AHDL Introduction

ALTERA Devices, Tools and Techniques

Contents

Using AHDL

- AHDL is an easy-to-use design entry method
  - text design entry
  - simple language
- AHDL is powerful and compact
  - can describe both simple and complex circuits efficiently
- AHDL is fully integrated into MAX+plusII
  - use AHDL subdesigns in the design hierarchy

AHDL File Structure

- AHDL files are ASCII text files
  - created and edited using built-in text editor (or any text editor)
- Major elements of an AHDL file
  - Title Statement
  - Include Statement
  - Constant Statement
  - Subdesign Section
    - inputs and outputs of the subdesign
  - Variable Section
    - declaration of instances, nodes, registers, state machines
  - Logic Section
    - Boolean equations, conditional logic (if and case), macrofunctions, truth-tables

Subdesign Section

- specifies the interface between this subdesign and other modules in the design
- INPUTs and OUTPUTs are named
  - names have scope limited to a subdesign
- groups can be referred to using the syntax :
  - E.g. x[3..0] represents x3, x2, x1, x0
- examples:

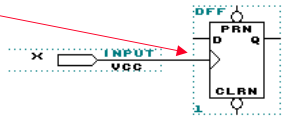
```
SUBDESIGN divide2
(
    x           : INPUT;
    div2        : OUTPUT;
)
```

```
SUBDESIGN top
(
    clk1, clk2 : INPUT;
    a0, a1, a2 : OUTPUT;
    b[7..0]    : BIDIR;
)
```


Variables Section

- declares the variables used in the Logic Section
- types of variables
  - instances
    - declares an instance of a primitive or macrofunction
      - buffers, flip-flops, I/Os and gates
      - use . (dot) syntax to refer to a port of a primitive in Logic Section
  - nodes
    - declares an intermediate node (not an input or output)

```
VARIABLE
    ff  : DFF
BEGIN
    ff.clk = x;
END
```



This syntax describes a “group” :-
key[3..0] means key3, key2, key1, key0.

```
VARIABLE
    key[3..0]   : NODE;
```


Variables Section - State Machines

- state machines are created in the Variables Section
- name of the machine
- keywords
  - MACHINE
    - specifies a state machine
  - OF BITS
    - optional
    - names the output bits of the machine (register outputs)
  - WITH STATES
    - comma separated list of states
    - optional numerical value of the output bits for each state
    - first state in list is reset state

```
VARIABLE
    row_seq: MACHINE
             OF BITS (row[4..1])
             WITH STATES (
                r0 = 0,
                r1 = 1,
                r2 = 2,
                r3 = 4,
                r4 = 8);
```


Logic Section - Boolean Equations

- Logic section follows the Variables Section
- enclosed by BEGIN and END
- main operators
  - ! NOT
  - & AND
  - # OR
  - $ XOR
  - !&, !#, !$ NAND, NOR, XNOR
  - - negate
- Boolean expressions can involve groups
- constants can be used
  - B"1100" is a binary constant

```
BEGIN
    s = a $ b;
    c = a & b;
    p = ((a & b) # c # (!a & !b &!c));
END;
```

```
a[3..0] = b[3..0] & c[3..0];

means :   a3 = b3 & c3, a2 = b2 & c2,
          a1 = b1 & c1, a0 = b0 & c0


x[4..1] = y[2..1];

means :   x4 = y2, x3 = y1,
          x2 = y2, x1 = y1
```

Logic Section - IF and CASE

- IF statement
  - used for conditional logic based on comparisons
    - ==, !=, >, <, >=, <=
  - IF/ELSEIF can be expensive in resources - use CASE instead where appropriate
- CASE statement
  - more efficient for long ELSEIF constructions
  - a structure for implementing many comparisons on just one variable
    - E.g. a 2-bit number `code[]` is tested. The output `out[]` is Gray coded

```
IF a[3..0] == b{3..0]
    equal = VCC;
    smaller = GND;
ELSEIF a[3..0] < b[3..0]
    equal = GND;
    smaller = VCC;
ELSE
    equal = GND;
    smaller = GND;
END IF;
```

```
CASE code[] IS
    WHEN 0 => out[] = B"00";
    WHEN 1 => out[] = B"01";
    WHEN 2 => out[] = B"11";
    WHEN 3 => out[] = B"10";
END CASE;
```


Logic Section - Truth Tables

- combinational logic can be entered as a truth table
  - keywords:
    - TABLE and END TABLE
  - inputs on left of =>
  - outputs on right of =>
- legal entries in a truth table
  - nodes
  - groups
  - VCC and GND
  - X (don't care)
  - constants

```
TABLE
    a,b  => sum,carry,gray[1..0];

    0,0  =>  0,   0,   B"00";
    0,1  =>  1,   0,   B"01";
    1,0  =>  1,   0,   B"01";
    1,1  =>  0,   1,   B"11";
END TABLE;
```


Examples

- Combinational

```
% Half adder to add %
% together a and b  %
% giving sum s and  %
% carry c.          %

TITLE "Full Adder"
SUBDESIGN myadder
(
    a,b           : INPUT;
    s,c           : OUTPUT;
)

BEGIN
    s = a $ b;
    c = a & b;
END;
```

- Sequential

```
% Frequency division by  %
% 2. Note the use of the %
% GLOBAL clock.          %

TITLE "Divide by 2"
SUBDESIGN divide2
(
    x             : INPUT;
    div2          : OUPUT;
)

VARIABLE
    ff            : DFF;

BEGIN
    ff.clk = GLOBAL(x);
    ff.d   = !ff.q;
    div2   = ff.q;
END;
```


Examples

- State Machine

```
TITLE "Keyboard Scanning State Machine";
SUBDESIGN kbd_scan
(
    row_en, clock38, key[1..0] : INPUT;
    row[4..1]                  : OUTPUT;
)
VARIABLE
    row_seq: MACHINE
             OF BITS (row[4..1])
             WITH STATES (
                    r0 = 0,
                    r1 = 1,
                    r2 = 2,
                    r3 = 4,
                    r4 = 8);
```

(contd ...)

```
BEGIN
    row_seq.clk = clock38;
    row_seq.ena = row_en;
    CASE row_seq IS
        WHEN r1 =>
            IF key[] == 0 THEN row_seq = r2;
            ELSE    row_seq = r4;
            END IF;
        WHEN r2 =>
            IF key[] == 0 THEN row_seq = r3;
            ELSE    row_seq = r1;
            END IF;
        WHEN r3 =>
            IF key[] == 0 THEN row_seq = r4;
            ELSE    row_seq = r2;
            END IF;
        WHEN r4 =>
            IF key[] == 0 THEN row_seq = r1;
            ELSE    row_seq = r3;
            END IF;
       WHEN OTHERS => row_seq = r1;
    END CASE;

END;
```