ECE4095 Final Report

H2V — a Haskell to Verilog Compiler

Reuben D'Netto (22096620)

March 21, 2015

A proof-of-concept Haskell to Verilog compiler (H2V) was designed and implemented. Recursive functions, higher-order functions and parallel computation of lists are supported. This demonstrated the feasibility of generating hardware modules from functional programs, which can significantly reduce the labour and expertise needed to design hardware accelerators. Additional work is recommended to convert this into a production-ready tool.

Contents

1. Significant Contributions

- Designed and implemented a Haskell to Verilog compiler, with support for the following language features:
  - Pattern matching
  - Pattern guards
  - Tail-recursive functions
  - Higher-order functions (evaluated at compile-time)
  - Partial application
- Designed and implemented support for the following functions through a combination of generated and hard-coded Verilog:
  - List operators: cons (:), concat (++)
  - Higher-order list functions: map, fold/reduce, zipWith
- Designed and implemented support for N-degree parallel computation of lists, as defined by user
- Designed and implemented data flow graph generation
- Verified hardware generated for test cases using SignalTap

MONASH University

Department of Electrical and Computer Systems Engineering

ECE4095 Final Year Project, Semester 2, 2014

Reuben D'Netto

H2V – a Haskell to Verilog Compiler

Supervisor: Dr. David Boland

Verilog is often used to implement hardware accelerators, which are used to perform expensive computations faster than a general purpose CPU would allow.

H2V generates Verilog modules from concise functional descriptions of logic, making it trivial to leverage data-level parallelism.

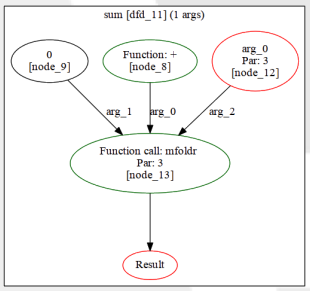
Logic can be tested with desktop Haskell compilers, reducing development time.

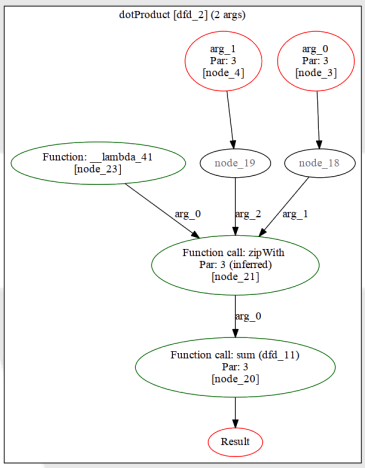
- Trivial composition of modules
- Compatible with existing Haskell compilers
- Easily tuned N-degree parallelization

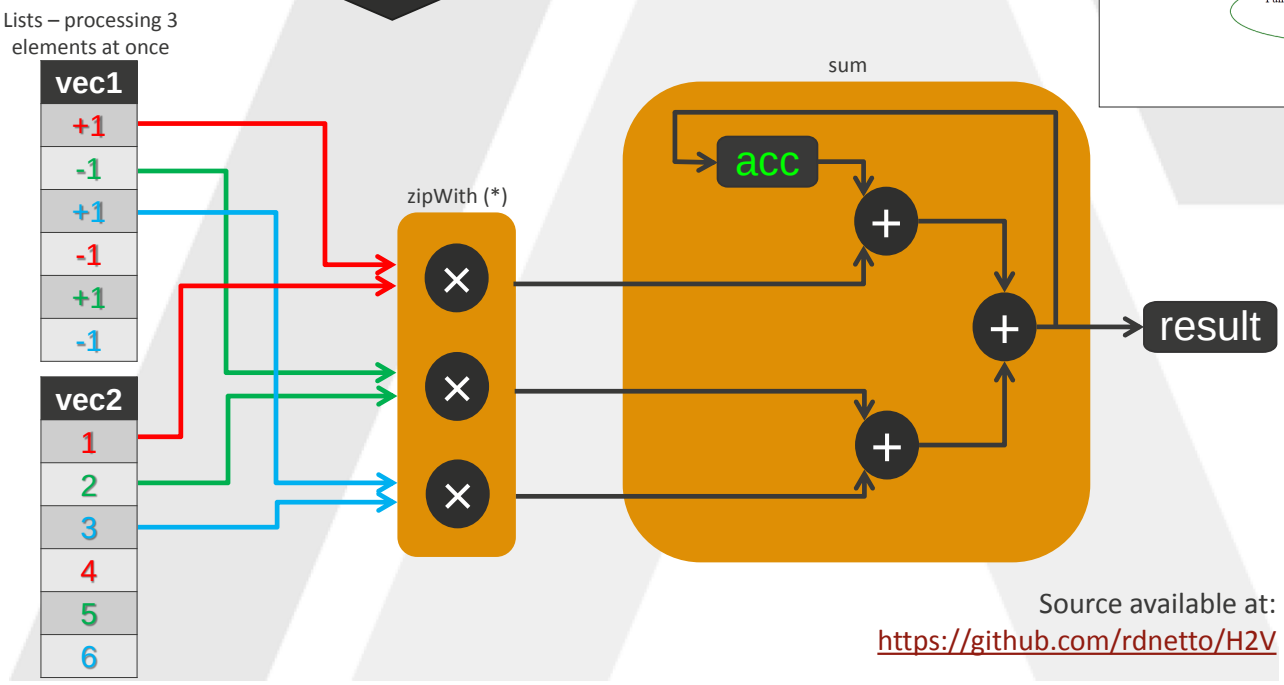
```
sum :: [Int] -> Int
sum = mfoldr (+) 0 ||| 3

dotProduct :: [Int] -> [Int] -> Int
dotProduct u v = sum $ zipWith (*) u v ||| 3

demo _ = dotProduct vec1 vec2 ||| 3 where
    vec1 = [+1, -1, +1, -1, +1, -1]
    vec2 = [1 .. 6]
```







Source available at:
https://github.com/rdnetto/H2V

3. Introduction

3.1. Context & Rationale

Verilog is a commonly used hardware description language used for programming field programmable gate arrays (FPGAs). Its uses may be broadly grouped into two categories: interfacing with low-level hardware, and accelerating computations which would otherwise be performed sequentially on a microprocessor. Due to its low-level nature (which is necessary for the first class of use cases), implementing computations directly in Verilog is extremely time consuming, and often complex as a result of the parallelism that it enables.

H2V is a Haskell to Verilog compiler. Programs can be defined extremely concisely and simply in Haskell, and then compiled into an appropriate set of Verilog modules. Since H2V is compatible with a subset of the Haskell standard, the same program code can be compiled and tested using existing tools, facilitating faster and more thorough testing.

3.2. Haskell

Haskell is a strongly typed pure functional language with lazy evaluation, in contrast to C, which is a weakly typed imperative language with strict evaluation. This means that all variables in Haskell are immutable, and all functions are unable to access, modify, or preserve any state. These features make it ideal for use in parallel computations, where state translates directly to bottlenecks.[1] Furthermore, because Haskell programs are defined in terms of the expressions assigned to variables (instead of the instructions to be evaluated by a microprocessor), H2V has a much cleaner and more direct mapping between expressions and hardware modules than C-based tools have.

3.3. Project Definition

The goal of this project is to demonstrate the viability of a Haskell to Verilog compiler, and its advantages over similar tools which compile imperative programs (e.g. in C) to hardware. A Haskell to Verilog compiler (H2V) was implemented, but as a proof of concept rather than a completed tool. H2V supports only a subset of the Haskell 2010 standard and the associated standard libraries.

4. Literature Review

4.1. Academia

The idea of using functional programming concepts to design hardware is not new; the concept dates back to the early 1980s,[2, p. 1135] prior to the standardization of Verilog[3] and VHDL.[4] More modern attempts at doing so include Lava[5] and CλaSH.[6]

[1] In fact, these features overlap considerably with the recommendations in many C to hardware tools.[1]

Lava is an embedded domain-specific hardware description language, and as such intends each syntactic element to map to a hardware block. This is achieved by encapsulating all signals in a abstract data type (called `Signal`), and breaks compatibility with existing Haskell tools. In contrast, H2V programs describe the computation of a value (or set of values), allowing the compiler more leeway for optimization. For example, where a H2V program contains two multiplication operations as the children of a ternary operator (i.e. a multiplexor), a single hardware multiplexor could be used without complicating the functional description.[2]

CλaSH is more similar to H2V as it compiles a proper subset of Haskell to VHDL. It however lacks support for recursive functions.[7]

4.2. Commercial Tools

There exists a large number of tools for compiling programs in C–like languages to hardware description languages, such as Altera's C2H.[1] Many of them use proprietary extensions to compensate for the intrinsic deficiencies of using a imperative system programming language for hardware, resulting in a highly fragmented landscape and the inability to use existing C libraries. In contrast to this, H2V programs are written in standard Haskell, and can use code from existing Haskell libraries (subject to the limitations of its feature-set). Furthermore, using a functional language gives H2V some significant advantages over tools based on imperative languages. (For a detailed discussion of this, please refer to s5.3.

[2] This approach is already used in H2V for recursive functions, and could easily be applied to other cases such as expensive function calls.

5. Supported Features

5.1. Comparison to Requirements Analysis

ID	Type	Description	Implemented
3.1.1a	Requirement	Support a subset of Haskell 2010.	Yes
3.1.1b	Requirement	Support tail-recursive functions.	Yes
3.1.2.1	Optional	Retain compatibility with existing Haskell compilers.	Yes
3.1.2.2	Optional	Support first class functions.	Yes
3.1.2.3	Optional	Support unary non-tail recursive functions with bijective mappings for arguments.	No
3.1.2.4	Optional	Support for compiling multiple files at once.	No
3.1.2.5	Optional	Loop vectorization of recursive functions.	No
3.1.2.6	Optional	Restructuring of expression trees to reduce critical path.	No
3.1.2.7	Optional	Partial type inference.	Yes[3]
3.2.1	Requirement	QSys Integration — generation of C headers and component definitions.	No[4]
3.2.2.1	Optional	Use optional features in Avalon bus protocol to improve performance.	N/A
3.2.2.2	Optional	Support Avalon Streaming interfaces.	No
3.3.1	Requirement	Support for sequential reads from RAM.	Implicit[5]
3.3.2	Optional	Support for sequential writes to RAM.	Implicit[5]
3.3.3	Optional	Support for non-sequential reads from RAM.	No
3.3.4	Optional	Support for caching subsequent accesses.	N/A
3.4.1.1	Requirement	Results shall include benchmarks comparing H2V and C2H.	No
3.4.1.2	Requirement	The execution time of H2V accelerators shall be equal or better to those of C2H.	Yes[6]
3.5.1	Requirement	H2V shall be published under a free software license.	Yes[7]

[3] Limited type inference is supported for scalar and list types. Type inference for higher-order types is not supported.
[4] See s5.1.1.
[5] See s5.1.2.
[6] H2V is significantly faster than C2H in list-based benchmarks due to its support for parallelism (refer to s5.2.3.)

5.1.1. QSys Integration

QSys integration was abandoned as the goal of the project shifted from implementing a finished tool (a task that would likely take years) to demonstrating the viability of the concept. QSys integration is trivial to implement, to the extent that it could easily be performed by hand. Consequently, this would have added little to the project.

A second factor in abandoning QSys integration was that it relied on the inherent assumption that the main application of H2V was augmenting Nios soft-processors. This proved to be false, as accessing data stored in the processor's RAM would be limited to two words per cycle in most cases (due to the constraints of the hardware). In contrast, H2V achieves optimal performance in applications where it can read several words in the same clock cycle. There would therefore be little point in implementing QSys integration, as it would be advantageous only in situations where the Avalon bus was not memory-mapped.[8]

5.1.2. Direct Memory Access

Instead of implementing direct memory access, a well-defined list interface protocol was implemented. This has the advantage of enabling parallelism beyond what the memory would support, as well as offering increased flexibility in allowing a variety of data sources and sinks to be used. Since all related use cases in the requirements analysis have been satisfied, this requirement can be regarded as completed.

5.2. Supported Features

5.2.1. Language Features

H2V is capable of correctly resolving functions and variables defined out of order in multiple scopes, including shadowing. i.e. the definition in the innermost scope has precedence over definitions in outer scopes with the same name. Pattern guards and pattern matching are also supported; the function definitions are lowered into a single definition with appropriate if statements / multiplexors inserted at the root node. Tail-recursive functions are supported, with pattern matching is used to define the base and recursive cases.

Refer to sA.1 for test cases corresponding to these features.

5.2.2. Higher-Order Functions

Introduction The following section is for the benefit of readers who are unfamiliar with the concept of higher-order functions, and may be skipped.

Consider a function of the form $f : (x, y) \to z | f(x, y) = x + y$. This is a function of arity 2 (i.e. it takes two arguments) and order one; it is a first-order function. In general,

[7] H2V is available at `https://github.com/rdnetto/H2V` under the GNU General Public License version 2 (GPLv2).

[8] The Avalon-MM bus has a maximum bandwidth of 128 bytes per clock cycle,[8, pp. 3-4] and so could be connected to H2V modules without a significant reduction in throughput.

any function whose argument and return types are scalar types (or lists thereof) is a first-order function. Variables, which do not take any arguments, may be regarded as zeroth-order functions. In other words, first-order functions take arguments and return values which are zeroth-order functions.

This may then be generalized in a manner similar to induction, to define an Nth-order function as one which takes arguments and returns a value of order $N - 1$ or lower.[9]Functions of order two or higher are generally referred to as higher-order functions, in contrast to first-order functions.

In Haskell, all functions of arity two or higher are considered higher-order functions, since partial application (i.e. the application of fewer arguments than the arity of the function) of a function returns a function which takes the remaining arguments.[10] While H2V supports partial application, the following discussion will refer to first-order functions of any arity as first-order functions, in order to simplify the terminology.

It is worth taking the time to distinguish higher-order functions in Haskell from function pointers in C. Function pointers allow functions to take arguments and return values which are pointers to existing functions. This is more limited than higher-order functions, which allow completely new functions to be returned.[11] Furthermore, the function pointer cannot be modified in anyway, while higher-order functions in Haskell can have arguments applied to them, to yield new functions via partial application. This is a direct consequence of the languages' different philosophies: Haskell was designed to represent computations, while C was designed to represent machine code, in which a function is merely a memory address to be jumped to.

Implementation Because a functional argument may be evaluated multiple times within a higher-order function, it is impractical to pass it at run-time. Therefore, H2V implements support for higher-order functions by evaluating them at compile-time, effectively rewriting them as first-order functions which are evaluated at run-time. This is sufficient as higher-order functions are used primarily to enable code re-use rather perform expensive computations. For example, Haskell's map function (see s5.2.3) is a higher-order function which maps a list over its functional argument. The partial application `map f` yields a first-order function which performs that same mapping. The application of the functional argument is a relatively cheap operation that can be performed at compile-time (simply substituting the function into the data flow graph), while the evaluation of the resulting first-order function is an expensive operation depending on the contents of the list at run-time.

Examples of higher-order functions supported by H2V may be found in sA.2.

[9] Any function can trivially be expressed as one of a higher order, by defining a function which returns the original function.

[10] This is referred to as *currying*, after the logician Haskell Curry.

[11] While one could theoretically allocate some memory, write assembly to it, and return a pointer to it in C, the resulting code would not be portable, and would certainly not be usable in C2H. A more sensible approach would be provide a pointer to a data structure, but this would be shared state which would prevent the function being called multiple times in parallel.

5.2.3. Lists

Concept Lists in H2V are implemented as a streaming data source. This is mirrors Haskell's lazy evaluation, where values (such as the elements of the list) are not computed until they are required. This means that lists can be used to encapsulate data without adding significant overheads, in contrast to C where arrays are mapped to blocks of memory. Furthermore, it means that the total memory requirements of the data structure are restricted to the number of elements accessed at any one time. This means that lists can be of infinite length. e.g. `[1, 3 ..]` will evaluate to the list of all positive odd numbers.

Parallelism List operations are a prime candidate for parallelism, since the data is already organized into discrete elements which are subject to the same repeated operations. To facilitate this, H2V defines a parallelism operator `|||` which may be applied to any expression which returns a list to define the number of elements it should read/process simultaneously. Parallelism inference is supported within functions; that is, any parallelism assignment will be propagated to other adjacent nodes in the data flow graph.

Higher-order List Functions H2V supports three higher-order list functions: `map`, `mfoldr`, and `zipWith`. `map` applies a function to each element of a list, returning a list containing the result of each application. `zipWith` generalizes map to functions with two arguments, taking the first and second arguments from their respective lists. Both of these functions request values from the source list and start computing the result before the consumer of the resulting list attempts to read from it, ensuring that all reads after the first take only one clock cycle (provided that the computation has had enough time to complete).

`mfoldr` is a monoidic right-fold. Fold (or reduce as it is sometimes known) recursively applies a function to an element of the list and an accumulator, storing its value in the accumulator. Right and left folds refer to the order in which the elements of the list are processed. Because the order in which elements are folded is defined for `foldr` and `foldl`, it is not possible to compute folds in parallel. For this reason the function `mfoldr` was defined instead. `mfoldr` imposes the additional constraint that the function and its domain form a *monoid*.[12] By adding the requirement that the function must be associative, intermediate values can be computed in parallel using a binary reduce tree. The use of a right-identity instead of an initial value for the accumulator also means that the number of values read from the list can be an arbitrary number instead of a power of two, since the empty slots can simply be set to the identity, thereby having no effect on the result.

[12] A monoid is an algebraic structure consisting of a set and a binary function which is associative and has a right-identity over the that set. For example, addition is a monoid over the set of real numbers with a right-identity of 0.

5.3. Case Study — Dot Product

Consider the following C code for computing a Euclidean inner product. (i.e. a vector dot product) The `__restrict__` type qualifier allows the compiler to assume pointer aliasing does not occur,[9, p. 122] and a hypothetical pragma is used to instruct the compiler to unroll three iterations of the loop.

```
int dotProduct(const int* __restrict__ listA, const int* __restrict__
    listB, int length){
    int result = 0;

    #pragma unroll_loop_with_parallelism 3
    for(int i = 0; i < length; i++)
        result += listA[i] * listB[i];

    return result;
}
```

If we consider how it maps to hardware, the following questions arise:

- How is the summation performed? Does each sum occur after `result` is updated, or is there an intermediate reduction tree?
- Does the pragma indicate that three elements should be read from each list, or that three operations should be performed? Are either of these interpretations even logically possible, and if not, will the compiler issue an error?
- How many values are read from `listA` and `listB` each cycle?
- Does the hardware attempt to sum and multiply the same number of values per clock cycle, or is it aware that these operations take different durations to complete?
- Is it possible for the values in `listA` and `listB` to be modified during the computation? C's `const` keyword only protects access to a location in memory via a given variable, not the memory location itself. i.e. the elements of the list are not *immutable*.

It quickly becomes obvious that even with an explicit pragma for controlling loop unrolling, there is a lot of ambiguity as to how the compiler generates the hardware. This is partly due to the conflation of two different operations: the addition and multiplication. Because they are combined into construct in idiomatic C, it becomes difficult to analyze them in isolation, even though they are separate operations computationally and would ideally map to separate blocks of hardware. The following code addresses this problem.

```
int dotProduct(const int* __restrict__ listA, const int* __restrict__
    listB, int length){
    int result = 0;
```

```
        int tmp[length];

        #pragma unroll_loop_with_parallelism 3
        for(int i = 0; i < length; i++)
            tmp[i] = listA[i] * listB[i];

        #pragma unroll_loop_with_parallelism 3
        for(int i = 0; i < length; i++)
            result += tmp[i];

        return result;
    }
```

This allows us to define the parallelism of the addition and multiplication separately, but the other problems remain. Furthermore, we now have to consider how the compiler treats `tmp[]`:

- Is a block of memory allocated for `tmp`, or does the compiler recognize it is used as an intermediate variable? Can the compiler identify that its elements are consumed in order, or does it store the contents to allow for random-access?
- How does the compiler handle cases where the loops have different parallelisms? Will it warn the programmer, or simply assume it was intended? This is significant, as the connection between data sources and sinks may be completely unobvious in a large program.

Perhaps most importantly, the code is no longer idiomatic, and can hardly be considered elegant. Elegance and idiomacy are important because they indicate syntactic and computational structures which are easily understood and recognized by both humans and computers, and are therefore easily optimized by compilers.

Finally, consider the following function, which generalizes the previous definition by replacing the operators with arbitrary functions.

```
    int innerProd(const int* __restrict__ listA, const int* __restrict__
        listB, int length){
        int result = 0;
        int tmp[length];

        #pragma unroll_loop_with_parallelism 3
        for(int i = 0; i < length; i++)
            tmp[i] = func1(listA[i], listB[i]);

        #pragma unroll_loop_with_parallelism 3
        for(int i = 0; i < length; i++)
            result = func2(result, tmp[i]);

        return result;
    }
```

In addition to the previous concerns, the programmer must now consider whether `func1` and `func2` are *pure*. A pure function is one whose evaluation does not depend on any shared state. Such state restricts parallelism, creating bottlenecks and race conditions. In a small program it may be reasonable for the programmer to keep track of such things mentally, but this cannot be said of larger programs authored by entire teams, and any solution which assumes humans will not make mistakes is no solution at all.

Now let's consider the idiomatic solution to the same problem in Haskell.

```
sum :: [Int] -> Int
sum = mfoldr (+) 0 ||| 3

dotProduct :: [Int] -> [Int] -> Int
dotProduct listA listB = sum $ zipWith (*) listA listB ||| 3
```

We have defined `sum` as a separate function for clarity, but it is immaterial whether it is manually inlined or not. Given this code, we can now easily answer the questions from before.

- Summation (`mfoldr (+)`) and multiplication (`zipWith (*)`) are clearly distinct operations, and can be annotated using the parallelism operator (`|||`) to indicate how many elements should be read at a time. Because this is an operator and not a pragma, it can be used multiple times per line on different expressions. An additional benefit of clearly separating these operations is that it simplifies resource sharing. (see s6.5.)
- Summation is performed using `mfoldr`, which uses a reduction tree by definition.
- As Haskell is a functional language, all variables are guaranteed to be immutable and all functions to be pure, so the compiler is able to optimize the function as aggressively as possible without needing to perform expensive, link-time analysis to verify this.
- Since Haskell uses lazy evaluation, the output of `zipWith` is not even computed until it is needed by `sum`. This means that there is no overhead, and both operations can be performed in parallel on the data stream.
- If `sum` has a different parallelism to `dotProduct`, H2V will give the user an error message.

Furthermore, the code is idiomatic and concise. It is identical to the equivalent code in standard Haskell (minus the parallelism annotations), and a mere four lines long (reducible to two if `sum` is inlined), in contrast to the eleven lines of C. Put simply, the Haskell version maps as cleanly to hardware as C does to assembly, as shown in Figure 1.

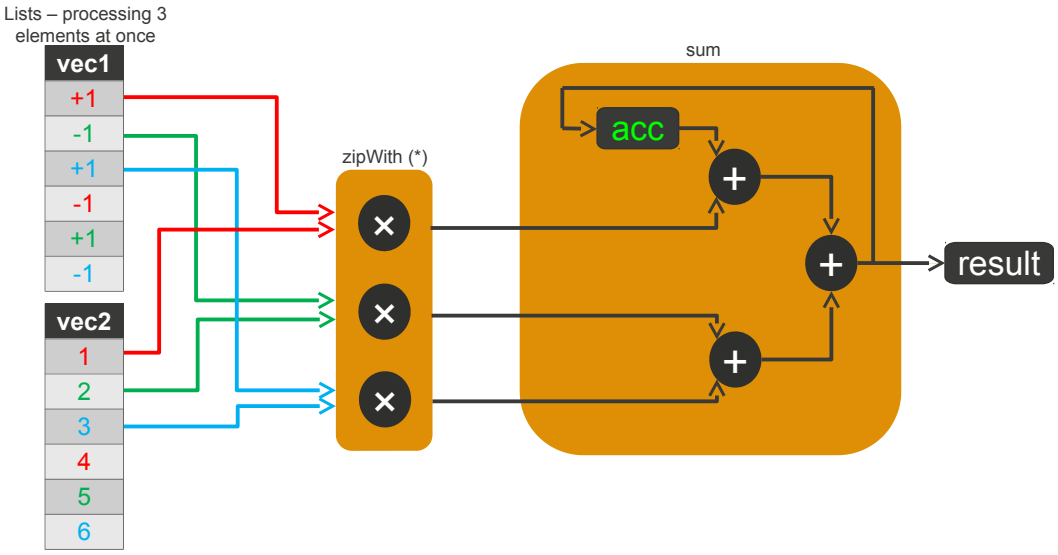


Figure 1: Hardware generated by H2V for dot product definition.

5.4. Interfaces & Protocols

5.4.1. Function Interface

All functions compiled into Verilog modules by H2V use the following interface.

Name	Width	Type	Description
`clock`	1 bit	Input	Clock signal.
`ready`	1 bit	Input	Clock enable — indicates that the other input values are now valid.
`done`	1 bit	Output	Asserted high when `result` is valid.
`arg0, arg1, ...`	8 bits[13]	Input	The values passed to the function as arguments.
`result`	8 bits[13]	Output	The return value of the function.

5.4.2. List Interface

Arguments which are lists are passed using the following signals:

[13] 8 bits is the default width for scalar values — see s6.4. Refer to s5.4.2 if the argument is a list.

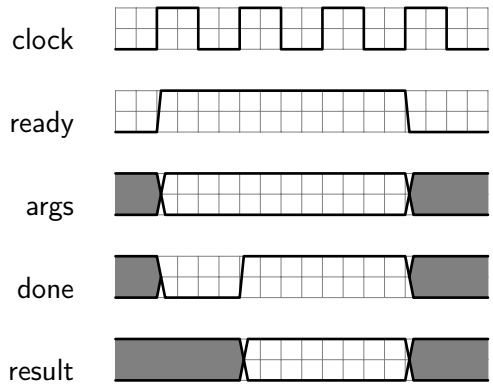


Figure 2: Timing diagram for function interface.

Name	Width	Type	Description
`REQ`	1 bit	Input	Asserted high when a new (set of) elements are requested. One asserted, must be reset before another request can be made. Note that `value_0, ...` will only remain valid as long as `REQ` is high.
`ACK`	1 bit	Output	Asserted high for one clock cycle when `value_0, ...` become valid.
`value_0, ...`	8 bits	Output	The values of the next N elements, where N is the parallelism of the list instance.
`value_0_valid, ...`	1 bit	Output	Asserted high if `value_0, ...` are valid.

Note that each list has an integral parallelism. i.e. the number of values read per `REQ-ACK` cycle. It is an invariant of the interface that

$$\texttt{value\_i\_valid} \implies \texttt{value\_j\_valid} \; \forall \; i > j$$

If `value_0_valid` is asserted low, then the end of the list has been reached. Note that as H2V supports infinite lists, there is no guarantee that this signal will ever be asserted low.

The signals `value_i` and `value_i_valid` are only valid for the interval between `ACK` being asserted and `REQ` being unasserted.

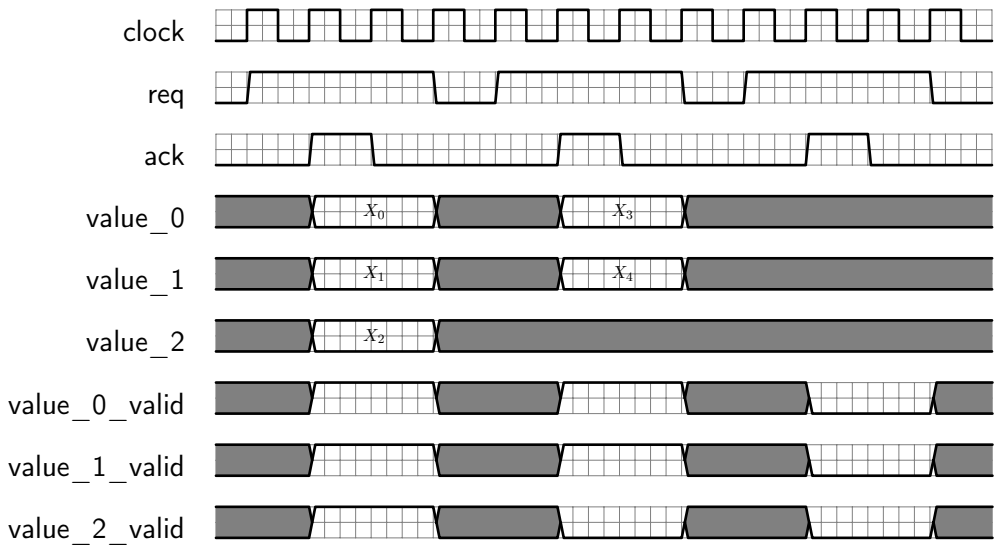


Figure 3: Timing diagram for the list interface of $\{X_0, X_1, X_2, X_3, X_4\}$.

6. Future Work

As stated previously, H2V is a proof of concept; it provides a foundation for a complete tool to be built upon. The following sections describe features that would be needed to realize this vision, and discuss the extent to which the existing codebase supports them.

6.1. Improved Type Support

6.1.1. Variable Width Integers

H2V currently supports only two scalar types: booleans and unsigned 8-bit integers. To be useful in real applications, support for signed integers and integers of other sizes will be needed. H2V's type system already supports these,[14] so implementing support is a simple matter of extending the parsing and rendering logic. Bit-widths could be described using parameterised types. For example, a signed 16-bit integer would be expressed as `SInt 16`. An advantage of using parameterised types is that the user can specify any width, rather than just one of a predefined set. This can result in significant reductions in hardware used for expensive computations, such as multiplication. Aliases could also be defined for `Word8`, `Word16`, etc. as unsigned integers and `Int8`, `Int16`, etc. as signed integers to preserve compatibility with existing Haskell code.

6.1.2. Fixed-point Types

Similarly, fixed-point types could also be defined. e.g. `FP 4 8` would describe a fixed-point number with 4 integer bits and 8 fractional bits. An advantage of this over existing

[14] See `DType` at `H2V/DfdDef.hs:100` for a complete description of H2V's type system.

approaches in C is that the compiler would be able to combine types correctly. Consider the following Haskell, and the equivalent C.

```c
int add1_to_fp8(int x){
    return x + (1 << 8);
}
```

```haskell
add1 :: FP 4 8 -> FP 4 8
add1 x = x + 1
```

In the C example, a bit-shift is needed to convert 1 from an integer to a fixed-point type, which obscures the actual function of the code. This is completely unnecessary in the Haskell example, because the compiler can recognize the need for type conversion and perform it automatically. Given that bit-shifts are effectively free when performed in hardware, there is no reason for the programmer concern themselves with this detail, and the bit-shift present in the C obscures the true purpose of the code. While this may seem insignificant for such a trivial example, reading such code becomes considerably more difficult when evaluating complex expressions.

6.1.3. Nested Lists

A significant limitation of H2V is that it is currently only able to manipulate one-dimensional lists. This is particularly limiting since H2V's streaming approach to list manipulation means it is impossible to index an element of the list without discarding everything prior to it. Nested lists would allow representation of multi-dimensional data, such as images. For instance, matrices could also be intuitively represented as a list of rows with a type such as `[[Int]]`.

H2V's type system already supports nested lists, since it defines lists recursively (i.e. a list is a type parameterised in any other type, including itself). As the function interface[15] currently states that lists are handled by replacing the single scalar port for those described by the list interface,[16] it would be logical to do the same for the list interface. i.e. instead of requiring `list_value_0` to be a scalar, it could be replaced by the full set of ports defined by the list interface. This would also be particularly easy to implement, since functions for rendering nodes as either scalars or lists (depending on type) with a prefix already exist.[17]

One complexity that arises from this is the semantics of the parallelism operator when used with nested lists, and whether it applies to all levels of the list or just the outermost. In order to be consistent, it should apply to the outermost level only, so that the element type (i.e. the inner lists) are treated the same regardless of whether they are scalar or not. In order to describe the parallelism of nested lists, it becomes desirable to define a new

[15] See s5.4.1.
[16] See s5.4.2.
[17] E.g. `renderArg` at `H2V/RenderVerilog.hs:340`.

operator. This could be called the multi-dimensional parallelism operator, and would take a list of integers instead of a single one, with each element of the list assigning a parallelism to the lists of corresponding depth. e.g. `[[1, 2], [3, 4], [5, 6]] ||* [3, 2]` would assign a parallelism of 3 to the outer list and 2 to the inner list, while `[[1, 2], [3, 4], [5, 6]] ||| 3` would only assign a parallelism to the outer list.

6.2. Type & Parallelism Inference

H2V currently supports type and parallelism inference within functions. However, each function must have its own type signature and parallelism assignment. This quickly becomes unwieldy, as demonstrated by `demo.hs` on page 28. Allowing parallelism to be inferred across function boundaries would render the redundant assignments unnecessary and improve code re-use by allowing the same function to be called in different places with different parallelisms.

Type inference would facilitate code re-use by enabling generic functions. For example, consider the following second-order functions:

```
applyTwice1 :: (Int -> Int) -> (Int -> Int)
applyTwice1 f = \x -> f (f x)

applyTwice2 :: (a -> a) -> (a -> a)
applyTwice2 f = \x -> f (f x)
```

The first instance can only be used for functions of `Int`s, while the second can be used for any function whose argument and return value are of the same type. In order for the same function to be called with differently typed arguments in different places, the compiler would need to track the types it was called with, which is the same information that would be needed for type inference.

6.3. Closures

Closures extend first-order functions by allowing them to reference variables in their defining context. i.e. they are said to *close over* said variables. Consider Listing 4, which defines a function that increments every element of a list by the provided value. The inner function `f` closes over the variable `a`, which is a member of `incrementAll`. As Verilog does not allow for nested modules, the function must be rewritten by the compiler so that the only inputs take the form of module parameters.

H2V currently has experimental, untested support for closures.[18] As part of the linking process, it identifies each function which contains references variables defined outside it, adds them to its list of arguments, then updates all of the calls to that function to include the closed over variables.

[18] See `H2V/GenerateDFD.hs:730`.

```
incrementAll :: Int -> [Int] -> [Int]
incrementAll a xs = map f xs where
    f x = x + a
```

Figure 4: A function with a closure.

```
map :: (a -> b) -> [a] -> [b]
map f (x0:xs) = (f x0) : (map f xs)

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x0:xs) (y0:ys) = (f x0 y0) : (zipWith f xs ys)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z (x0:xs) = foldl f (f z x0) xs
foldl f z [] = z

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z (x0:xs) = f x0 (foldr f z xs)
foldr f z [] = z

sumEachPair :: [Int] -> [Int]
sumEachPair (x0:x1:xs) = (x0 + x1) : (sumEachPair xs)
```

Figure 5: Haskell definitions of recursive list functions.

6.4. Compilation of Recursive List Functions

H2V currently implements `map`, `zipWith` and `mfoldr` using hard-coded functions for generating their logic. This approach scales extremely poorly, requiring approximately 150 lines of dedicated code for each one,[19] and means imposes a significant implementation cost for each such function. Ideally, these functions could be defined in Haskell (as shown in Listing 5), and generalized logic used to compile them. This would enable the re-use of existing list functions from the standard library, significantly boosting the expressive power of H2V.

This feature could be implemented by recognizing that all of these cases involve recursive functions which consume or produce lists, in both cases via the cons operator (`:`). We can then extract the expressions for the elements to be prepended to the list, and compile them to the appropriate logic, connected to a list interface. Loop unrolling can be applied as usual via the parallelism operator. Note that specialized logic would still be required for the monoidic fold functions (e.g. `mfoldr`), as their implementation relies

[19] See `H2V/RenderVerilog.hs:669`.

on the non-syntactic constraint that the folding function and its domain are a monoid.[20]

An attempt was made at implementing this, but was not completed due to time constraints. It can be found on the topic branch `recursive_list_functions`.

6.5. Resource Sharing

In complex programs, it is rare for all hardware blocks to be in use simultaneously. Because functional programming encourages the expression of programs as many small, modular functions, it becomes viable to identify commonly used functions and re-use existing hardware blocks when the calls are known to happen at mutually exclusive times. For example, consider the example code in s5.3. Identifying the re-usable code in the C example would be extremely difficult; at most one could make some arguments about individual multipliers. However, the Haskell code clearly groups the operations using `mfoldr (+)` and `zipWith (*)`, and even explicitly assigns them parallelisms. Generalizing further, the `dotProduct` function as a whole could be re-used in multiple places. This would require a significant amount of effort as H2V currently does not consider timing requirements or the relationships between different modules, but could result in some fairly impressive reductions in hardware requirements, especially for applications like CPUs where only a small fraction of the ALU is in use at any given time.

7. Conclusions

The results of this project demonstrate the feasibility of generating hardware descriptions from functional programs. Language features are easily rewritten to equivalent Verilog constructs and higher-order functions can be effectively evaluated at compile-time. The streaming model for lists was found to work extremely well, though the lack of support for nested lists and recursive functions of lists prevented its application to problems involving matrices. Extending the range of supported types, improving inference and adding features such as closures and native compilation for recursive functions are recommended, as this will result in the creation of an extremely powerful and elegant hardware synthesis tool.

[20] See s5.2.3.

A. Appendix — Test Cases

The following test suite is representative of the kinds of programs H2V is capable of compiling. Each test case is followed by the DFD generated by H2V. Verilog output has been omitted for brevity.

The test suite can be run on a UNIX-compatible system using the following commands:

```
git checkout https://github.com/rdnetto/H2V.git
cd H2V/tests
./test-all.sh
```

A.1. Language Features

../../H2V/tests/f0.hs

```
--due to a bug in haskell-src, this line is parsed incorrectly as (a2*x + b)*x + c without brackets
f x a b c = (a2 * x) + (b * x) + c where
    a2 = a * a
```

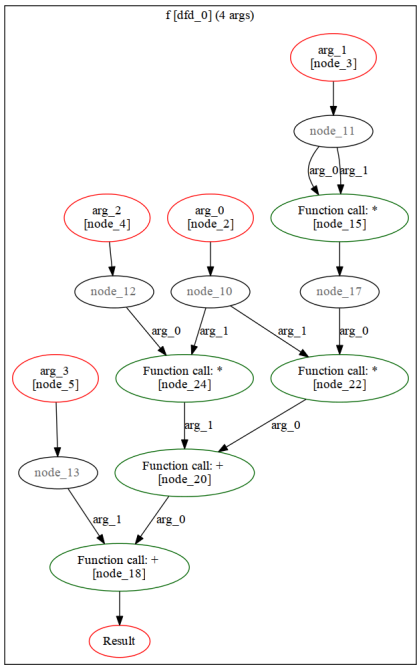


Figure 6: DFD for `f0.hs`

../../H2V/tests/f1.hs

```haskell
f1 :: Int -> Int
f1 x = x + c where
    a = 1
    b = 2
    c = d - a
    d = b + 2
```

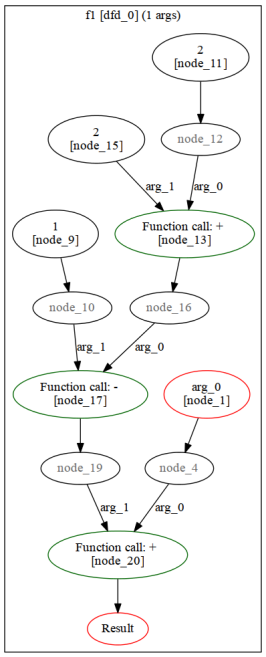


Figure 7: DFD for `f1.hs`

../../H2V/tests/guards.hs

```haskell
--pattern guards
f4 x
    | x < 0      = 0
    | x > 0      = 1
    | otherwise = 2
```

../../H2V/tests/pattern_match.hs

```haskell
--pattern matching
f3 0 = 0
f3 x = if x < 0
       then -1
       else 1
```

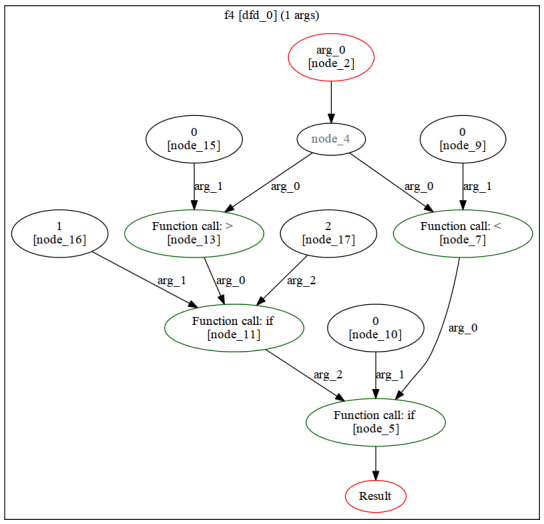


Figure 8: DFD for `guards.hs`

../../H2V/tests/fib.hs

```
-- Test case for nested, recursive functions
-- Returns the nth fibonnacci number
-- Could be implemented using an infinite list, but that would be more
    painful to compile to Verilog
-- x0, x1 are accumulators, n counts down to the required index
fib n = fib' 0 1 n where
    fib' x0 _ 0 = x0
    fib' x0 x1 n = fib' x1 (x0 + x1) (n - 1)
```

A.2. Higher-Order Functions

../../H2V/tests/ho_arg.hs

```
--accept a function
applyTwice :: (Int -> Int) -> (Int -> Int)
applyTwice f = \x -> f (f x)
f8 x = applyTwice (\x -> x + 1) 0
```

../../H2V/tests/ho_flip.hs

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \a b -> f b a

revsub :: Int -> Int
revsub = flip (-)

f5 x = x + (revsub 2 7) + (revsub 9 10) where
```

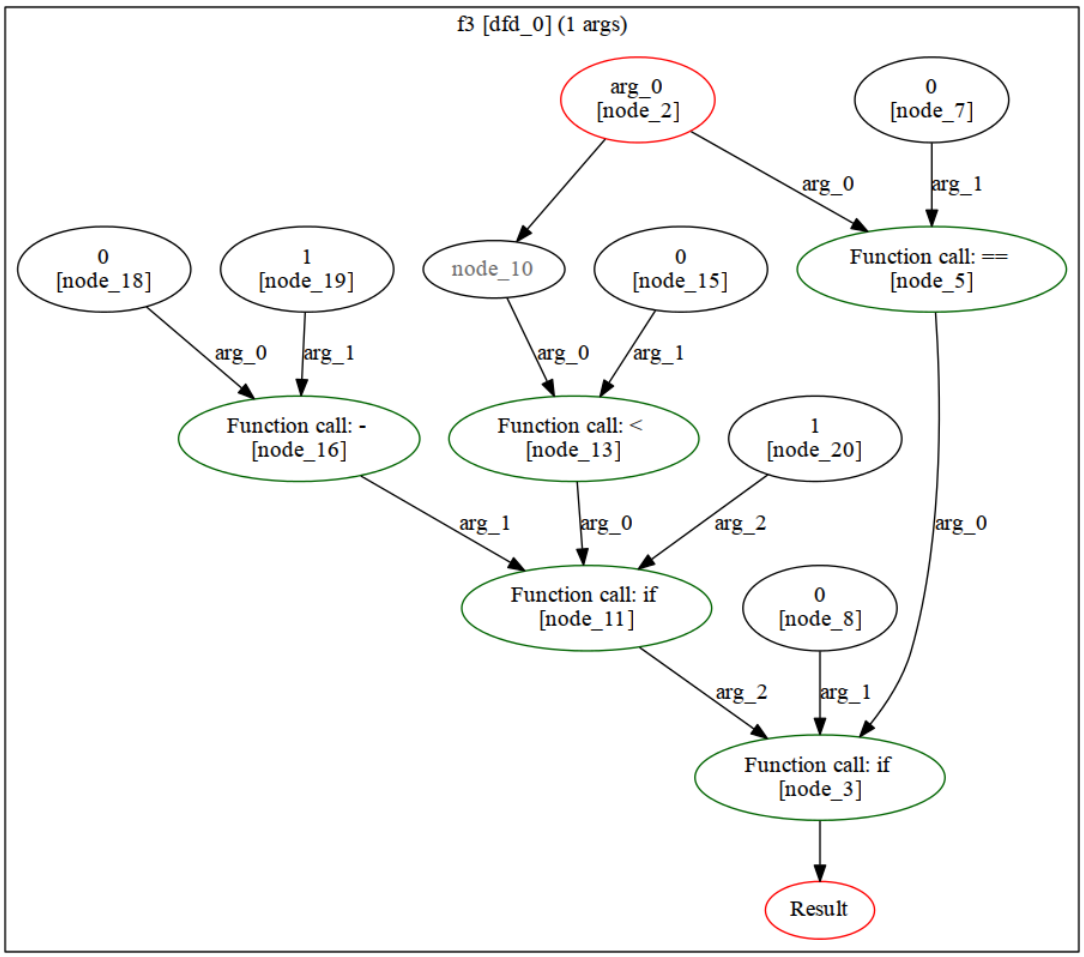


Figure 9: DFD for `pattern_match.hs`

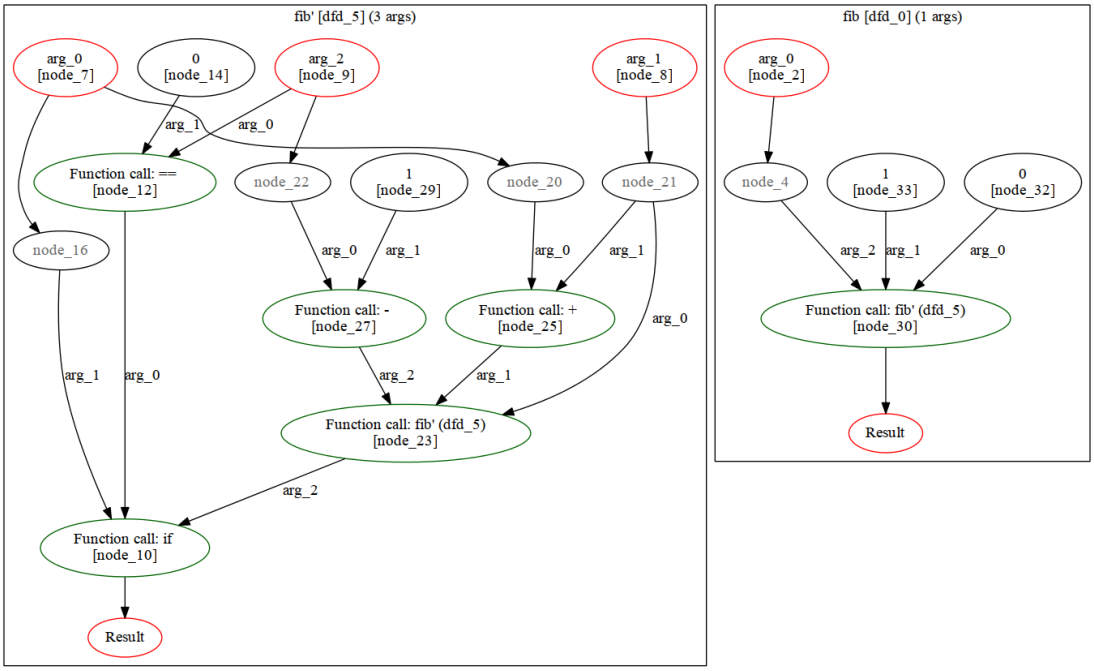


Figure 10: DFD for `fib.hs`

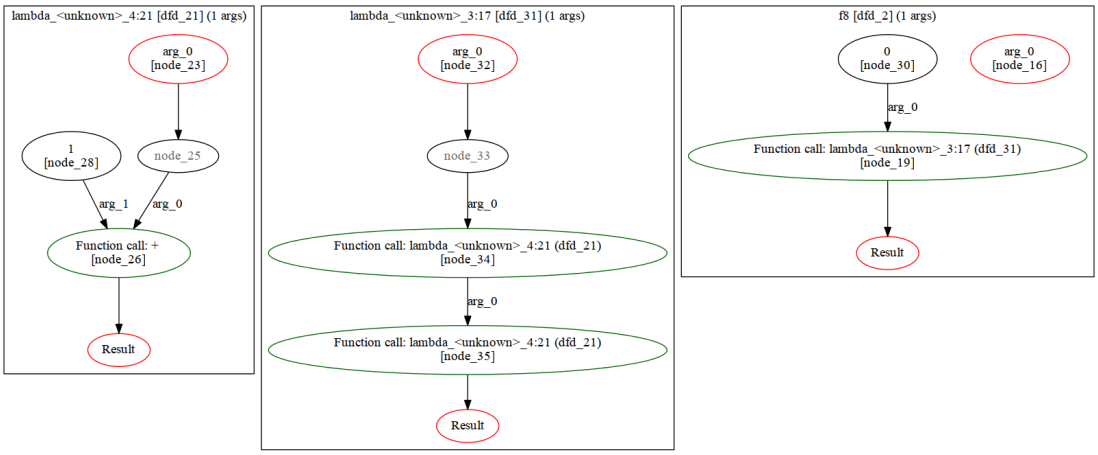


Figure 11: DFD for `ho_arg.hs`

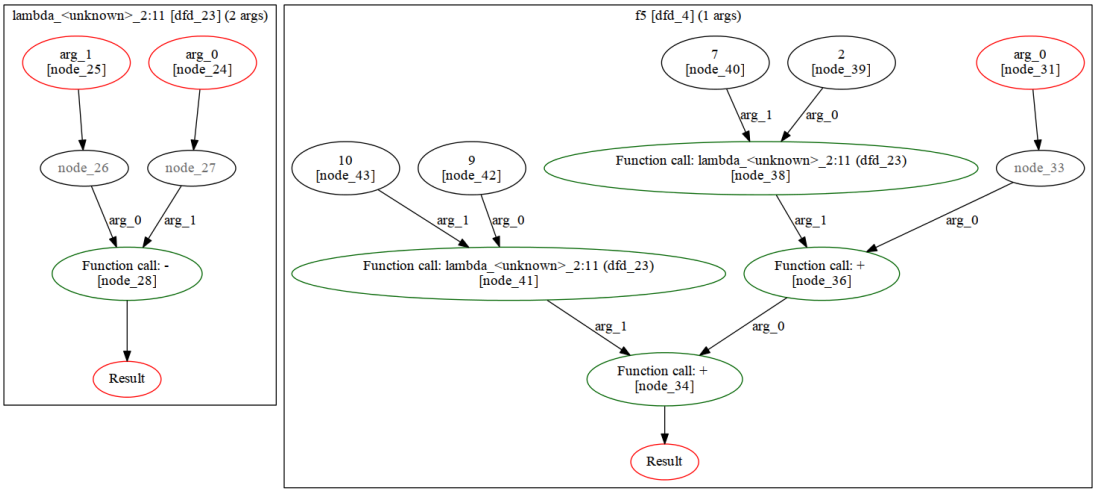


Figure 12: DFD for `ho_flip.hs`

../../H2V/tests/ho_pa.hs

```
--partial application
plus x y = x + y
add1 = plus 1
add2 = (+2)
add3 = (3+)
f7 x = (add1 x) + (add2 x) + (add3 x)
```

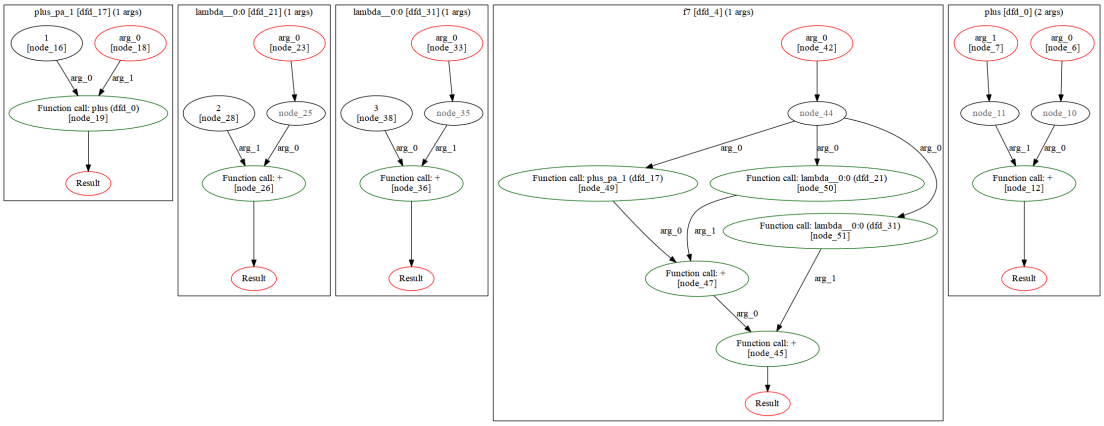


Figure 13: DFD for `ho_pa.hs`

../../H2V/tests/ho_return.hs

```
--higher order functions
--return a function
f6a :: Int -> (Int -> Int)
f6a a = \b -> a + b
f6 x = (f6a 1) x + (f6a 2) (x + 1)
```

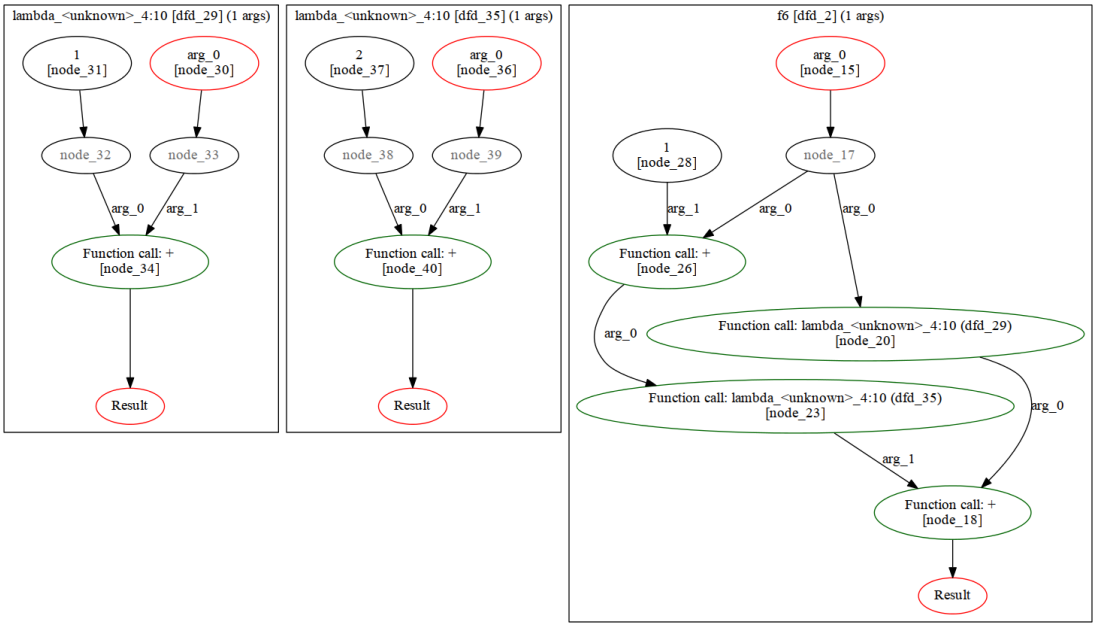


Figure 14: DFD for `ho_return.hs`

A.3. Lists

../../H2V/tests/lists.hs

```
import H2V_Compat

concatTest :: Int -> Int -> [Int]
concatTest a b = [a .. a + 3] ++ [b .. b + 3]

concatTest2 :: Int -> [Int]
concatTest2 a  = [0, 1] ++ [2, 3] ++ [4, 5]

rangeTest :: Int -> Int -> [Int]
rangeTest m n = [m,m+2 .. n]

basicTest :: Int -> [Int]
basicTest x = [1, x, 2]

consTest1 :: Int -> [Int]
consTest1 a = a:[3, 4, 5]

consTest2 :: Int -> [Int]
consTest2 a = a:[]

consTest3 :: Int -> Int -> [Int]
consTest3 a b = a:b:[3, 4, 5]
```

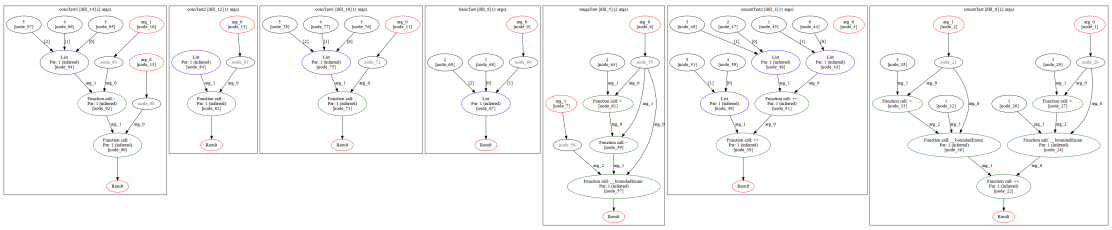


Figure 15: DFD for `lists.hs`

../../H2V/tests/lists2.hs

```
import H2V_Compat

head :: [Int] -> Int
head (x0:xs) = x0
head [] = -1

tail :: [Int] -> [Int]
tail (x0:xs) = xs
tail [] = []

headTest a = head [1 .. 4]
tailTest a = tail [1 .. 4]

mapTest1 :: Int -> [Int]
mapTest1 a = map (+10) ([0 .. 16] ||| 4)

--this is equivalent to the case without any brackets around the call to
    map
mapTest2 :: Int -> [Int]
mapTest2 a = (map (+10) [0 .. 16]) ||| 4

foldTest :: Int -> Int
foldTest x = mfoldr (+) 0 [0 .. 10] ||| 4

zipTest :: Int -> [Int]
zipTest _ = zipWith (+) [1, 2, 3, 4, 5] [10, 20, 30, 40, 50] ||| 3
```

Figure 16: DFD for `lists2.hs`

../../H2V/tests/demo.hs

```
import H2V_Compat

sum :: [Int] -> Int
sum = mfoldr (+) 0 ||| 3

dotProduct :: [Int] -> [Int] -> Int
dotProduct u v = sum $ zipWith (*) u v ||| 3

demo _ = dotProduct [1, 2, 3] [4, 5, 6] ||| 3
```

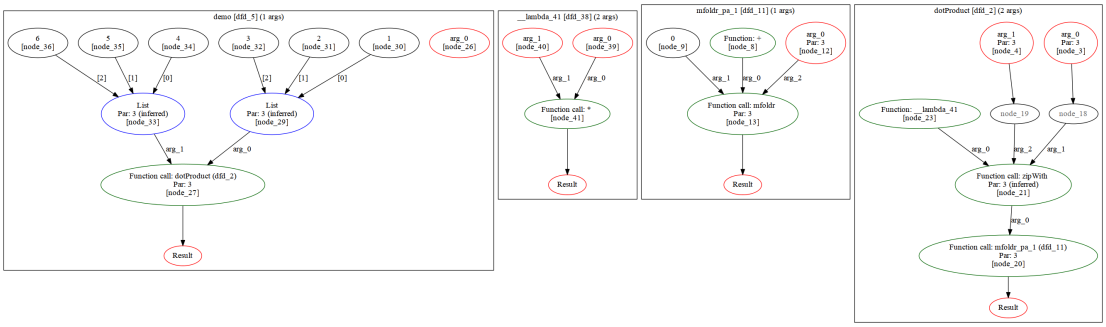


Figure 17: DFD for `demo.hs`

B. Appendix — Source Code

The source code is available at `https://github.com/rdnetto/H2V`.

C. Appendix — Include Files for H2V Programs

The following files should be included by H2V programs. `H2V_Compat` enables existing Haskell compilers to work with H2V code by defining non-standard functions and operators. `include.v` contains Verilog modules used by generated code and must be added to any Quartus project using H2V modules.

../../H2V/include/hs/H2V_Compat.hs

```
--This file enables compatility between H2V programs and normal Haskell
    compilers

module H2V_Compat where
import Data.Bits (Bits, shiftL, shiftR)
import qualified Data.Bits

--Parallelism operator - no-op on PCs
(||||) :: a -> Int -> a
(||||) x _ = x

--Monoid fold. Requires an associative folding function with a right-
    identity.
--Associative:   (a `f` b) `f` c == a `f` (b `f` c). Note that order is
    preserved.
--Right-identity: a `f` i == a
mfoldr :: (a -> b -> b) -> b -> [a] -> b
mfoldr = foldr

--Bitwise shift operators
(.<<.) :: Bits a => a -> Int -> a
(.<<.) = shiftL
infixl 7 .<<.
```

```
(.>>.) :: Bits a => a -> Int -> a
(.>>.) = shiftR
infixl 5 .>>.

xor :: Bits a => a -> a -> a
xor = Data.Bits.xor

complement :: Bits a => a -> a
complement = Data.Bits.complement
```

../../H2V/include/v/include.v

```
module Concat(
    input clock,
    input ready,

        output reg  listA_req,
    input       listA_ack,
    input [7:0] listA_value,
    input       listA_value_valid,

        output reg  listB_req,
    input       listB_ack,
    input [7:0] listB_value,
    input       listB_value_valid,

        input            req,
        output reg       ack,
        output reg [7:0] value,
        output reg       value_valid
    );

    reg lastSelectA;
    wire selectA;
    assign selectA = lastSelectA & (listA_ack ? listA_value_valid : 1'b1);

    always @(posedge clock) begin
        if(ready)
            lastSelectA <= selectA;
        else
            lastSelectA <= 1;
    end

    always @(*) begin
        if(selectA) begin
            listA_req = req;
            ack = listA_ack;
            value = listA_value;
            value_valid = listA_value_valid;
            listB_req = 0;

        end else begin
            listB_req = req;
            ack = listB_ack;
```

```
            value = listB_value;
            value_valid = listB_value_valid;
            listA_req = 0;
        end
    end
endmodule

module Cons(
    input clock,
    input ready,
    input [7:0] head,

        output reg  tail_req,
    input       tail_ack,
    input [7:0] tail_value,
    input       tail_value_valid,

        input            req,
        output reg       ack,
        output reg [7:0] value,
    output reg       value_valid
    );

    reg headShown;
    reg selectHead;
    reg lastReq;
    reg headAck;

    always @(posedge clock) begin
        lastReq <= req;

        if(ready) begin
            if(~lastReq & req) begin
                headAck <= 1;
                headShown <= 1;

                if(headShown)
                    selectHead <= 0;

            end else begin
                headAck <= 0;
            end

        end else begin
            headShown <= 0;
            selectHead <= 1;
            headAck <= 0;
        end
    end

    always @(*) begin
        if(selectHead) begin
            ack = headAck;
            value = head;
```

```
            value_valid = 1;
            tail_req = 0;

        end else begin
            tail_req = req;
            ack = tail_ack;
            value = tail_value;
            value_valid = tail_value_valid;
        end
    end
endmodule

module Hold(
    //Utility module for keeping y high after x goes high (while ready is
        high). Used in generated code.
    input clock,
    input ready,
    input x,
    output reg y
    );

    always @(posedge clock)
        y <= ready ? y | x : 0;
endmodule

module ListMux(
    //Utility module for implementing the ternary operator on lists.
    input ready,
    input cond,
    input             out_req,
    output reg        out_ack,
    output reg [7:0]  out_value,
    output reg        out_value_valid,

    output reg        true_req,
    input             true_ack,
    input [7:0]       true_value,
    input             true_value_valid,

    output reg        false_req,
    input             false_ack,
    input [7:0]       false_value,
    input             false_value_valid
    );

    always @(*) begin
        if(~ready) begin
            true_req = 1'b0;
            false_req = 1'b0;
            out_ack = 1'b0;
            out_value = 8'hFF;
            out_value_valid = 1'b0;

        end else if(cond) begin
```

```
            true_req = out_req;
            out_ack = true_ack;
            out_value = true_value;
            out_value_valid = true_value_valid;

            false_req = 1'b0;

        end else begin
            false_req = out_req;
            out_ack = false_ack;
            out_value = false_value;
            out_value_valid = false_value_valid;

            true_req = 1'b0;
        end
    end
endmodule

module Decons(
    input clock,
    input ready,
    output reg  done,

        output reg  list_req,
    input       list_ack,
    input [7:0] list_value,
    input       list_value_valid,

    output reg [7:0] head,
    output reg       head_valid,

        input            tail_req,
        output reg       tail_ack,
        output reg [7:0] tail_value,
    output reg       tail_value_valid
    );

    reg nextDone;

    always @(posedge clock) begin
        if(ready) begin
            done <= done | nextDone;

            if(~done & list_ack) begin
                nextDone <= 1'b1;
                head <= list_value;
                head_valid <= list_value_valid;
            end else begin
                nextDone <= 1'b0;
            end

        end else begin
            done <= 1'b0;
            nextDone <= 1'b0;
```

```
            head <= 8'hFF;
            head_valid = 1'b0;
        end
    end

    always @(*) begin
        list_req = ready & ~done;

        if(done) begin
            list_req = tail_req;
            tail_ack = list_ack;
            tail_value = list_value;
            tail_value_valid = list_value_valid;
        end else begin
            list_req = ready & ~nextDone;
            tail_ack = 1'b0;
            tail_value = 8'hFF;
            tail_value_valid = 1'b0;
        end
    end
endmodule
```

References

[1] *Nios II C2H Compiler — User Guide*, Available: `http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf`, Altera Corporation, Nov. 2009.

[2] M. Sheeran, "Hardware design and functional programming — a perfect match," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1135–1158, 2005.

[3] "IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language," IEEE, Tech. Rep. 1364, 1995.

[4] "IEEE Standard VHDL Language Reference Manual," IEEE, Tech. Rep. 1076, 1987.

[5] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," *Association for Computing Machinery*, vol. 34, no. 1, pp. 174–184, 1998.

[6] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "CλaSH: structural descriptions of synchronous hardware using haskell," *Digital System Design: Architectures, Methods and Tools*, pp. 714–721, Sep. 2010.

[7] C. Baaij. (2014). Clash tutorial, [Online]. Available: `http://hackage.haskell.org/package/clash-prelude-0.5.1/docs/CLaSH-Tutorial.html`.

[8] *Avalon interface specifications*, Available: `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf`, Altera Corporation, Jun. 2014.

[9] "C99 Specification — 9899:TC3," ISO/IEC, Tech. Rep. WG14/N1256, Sep. 2007, Available: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.