Normale Supérieure, France

David J. Pearce
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

Alex Potanin
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

Abstract

Field-Programmable Gate Arrays (Fig's) have been around since the early 1980s and have now achieved relatively widespread use. For example, FPGAs are routinely used for high-performance computing, financial applications, seismic modelling, DNA sequence alignment, software defined networking and, occasionally, are even found in smartphones. And yet, despite their success, there still remains something of a gap between programming languages and circuit designs for an FPGA. We consider the compilation of an imperative programming language, Whiley, to VHDL for use on an FPGA. Whiley supports compile-time verification of function pre- and post-conditions. Hence, our motivation is to adapt this technology for verifying properties of hardware designs.

ACM Reference format:
Baptiste Pauget, David J. Pearce, and Alex Potanin. 2017. Towards Compilation of an Imperative Language for FPGAs. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, New York, NY, USA, January 01–03, 2017 (PL'17),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Field-Programmable Gate Arrays (FGPAs) can no longer be considered a "new" technology. Since their introduction in the early 1980's by Xilinx they have finally reached the mass market. FPGAs are used in a variety of arenas such as *HPC acceleration* [14], *seismic modelling* [31], *encryption* [1, 28], *financial applications* [33, 44], *DNA sequence alignment* [15, 35], *software-defined networking* [30, 41], and are even found in smartphones (e.g. Samsung Galaxy S5 and iPhone 7). FPGAs provide a middle-ground between Application-Specific Integrated Circuits (ASICs) and general purpose CPUs. Compared with a CPU, an FPGA offers the performance benefits from a gate-level abstraction [4] and can provide higher throughput with lower power usage than a CPU or GPU [43, 44]. Bacon *et al.* note the following [4]:

> *"When it comes to power efficiency (performance per watt), however, both CPUs and GPUs significantly lag behind FPGAs"*

PL'17, January 01–03, 2017, New York, NY, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

When compared with traditional ASIC manufacturing processes, FPGAs offer greater flexibility as, in some sense, hardware becomes more like software. This is more even more apparent today since modern FPGAs can be reconfigured (or partially reconfigured) in a matter of milliseconds.

Roughly speaking, an FPGA is made up of a large number of configurable logical blocks (CLBs). Each logic block typically consists of some number of lookup tables (LUTs), flip-flops, and full-adders. Modern FPGAs, such as Xilinx's Virtex-7 series, are manufactured at the nanometer scale (e.g. 28nm), operate in the 500Mhz frequency range and provide millions of logic blocks. They may also contain other components, such as block RAMs (BRAMs), Digital Signal Processing slices (DSPs), communication interfaces (e.g. Ethernet, PCI), and processor cores (e.g. ARM) [46]. The design process for an FPGA typically revolves around the use of Hardware Description Languages (HDLs) such as VHDL [24] or Verilog [25]. These languages offer relatively low-level abstractions and, as such, are ideally suited to thinking in terms of low-level building blocks such as gates, registers and multiplexers [4]. However, they offer few high-level abstractions and this makes developing for FPGAs costly and time consuming [10].

We are interested in compiling programs written for an imperative programming language, Whiley, for an FPGA [40]. Whiley is unusual in providing support for function specifications (i.e. pre-/post-conditions) which can be verified at compile-time. Whiley follows in a long lineage of tools which emphasise the use of automated theorem provers, such as Simplify [13] or Z3 [12], for static verification. Similar languages include ESC/Java [19], Spec# [7], Dafny [32], Why3 [18], VeriFast [27], and SPARK/Ada [6]. Such languages, of course, are used to verify properties of *software* systems. From this, a natural question arises: *can they verify properties of hardware systems?* This provides the general motivation for our work, but raises an immediate challenge: *how do we compile imperative languages, such as Whiley, for FPGAs?* This is the issue we examine here.

The problem of compiling high-level languages for an FPGA is, of course, not a new problem. Many so-called "C-to-gates" systems exist for compiling subsets of C for FPGAs, such as Stream-C [21], Handle-C [9], Autopilot [47] and more. Other approaches implement libraries or DSLs for *generating* circuit designs from program code, such as ASC [36, 37], JHDL [8] and more. And yet, we find these systems remain unsatisfactory. Either they are too "gate-oriented" or overly restrictive in terms of supported language features. Part of the challenge is that it remains unclear what it even means to compile a program for an FPGA. Certainly, FPGAs are well suited for particular kinds of computation. Perhaps the most common paradigm is that of hardware acceleration, whereby a loop nest — or *kernel* — is converted into a data pipeline on the FPGA. Brodtkorb *et al.* make the following statement in this regard [10]:

> *"A key element to obtaining high-performance on FPGAs is to use as many slices as possible for parallel computation. This can be achieved by pipelining the blocks, trading latency for throughput; by data-parallelism, where data-paths are replicated; or a combination of both"*

This gives insight into the difference between programming for a CPU versus for an FPGA. Specifically, when programming for an FPGA we must utilise as much silicon as possible to maximise performance. Thus, we cannot view a function as an entity to execute sequentially; rather, we must view it as something to be broken up into a pipeline.

A key starting point here is the Lime language developed for the Liquid Metal project [2, 4, 23]. This extended Java in a number of ways to provide better support for FPGA programming. We find that there are a number of similarities between their extensions and the Whiley language. However, at the same time, Lime remained focused on hardware acceleration, rather than more general forms of computation. For example, one could not implement a CPU core within the Lime framework as this does not fit the accelerator model.

The contributions of this paper are as follows:

- We present a novel approach to compiling imperative languages for an FPGA. This is based around a language primitive which allows fine-grained control over the generated pipeline.
- We have developed a prototype implementation which compiles programs in the Whiley language into VHDL which can then be synthesised for an FPGA.

2 Background

In this section, we provide some background on FPGAs and a short introduction to the Whiley language.

2.1 Field-Programmable Gate Arrays

An FPGA consists, roughly speaking, of an array of Configurable Logic Blocks surrounded by I/O pads and a programmable interconnect network which, typically, constitutes most of the area [17]. Figure 1 provides a simplistic illustration following a traditional "island-style" layout.

Configurable Logic Blocks (CLBs). Roughly speaking, a CLB contains various components, normally including some

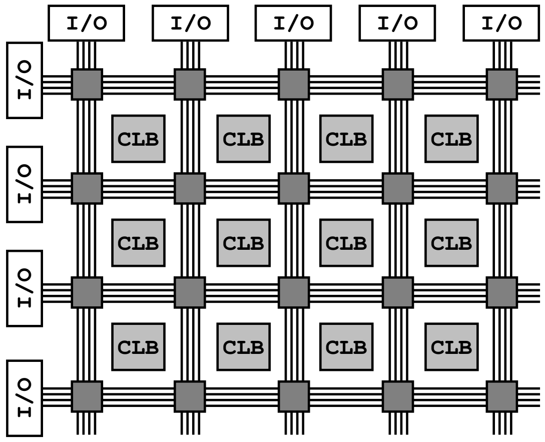


Figure 1. A simplistic architectural view of an FPGA.

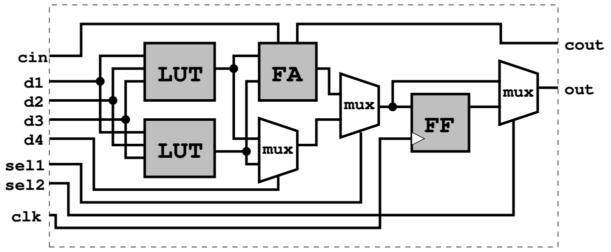
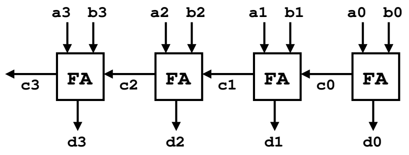


Figure 2. A simplistic view of a CLB.

lookup tables (LUTs), storage elements (e.g. D-type flip-flops), and other machinery such as carry logic, multiplexers, etc (the exact details differ between devices).

Figure 2 illustrates an example CLB made up from two three-bit lookup tables (LUT), one full adder (FA), one D-type flip flop (FF) and three multiplexers (MUX). We can, for example, implement a four-bit LUT by combining the two LUTs together and bypassing the adder and the flip flop. Or, we could implement a simple ripple-carry adder by chaining several CLBs together (i.e. by connecting *carry out* to successive *carry in*, etc), making the LUTs sensitive only to `d1` and, again, bypassing the flip flop. Finally, we could implement one-bit of a register by again making the LUTs sensitive only to `d1`, and bypassing the adder.

Building Blocks. A CLB can implement any of the classical gates (e.g. *nand*-, *nor*-gates, etc) or act as a flip-flop, multiplexer or adder. As such, the basic building blocks of digital electronics are available. A four-bit *ripple-carry* adder can be constructed by chaining four CLBs together as follows:

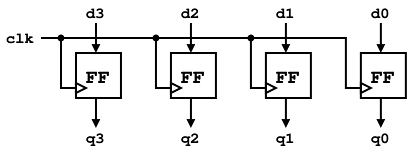


This is about the simplest possible implementation of an adder and trades performance for a minimal number of gates.

The time taken for the result to propagate through the adder once the inputs are set (latency) is proportional to the bit-width. In this case, it is four times the *gate delay* for an adder in our CLB implementation. Other, faster, implementations are possible, such as the *carry lookahead* and *carry skip* adders which reduce the delay at the expense of more gates.

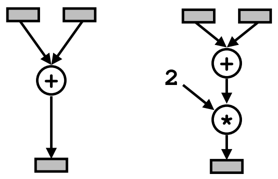
The implementations of other arithmetic operators follow in roughly the same fashion and, likewise, different choices are available with different trade-offs. We note that multiplication is more expensive in terms of both gates and latency compared with an adder, and division even more so. In contrast, bitwise operators have constant latency determined by the gate delay across a CLB.

Aside from the basic arithmetic operators, the next most important building block is an N-bit *register*. The following illustrates a simple four-bit register:



Unlike for arithmetic operators, a register is *clocked*. When the clock changes (usually either on a *rise-edge* or *falling-edge* signal), the current state of the inputs is fixed on the outputs follow this until the next clock. Registers are import as they offer precise timing determined by the clock.

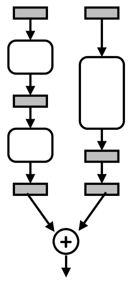
Timing. In the construction of digital circuits, timing is of course critically important. Consider the following circuits:



The circuit on the left reads from two N-bit registers, passes their values through an N-bit adder, and writes the result into another N-bit register. The datapath on the right is more complex, first adding the inputs then multiplying the result by 2. The width of the datapaths (i.e. the value of N) is not specifically important here (though must be statically known). Assuming the input and output registers are clocked together then the maximum clock frequency is determined by the minimum gate delay between the input and output registers. Let's assume (as seems likely) that the gate delay for the right circuit is greater than the left circuit. Thus, the maximum frequency for clocking the right circuit is lower than for the left. Further, if both are part of some (larger) circuit and clocked together then, clearly, we must conservatively choose the lower clock rate.

Our circuits above help highlight the distinction between *combinatorial* (or *time-independent*) and *sequential* logic. In the former, the outputs are a real-time function of the inputs (subject to latency). In the latter, the outputs change only on clock signals, rather than in real-time. Thus, each of our circuits taken as a whole is sequential whilst their internals sandwiched between registers are combinatorial.

Register Balancing. The final piece of the jigsaw relates to the problem of combining sequential circuits with different *cycle latency's* (i.e. clock cycles from when an input is received to when the corresponding output is produced). For example, consider the following high-level circuit:



Here, we assume that between the registers represents arbitrary blocks of combinatorial logic. Furthermore, we desire that each output produced corresponds to inputs *which arrived at the same time*. To ensure this happens, we have added an additional register after the combinatorial block on the right-hand side to balance the numbers on each path. This simply ensures that values on the right-hand side remain in sync with their corresponding values on the left-hand side.

At this point, we have now covered the necessary background on digital circuit design required for the remainder. As such, we now turn our focus to the Whiley language.

2.2 Whiley

The Whiley programming language has been developed from the ground up to enable compile-time verification of programs [40]. The Whiley Compiler (WyC) attempts to ensure that every function in a program meets its specification. When it succeeds in this endeavour, we know that: 1) all function post-conditions are met (assuming their pre-conditions held on entry); 2) all invocations meet their respective function's pre-condition; 3) runtime errors such as divide-by-zero, out-of-bounds accesses and null-pointer dereferences are impossible. Note, however, such programs may still loop indefinitely and/or exhaust available resources (e.g. RAM).

Specifications. Whiley allows explicit *pre-* and *post-conditions* to be given for functions. For example, the following function accepts a positive integer and returns a natural number:

```
function decrement(int x) -> (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0:
    //
    return x - 1
```

Here, `decrement()` includes **`requires`** and **`ensures`** clauses which correspond (respectively) to its *precondition* and *postcondition*. In this context, `y` represents the return value and may be used only within the **`ensures`** clause.

The Whiley compiler reasons by exploring control-flow paths. For example, it easily verifies the following:

```
function max(int x, int y) -> (int z)
// Must return either x or y
ensures x == z || y == z
// Return must be as large as x and y
ensures x <= z && y <= z:
    //
    if x > y:
        return x
    else:
        return y
```

Here, multiple **`ensures`** clauses are given which are conjoined to form the function's postcondition. We find that allowing multiple **`ensures`** clauses helps readability and, likewise, multiple **`requires`** clauses are permitted as well.

Data Type Invariants. Type invariants over data can also be explicitly defined:

```
// A natural number is an integer greater-than-or-equal-to zero
type nat is (int n) where n >= 0
// A positive number is an integer greater-than zero
type pos is (int p) where p > 0
```

Here, the **`type`** declaration includes a **`where`** clause constraining the permitted values. The declared variable (e. g., `n` or `p`) represents an arbitrary value of the given type. Thus, `nat` defines the type of natural numbers. Likewise, `pos` gives the type of positive integers. Constrained types are helpful for ensuring specifications remain as readable as possible. For example, we can update `decrement()` as follows:

```
function decrement(pos x) -> (nat n):
    //
    return x - 1
```

Loop Invariants. Whiley supports loop invariants as necessary for reasoning about loops. The following illustrates:

```
function sum(int[] xs) -> (nat r)
// Every item in xs is greater or equal to zero
requires all { i in 0..|xs| | xs[i] >= 0 }:
    //
    int s = 0
    nat i = 0
    while i < |xs| where s >= 0:
        s = s + xs[i]
        i = i + 1
    return s
```

Here, a bounded quantifier enforces that `sum()` accepts an array of natural numbers (which could equally have been expressed as type `nat[]`). As expected, summing an array of natural numbers should yield a natural number (recall arithmetic does not overflow). The loop invariant helps the compiler generate a sufficiently powerful verification condition to statically verify that `sum()` meets its specification.

Flow Typing & Unions. An unusual feature of Whiley is the use of a *flow typing system* (see e. g., [38, 45]) coupled with *union types* (see e. g., [5, 26]). To illustrate, we consider null values. These have been a significant source of error in languages like Java and C#. The issue is that, in such languages, one can treat *nullable* references as though they are *non-null* references (Hoare calls this his billion dollar mistake [22]). Whilst many approaches have been proposed [16, 34], flow typing provide an elegant solution:

```
// Return index of first occurrence of c in str, or null if none
function index(string s, char c)->int|null:
    ...
```

Here, `index()` returns the first index of a character in the string, or **`null`** if there is none. The type **`int|null`** is a union type, meaning it is either an **`int`** *or* **`null`**. Furthermore, to use the result, one must first check whether it *is* an **`int`** using the **`is`** operator (similar to `instanceof` in Java):

```
    ...
    int|null idx = index(...)
    if idx is int:
        ...              // idx has type int
    else:
        ...              // idx has type null
```

This first ensures `idx` is an **`int`** before using it in the true branch and Whiley's flow type system automatically retypes `idx` on the true (resp. false) branch to have type **`int`** (resp. **`null`**). The use of union types here to manage **`null`** values is closely related to the use of option types in languages like Haskell, Scala and, more recently, Java 8.

3 Implementation

We now present our prototype implementation for compiling Whiley programs to VHDL for synthesis onto an FPGA. The `Whiley2VHDLCompiler` is an open source plugin available from `github.com/BaptP/Whiley2VHDLCompiler`.

3.1 Overview

Figure 3 provides an overview of the information flow through the Whiley Compiler. Whiley source files are converted into (binary) `wyil` files; in turn, these are converted into binary `class` files (for execution on the JVM) or, using the extension presented here, to VHDL files. The Whiley Intermediate Language (WyIL) is essentially a binary form of the Whiley source with additional information embedded (such as fully

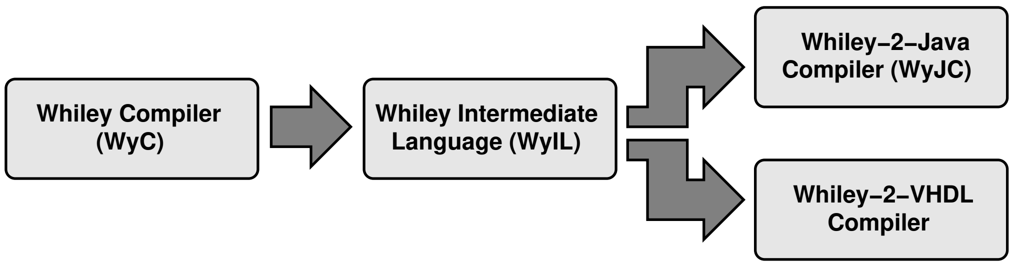

Whiley Compiler (WyC)
Whiley Intermediate Language (WyIL)
Whiley–2–Java Compiler (WyJC)
Whiley–2–VHDL Compiler

Figure 3. Illustrating the compilation pipeline.

resolved names, etc). Generated VHDL files can be synthesised into bit files for uploading to an FPGA using, for example, the Xilinx ISE toolsuite. Likewise, they can be simulated using other third-party tools, such as GHDL.

3.2 Data Representation

The compilation of Whiley data types presents the first challenge as, for example, they were not designed with the finite constraints of an FPGA in mind. In particular, all data types on an FPGA must have statically-known bounds on the number of bits they occupy.

Integers. The representation of integer datatypes is relatively straightforward as these are directly supported in VHDL. One challenge is that integers in Whiley are *unbounded* which, although useful for verification, is something of a hindrance here. However, an arbitrary number of finite datatypes can be supported through the use of type invariants. The following illustrates:

```
type i8 is (int x)
where x >= -128 && x <= 127
```

This defines a signed integer within the range $-128 \ldots 127$ which can be encoded in one byte using a twos-complement representation. We can specify arbitrary bounds here and rely on the compiler to determine the minimal representation (e.g. using *bitwidth analysis* [11, 20, 39, 42]).

Records. These are relatively easy to represent since they have a finite number of fields and, hence, have finite representation if their fields do. For example:

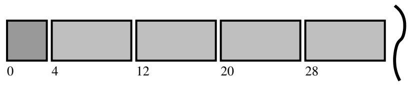
```
type Point is { i8 x, i8 y }
```

This type can be encoded easily enough in two bytes. Furthermore, since records in Whiley have value semantics, assigning a variable of type `Point` to another simply results in a bitwise copy.

Arrays. These can also be represented provided they have statically known lengths. For example:

```
type string is (i8[] s) where |s| < 10
```

This defines a **`string`** to be a sequence of at most ten bytes. To encode an array we require all elements have finite representation and, additionally, we must determine the minimal representation for the length variable. Thus we can encode a **`string`** like so:

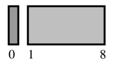


Here, the minimal representation for the length is determined as a four-bit nibble. Thus, we begin to see the flexibility offered by an FPGA in terms of data representation. Furthermore, we begin to see the benefits of type invariants. Since the length is at most ten, we know that not every combination of bits is possible in the length variable. Thus, in principle, our compiler can exploit this knowledge to further optimise the design in a way that is difficult or impossible for e.g. a general-purpose VHDL compiler.

Unions. These represent something of a challenge as we must not only encode the payload itself, but also an appropriate tag. Consider this simple union:

```
type ni8 is (i8|null)
```

This defines `ni8` as a union of a byte and **`null`**. As such, we determine the worst-case payload size by examining each case and the number of bits required to distinguish cases. Thus, we can encode an `ni8` as follows:



Here, the maximum payload size is one byte since `i8` has the largest bitwidth of any case. Furthermore, there are only two cases and, hence, only one bit is required for the tag.

Others. Aside from the relatively standard data types discussed above, Whiley supports a number of other types which are not easily encoded. This includes *references*, *function pointers*, *recursive types* and *open records*. In addition, the primitive type **`any`** does not have a finite representation. The following illustrates a recursive type:

```
type LinkedList is null | Link
type Link is { LinkedList next, i8 data }
```

In principle this could be encoded onto an FPGA if its maximum depth was bounded. Unfortunately, there is no operator for this (unlike for arrays) and, hence, it is very difficult for the compiler to determine a bound.

Another example is that of open records which, in some ways, are similar to interfaces in Java:

```
type Item is { i8 kind, ... }
```

The "`...`" signals that this type is *open* (i.e. may include zero or more additional fields). Thus, `Item` represents *any* record type which has at least a field `kind`, and may have arbitrarily many additional fields.

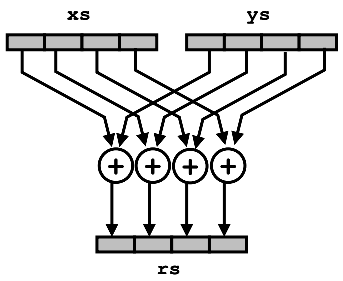
3.3 Control Flow

Whilst the representation of Whiley data types on an FPGA is relatively straightforward, the translation of control-flow is more challenging. The inherent problem when compiling for an FPGA is to determine how and when to perform operations in parallel. In short, if one does not expose parallelism in some way then the benefits from using an FPGA are likely to be limited. Most existing high-level synthesis systems (e.g. Stream-C [21], Handle-C [9], etc) provide explicit constructs for specifying parallelism. On an FPGA (or indeed in any digital circuit) the concept of parallelism has two dimensions. We can think of these as roughly similar to the usual notions of *data-* and *task-*parallelism. For example:

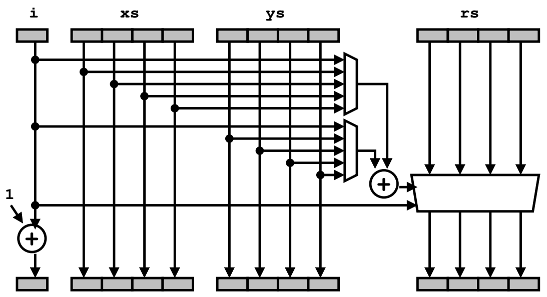
```
type vec is (i8[] vs) where |vs| == 4

function add(vec xs, vec ys) -> (vec zs):
   i8 i = 0
   vec rs = [0,0,0,0]
   while i < 4:
      rs[i] = xs[i] + ys[i]
      i = i + 1
   return rs
```

Here, `vec` is defined as an array of four bytes, with `add()` as a function for adding two `vec`s. We can parallelise this loop in different ways. For example, we could employ a simple data-parallel implementation:



Although perhaps an optimal implementation, this relies on our compiler's ability to statically determine the number of loop iterations and, furthermore, that there are no real loop-carried dependencies. In situations where such information cannot be determined (e.g. for more complex loops), an alternative is a task parallel implementation. That is, a pipeline where each stage looks roughly like this:



Here, two multiplexers are used to select the relevant array elements from `xs` and `ys` and another (larger) multiplexer controls writing the result to the appropriate element. If the compiler can ascertain that there are exactly four loop iterations, then it can construct a pipeline consisting of four such stages. If not, then it must feed the output of the stage back into its input, and include additional machinery for handling loop termination.

The key point here is the way these two implementations handle parallelism. In the first we have a vector operation which, for given inputs, produces new outputs in one cycle. In the second, we have a pipeline operation which, for given inputs, produces new outputs after, say, four cycles. Furthermore, in the latter, at any given moment we have multiple executions of `add()` in progress at (i.e. assuming all pipeline stages are full). Thus, the challenge for compiling to an FPGA is that we have one source-level representation of our program but multiple possible implementations. Whilst this is already true to some extent when targeting a normal CPU, the effect is more profound on an FPGA. Indeed, there are many more implementations of our loop than we have considered. For example, we might try to increase CLB utilisation (hence, throughput) by introducing additional pipeline stages (e.g. before and after the array element addition). And, of course, as discussed before there are multiple different implementations of our arithmetic operators that we could choose between. For example, in the above pipeline stage, the increment operation on `i` can afford to be relatively slow (hence, use fewer CLBs) since it is not the critical bottleneck.

Approach. We adopt a novel approach to compiling imperative code for an FPGA which allows fine-grained control over the placement of registers. This allows the programmer to tune the amount of pipelining employed to suite his/her needs. There are several facets:

- **Instruction Parallelism**. Our compiler automatically identifies instruction-level parallelism by translating functions into a data-flow graph representation (e.g. similar to that done elsewhere [2, 36]).
- **Data Parallelism**. Our compiler assumes that any data-parallelism present in the target function has already been extracted (e.g. by a previous compiler stage). That

is, any loops with statically known bounds are assumed to have already been unrolled. This is a simplifying assumption as loop analysis is not our focus.

- **Explicit Staging**. An explicit statement is added to Whiley with which the programmer can signal the placement of registers. We refer to this as the **`stage`** statement (though, for simplicity, our prototype implementation overloads the existing **`skip`** statement).
- **Register Balancing**. Our compiler performs *register balancing* to ensure that, whatever the placement of **`stage`** statements, values propagate through the pipeline in sync (i.e. intermediate values generated from the same inputs are propagated together — recall §2.1).
- **Pipeline Stalls**. Our compiler supports the compilation of arbitrary loops. Such loops are assumed to terminate, though may have an unknown number of iterations. Since progress through the pipeline may stall waiting for a given loop to terminate, control-signals are added to dictate when the next data item is available.

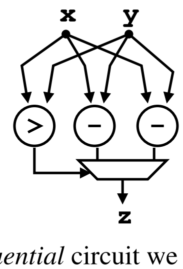
This approach provides a novel mechanism for controlling compilation for the FPGAs. We note, however, that it does not allow control along all dimensions. For example, it provides no means of dictating what underlying implementation to use for a given operator. Likewise, instruction-level parallelism is identified automatically by our static analysis, and the programmer has limited control over this.

3.4 Conditionals

We now present some examples to illustrate the main ideas. Consider the following function:

```
function diff(int x, int y) -> (int z):
    if x > y:
        return x - y
    else:
        return y - x
```

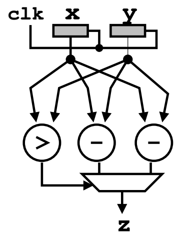
Our compiler produces a combinatorial circuit for this:



To generate a *sequential* circuit we need to introduce one or more pipeline stages, such as in the following:

```
function diff(int x, int y) -> (int z):
    stage
    if x > y:
        return x - y
    else:
        return y - x
```

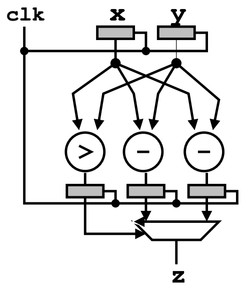
This now generates the following sequential circuit containing (effectively) one pipeline stage:



This implementation has the advantage that we can now clock data values through the pipeline and, to accommodate this, an additional `clock` signal is required. An important issue is the handling of register balancing. For example, consider this variation on the above:

```
function diff(int x, int y) -> (int z):
    stage
    if x > y:
        int tmp = x - y
        stage
        return tmp
    else:
        return y - x
```

This now generates the following sequential circuit containing (effectively) two pipeline stages:

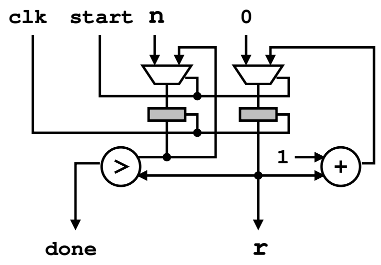


The original Whiley source code dictated a register in the true branch only. However, our compiler automatically performs register balancing to ensure that intermediate values generated from the same inputs always arrive together.

Loops. Unfortunately, supporting general loops is much more involved. For example, consider this very simple loop:

```
function count(int n) -> (int r):
    int i = 0
    //
    while i <= n:
        i = i + 1
    //
    return i
```

One of the key challenges is that we cannot know a priori how many iterations the loop will take. Furthermore, we cannot implement the loop using purely combinatorial circuits as the potential for race conditions can lead to spurious or incorrect outputs. Instead, we can translate this as follows:



This is more complex and, frankly, less elegant than our translation of condition statements. Two registers have been introduced to maintain the values of `i` and `n` as the loop proceeds.[1]

In addition to the expected inputs and outputs above we also have a new input, `start`, and a new output, `done`. The latter is relatively straightforward. Since the loop may execute an unknown number of times, it must signal when its complete and the output value `r` is available. For a similar reason we also require the `start` signal. The fundamental issue is that, whilst the loop is executing, no more values can be passed into the pipeline. That is, all input values are stalled until the loop is complete. The `start` signal is used to indicate when to start the next loop. It is assumed that, when this is signalled, any previous output value was already read.

Our loop implementation above is limited to executing one set of inputs at a time. In other words, we have lost the task parallelism present in our previous examples. This is somewhat undesirable since it limits the benefits from using an FPGA. However, we can still benefit from pipelining by using the **`stage`** statement. For example if we had two loops, one after the other, we could insert a **`stage`** between them.

Timing Analysis. In order to ensure registers are properly balanced, our implementation performs a rudimentary *timing analysis*. To do this, we simply enumerate all paths through the data-flow graph and, for each node, determine: the number of registers encountered on this path; and, the maximum number of registers encountered on any path. Using this information, we can identify where to insert registers for balancing.

Testing. To test our system, we compiled several small examples and uploaded them onto an FPGA. The FPGA used was a Papilio One 500K which is built around a Xilinx Spartan 3, along with the LogicStart MegaWing (which provides buttons and LEDs, etc). See Figure 4.

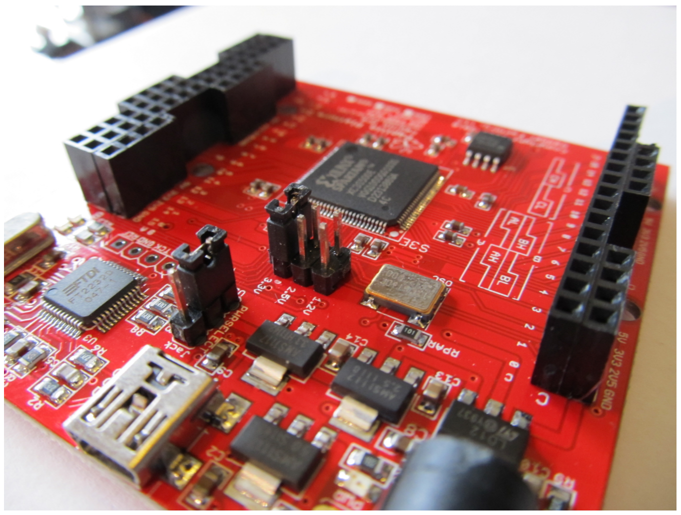


Figure 4. The Papilio One 500K

4 Related Work

We now consider related work and, we note, most approaches are either focused around very restrictive subsets of C (e.g. Stream-C [21], Handle-C [9], Autopilot [47], etc) or using libraries written in various languages (e.g. C++ [36], Java [8], Scala [3], etc) which, when executed, generate circuit designs.

The *Liquid Metal* project perhaps provides one of the most comparable efforts to this work [4]. Their primary motivation was that FPGAs can *“offer extremely high performance with very low power compared to more general purpose designs”*, but that the skills currently required for programming FPGAs are beyond regular developers [23]. The Lime programming language was developed in an effort to bridge this gap [2]. Lime is a Java-based language with significant extensions focused on both GPU and FPGA accelerators. For example, Lime extends Java with immutable value types which the authors claim: *“For compilation to hardware, Lime’s value classes are essential since they can be freely moved across a chip as a chunk of bits”* [2]. This includes immutable array values (and, we note, Whiley shares some strong similarities here). Indeed, the authors argue *“... the goals of Lime require a classification of methods into those that are pure functions”* and, again, we can draw parallels with Whiley which has first-class support for pure functions. The paradigm of computation promoted by Lime remains that of hardware acceleration. In particular, Lime provides a notion of tasks operating over data streams which can be composed and compiled into hardware pipelines.

Another relevant work is that of Kou and Palsberg who were interested in bridging the gap between FPGAs and the high-level abstractions offered by regular programming languages [29]. Their approach was to compile a regular object-oriented language into a C subset which could then be synthesised using Autopilot [47]. One of their key focuses was the fact that existing synthesis tools for C (such as Autopilot) exclude language features such as pointers and dynamic memory, etc. Nevertheless, their system only supports recursion-free programs only and requires all memory requirements be

[1]Note, technically speaking, we could have avoided a register for `n`, but our compiler does not perform the necessary analysis to determine it’s a loop constant.

known beforehand. They refer to the standard method of compiling object-oriented programs to C as a *horizontal object layout* (i.e. where classes become `structs`, etc). However, instead of this, they adopt a so-called *vertical object layout* which simplifies the handling of polymorphism.

Libraries as Generators. The ASC system of Mencer *et al.* follows a lineage of tools which provide libraries on top of standard programming languages [36, 37]. In this case, C++ is used to good effect to provide an embedded DSL for generating FPGA accelerators. Specifically, the user implements their kernel in C++ using those primitives provided by the ASC library. They may also include additional client code which interacts with the kernel by reading/writing data streams. Then, when the compiled program is executed it generates a hardware netlist on-the-fly and uploads it to the FPGA. Alternatively, a gate-level simulator can be run for debugging and testing. The primitives provided by ASC are essentially generators for arithmetic operators and comparators which support a reasonable range of implementations (e.g. two's-complement versus sign-magnitude, fast-carry versus constant-time adders, etc). The user has complete control over which implementation is selected at any point making this a relatively low-level solution. Regarding the generated datapath, the user can choose to optimise for throughput or latency. For example, optimising for throughput results in ASC inserting registers at almost every point (and, likewise, balancing each stage of the computation using FIFO buffers to ensure operands arrive together). In contrast, optimising for latency results in a purely combinatorial circuit. Thus, whilst ASC supports fine-grained control over operator implementations, it provides only coarse-grained control over the datapath itself. This contrasts with our approach which offers fine-grained control over the datapath, but not operator implementations.

JHDL is another library for generating FPGA accelerators, this time based on Java [8]. Again, simulation and circuit generation are easily interchangeable. The rough idea here is that, when a circuit element is constructed, it can be regarded as having been physically placed on the hardware. For languages with destructors, the destruction of an object would then correspond to its removal from the hardware. Since Java doesn't support destructors per se, a similar effect was achieved in JHDL through a special purpose `delete()` method. The library supports three fundamental base classes for use in implementing a circuit. The first is the *Combinatorial* class which represents a circuit that simply propagates inputs to outputs. The second is the *Synchronous* class which is clocked and produces new outputs on the clock. Finally, a *Structural* class is supported for combinations of combinatorial and synchronous circuits. An abstraction representing input/output ports for buffering data to/from the device is also provided. Thus, the Java control program writes bytes into the input port and reads results from the output port.

A similar approach to JHDL is taken in Chisel which, in this case, builds on Scala and generates Verilog [3]. One of the advantages of Scala over Java is that Chisel can make good use of operator overloading (like ASC does). Chisel also performs bitwidth analysis on the internal dataflow graph generated.

Bitwidth Analysis. Of relevance is the literature on *bitwidth analysis*. Budiu *et al* claim that "*on average 14% of the computed bytes in programs from SpecINT95 and Mediabench are useless*" [11]. Stephenson *et al.* were interested in optimising bitwidths for the purpose of compiling to silicon or FPGAs [42]. Their approach employed a dataflow analysis which propagated integer ranges in both a forwards and backwards direction. The latter helps to further narrow ranges for discoveries made downstream. For example, consider a variable which is used to index into an array. If the range of the variable is greater than the known bounds of the array, one can clip the variable's range to match the array bounds (i.e. by assuming that no actual error exists). In such case, we want to propagate this discovery back through the control-flow graph to further prune any variables flowing into this.

As another illustrative example, it is interesting to note that the majority of previous work on bitwidth analysis focuses on integer variables. In contrast, the work of Gaffar *et al* focus on the bitwidth analysis of floating pointer variables for compilation onto FPGAs [20]. Their approach is also unusual in that it focuses on the sensitivity of output variables, and correlates higher sensitivity with a need for increased bitwidth.

5 Conclusion

In this paper, we have considered the process of compiling a simple imperative language with specifications and type invariants for an FPGA. The benefit of type invariants is that they provide a powerful means to express different finite representations of data. Our implementation employs a language primitive for dividing up the datapath into stages. This requires a simple timing analysis to ensure registers are balanced. In addition, it automatically exploits task-level parallelism by converting the input program into a dataflow graph. Unbound loops present a challenge and further work is needed to investigate optimal solutions. In the future, we wish to exploit compile-time verification in Whiley to verify properties of the generated hardware systems.

References

[1] P. Anghelescu. 2016. FPGA implementation of programmable cellular automata encryption algorithm for network communications. *Comput. Syst. Sci. Eng* 31, 5 (2016).

[2] J. Auerbach, D. Bacon, P. Cheng, and R. Rabbah. 2010. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proc. OOPSLA*. 89–108.

[3] J. Bachrach, H. Vo, B. C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proc. DAC*. ACM Press, 1216–1225.

[4] D. F. Bacon, R. M. Rabbah, and S. Shukla. 2013. FPGA programming for the masses. *CACM* 56, 4 (2013), 56–63.

[5] F. Barbanera and M. Dezani-Cian Caglini. 1991. Intersection and Union Types. 651–674.

[6] J. Barnes. 1997. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Inc., Reading.

[7] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. 2011. Specification and verification: the Spec# experience. *CACM* 54, 6 (2011), 81–91.

[8] P. Bellows and B. L. Hutchings. 1998. JHDL - An HDL for Reconfigurable Systems. In *Proc. FCCM*. IEEE Computer Society, 175–184.

[9] M. Bowen. *Handel-C: Language Reference Manual*. Technical Report. Embedded Solutions Ltd.

[10] A. Rigland Brodtkorb, C. Dyken, T. Runar Hagen, J. M. Hjelmervik, and O. Storaasli. 2010. State-of-the-art in heterogeneous computing. *Scientific Programming* 18, 1 (2010), 1–33.

[11] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. 2000. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *Proc. Euro-Par*. Springer-Verlag, 969–979.

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. TACAS*. 337–340.

[13] D. Detlefs, G. Nelson, and J. B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *JACM* 52, 3 (2005), 365–473.

[14] R Dimond, S. Racanière, and O. Pell. 2011. Accelerating Large-Scale HPC Applications Using FPGAs. In *Proceedings of the IEEE Symposium on Computer Arithmetic*. IEEE, 191–192.

[15] S. Dydel and P. Bala. 2004. Large Scale Protein Sequence Alignment Using FPGA Reprogrammable Logic Devices. In *Proc. FPL*. Springer-Verlag, 23–32.

[16] M. Fähndrich and K. R. M. Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*. ACM Press, 302–312.

[17] Umer Farooq, Zied Marrakchi, and Habib Mehrez. 2012. *FPGA Architectures: An Overview*. Springer New York, New York, NY, 7–48.

[18] J. Filliâtre and A. Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proc. ESOP*. 125–128.

[19] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. 2002. Extended Static Checking for Java. In *Proc. PLDI*. 234–245.

[20] A. Abdul Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi. 2002. Floating-point bitwidth analysis via automatic differentiation. In *FPT*. IEEE, 158–165.

[21] M. Gokhale, J. M. Stone, J. M. Arnold, and M. Kalinowski. 2000. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *Proc. FCCM*. IEEE, 49–58.

[22] Tony Hoare. 2009. Null References: The Billion Dollar Mistake, Presentation at QCon. (2009).

[23] S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. 2008. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *Proc. ECOOP*. Springer-Verlag, 76–103.

[24] 2008. *1076-2008 - IEEE Standard VHDL Language Reference Manual*. IEEE.

[25] 2008. *1364-2005 - IEEE Standard for Verilog Hardware Description Language*. IEEE.

[26] Atsushi Igarashi and Hideshi Nagira. 2007. Union Types for Object-Oriented Programming. *JOT* 6, 2 (2007).

[27] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proc. NFM*. Springer-Verlag, 41–55.

[28] A. A. Kamal and A. M. Youssef. 2009. An FPGA implementation of the NTRUEncrypt cryptosystem. In *International Conference on Microelectronics (ICM)*. IEEE, 209–.

[29] S. Kou and J. Palsberg. 2010. From OO to FPGA: fitting round objects into square hardware?. In *Proc. OOPSLA*. ACM Press, 109–124.

[30] D. Kreutz, F. M. V. Ramos, P. Jorge E. Veríssimo, C. Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (2015), 14–76.

[31] J. Krueger, D. Donofrio, J. Shalf, M. Mohiyuddin, S. Williams, L. Oliker, and F.-J. Pfreund. 2011. Hardware/software co-design for energy-efficient seismic modeling. In *Conference on High Performance Computing Networking, Storage and Analysis*. ACM Press, 73:1–73:12.

[32] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. LPAR (LNCS)*, Vol. 6355. Springer-Verlag, 348–370.

[33] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. A. Vissers. 2012. A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT). In *Proceedings Symposium on High-Performance Interconnects*. IEEE, 9–16.

[34] C. Male, D.J. Pearce, A. Potanin, and C. Dymnikov. 2008. Java Bytecode Verification for @NonNull Types. In *Proc. CC*. 229–244.

[35] S. Masuno, T. Maruyama, Y. Yamaguchi, and A. Konagaya. 2007. An FPGA Implementation of Multiple Sequence Alignment Based on Carrillo-Lipman Method. In *Proc. FPL*. IEEE, 489–492.

[36] O. Mencer. 2006. ASC: a stream compiler for computing with FPGAs. *IEEE Trans. on CAD of Integrated Circuits and Systems* 25, 9 (2006), 1603–1617.

[37] O. Mencer, D. J. Pearce, L. W. Howes, and W. Luk. 2003. Design space exploration with A Stream Compiler. IEEE, 270–277.

[38] D. J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Proc. VMCAI*. 335–354.

[39] D. J. Pearce. 2015. Integer Range Analysis for Whiley on Embedded Systems. 26–33.

[40] D. J. Pearce and L. Groves. 2015. Designing a Verifying Compiler: Lessons Learned from Developing Whiley. *SCP* (2015), 191–220.

[41] S. Sezer, S. Scott-Hayward, P.-K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao. 2013. Are we ready for SDN? Implementation challenges for software-defined networks. *IEEE Communications Magazine* 51, 7 (2013), 36–43.

[42] M. Stephenson, J. Babb, and S. Amarasinghe. 2000. Bitwidth Analysis with Application to Silicon Compilation. In *Proc. PLDI*. 108–120.

[43] D. B. Thomas, L. W. Howes, and W. Luk. 2009. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proc. FPGA*. ACM Press, 63–72.

[44] X. Tian and K. Benkrid. 2010. High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU. *ACM TRETS* 3, 4 (2010), 26:1–26:22.

[45] S. Tobin-Hochstadt and M. Felleisen. 2010. Logical types for untyped languages. In *Proc. ICFP*. 117–128.

[46] xilinx7 2012. *Xilinx 7 series FPGAs Product Brief*. Technical Report.

[47] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. 2008. *AutoPilot: A Platform-Based ESL Synthesis System*. Springer-Verlag, 99–112.