H2V — a Haskell to Verilog Compiler

Reuben D'Netto (22096620)

Supervised by: David Boland

30th October 2014

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

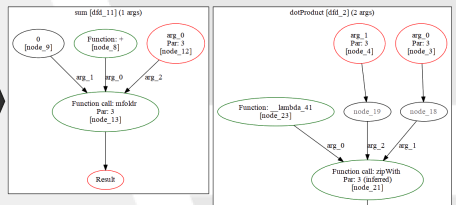
Verilog is often used to implement hardware accelerators, which are used to perform expensive computations faster than a general purpose CPU would allow.

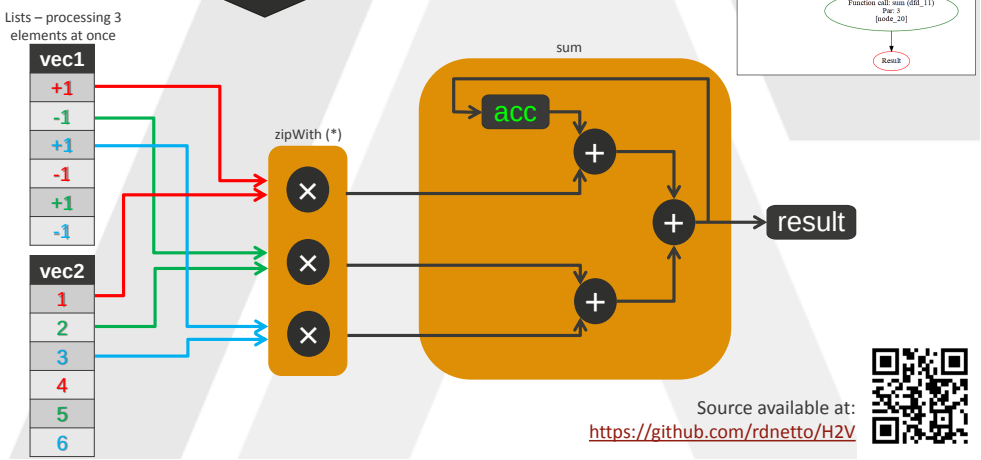
H2V generates Verilog modules from concise functional descriptions of logic, making it trivial to leverage data-level parallelism.

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

Section 1

Introduction

Introduction

- Verilog
  - Interfacing with digital integrated circuits
  - Accelerating expensive computations
- Haskell
  - Pure
  - Functional
  - Concise
- Proof of concept

Purity is good for optimization!

Pure function

A function whose result does not depend on any shared or global state. Modifications to such state are referred to as *side-effects*.

```
main :: Int -> Int
main a = x + z where
    x = foo a
    y = bar a                --can be optimized out
    z = baz a
```

```
int main(int a){
    int x = foo(a);
    int y = bar(a);      //might have side-effects
    int z = baz(a);
    return x + z;
}
```

Section 2

Supported Features

Recursive Functions

Tail-recursive function

A function which performs recursion by returning the result of the recursive call.

```haskell
fib :: Int -> Int
fib n = fib' 0 1 n where
    fib' :: Int -> Int -> Int -> Int
    fib' x0 _ 0 = x0
    fib' x0 x1 n = fib' x1 (x0 + x1) (n - 1)
```

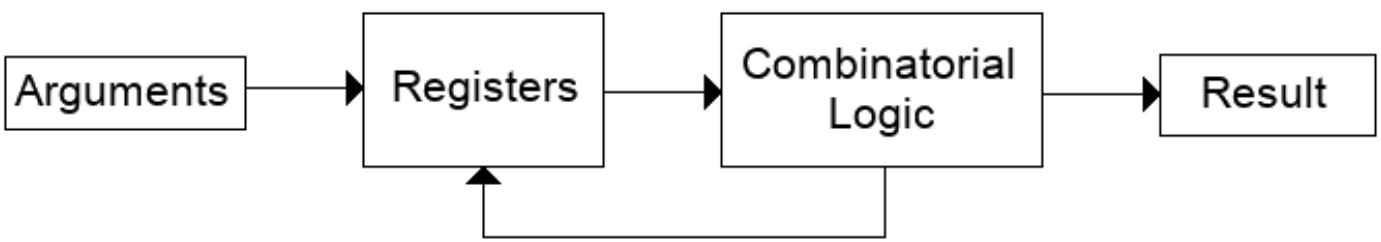
```c
int fib(int n){
    return _fib(0, 1, n);
}
static int _fib(int x0, int x1, int n){
    if(n == 0)
        return x0;
    else
        return fib2(x1, x0 + x1, n - 1);
}
```

Recursive Functions

Tail-recursive function

A function which performs recursion by returning the result of the recursive call.

```
fib :: Int -> Int
fib n = fib' 0 1 n where
    fib' :: Int -> Int -> Int -> Int
    fib' x0 _ 0 = x0
    fib' x0 x1 n = fib' x1 (x0 + x1) (n - 1)
```



Higher-Order Functions — Introduction

Order	Haskell	C
0	`f :: Int`	`int f(){return 0;}`

Higher-Order Functions — Introduction

Order	Haskell	C
0	`f :: Int`	`int f(){return 0;}`
1	`f :: Int -> Int`	`int f(int);`

Higher-Order Functions — Introduction

Order	Haskell	C
0	`f :: Int`	`int f(){return 0;}`
1	`f :: Int -> Int`	`int f(int);`
2	`f :: (Int -> Int) -> (Int -> Int)`	`typedef int (*int2int)(int);` `int2int f(int2int);`

Higher-Order Functions — Introduction

Order	Haskell	C
0	`f :: Int`	`int f(){return 0;}`
1	`f :: Int -> Int`	`int f(int);`
2	`f :: (Int -> Int) -> (Int -> Int)`	`typedef int (*int2int)(int);` `int2int f(int2int);`

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

--Using partial application to omit arguments
revsub :: Int -> Int -> Int
revsub = flip (-)
```

```haskell
--Equivalent to:
revsub x y = y - x
```

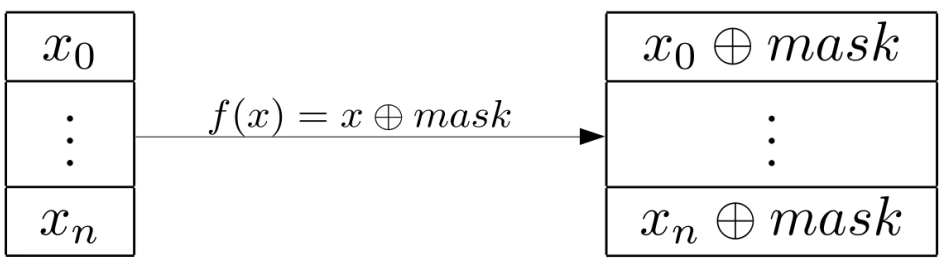
Higher-Order List Functions

- List functions
  - `map` — Applies a function to each element of a list
  - `zipWith` — Map generalized to two arguments/lists
  - `mfoldr` — Reduce a list of elements to a single value
- Parallelism operator: `list ||| N`

List Functions — Maps

```
xorArray :: Int -> [Int] -> [Int]
xorArray mask input = map (xor mask) input

map :: (a -> b) -> [a] -> [b]          --built-in
xor :: Int -> Int -> Int              --built-in
```

```
int* xorArray(int mask, const int* input, int* output,
    int N){
    for(int i = 0; i < N; i++){
        output[i] = input[i] ^ mask;
    } //end for

    return output;
}
```



List Functions — Folds/Reduce

- Problem: Traditional folds are sequential
- Solution: Define a new kind of fold

```
void foldl(int* array, int N, int (*f)(int), int x0){
    int res = x0;
    for(int i = 0; i < N; i++)
        res = f(res, array[i]);
    return res;
}
```

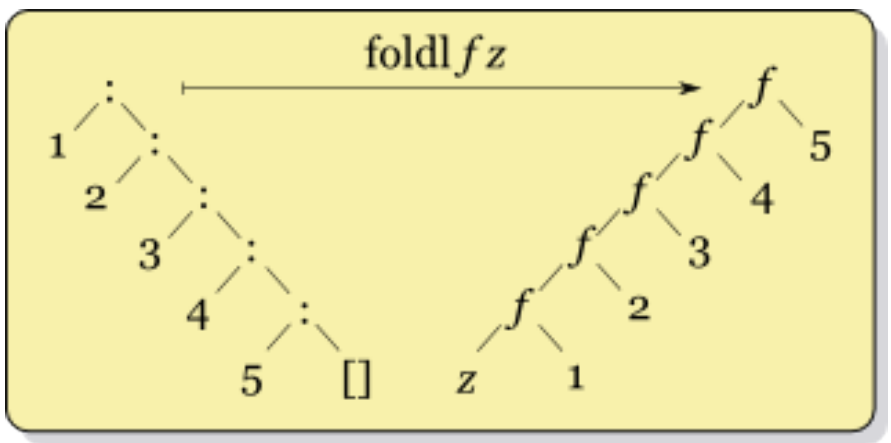


Image: Cale Gibbard (21 October 2006). *Left fold transformation* [Image]. Retrieved from

List Functions — Folds/Reduce

- Problem: Traditional folds are sequential
- Solution: Define a new kind of fold

- Monoidic Folds
  - Monoid — a algebraic structure consisting of a function and a set over which it is associative and has a right-identity.
  - This means that we can insert the right-identity at arbitrary points in the list, without changing the result.
  - $\Longrightarrow$ We can use reduction trees!
  - Added bonus: doesn't matter if the parallelism is a factor of the list length

Section 3

Case Study — Dot Product

Idiomatic C

```
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;
}
```

With Separate Addition and Multiplication

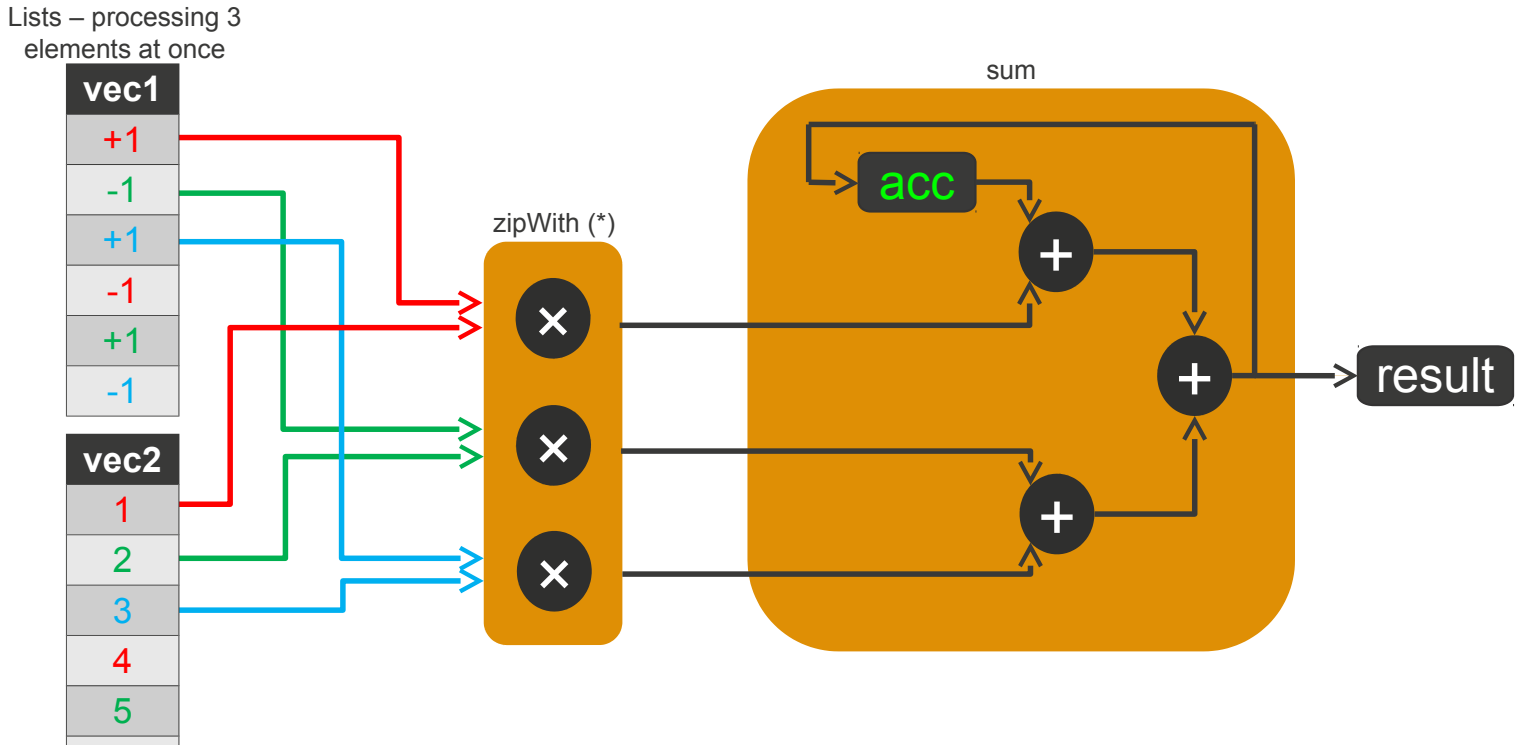
```
int dotProduct(const int* __restrict__ listA, const int
    * __restrict__ listB, int length){
    int result = 0;
    int tmp[length];

    #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;
}
```

Generalized to arbitrary functions

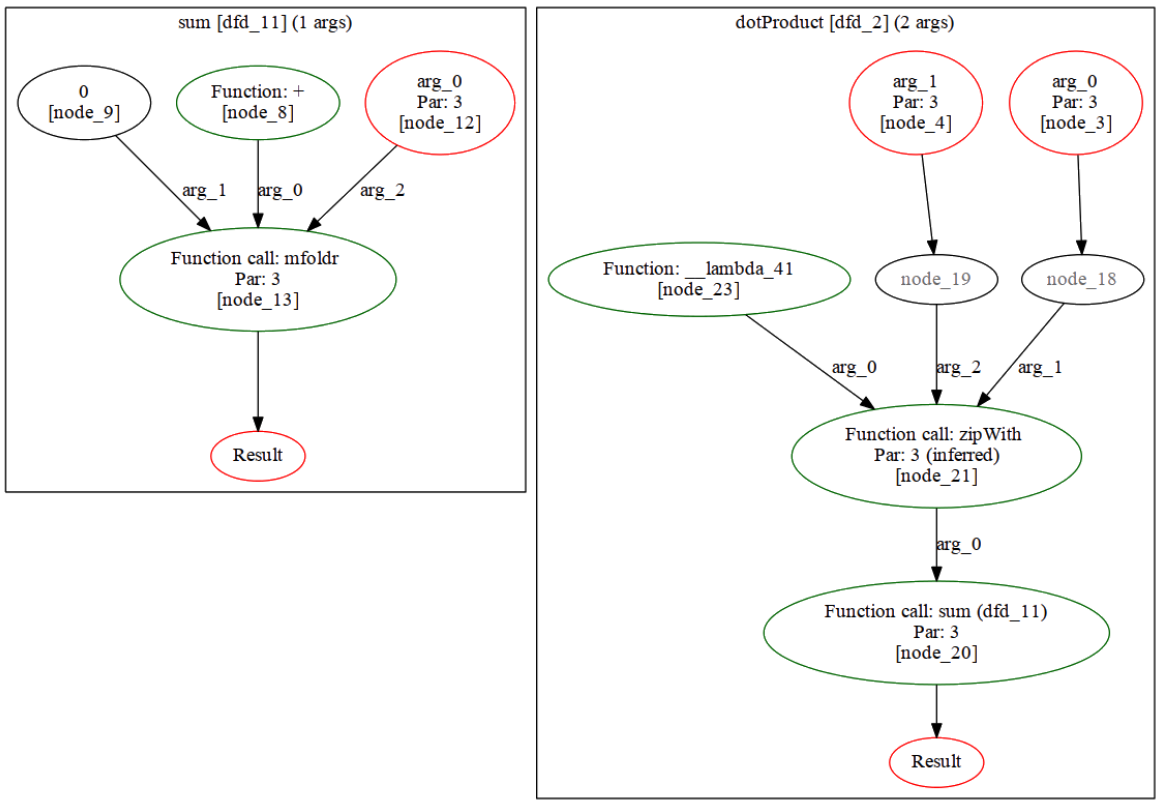
```
int innerProd(const int* __restrict__ listA, const int*
    __restrict__ listB, int length){
    int result = 0;
    int tmp[length];

    #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 Haskell

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

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



In Haskell


sum [dfd_11] (1 args)
0
[node_9]
Function: +
[node_8]
arg_0
Par: 3
[node_12]
arg_1
arg_0
arg_2
Function call: mfoldr
Par: 3
[node_13]
Result
dotProduct [dfd_2] (2 args)
arg_1
Par: 3
[node_4]
arg_0
Par: 3
[node_3]
Function: __lambda_41
[node_23]
node_19
node_18
arg_0
arg_2
arg_1
Function call: zipWith
Par: 3 (inferred)
[node_21]
arg_0
Function call: sum (dfd_11)
Par: 3
[node_20]
Result

Section 4

Future Work

Future Work

- Improved type support
  - Variable width integers
  - Fixed-point types
  - Nested lists
- Complete type & parallelism inference
- Closures
- Compilation of recursive list functions
- Resource sharing

Demo & Questions