HWML:
A Polymorphic Functional Hardware Description Library for ML

Sven Woop and Philipp Slusallek
Technical Report, Computer Graphics Lab
Saarland University
Email: woop@cs.uni-sb.de slusallek@cs.uni-sb.de

Abstract

This paper describes HWML, a structural hardware description library for the functional programming language ML. The usage of functional programming techniques allows high level abstraction of the circuit structure. Components are denoted by ML functions which allows to specify components recursively or to write high level components that get components as argument or return components. Polymorphic components in conjunction with abstract wires and typed multi-ported memories are supported which increases reusability of code. Automatic pipelining of non-cyclic circuits, that might even contain memory reads and writes, abstracts the computation flow graph from the cycle domain. For concurrent usage of long latency pipelines, the pipeline can by specified for a single thread and later be multi-threaded automatically.

The library functions are optimized to the target platform, such that block multipliers or block RAMs available in Xilinx FPGAs are used automatically for instance.

These features all together make HWML a powerful library for hardware description that allows writing compact and expressive code.

The essential interface functions to the HWML library are explained and example code demonstrates the advantages of polymorphic and functional high level hardware description.

1 Introduction

In 1965, Gordon Moore forecasted an exponential increase of the number of transistors on a single die. In parallel to this increase of complexity the tools to handle the larger and larger amount of transistors have raised their level of abstraction by providing higher level hardware description languages (HDLs) and powerful tools for hardware synthesis.

There are two main types of hardware description languages using either a structural or a behavioral approach. Structural hardware description languages describe the connection between atomic components, while behavioral languages describe the semantics of the circuit. These behavioral languages abstract from low level details as much as possible by mapping the semantic description in a high level synthesis step to the target platform.

Structural hardware descriptions using schematic entry approach has been a standard technology for a long period. Later VHDL and Verilog put structural and behavioral description together into text based languages. Both VHDL and Verilog are complex languages, require attention to low level details, and lead to rather verbose code with many lines for little functionality. Despite these drawbacks they are well established HDLs in the industry.

Functional aspects in hardware description provide many advantages (see below) and have been analyzed by many researchers [Tom Hawkins ; Mycroft and Sharp 2000; Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh], but none of them supports polymorphism or multi-ported typed memories, which are essen-

tial for hardware designs. On the other hand, Viva [Starbridge] is a visual schematic entry tool that does support polymorphism.

Several hardware description libraries for existing programming languages have been developed, like: System C [SystemC Community] for C, JHDL [Peter Bellows and Brad Hutchings] for Java, and Lava [Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh] for Haskel.

Especially for FPGAs the availability of high level hardware description languages is important to give the designer the possibility to quickly develop, change, or extent the design. Ideally, high level hardware description languages would allow for programming FPGAs just as easily as writing and compiling parallel code for a multi-processor system.

A language that allows this is Handel-C [Celoxica], which is a reduced C dialect with parallel statements that allows to write C code that is later compiled to a low level hardware description. A more abstract behavioral hardware description language is Mitrion-C [Mitrion]. The main difference to Handel-C is that the Mitrion-C programmer has no cycle accurate control to the generated circuit, as the Mitrion-C compiler performs cycle allocation and scheduling of the computation as well. This makes hardware description principally as easy as writing standard C code, but still the programmer needs to have the hardware platform in mind, as the length of its program is limited by the comparatively small amount of hardware resources on its FPGA. Furthermore the programmer has hardly any control to the generated circuit, which makes hardware designer dependent from the quality of the high level HDL compiler.

Behavioral hardware descriptions require less knowledge about hardware design but often produce less efficient circuits. Pure behavioral hardware description languages like Handel-C or Mitrion-C are not used for large industrial projects, mainly because efficiency is often the motivating factor for a hardware design. Structural languages generally provide more options for optimizing a hardware design.

In this paper we focus on the use of higher level abstractions in structural hardware description based on an implementation in the functional programming language ML. The hardware meta library HWML combines the ideas of polymorphic components, abstract wires, and typed memories to a powerful functional programming environment. High level operations on components are supported, such as automatic pipelining and multi-threading. Functional programming techniques such as higher order functions (which lead to higher order components) are very useful for abstract structural hardware design.

The HWML library has been used in the SaarCOR project [Jörg Schmittler, Sven Woop, and Philipp Slusallek] for the design of special purpose ray tracing hardware. This strongly multi-threaded ray tracing processor [Sven Woop, Jörg Schmittler and Philipp Slusallek] consists of four general-purpose shading processors (SPUs) each operating on four-component floating point or integer vectors. The SPUs are augmented with four dedicated ray traversal units (TPUs) that allow for fast traversal of rays through spatial index structures. The architecture is fully programmable and implements the typical processor instruction set of current graphics hardware GPUs but extended to also support efficient recursion and fast shooting of rays. Ray tracing performance levels that can be achieved with an initial FPGA-based prototype are up to 30 times faster than traditional CPU based ray tracing implementations. The core SPU processor has been implemented with the HWML library in about 1000 lines of code, which shows that efficient, compact, and expressive code can be created with HWML. HWML has also greatly improved the ability to test and debug the hardware design.

2 Hardware Meta Library

Describing hardware with the hardware meta library HWML works in two different levels: the first level is defined by ML as the underlying language, and the second level by the library itself. All ML statements are executed directly by the ML system, while the HWML library functions build an internal gate level graph structure, which can later be executed on an FPGA or by a simulation.

Embedding the library into ML has the advantage, that one can take advantage of all the powerful features of ML, like higher order functions, pattern matching, build-in lists, defining new operators, and so on. Knowledge about ML programming is essential to understand some of the following sections. For a brief introduction to ML see [Scott Smith]. In our lab we use the Moscow SML dialect [Kokholm and Sestoft] to test the library, but a mapping to other ML implementation should be trivial.

2.1 Abstract Wires

Typically hardware description libraries for functional programming languages use the type system of the functional language to type their wires. The advantage of this concept is that static type checking can be used. However, the disadvantage is that polymorphic functions can not be supported, as the type system forbids that. Even a separate register needs to be present for each wire type like boolean wires or integer wires for instance.

The HWML library goes a different way and defines a single basic data type `wire` to represent wires of various type. Because each wire has the same ML type `wire`, polymorphic function on various different types of wires can be implemented in the library. An example is a "register" function that can delay an arbitrary wire by a single cycle. A drawback of this concept is that a wire can no longer be statically type checked by the ML system. This problem is solved by a dynamic runtime type checking performed internally by the library, thus adding an integer to a floating point value will cause an exception to be raised for instance. The runtime type of a wire can be accessed using the `tyOf` library function.

Wires can have a highly complex structure and a unique runtime type is connected to each wire. The library works by connecting wires using library functions and special operators to a circuit. Internally the library breaks the high level operations down to the gate level and represents the circuit in a graph labeled with atomic gates like `and, or, not, mux, ...`

The HWML library supports several basic types of wires: boolean wires, integer wires with an arbitrary number of bits and even floating point wires with arbitrary precision. Using the `L` constructor nested hierarchical wires can be created, which can be used to create tuples or lists of wires. The statement `L[A,B]` creates a wire that is a tuple of two wires `A` and `B` for instance. There are functions to create constant wires, such as the boolean constants `B0()` and `B1()` or integer constants `mkI width value`.

Wires are called *abstract wires* in the following, as their exact structure can remain unknown for its connected component. Accessing the structure of such a wire is provided by pattern matching, which is standard ML functionality. The statement `val L[L[x,y,z],_,L[a,b]] = A` for instance decomposes an arbitrary wire `A`, that consists of three components, a three component vector `L[x,y,z]`, an arbitrary part `_` and a two component vector `L[a,b]`. Subsequently, the variables `x,y,z,a,b` are bound to the specified sub-wires of `A`.

2.2 Logical and Arithmetic Operations

ML supports functionality to introduce new operators to the language. The HWML library uses this functionality to define new operators to perform logical and arithmetic operations on wires. Standard logical operations such as negation (`not A`), logical and (`A && B`), logical or (`A || B`) and others are supported. Integers and floating point wires can be compared to each other using the compare operators for equality (`A == B`), inequality (`A != B`), smaller than (`A << B`), greater than (`A >> B`) and others. Multiplexing between arbitrary wires is supported and arithmetic operations such as addition (`A ++ B`), subtraction (`A -- B`) and multiplication (`A ** B`) of two wires.

These library functions are optimized for the target platform, such that block multipliers available in Xilinx FPGAs are used by a multiplication if advantageous for instance. This is possible by a special internal treatment of the multiplication which causes a block multiplier to be instantiated instead of synthesizing a gate netlist for it.

All the operators are polymorphic, which means that them can be applied to wires of different runtime type. One can for instance add two integers together or two matching tuples of integers. More general, the library functions and operators operate recursively on the wires and perform some scalar extension. For instance, let `a,b,c,d` be boolean wires then `L[a,b] && L[c,d]` is the same as writing `L[a && c,b && d]` and the expression `a && L[c,d]` is scalar extended to `L[a && c,a && d]`. This scheme is applied recursively to each library operator and function. This polymorphism is inherited to functions build out of these library operators and functions.

The main advantage of polymorphism is that the same code can be used in several contexts, which increases reusability. A problem that often happens with pipelined designs is that some wires need to be by-passed by a component, as them are only used by a later one in the pipeline. This is easily possible in our approach, by packing these wires to be by-passed to a single complex wire and pass it through the component by taking advantage of the polymorphic registers and memories.

Polymorphism together with higher order compo-

nents allows to implement a fold component that collects stream items of different type, see Section 2.10.

The library further includes, functions to determine the bit width of a wire, advanced mathematical operations on floating point wires like reciprocal and reciprocal square root, FIFOs, primuxes, and many more. Not all of this functionality can be described here in detail. Of course, existing ML libraries can be used as well, e.g. to operate on ML lists, vectors, file systems, ...

2.3 Register Transfer Level

To this point, only combinatorial circuits could be created with the library. But, the abstraction of clock cycles is essential in hardware design thus supported by a register `reg A` function that simply delays a wire `A` by a single cycle. A register `reg_en ce A` with a clock enable signal `ce` is also supported, as well as an n-cycles delay line `delay n A` or `delay_en ce n A`.

2.4 Cyclic Circuits

Pure functional programming techniques would not be sufficient to build circuits, as cyclic data types are normally not supported by them. ML contains besides functional also imperative language constructs, like references, that can be used to build a cyclic graph representation of the circuit.

This allows to create cycles by using two special functions. The `wire ty_wire` function creates a fresh wire of the specified runtime type and the `<-` assign operator can later be used to connect a different wire to it. This technique can be used to close a cycle as shown in Figure 1. Synchronous and asynchronous cycles can be created this way. The latter one will generate a warning during circuit export, because they are normally programming errors and cannot be simulated.

2.5 Output Arguments and Input Results

A function always takes arguments and returns a result. Wires in arguments of ML functions are mostly used as input wires. Often it is usefull to return a value even to an argument wire of a function. For instance, the full signal of a fifo is read by the component writing to it, thus a fifo might get an input stream, consisting of the stream data and the full signal as argument. This full signal is going to the different direction as the data stream itself, as it is computed by the fifo.

```
fun my_reg_en ce in =
let val out = wire (tyOf in)
    val _ = out <- reg(mux(ce,out,in))
in out end
```

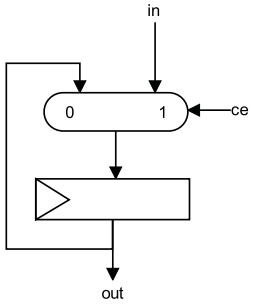

Figure 1: Register with clock enable build from a standard register using a back loop. If clock enable is false, the current output value is selected and clocked again to the output of the register.

This can be performed by using the `wire` and `<-` assign statement to assign a wire to a fresh wire created with the `wire` function. The other way around a result of a component can also be an logical input for it, by returning a fresh wire created with the `wire` function which is later assigned, but can still be used in computations.

The next example compares two wires for inequality by using a compare function that assigns the result to its third argument. The call of the compare function returns no value of interest, which is simply ignored by the `_` pattern.

```
fun not_equal (in0,in1) =
let
    fun compare (a,b,out) =
      out <- a == b

    val eq = wire TyB
    val _ = compare (in0,in1,eq)
in Not eq end
```

2.6 Runtime Type Casting

Internally HWML is strongly runtime typed requiring a type cast to use the bits of a wire in different ways. The expression `static_cast ty_wire A` casts a wire `A` to a runtime type `ty_wire` of the same bit width by re-interpreting the bits of `A` us-

ing the specified type. This type of cast is especially useful for decoding data read from external memory, like transforming a 32 bit integer into an RGBA color: `val L[r,g,b,a] = static_cast (TyL[TyI 8, TyI 8, TyI 8, TyI 8]) int32`.

The cast operation `dynamic_cast ty_wire A` converts the value of the wire `A` to the specified type if possible or raises an exception if the conversion makes no sense. The following statement for instance converts a floating point wire to an integer wire by returning the integer part of the floating point value `float_wire`: `val integer_wire = dynamic_cast (TyI 32) float_wire`.

2.7 Typed Multi-ported Memories

A key feature of the HWML library is the possibility to handle typed multi-ported memories by abstracting the memory ports from the memory itself. Only pure read and pure write ports are currently supported. Depending on the target platform multiple write ports may cause problems, as on FPGAs for instance a memory with 2 independent write ports and a read port can not be created efficiently.

To create a memory, the designer specifies only the read and write ports, the addressing mode, and the data type of the memory and gets the corresponding memory ports.

```
val [read0, read1,write0] =
   mem "my_memory" [READ,READ,WRITE]
     (TyI 2, TyL[TyI 8, TyI 8])
```

This example creates a memory named `my_memory`, with two read and a single write port which is addressed by a 2 bit integer containing tuples of 8 bit integers. The readable memory ports can be accessed by the `read port addr` or `read_en port ce addr` library functions. Writable ports are accessible by the `write port (addr, din)` or `write_en port we (addr, din)` function, which connects the specified wires to the memory port.

```
val data0 = read read0 addr0
val L[da1,db1] = read read1 addr1
val _ = write_en write0
                 (B1()) (addr2,L[a,b])
```

When exporting a memory component, the library automatically compiles the memories to the correct atomic memory units that are available on the target platform. For Xilinx FPGAs this implies the use of block RAM and select RAM available in those chips. For ASICs it is necessary to invoke a memory compiler which generates the required memories for the used process.

2.8 Automatic Pipelining

Pipelining is a common concept to increase the maximum operating frequency of a circuit. If there is enough parallelism in the mapped algorithm, pipelining may improve operation speed dramatically.

The library supports automatic pipelining of components that introduce no cycles, neither synchronous nor asynchronous ones. As a special feature, memory read and write accesses are allowed in the component to be pipelined. The only limitation is that a write operation must depend on a read operation to the same memory to avoid conflicts in the case that the same address is being used.

Two pipelining strategies are supported: pipeline to a defined latency or pipeline to a globally specified frequency. The fixed latency pipelining strategy is useful if the control mechanism of the pipeline only supports a fixed pipeline length. The algorithm tries to distribute the circuit depth equally between the set of pipeline stages, which may change the maximal operation frequency.

The second strategy inserts registers to meet a certain frequency. This requires a pipeline control mechanism that supports arbitrary latencies as it may change when varying the target frequency or the circuit. The pipelining function returns the pipelined component as well as its latency, which can be used to adjust the control circuits when necessary.

Before pipelining a number of optimizations are performed including constant propagation, which improves the division into pipeline stages.

At this abstract stage of the hardware design pipelining can only take the gate netlist into account. This is not perfect, as later synthesis tools might introduce different atomic components (such as LUTs for FPGAs) with timing properties unknown to HWML. Despite that, the pipelining performed by HWML is the best possible at this level of abstraction. Moreover, the HWML pipelining functionality is easy to use by simply applying a pipelining statement to a component (see the next CPU example).

Register re-balancing frequently available in lower

level synthesis tools are usually limited to a single level of the hierarchy and often consumes significant synthesis time. By providing a good initial pipelining synthesis time is greatly reduced and we become more independent from the quality of synthesis tools for the target platform.

The following simple example shows the main part of a very simple CPU, supporting an add and multiply operation. Firstly, the instruction memory `IM` and the register file `RF` are created. The following lines read the instruction from the address specified by the program counter (PC), access the register file, perform the addition and multiplication in parallel, then select between both computations depending on the instruction, and write the data back to the register file. The CPU is then pipelined to latency of three and instantiated.

```
fun CPU pc =
let
  (* create instruction memory *)
  val [IM] = mem "IM" [READ]
     (TyI 2, TyL[TyB, TyI 2,
      TyI 2, TyI 2]]) nil

  (* create register file *)
  val [R0, R1, W0] =
    mem "RF" [READ,READ,WRITE]
     (TyI 2, TyF (7, 24)]) nil

  (* cpu code starts here *)
  val L[op, srcA, srcB, dst] = read IM pc
  val A = read R0 srcA
  val B = read R1 srcB
  val AopB = mux(op, A ++ B, A ** B)
  val _ = write_en W0 (B1()) (dst, AopB)
in
  L[inc pc, AopB]
end

val CPU_piped = pipe_lat (CPU, 3)

val (valid,pc,busy) =
  CPU_piped (valid, pc, busy)
```

As one sees, the pipelining adds a `valid` signal and a `busy` signal to the component. The valid signal specifies if the input or output wires contain valid data in the current cycle. The `busy` signals can be used to stall the pipeline when necessary. See Figure 2 for a possible pipelining result of the CPU.

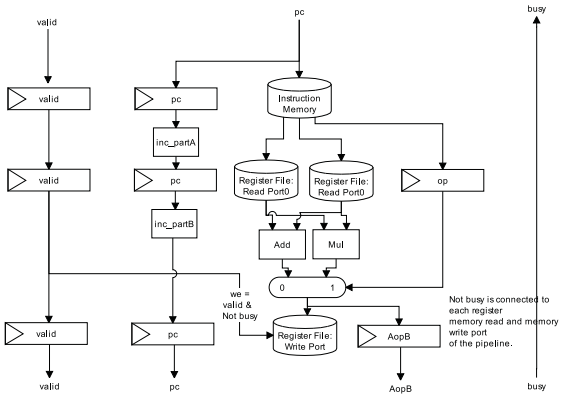

Figure 2: A result of applying automatic pipelining to the simple CPU. The incrementer is balanced to several pipeline stages, and the busy signal (going from bottom to top) is automatically connected negated to each clock enable.

2.8.1 Multi-Threading

The pipelining algorithm has been extended to also perform multi-threading of the circuit for a more efficient usage of pipeline resources. The pipeline is specified only for a single thread of control. All data paths are then augmented with a thread id and memories are replicated and indexed by this id.

2.9 Recursive Circuit Description

Many arithmetic and logical units have a natural recursive definition. Prime examples are: conditional sum adders, carry lookahead adders, decoders, encoders, and many more. Trying to define a parameterized version of these components without support for recursion it quite difficult.

For instance, recursively defining a decoder of arbitrary input width is easily possible, as shown by the next example.

```
fun decode nil = [B1()]
  | decode (x::xr) =
       ((not x) && (decode xr)) @
       (x  && (decode xr))
```

The decode function gets an integer as input that is represented by a bit list. This bit list is processed from the most significant bit to the least significant bit. The decode component consists of two rules. The decoding

of an empty bit list `nil` is a list containing only the bit 1 trivially. If the list can be split into the most significant bit `x` and a rest `xs`, the list `xs` is decoded recursively. Two of these decoded lists are concatenated together, but one of both is tied to zero depending on the most significant bit.

One sees the strength of recursive circuit description, as even parameterized components such as our decoder can mostly be specified in a few lines of code.

2.10 Higher Level Components

The concept of using functions as parameters for other functions, or to return functions as the result of a function call is standard for functional programming languages like ML. In hardware description this technique is very rarely used despite the fact that it can significantly raise levels of abstractions. Many components can be greatly simplified and provide a higher level component by abstracting out some generic functionality.

Data streams play an important role in many hardware designs, especially in signal processing. The following example shows a polymorphic high level component `fold` that can be used to implement various functions that take a stream and return a single value computed out of it.

This `fold` component gets an combinational binary operator `f` and a stream with control signals `first` (indicating the first stream element), `last` (indicating the last stream element), the stream element `din` and a `valid` signal that indicates that there is a valid stream element in the current cycle. The fold component folds the elements of the stream by the operator f, and is defined by the following lines of code:

```
fun fold f (valid,first,last,din) =
let
  val d' = wire (tyOf din)
  val _ = d' <- reg_en valid
               (mux(first,f(d',din),din))
  val valid' = reg (valid && last)
in
  (valid',d')
end
```

Let add be the function `fun add (a,b) = a ++ b` then we can instantiate a component `fold add` that computes the sum of its input stream. As `add` is a polymorphic function, the `fold add` component is polymorphic again, thus streams of arbitrary type such as integer items or floating point items can be added. Of course the fold component can be instantiated with many different components `f` to compute various operations on the stream items. Of course, for each instantiation a separate circuit is generated.

The example of folding a stream of items with an arbitrary polymorphic operation shows the strength of the HWML approach that significantly increases the possible levels of abstraction and code reuse.

2.11 Hierarchy

Despite the use of functions for representing components, the HWML library has no access to this structure of the circuit, as it is known only to the ML system. We use extra statements to impose a hierarchy onto a circuit. This additional structure may be used later, e.g. for applying synthesis constraints or performing manual placement of components. Furthermore, the imposed structure overcomes limitations in some vendor tools that have problems with very large VHDL files and require subdivision of the circuit into smaller pieces. The hierarchy also helps the library to provide meaningful error messages.

The structure is defined hierarchically by using the opening `down name` and the closing `up()` function calls. The gates created between these two function calls are labeled with the string `name` and are placed in separate VHDL files when exporting the circuit. The `down` and `up` functions can be nested to create a hierarchy.

With this mechanism the imposed component structure is independent from the logical structure of the circuit. One can for instance simply subdivide a single component into two hierarchy components for later manual placement.

2.12 Simulation

Circuits created with the HWML library can be simulated cycle accurately within the ML framework. The `simulate` function of the library simply traverses the in-memory circuit description repeatedly for each cycle and performs any necessary computation and propagation of signals across wires. For simulation purposes wires may also be connected to standard ML functions for cycle accurate emulation of external components such as DDR memories or a host controller.

Wires can be marked for debug output using the polymorphic `inspect` function. This function gets a wire,

a condition, and a string. The string together with the state of the wire is printed in a human readable form for each simulated cycle in which the condition evaluates to true. For example, floating point wires are printed in the usual floating point format instead of rather useless bit patterns.

The library also contains a polymorphic `random` function, that gets a runtime type and returns a random wire value for fast statistical test vector creation.

2.13 Exporting Circuits

The HWML library has the capability to export the netlist of a component to a VHDL file, for further processing by external tools. If the component contains some hierarchy created with the `down` and `up` functions, then several VHDL files are created according to this hierarchy. Depending on the target platform, FPGA specific features such as block multipliers and block RAM can be taken advantage of. On the other hand, if exporting to a ASIC platforms, these multipliers need to be synthesized and an auxiliary script is generated for creating the required memories using a memory compiler.

3 Conclusion

In this paper we presented the hardware description library HWML based on the functional programming language ML. Compared to other hardware description languages, the main advantages of the library are the abstract wire construct together with the polymorphic functions, and recursive and higher order components.

The embedding of the library into the powerful ML programming framework allows for integrating powerful optimization techniques, simulation, and debugging tools with the hardware description. The high level of abstraction is supported by means for defining memories of arbitrary address width and data type independent to the underlying memory structures provided by the target platform. The concept of memory ports allows to abstract from the details of the memory itself.

A powerful high level optimization technique is provided by automatic pipelining of non-cyclic components with memory accesses. This approach abstracts the control flow of operations from the cycle domain. Furthermore, pipelines may be automatically multi-threaded by replicating the memory for internal state and identifying streamed data with a thread id.

The HWML library offers cycle accurate simulation of the circuits, including a powerful conditional logging from within the same environment for debugging purposes.

In summary HWML provides a collection of tools that even individually increase productivity of a hardware designer. However, it is the combination of these tools into a single tightly integrated environment that leads to a unique set of features and capabilities that allow to write compact, expressive, and reusable hardware descriptions, simulate them quickly, and export them directly to low level synthesis tools for FPGA and ASIC chips.

4 References

CELOXICA. Handel-C language. www.celoxica.com.

JÖRG SCHMITTLER, SVEN WOOP, AND PHILIPP SLUSALLEK. The SaarCOR realtime ray tracing hardware project, www.saarcor.de.

KOKHOLM, S. R. C. R. N. J., AND SESTOFT, P. Moscow ML, www.dina.kvl.dk/ sestoft/mosml.html.

MITRION. Mitrion-C, www.mitrion.com.

MYCROFT, A., AND SHARP, R. 2000. A statically allocated parallel functional language. In *Automata, Languages and Programming*, 37–48.

PER BJESSE, KOEN CLAESSEN, MARY SHEERAN, AND SATNAM SINGH. Lava: Hardware Design in Haskell. In *ICFP '98 in Baltimore (Maryland, USA), 1998.*

PETER BELLOWS AND BRAD HUTCHINGS. JHDL - an HDL for reconfigurable systems. In *IEEE. Symposium on FPGAs for Custom Computing Machines, pages 175-184, Los Alamitos, CA, 1998, IEEE Computer Society Press.*

SCOTT SMITH. Introduction to the ML programming language, www.cs.jhu.edu/ scott/cw/lectures/sml-intro.html.

STARBRIDGE. Viva - The Ultimate Design Language for Programming and Managing (FPGAs). www.starbridgesystems.com/viva.htm.

SVEN WOOP, JÖRG SCHMITTLER AND PHILIPP SLUSALLEK. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing, to appear at SIGGRAPH 2005.

SYSTEMC COMMUNITY. SystemC, www.systemc.org.

TOM HAWKINS. Confluence tutorial and reference manual. www.launchbird.com.