HL-HDL (High-Level Hardware Description Language)

COMS W4115 PLT

Woon G. Lee (wgl2107)

April 25, 2014

Table of Contents

1. Introduction

There are two major HDLs (Hardware Description Languages), VHDL and Verilog, mostly used to describe the hardware behaviors. They can define signal types, components, functions, and procedures using their own syntax. Then, the program is compiled and synthesized by HDL compilers supported by multiple companies such as Altera, Xilinx, Mentor Graphics and so on. HDL designers should describe all the details using HDLs which are sometimes tedious tasks. When the designer plans hierarchical design, it often starts with top-level block diagram, then each block is filled up by writing HDL codes and that is when the designer can verify the whole design. In other words, the designer has to implement the design with HDL only from top to all the way down to bottom layer. HL-HDL is designed to support the VHDL implementation of the design engineer's concepts easily without direct VHDL coding. It can support basic building blocks such as buffer, latch, multiplexer, divider, state machine using pre-defined keywords. By doing so, the hardware design engineer does not need to set up the VHDL coding structure but simply implement the design block using HL-HDL, then it will be implemented into VHDL source code by HL-HDL compiler. As stated, the output of the HL-HDL compiler would be the VHDL source files which can be applied to the existing VHDL compiler. In this project, the output will be compiled and simulated using Quartus II (Altera's VDHL compilation software) and verified.

2. Tutorial

2.1 Example

The following example program shows that CPU tried to read data from the pre-defined registers by asserting control signals and register address. Overall structure is pretty similar to C-language. First, FUNC read_reg is used for a component declaration which reads data from a register at the address given as a parameter. It is called in main body and implemented as a process in VHDL output file. Register write function can be coded in similar way.

```
FUNC read_reg(signal in_clk, reset, CSN, OEN, bus reg_addr(5:0), read_data(7:0))

/* functional block declaration that read back register value at register address reg_addr(5:0) */

{

   Signal CS, OE;

   CS = INV(CSN); /* invert input signal CSN in order to use active-high input to RDATA */

   OE = INV(OEN); /* invert input signal OEN in order to use active-high input to RDATA */

   if (reset == 1) then read_data = 0x00;   /* load the data at the rising edge of the clock */

   else if (in_clk == 1) then RDATA(CS, OE, reg_addr(5:0), read_data(7:0));

   end;
```

```
}
/* main body */
main(
        signal CLK, CPU_RESET, CPU_RDN, CPU_WRN, CPU_CSN;
        bus CPU_ADDR(5:0), CPU_DATA(7:0)
    )
{
    constant VERSION_REG(7:0) = x10;
    bus GENERAL1(7:0), GENERAL2(7:0);
    signal clk_buf;
    bus addr_reg(5:0), data_reg(7:0);
    clk_buf = BUF(CPU_CLK);    /* buffered input clock */
    /* Define registers */
    REG(x00, VERSION_REG, R);
    REG(x01, GENERAL1, RW);
    REG(x02, GENERAL2, W);

    addr_reg = LAT(clk_buf, CPU_ADDR(5:0)); /* registered CPU address at rising edge of the clock */
    read_reg(clk_buf, CPU_CSN, CPU_RDN, addr_reg(5:0), data_reg(7:0))'
}
```

3. Reference Manual

3.1 Grammar Notation

Regular expression notation is used in this document. 's*' denotes that it has zero or more s's, 's+" that it has at least one s, (s|r) that s or r could be, (s & r) that s and r are concatenated, where parentheses are used to group

the symbols.

3.2 Lexical Conventions

A program is contained in a single file, which has a sequence of tokens to be processed.

3.2.1 Line Terminator

Semi-colon character (;) is used as line terminator.

3.2.2 Comments

C-like comment is supported. Comment begins with characters "/*" and ends with "*/". Any sequence of characters except for "*/" can be used between these two character combinations.

3.2.3 Tokens

There are five kinds of tokens: Identifiers, keywords, constants, expression operators, and separators. White spaces such as spaces, tabs, newlines, and carriage returns are used to delineate tokens.

3.2.4 Identifiers

An identifier is a sequence of alphanumeric characters which must begin with letter and/or can be followed by numbers. Underscore ('_') is considered as a letter and bus identifier must have parenthesized number range at the end. Identifier is not case-sensitive whose length is limited to 16.

letter = [a-zA-Z]

digit = [0-9]

identifier = letter (letter | digit | '_')*

3.2.5 Keywords

The following keywords are reserved for use as keywords and may not be used otherwise.

constant	signal	bus	and
or	inv	buf	lat
mux	reg	rdata	wdata
func	if	then	else
main			

3.2.6 Contants

There are two kinds of constants as follows:

3.2.6.1 Integer constants

An integer constant is a sequence of digits.

Integer constant: (digit)+

3.2.6.2 Binary constants

A binary constant is a sequence of binary digits ‘0’ or ‘1’, which could be single or multiple digits. A prefix ‘b’ is required.

Binary constant: ‘b’ (‘0’|’1’)+

For example, ‘0’ or ‘1’ is a single digit binary constant for a signal type and “0011”, “1010” are 4-digit binary constants for a bus type.

3.2.6.3 Hexadecimal constants

A hexadecimal constants is a sequence of hexadecimal digits, ‘0’ ~ ‘9’, ‘A’ ~ ‘F’. A prefix ‘x’ is required.

Hexadecimal constant: ‘x’ (‘0’ | … | ’9’ | ’A’ | … | ’F’)+

For example, 0x1FC is a valid hexadecimal constant.

3.2.7 Expression Operators

=	:	signal or bus assignment
AND	:	logical AND for signals or buses
OR	:	logical OR for signals or buses
INV	:	invert signal or bus
==	:	logical comparator (equal to) for signals or buses
!=	:	logical comparator (not equal to) for signals or buses

3.2.8 Separators

The following ASC II characters are used as separators:

;	:	line terminator
:	:	bus range specifier
{ }	:	main or user function definition
,	:	argument separator

3.3 Data Types

3.3.1 Primitive data types

The following primitive data types are supported in HL-HDL.

3.3.1.1 Integer

An integer type is used to define integral value to the identifier.

3.3.1.2 Signal

A signal type is used to define binary value to the single-bit identifier.

3.3.1.3 Bus

A bus type is used to define hexadecimal value to the multiple-bit identifier.

3.3.2 Literals

3.3.2.1 Integer Literals

Integer literal is a sequence of digits and defined as follows:

integer = (digit)+

3.3.2.2 Signal Literals

Signal literal is a single binary digit and defined as follows:

signal = (‘0’|’1’)

3.3.2.3 Bus Literals

Bus literal is a sequence of hexadecimal numbers and defined as follows:

hex = [‘0’-‘9’ |’a’-‘f’ ‘A’-‘F’]

bus = ‘x’ (hex)+

3.4 Expressions

Expressions are evaluated in the order as follows. Also, left- or right-associativity is specified in each subsection.

3.4.1 Primary expressions

Primary expressions have left-to-right associativity.

3.4.1.1 Identifier

An identifier is a primary expression whose type must be properly declared.

3.4.1.2 Constant

An integer, binary, or hexadecimal constant is a primary expression. Its type is integer, signal, and bus, respectively.

3.4.1.3 (*expression*)

A parenthesized expression is a primary expression and its type and value are identical to those of "expression".

3.4.2 *expression*$_1$ (*expression*$_2$)

Expression followed by an expression is a part of whole bus signals, where expression$_1$ is bus identifier and expression$_2$ is in the range of the bus space.

3.4.3 Logical Expressions

The following are the logical expressions and the first two have left-to-right and the last does right-to-left associativity.

expression$_1$ AND *expression*$_2$	:	logical AND
expression$_1$ OR *expression*$_2$	:	logical OR
INV (expression)	:	logical NOT

3.4.4 Numerical Expressions

The following numerical expressions have left-to-right associativity.

expression$_1$ == *expression*$_2$	:	equal to
expression$_1$!= *expression*$_2$	:	not equal to

3.4.5 Assignment Expressions

An assignment expression has right-to-left associativity.

expression$_1$ = *expression*$_2$

3.4.6 Operator precedence

Precedence	**Operator type**	**Operators**	**Associativity**
1	primary	( )	Left-to-right
2	unary	NOT	Right-to-left
3	binary	AND, OR	Left-to-right
4	numerical	==, !=	Left-to-right

5	assignment	=	Right-to-left

3.5 Statements

Statements are executed in sequence.

3.5.1 Expression statement

Most expression statements have the form as follows:

expression ;

3.5.2 IF statement

If statement is a conditional statement which has two forms as follows:

if (*expression*) then *statement* ;

if (*expression*) then *statement* else *statement* ;

In both cases, expression is evaluated first and if it is true, the first statement is executed. However, in the second case, if it is not true, the second statement is executed.

3.6 Functions

There are a couple of generic functions are provided in HL-HDL, which are the most generic functional block used in VHDL. Otherwise, user-defined function can be declared and called. In this case, user-defined function should be declared before main procedure. In regard to mapping to VHDL implementation, it is mapped to “component” declaration.

3.6.1 Generic functions

The following functions are provided in HL-HDL.

Function_identifier (*parameter list*)

3.6.1.1 BUF function

BUF (*expression)*: expression is either signal or bus and a buffer is added, whose output is the same type as input expression.

e.g. A = BUF(B);

3.6.1.2 LAT function

LAT (*clock_identifier, expression):* clock_identifier is a clock source for the latch and either signal or bus is

latched and output.

e.g. A = LAT(clk, B);

3.6.1.3 MUX function

MUX (*expression CTRL, list of expressions*): One of input expression is output based on CTRL value.

e.g. A = MUX (000, B, C, D, E, F, G, H, I);

3.6.1.4 REG function

REG (*expression_REG_ADDRESS, expression_DATA, expression_RW*): generates a register named DATA with property (0 = RO, 1=WO, 2 =RW) and its data width at REG_ADDRESS. New registers are saved in "constants_registers.vhd " file.

e.g. REG (x100, STATUS, 0);

3.6.1.5 RDATA function

RDATA (*CHIP_SEL_identifier, OUT_EN_identifier, expression_REG_ADDRESS, expression_READ_DATA*): reads REGISTER DATA at address REG_ADDRESS when CHIP_SEL and OUT_EN are logic '1' and saves the DATA on READ_DATA.

e.g. RDATA(CS, OE, x100, rdata);

3.6.1.6 WDATA function

WDATA (*CHIP_SEL_identifier, WR_EN_identifier, expression_REG_ADDRESS, expression_READ_DATA)*: writes READ_DATA to REGISTER_DATA at address REG_ADDRESS when CHIP_SEL and WR_EN are logic '1'.

e.g. WDATA(CS, WR, x100, DATA);

3.6.2 User-defined function

User-defined function can be declared with a keyword FUNC followed by list of parameters in parentheses and statements in { }.

FUNC *identifier* (*expressions, … , expressions*)

{

Statements

}

This corresponds a COMPONENT declaration in VHDL and must be declared prior to main procedure. It can

be instantiated by calling in main procedure.

3.7 Program Structure

The program structure is specified as follows:

FUNC *identifier* (*expression, … , expression*)

{

Statements;

}

main (*expression, … , expression*)

{

Statements;

}

Any user-defined functions must be declared prior to main procedure as shown above. The output of the compiler is one or two output files in VHDL format. If there is any register generation function calls, those registers are saved in constant_register.vhd file and VHDL code for main procedure in main.vhd file.