COLUMBIA UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

EHDL

Easy Hardware Description Language

Language Reference Manual

Paolo Mantovani (pm2613)
Mashooq Muhaimen (mm3858)
Neil Deshpande (nad2135)
Kaushik Kaul (kk2746)

October 28, 2011

1 Introduction

This manual describes the EHDL language. EHDL is a programming language that allows the programmer to use an imperative style to formally describe and design digital systems.

2 Syntax Notation

In the syntax notation used in this manual, nonterminals are indicated by *italic* type, terminals are indicated by single quotes. We make frequent use of regular expression notation to specify grammar patterns. r* means the pattern r may appear zero or more time, r+ means the r may appear one or more times, r? means r may appear zero or once. r1 | r2 denotes an option between two patterns, r1 r2 denotes r1 followed by r2.

3 Lexical Conventions

A program is a list of global constants and a list of functions. A function is a list of output buses, input buses and a body that describes the functionality of a portion of the hardware design that that function represents.

3.1 Tokens

There are 7 types of tokens: white space, comments, identifiers, keywords, literals, operators, and other separators. If the input string stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

3.2 Whitespace

Blanks, tabs, and newlines, collectively referred to as “white space” are ignored except to separate tokens.

3.3 Comments

There are two types of comments: single line and multiline. The characters // introduce a single line comment. The characters /* introduce a multiline comment, which terminate with the characters */.

// has no special meaning inside a /* ... */ block, and /* and */ lose their meaning if they come after // in a line.

3.4 Identifiers

An identifier consists of a letter followed by other letters and digits. The letters are the ascii characters a-z, A-Z and _. Digits are ascii characters 0-9. Upper and lower case characters are different (EhDl and ehdl are separate identifiers). There is no limit on the length of an identifier.

letter → ['a'-'z' 'A'-'Z' '_']

digit → ['0'-'9']

identifier → *letter(letter* | *digit)*+

3.5 Keywords

The following identifiers are reserved as keywords and may not be used otherwise:

if	Switch	int	POS
else	case	while	ASYNC
for	const	uint	return

3.6 Literals

A literal is a sequence of digits optionally preceded by the character '-' to indicate negativity. Some examples of literals are : 123, -123 , 0 etc.

literal → *-? digit+*

3.7 Operators

EHDL has the following operators :

+	-	*	/	%
<	>	<=	>=	!=
==	||	&&	^	<<
>>	^	=	(	)
[	]	!		

The precedence and associativity of the operators are described in section 5.3.

3.8 Separators

EHDL has the following separators and delimiters:

,	:	;	{	}

4 Meaning of Identifiers

Identifiers refer to a variety of things: functions, constants, and variables. A variable is defined solely by its type.

4.1 Types

There are two fundamental types: int(k) type and uint(k) type. There is also a derived type : the array type.

4.1.1 int(*k*) and uint(k) Type

type_specifier → int(*k*) | uint(k)

int(*k*) and uint(k) are used to indicate a *k* bit input or output bus. *k* has to be greater than 0. The value an int(k) bus takes is interpreted to be a signed integer, while uint(k) bus values are interpreted to

be unsigned. Examples of int(*k*), uint(k) types are: int(5), uint(32) etc.

4.1.2 Array Type

Arrays are vectors containing a particular type. e.g. uint(32) imem[512] is a 512 length vector of uint(32) types.

4.2 Functions

An EHDL function represents a portion of hardware design that has well defined inputs and outputs and that performs a well-defined function. [3]

5 Expressions

Expressions are constants, variables, operator expressions and function calls.

expr → *constants*
| *variables*
| *ops*
| *function_call*

5.1 Constants

A constant is a literal or a const type declared in accordance to section 6.

5.2 Variables

A variable has the following form:

variable → *identifier*
| *array-reference*
| *subbus*

5.2.1 Array References

array-reference → *identifier*[*expr*]

The first identifier must be an array type while the *expr* inside the square brackets must evaluate to a uint(*k*) type (a bus interpreted as its value, which is a positive integer) or a number (for instance the index of a *for-statement*). If an array reference is made inside the body of a for loop, and if the expr inside the square brackets includes a loop index, the expression must be a const expression.

5.2.2 Subbus

subbus → *identifier(const:const)*

Subbuses can be used to refer to a subset of bits that form a named bus. e.g. if m is an uint(32) input bus, m(0:4) denotes the first 5 bits of m.

5.3 Operator Expressions

Table 1 lists the operators in order of precedence (highest to lowest). Their associativity indicates in what order operators of equal precedence are applied.

Operator	Description	Associativity
( ) [ ] .	Parentheses. Used for grouping, also function calls Brackets (array subscript) Member selection via structure name	left to right
!	Logical Negation	right to left
* / %	Mult/div/modulus	left to right
+ -	Plus/minus	left to right
<< >>	Bitwise shift left/ bitwise shift right	left to right
< <= > >=	less than/less than equal to/greater than/greater than equal to	left to right
== !=	is equal to/is not equal to	left to right
&& || ^	logical and/logical or/xor	left to right
: =	array index range/ assignment	left to right
,	Comma (separate expressions)	left to right

Table 1. Operator precedence and associativity

5.4 Function Calls

An EHDL function call has the following syntax:

function_call → identifier(arglist?)

arglist is a comma separated list of expressions.

Examples: gcd(), gcd(a,b) gcd(a , b*c) etc.

6 Declarations

An EHDL declaration is a const declaration, int declaration, array declaration or a function declaration.

declaration → const-declaration

| *ASYNC? int-declaration*

| *array-declaration*

| *function-declaration*

6.1 const Declaration

A const declaration has the following form :

const-declaration → const type-specifier identifier = literals;

example: const uint(6) rtype = 0;

6.2 ASYNC Keyword

If a variable must be asynchronously connected to different logic blocks, separated by registers, it must be declared as an asynchronous variable through use of the keyword ASYNC. Asynchronous variables are never assigned by *pos-statements (* see Section 7.3) and they can be written only once (otherwise: conflict because we will end up with multiple drivers for the same signal).

6.3 int Declaration

An int-declaration has the following form :

int-declaration → type-specifier identifier;

| *type-specifier identifier = const;*

The second option enables the programmer to specify the initial value of the variable just declared. If the value is not initialized, it defaults to 0.

6.4 Array Declaration

An array declaration has the following form :

array-declaration → type-specifier identifier[digit+];

All the elements in an array are initialized to 0.

6.5 Function Declaration

function-declaration → (outputlist) identifier (inputlist) { stmt }

Both *inputlist* and *outputlist* are comma separated lists of int or array declarations.

stmt is described in the next section.

7 Statements

A statement has the following form:

stmt → { stmtlist }

| *expr;*

| *return expr;*

| *selection statements*

| *iterative statements*

| *POS (expr);*

stmtlist is a list of semicolon separated statements.

7.1 Selection statements

selection-statement → *if (expr) no-pos-statement;*

| *if (expr) no-pos-statement else no-pos-statement;*

| *switch (expr) case-statement;*

case-statement → *case-statement-list*

| *case (expr) : no-pos-statement;*

case-statement-list is a semicolon separated list of case-statements.

no-pos-statement is a *statement* without a single instance of the POS keyword. POS has the effect of synthesizing registers and if it was allowed to exist for example in the if block, but not in the else block, this would have no physical meaning. We can't just dynamically create a register based on a value that we figure out at “runtime”. We could unconditionally create a register if POS existed inside any of the branches of a selection statement, however we chose to force the programmer to put POS outside of any selection block so that it is explicit.

7.2 Iterative Statements

There are two types of iterative statements: while loop statements and for loop statements.

7.2.1 While Statements

while-statement → *while (expr) stmt-containing-atleast-one-pos*

While loops are used to describe logic blocks that implement iterative algorithms(e.g. multiply and accumulate unit). This statement can not be used with an index to process an array. Since this kind of logic is supposed to contain a feedback, it is mandatory to introduce a sequential element that breaks the combinational loop. The body of the “while” must contain therefore at least one POS statement.

7.2.2 For Statements

EHDL “for” statements are meant to be used for array processing. They have the form

for-statement → for (*id* = *init* ; id < *end ;* id = id + incr) { *stmt* }

id cannot be declared as a type, it is automatically interpreted as an unsigned integer number, and not as a bus identifier. init, incr and end are constant expressions. Constant expressions are expressions all whose operands are const types. init is the value id is initialized to, incr is the increment applied at the end of each loop iteration, end is the terminating condition of the loop. The loop instantiates a number of different copies of logic blocks, described by the “for” body. The number, which is equal to the number of iterations, can be derived from the difference between *end* and *init* divided by *incr*. It is not permissible to change the value of *id* inside the body of the loop.

"for" statements are not meant to represent logic blocks with a feedback and the POS statement is not mandatory. However, if the same bus (either a single bus or the same entry of an array) is assigned in different iterations, it is compulsory to add the POS statement in order to break the combinational loop.

If an array is being referenced using the loop index, a special restriction applies. The expression inside the square brackets must be a const expression (i.e. the expression evaluates to a constant positive integer).

7.3 POS statement

POS is the keyword that allows EHDL to instantiate registers. The statement has the following syntax:

pos-statement → POS(*expr*) ;

The expression in parenthesis is the enable of the register and must be of type *uint(1)*. EHDL will generate a register for each variable which has been assigned in the scope of the *pos-statement*, including the variables involved in a previous *pos-statement* or, if it is the first *POS* of a function, the arguments, assigned by the caller.

After a pos-statement the old identifiers of the variables refer to the output of the register, while the reference to the input is no longer available.

8 References

This manual borrows its style from [1] and [2]:

[1] Ritchie, Dennis. C Reference Manual. Retrieved on October 28, 2011 from http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf

[2] Conway et al. Retrieved on October 28, 2011 from http://www.cs.columbia.edu/~sedwards/classes/2003/w4115/conway-report.pdf.

[3]VHDL reference manual. Retrieved on October 28, 2011 from http://www.usna.edu/EE/ee462/manuals/vhdl_ref.pdf