**GEMINI**

*Designing and Implementing a Functional Hardware Description Language*

*Author Advisor*

Aditya Srinivasan Dr. Andrew D. Hilton



Department of Electrical & Computer Engineering

DUKE UNIVERSITY

A thesis submitted to DUKE UNIVERSITY in accordance with the requirements of Graduation with Departmental Distinction in ELECTRICAL & COMPUTER ENGINEERING in the Pratt School of Engineering.

APRIL 2018

**ABSTRACT**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam dictum, risus in finibus vulputate, eros lorem elementum dui, in ultricies est nisl non magna. Morbi laoreet ex vel justo dictum, id pulvinar erat mattis. Mauris scelerisque odio in neque tempus ornare. Nunc ut dolor vel odio faucibus mollis fringilla vitae ipsum. Suspendisse eu bibendum ex. Interdum et malesuada fames ac ante ipsum primis in faucibus. Suspendisse potenti. Curabitur quis quam mi. Praesent placerat lorem viverra sem vestibulum faucibus id ac felis. Nulla sit amet fermentum justo. Cras ac cursus purus. Nunc sit amet nibh vestibulum, posuere nisi sit amet, feugiat ipsum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec sed urna et ante tempor viverra ut in purus.

**TABLE OF CONTENTS**

**1 Introduction 3**

**2 Language Specification 3**

**2.1** Values **3**

**2.2** Expressions **4**

**2.3** Types **5**

**2.4** Declarations **6**

**2.5** Library **7**

**3 Formalizations ?**

**3.1** Software Grammar **?**

**3.2** Hardware Grammar **?**

**3.3** Derived Terms **?**

**3.4** Typing Rules **?**

**3.5** Evaluation Rules **?**

**4 Type Safety ?**

**4.1** Proof of Progress **?**

**4.2** Proof of Preservation **?**

**4.3** Proof of Safety **?**

**5 Implementation ?**

**5.1** Lexer **?**

**5.2** Parser **?**

**5.3** Semantic Analysis **?**

**5.3.1** Type Decoration **?**

**5.3.2** Type Inference **?**

**5.3.3** Software Evaluation **?**

**5.3.4** Hardware Type Checking **?**

**5.4** Instruction Selection **?**

**6 Examples ?**

**7 Future Works ?**

**8 References ?**

**A Appendix A ?**

**B Appendix B ?**

**C Appendix C ?**

**1** **INTRODUCTION**

Hardware description languages (HDLs) have existed since the 1960s as powerful tools to precisely represent the design of electrical components [1]. As very-large-scale-integration (VLSI) became more popular for design, the first modern HDL, Verilog, was introduced between 1983 and 1984 [2]. At around the same time, the Department of Defense began developing a new standard named VHDL [3].

Since their inception, these two HDLs and variant descendent languages have been the most widely-used in industry for the verification, synthesis, and simulation of circuits. While successive iterations have provided more and more powerful features, such as datatypes and strong type systems, they have not been able to keep up with the ways in which software programming languages have evolved. This hinders their ability to concisely and efficiently express the designs they are intended to represent.

The design of Gemini was motivated by the need to provide higher-level abstractions to hardware description languages in order to improve expressivity and modularity. Such features include a strong type system, parametric polymorphism, variant types, higher-order functions, recursive types, and recursion, which are borrowed from popular contemporary functional programming languages.

**2 LANGUAGE SPECIFICATION**

In this section, we will informally specify various features of Gemini with examples of syntax. Section 3 will provide formalizations of these.

**2.1 VALUES**

All programming languages compute on values. A value is an atomic unit of data that cannot be reduced or evaluated any further. Common examples of values in programming languages are integer literals (123), boolean literals (true), and string literals (“abc”).

In Gemini, there are four types of *hardware values*, as exhibited in Table 2-1.

**Table 2-1: Hardware values**

|  |  |
| --- | --- |
| Type | Examples |
| bit | ‘b:0, ‘b:1 |
| array | #[‘b:0, ‘b:1], #[#[‘b:1], #[‘b:0]] |
| record | #{valid = ‘b:1, dirty = ‘b:0} |
| tuple | #(‘b:0, ‘b:1) |

All hardware values are arrangements of bits. Elements in an array or record may be of any hardware type, thus supporting multidimensional arrays. Record values have explicitly labeled fields, as the first example shows with valid and dirty, and tuples have implicitly labeld fields corresponding to a numerical index, as the second example shows with the first element having label 1 and the second element having label 2.

In Gemini, there are several types of *software values*, as exhibited in Table 2-2.

**Table 2-2: Software values**

|  |  |
| --- | --- |
| Type | Examples |
| integer | 0, ~10, 42 |
| real | 9.8, ~3.14, 2.998e8 |
| string | “”, “hello, world”, “abc\n123” |
| list | [], [1, 2, 3], [[“abc”, “def”], [“123”, “456”]] |
| record | {name = “John”, age = 42} |
| tuple | ([“a”, “b”], 2), () |
| ref | **ref** 0, **ref** [1, 2, 3] |
| software-wrapper | **sw** ‘b:0, **sw** #[‘b:0, ‘b:1] |
| variant | SOME 100, NONE |
| function | **fun** double x = x \* 2 |

We devote a special discussion to the software-wrapper value in Section 2.3.

In Gemini, there is one type of *module value*, which is similar to a function except it operates on hardware values. Further, higher-order modules are unsupported. This is intentional by design, as will be discussed in Section 2.3.

**2.2 EXPRESSIONS**

Expressions operate on values and subexpressions, and can be reduced by evaluation rules to other expressions or values. These evaluation rules will be formalized in Section 3.4; here we give an overview.

**2.2.1 Operators**

Operators are analogous to functions, except their application is infix and the operands are the arguments. In Gemini, operator overloading is not as common as in other languages. For example, the addition of integers and reals involve separate operators (+ and +. respectively). This is common in strongly typed languages, since it enables the exact determination of types when performing inference. For a complete list of the operators and their semantic results, please consult Figures A-1 through A-5 of Appendix A.

**2.2.2 Accesses**

Records (both software and hardware), tuples (both software and hardware), arrays, and refs can be accessed to retrieve data. Record accesses are performed with the syntax #f e where f is the field name and e is the record to access. Tuple accesses are performed similarly, except f must be an integer literal corresponding to the tuple index. Array accesses are made with the syntax e[:i:], where e is the array and i is the index to access. Ref accesses are made with the syntax $e, where e is the ref to access.

**2.2.3 Conditional**

Conditional expressions allow for control flow. There are two types of conditionals: if-then-else expressions and if-then expressions. The former is produced with the syntax **if** e1 **then** e2 **else** e3, and the result is e2 if e1 is nonzero, else it is e3. The latter is a special case where the else clause is omitted and implicitly returns the empty tuple: **if** e1 **then** e2.

**2.2.4 Assignment**

Assignment expressions are used to assign values to refs. The syntax is e1 := e2, where e1 is the ref.

**2.2.5 Sequence**

Sequence expressions allow multiple expressions to be evaluated for side-effect, with the last one being returned as the value. They are made with syntax (e1; e2; en-1; en)

**2.2.6 Pattern-match**

Pattern-match expressions are used as another form of control flow. A pattern-match expression consists of a test expression e and an ordered set of match-result pairs (m1, r1), …, (mn, rn). The value of e is compared to each match, and the first mi for which there is a match, ri is returned. The syntax is **case** e **of** m1 => r1 |: m2 => r2 |: mn-1 => rn-1 |: mn => rn.

**2.2.6 Let-bindings**

Let-bindings are used to bind identifiers with values or types and to then evaluate an expression in the same scope. The syntax is **let** <decs> **in** e **end**, where <decs> is a series of zero or more declarations, which will be discussed further in Section 2.4.

**2.3 TYPES**

In the previous discussion we have mentioned the concept of types without providing a formal definition. A type is a category to which a value can be assigned. Alternatively, a type is a set of values, with each type being disjoint from the others.

In type theory, there is a special term given to the type of types: kinds. Kinding is necessary since not all types have the same structure. For example, the type list is itself not a type, but a type constructor. It is a type-level function that takes a type, such as int, and produces a type, such as int list.

In conventional type systems, kinds are built from an atomic kind, written as and pronounced “type”, and the constructor [5]. The kind of proper types – such as int, real, and string – is , the kind of type constructors – such as list and ref – is , and the kind of functions from proper types to type constructors – such as pair – is .

In Gemini, the kinding system is composed of three atomic kinds: (the software kind), (the hardware kind), and (the module kind). The reason for the separation is to enforce the trichotomy between software, hardware, and module types in the type system, since types from two different kinds are not interchangeable. The BNF diagram in Figure 2-1 lists the member types of each kind.

**Figure 2-1**: Kind definitions

::= int

| real

| string

| list

| {l1: , ..., ln: }

| ref

| sw

| Ci

|

::= bit

| []

| @

| #{l1: , ..., ln: }

::=

Here, and refer to types belonging to kind and respectively. The kinding definition of Gemini intentionally prohibits certain expressions and values from being defined. For example, higher-order modules cannot exist since the module type only allows types of kind as the argument and result. Further, hardware values cannot persist in software values such as lists and functions, and vice versa. This is an important property that enables the compilation of Gemini by first evaluating all software expressions and then processing the resulting program consisting only of hardware values.

We note that records and tuples are both represented by the same type. This is because tuples are a special kind of record with implicitly numbered fields, and apart from syntax these two are treated as the same.

In addition to having three atomic kinds, the Gemini type system is interesting for two more reasons. Firstly, array types and temporal types are parameterized by some integer . These differ from type constructors such as lists, which accept a type to produce a new type, in that they accept a value to produce a new type. This is necessary since hardware arrays of differing lengths represent unique types, as do temporal values of differing periods. Secondly, the software-wrapper is a vital operator allowing for inter-kind manipulation of data. When a hardware value is wrapped, it can be persisted in any software data type, such as lists, records, or functions. The actual value can never be read, and the operator unsw unwraps the value for it to be used as a hardware value.

**2.4 DECLARATIONS**

There are five types of declarations that can be made: values, functions, types, datatypes, and modules. Declarations can be made in structures or the **let**...**in** section of let-bindings.

**2.4.1** **Values**

Value declarations bind an identifier x to a value with one of two syntaxes. The value can be declared with implicit type using the syntax **val** x = e. Alternatively, the value can be declared with an explicit type using the syntax **val** x : T = e, for some type T.

**2.4.2 Functions**

Function declarations bind an identifier f to a function with the syntax **fun** f arg1 argn = e, where argi can be an identifier, a record of arguments, or a tuple of arguments. The function and any arguments may be explicitly typed by suffixing with :T for some type T.

**2.4.3 Types**

Type declarations bind an identifier t to a type with the syntax **type** t = T, for some type T. Type constructors that are parametrically polymorphic may also be declared by introducing type variables using the syntax **type** ‘a t = T’a, where T’a is some type that may use ’a. The syntax for type declarations is shown in Figure A-6 of Appendix A.

**2.4.4 Datatypes**

Datatype declarations bind an identifier d to a variant type. Software datatypes are declared with the syntax **sdatatype** d = C1 **of** TS1 |: Cn **of** TSn, where Ci is a datatype constructor that accepts software type TSi and produces datatype d. Similarly, hardware datatypes are declared with the syntax **hdatatype** d = C1 **of** TH1 |: Cn **of** THn. Constructors need not require any type to be constructed, in which case the **of** Ti is ommitted.

**2.4.5 Modules**

Module declarations bind an identifier m to a module with the syntax **module** m arg = e, where arg can be an identifier, a hardware record of arguments, or a hardware tuple of arguments. The module and argument may be explicitly typed by suffixing with :T for some type T. Modules can also be parameterized with software values using angle-bracket notation: **module** m <param> arg = e.

**2.5 LIBRARY**

The Gemini library provides useful built-in functions and modules, for operating on software and hardware values respectively. A complete list of these can be found in Figure A-7 of Appendix A.

**3 FORMALIZATIONS**

Having specified the type system and syntax of values, expressions, and declarations, we now formalize our language grammar and rules. These completely define the semantics of Gemini.

**3.1 SOFTWARE GRAMMAR**

We first define the grammars for software-typed values and expressions.

**3.1.1 Software Value Grammar**

An excerpt of the grammar is shown below. Figure B-1 of Appendix B illustrates the grammar in full.

**Figure 3-1**: Excerpt of software value grammar

::=

|

|

|

|

|

|

|

|

::=

::=

::=

::=

This grammar represents each value abstractly. The value for refs is the location store augmented by binding the location to some software value. The value for functions is a lambda expression. The syntax grammar for declaring values and expressions follows.

**3.1.2** **Software Expression Grammar**

An excerpt of the grammar is shown below. Figure B-2 of Appendix B illustrates the grammar in full.

**Figure 3-2**: Excerpt of software expression grammar

::=

|

|

|

|

|

|

|

|

::=

|

|

|

|

|

|

|

::= [ ]

| nil

::=

|

::= ,

|

**3.1.3 Derived Expressions**

Some expressions can be expressed in terms of other expressions, and are therefore denoted as ‘derived expressions’. Beyond the syntactic layer of the compiler, these are treated analogously to the expressions from which they are derived. The set of derived terms can be found in Figure B-3 of Appendix B.

**3.2 HARDWARE GRAMMAR**

There exists both software-typed values and expressions. However, there only exists hardware-typed values. The semantics of a hardware-typed expression is ill-defined since no hardware circuit can be evaluated or reduced any further than the structure it takes. Thus, here we define the grammars for hardware values and their syntax.

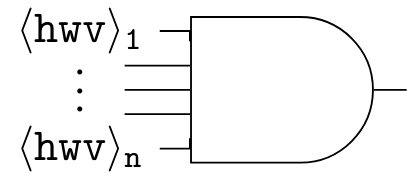
**3.2.1 Hardware Value Grammar**

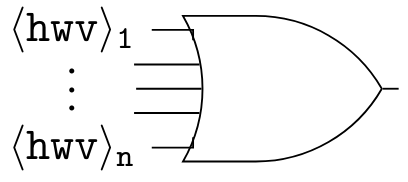
The grammar in full is shown below.

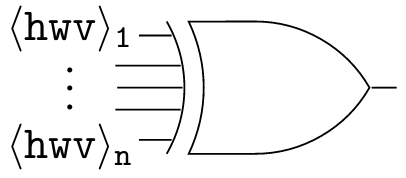
**Figure 3-3**: Hardware value grammar

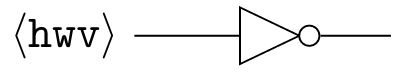
::=

|









|

| dff()

**3.2.2** **Hardware Syntax Grammar**

An excerpt of the grammar is shown below. Figure B-3 of Appendix B illustrates the grammar in full.

**Figure 3-4**: Excerpt of hardware syntax grammar

::=

|

|

|

|

|

|

|

::=

|

|

|

|

It can be observed that the grammars for software and hardware syntax both feature non-terminals such as conditional and pattern match. This is because these expressions may be used to return either software- or hardware-typed values. The grammar itself is not responsible for enforcing typing rules, which brings us to the next section.

**3.4 TYPING RULES**

With the language grammar formalized, it is possible to identify whether a program is valid under the language, and to construct an abstract syntax tree. This section provides the tools to verify that the program is well-typed, which is imperative for correctness in compilation.

Each typing rule is a theorem, with a set of propositions or hypotheses and a conclusion. Diagramatically, the antecedent clause is written above the horizontal line, with the consequent clause below.

An excerpt of the set of typing rules is shown below. Figure B-4 of Appendix B illustrates the set of typing rules in completion.

**Figure 3-5**: Excerpt of typing rules

|  |  |
| --- | --- |
| x : T  (T-VAR)  x : T | , x : T1 t2 : T2  (T-ABS)  x:T1.t2 : T1 T2 |
| t1:T1 T2 t2:T1  (T-APP)  t1t2 : T2 | t1:int t2:int  (T-INT-ADD)  t1 + t2 : int |
| t1 : TH t2 : TH  (T-AND)  t1 & t2 : TH | t1:int t2:T t3:T  (T-IF)  if t1 then t2 else t3 : T |

As an example, rule T-IF is pronounced “if t1 has type int, t2 has type T, and t3 has type T, then if t1 then t2 else t3 has type T”. Note in this example that the then- and else-clause may have type T belonging to any kind, whereas typing rule T-AND restricts t1 and t1 to a type of kind hardware.

**3.5 EVALUATION RULES**

Now having a rigorous formulation of the syntax and typing rules of our language, we need to precisely define how expressions are evaluated. This is known as defining the semantics of the language, for which there are three basic approaches: operational semantics, denotational semantics, and axiomatic semantics [5]. In this paper, we choose to use operational semantics to define the behavior of Gemini for its simplicity and flexibility.

Operational semantics define an abstract state machine for the language, where each state is an expression. The machine’s behavior is defined by a transition function that either yields the next state by performing a computational step, or declares that the machine has halted by reaching some terminal value [5]. Operational semantics can be further partitioned into small-step and big-step semantics. Small-step semantics, or structural operational semantics, consider how evaluation take place one step at a time. Big-step semantics, or natural semantics, describe the overall results of evaluation by describing the final value to which some expression evaluates [5].

In general, big-step semantics are less verbose since intermediate expression states need not be encoded in the machine behavior. However, small-step semantics are more readily translatable for implementation, and since our ultimate goal is developing a compiler we choose to use small-step semantics.

Similarly to typing rules, each evaluation rule is a theorem. An excerpt of the set of evaluation rules is shown below. Figure B-5 of Appendix B illustrates the set of evaluation rules in completion.

**Figure 3-6**: Excerpt of evaluation rules

|  |  |
| --- | --- |
| t1 t1’  (E-APP1)  t1 t2 t1’ t2 | v1 0 (E-IFELSE-T)    if v1 then t2 else t3 t2 |
| t2 t2’  (E-APP2)  v1 t2 v1 t2’ | (E-IFELSE-F)  if 0 then t2 else t3 t3 |
| (x.t1)v1 [xv1]t1 (E-APPABS) | t1 t1’ (E-IFELSE)    if t1 then t2 else t3  if t1’ then t2 else t3 |

As an example, rule E-IFELSE is pronounced “if t1 evaluates to t1’ in one step, then the whole expression if t1 then t2 else t3 evaluates to if t1’ then t2 else t3”.

The evaluation rules together define a precise evaluation strategy. For example, consider the expression if x then (if 1 then “a” else “b”) else “c”. Under the evaluation rules of our language, it is not possible for this to evaluate to if x then “a” else “c”, despite this being a state that would evaluate to an equivalent value. We must first evaluate the guard of the outer-conditional by E-IFELSE. Once it is a value, then we pick one of the then- and else-clause based on rules E-IFELSE-T and E-IFELSE-F and compute on it. A useful property is the determinacy of one-step evaluation, stating that if t t’ and t t’’, then t’ = t’’. This ensures that evaluation is a deterministic process.

**4 TYPE SAFETY**

Given formalizations of the syntax, typing rules, and evaluation relations of the language, it is desirable to prove a basic property of Gemini’s type system: safety. To understand safety we must first define a few terms.

**DEFINITION**: A term t is in *normal form* if no evaluation rule applies to it. (D-1)

**DEFINITION**: A term t is in a *stuck state* if it is in normal form but it is not a value. (D-2)

**DEFINITION**: A type system possesses *safety* if a well-typed term can never reach a stuck state during evaluation. (D-3)

We show the safety of our type system with two propreties.

**DEFINITION**: A type system has the property of progress if a well-typed term is never in a stuck state; either it is a value or it can take a step according to some evaluation rule. (D-4)

**DEFINITION**: A type system has the property of preservation if when a well-typed term takes an evaluation step, the resulting term is also well-typed. (D-5)

In order to prove the progress theorem, we first posit two lemmas.

**LEMMA**: The following are true, and constitute the inversion of the typing relation: (L-1)

1. If x : R, then x : R
2. If x:T1.t2:R, then R = T1 R2 for some R2 with ,x:T1 t2:R2
3. If t1 t2:R then there is some type T11 such that t1:T11 R and that t2:T11
4. If : R, then R = int

The remainder of the cases are ommitted here and are shown in full in Figure C-1 of Appendix C.

*Proof*: Immediate from the definition of the typing rules.

**LEMMA**: The following are true, and constitute the canonical forms: (L-2)

1. If v is a value of type int, then v is an integer value according to the software value grammar.
2. If v is a value of type real, then v is a real value according to the software value grammar.
3. If v is a value of type string, then v is a string value according to the software value grammar.
4. If v is a value of type bit, then v is either 0 or 1.
5. If v is a value of type T1 T2, then v = T1.t2.
6. If v is a value of type TS ref, then v is a location in store .
7. If v is a value of type {li : Tii 1..n}, then v is a value with the form {li = vii 1..n}.
8. If C is a constructor of datatype D accepting type T1 and v is a value of type T1 then C v is a value of type D with form .
9. If v is a value of type TH[n], then v is a value according to the hardware value grammar.
10. If v is a value of type #{li : Tii 1..n}, then v is a value with the form #{li = vii 1..n}.
11. If v is a value of type TH sw, then v is a value with the for some vH of type TH.

*Proof*: We refer to the first n cases of the Lemma L-1 as they pertain to values in this language.

*Case* 1: Values in this language can take several forms. The case of an integer gives us our desired result immediately. All other forms cannot occur since we assumed that v has type int and among the cases in consideration from Lemma L-1, only case 4 tells us that the value has type int.

The remaining cases are similar.

We are now equipped to prove the theorem of progress

**THEOREM OF PROGRESS**: Suppose t is a closed, well-typed term ( t : T for some T). Then either t is a value or else there is some t’ with t t’. (TH-1)

*Proof*: By structural induction on a derivation of t : T.

*Case* T-INT, T-REAL, T-STRING, T-BIT, T-NIL:

Immediate since t is a value.

*Case* T-APP:

t = t1 t2

t1 : T11 T12

t2 : T12

By the induction hypothesis, either t1 is a value or else there is some other t1’ for which t1  t1’, and likewise for t2. If t1 t1’ then by E-APP1, t t1’ t2. On the other hand, if t1 is a value and t2 t2’, then by E-APP2, t t1 t2’. Finally, if both t1 and t2 are values, then case 5 of the canonical forms lemma tells us that t1 has the form : T11.t12 and so by E-APPABS, t [xt2]t12 which is a value.

The remaining cases are shown in full in Proof C-2 of Appendix C.

**THEOREM OF PRESERVATION**: If t : T and t t’, then t’ : T. (TH-2)

*Proof*: By structural induction on a derivation of t : T. At each step of the induction, we assume that the desired property holds for all subderivations (i.e. that if s : S and s s’, then s’ : S, whenever s : S is proved by a subderivation of the present one) and proceed by case analysis on the final rule in the derivation.

*Case* T-VAR:

t = x

x : T

If the last rule in the derivation is T-VAR, then we know from the form of this rule that t must be a variable of type T. Thus t is a value, so it cannot be the case that t t’ for any t’, and the requirements of the theorem are vacuously satisfied.

*Case* T-APP:

t = t1 t2

t1 : T11 T12

t2 : T11

T = T12

Looking at the evaluation rules with application on the left-hand side, we find that there are three rules by which t t’ can be derived: E-APP1, E-APP2, and E-APPABS. We consider each case separately.

*Subcase* E-APP1:

t1 t1’

t’ = t1’ t2

From the assumptions of the T-APP case, we have a subderivation of the original typing derivation whose conclusion is t1 : T11 T12. We can apply the induction hypothesis to this subderivation obtaining t1’ : T11 T12. Combining this with the fact that t2 : T11, we can apply rule T-APP to conclude that t’: T.

*Subcase* E-APP2:

Similar to E-APP1.

*Subcase* E-APPABS:

t1 = : T11.t12

t2 = v2

t’ = [xv2]t12

Using Lemma L-1, we can desconstruct the typing derivation for : T11.t12 yielding : T11 t12 : T12. From this we obtain t’ : T12.

The remaining cases are shown in full in Proof C-3 of Appendix C.

**THEOREM OF SAFETY**: A well-typed term can never reach a stuck state in evaluation. (TH-3)

*Proof*: Theorem TH-1 demonstrates that a well-typed term is not stuck, and Theorem TH-2 demonstrates that if a well-typed term takes a step of evaluation, then the resulting term is also well-typed. In combination and inductively, these guarantee safety.

**ACKNOWLEDGEMENTS**

I would like to express my sincere gratitude to my advisor Dr. Hilton for his involvement and support throughout the course of this project. His immense knowledge and passion for type theory and compilers encouraged me to pursue an ambitious endeavor.

**7 REFERENCES**

[1] Barbacci, M. "A comparison of register transfer languages for describing computers and digital systems," Carnegie-Mellon Univ., Dept. of Computer Science, March 1973

[2] ["Verilog's inventor nabs EDA's Kaufman award"](http://www.eetimes.com/document.asp?doc_id=1157349). EE Times. November 7, 2005

[3] Department of Defense (1992). [Military Standard, Standard general requirements for electronic equipment](http://everyspec.com/MIL-STD/MIL-STD-0300-0499/MIL-STD-454N_9160/). Retrieved November 15, 2017

[4] Barbacci, M., Grout S., Lindstrom, G., Maloney, M.P. "Ada as a hardware description language : an initial report," Carnegie-Mellon Univ., Dept. of Computer Science, 1984

[5] Pierce, Benjamin C. Types and Programming Languages. MIT Press, 2002

**A APPENDIX A**

**Figure A-1: Hardware Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| & | e1 & e2 | Bitwise logical “and” of composing bits\* |
| | | e1 | e2 | Bitwise logical “or” of composing bits\* |
| ^ | e1 ^ e2 | Bitwise logical “xor” of composing bits\* |
| ! | !e1 | Negation of bit operand |
| << | e1 << e2 | Shifts left bit array operand to the left by the amount specified (as an unsigned integer) by the right bit array |
| >> | e1 >> e2 | Shifts left bit array operand to the right by the amount specified (as an unsigned integer) by the right bit array, filling with zero bit value |
| >>> | e1 >>> e2 | Shifts left bit array operand to the right by the amount specified (as an unsigned integer) by the right bit array, filling with most significant bit value |
| &-> | &->e1 | Bitwise and-reduction of bit array operand |
| |-> | |->e1 | Bitwise or-reduction of bit array operand |
| ^-> | ^->e1 | Bitwise xor-reduction of bit array operand |
| && | e1 && e2 | Or-reduction of both bit array operands, followed by bitwise logical “and” of resulting bits |
| || | e1 || e2 | Or-reduction of both bit array operands, followed by bitwise logical “or” of resulting bits |
| ^^ | e1 ^^ e2 | Or-reduction of both bit array operands, followed by bitwise logical “xor” of resulting bits |

\* these operators pervade through the structures of the subexpressions to perform the bitwise operation on the composing bits while retaining the overall structure

**Figure A-2: Arithmetic Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| ~ | ~e | Negation of integer operand |
| + | e1 + e2 | Addition of both integer operands |
| - | e1 – e2 | Subtraction of right integer operand from left integer operand |
| / | e1 / e2 | Division of left integer operand by right integer operand, rounded towards negative infinity |
| \* | e1 \* e2 | Multiplication of both integer operands |
| % | e1 % e­2 | Modulo of dividend left integer operand with divisor right integer operand |
| +. | e1 +. e2 | Addition of both real operands |
| -. | e1 -. e2 | Subtraction of right real operand from left real operand |
| /. | e1 /. e2 | Division of left real operand by right real operand |
| \*. | e1 \*. e2 | Multiplication of both real operands |

**Figure A-3: Conditional Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| andalso | e1 andalso e2 | Logical conjunction of both integer operands |
| orelse | e1 orelse e2 | Logical disjunction of both integer operands |
| not | not e1 | Logical complementation of integer operand |

**Figure A-4: Comparison Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| = | e1 = e2 | Equality of both operands |
| <> | e1 <> e2 | Non-equality of both operands |
| > | e1 > e2 | Left operand has a strictly greater order than right operand |
| < | e1 < e2 | Left operand has a strictly lesser order than right operand |
| >= | e1 >= e2 | Left operand has a greater or equal order compared to right operand |
| <= | e1 <= e2 | Left operand has a lesser or equal order compared to right operand |

**Figure A-5: List Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Semantic Result |
| :: | e1::e2 | Append left element operand to right list operand |

**Figure A-6: Types**

|  |  |
| --- | --- |
| Type | Syntax |
| integer | int |
| string | string |
| real | real |
| list of TS | TS list |
| ref of TS | TS ref |
| software-wrapped TH | TH sw |
| record of TS1, …, TSn with labels l1, …, ln | {l1: TS1, …, ln: TSn} |
| tuple of TS1, …, TSn | TS1 \* … \* TSn |
| function from TS1 to TS2 | TS1 -> TS2 |
| bit | bit |
| array of TH | TH[n] where n is some integer literal |
| temporal of TH | TH @ n where n is some integeral literal |
| record of TH1, …, THn with labels l1, …, ln | #{l1: TH1, …, ln: THn} |
| tuple of TH1, …, THn | TH1 #\* … #\* THn |

**Figure A-7: Library Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Structure | Function | Type | Semantic Result |
| Core | print | string -> unit | Write a string to the standard output |
| read | string -> string | Read the contents of a file |
| List | nth | (‘a list \* int) -> ‘a | Return an element from a list given an index; raises an exception if the index is out of bounds |
| length | ‘a list -> int | Return the length of a list |
| rev | ‘a list -> ‘a list | Return the reversed list |
| map | (‘a -> ‘b) -> ‘a list -> ‘b list | Return a list after applying a function to each element |
| filter | (‘a -> int) -> ‘a list -> ‘a list | Return a list containing only elements that satisfy the predicate function |
| foldl | (‘a \* ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b | Accumulate a value by iterating over a list from left to right |
| foldr | (‘a \* ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b | Same as foldl except iteration is right to left |
| Int | toString | int -> string | Return a string representation of an int |
| String | size | string -> int | Return the number of characters in a string |
| substring | (string \* int \* int) -> string | Return the substring from a start to end location of a string; raises an exception if either index is out of bounds |
| concat | string list -> string | Return the concatenation of all strings in a list |
| split | string -> string -> string list | Return a list of strings resulting from splitting an original string over some delimiter |
| Real | floor | real -> int | Return a real rounded towards negative infinity |
| ceil | real -> int | Return a real rounded towards positive infinity |
| round | real -> int | Return a real rounded towards the closest integer |
| fromInt | int -> real | Return a real converted from an integer |
| toString | real -> string | Return a string representation of a real |
| Array | toList | ‘a[n] sw -> ‘a sw list | Return a list of software-wrapped hardware values from a software-wrapped hardware array |
| fromList | ‘a sw list -> ‘a[n] sw | Return a software-wrapped hardware array from a list of software-wrapped hardware values |
| BitArray | twosComp | bit[n] ~> bit[n] | Return a circuit performing twos-complement of a bit array |
| HW | dff | ‘a ~> ‘a @ 1 | Return a DFF circuit with a given hardware input |

**A APPENDIX B**

**Figure B-3: Derived Terms**

|  |  |
| --- | --- |
| Name | Equivalence |
| tuple |  |
| unit | () {} |
| logical and | e1 andalso e2 if e1 then e2 else 0 |
| logical or | e1 orelse e2 if e1 then 1 else e2 |
| logical not | not e1 if e1 then 0 else 1 |
| and-reduction | &->#[e1, e2] e1 & e2 |
| or-reduction | |->#[e1, e2] e1 | e2 |
| xor-reduction | ^->#[e1, e2] e1 ^ e2 |
| and collapse | e1 && e2 (|->e1) & (|->e­2) |
| or collapse | e1 || e2 (|->e1) | (|->e­2) |
| xor collapse | e1 ^^ e2 (|->e1) ^ (|->e­2) |
| if-then | if e1 then e2 if e1 then e2 else {} |
| sequence | (e1; e2) (T.t2) t1 where t2 |