EE 231L

Using AHDL to Design Combinational Circuits

Altera Hardware Description Language (AHDL) is a language developed by Altera Corporation for programming their programmable logic devices (PLDs). There are other HDLs which can be used to program PLDs (VHDL and Verilog HDL are two industry standards), but these other, more general, languages are harder to learn and use. In this lab you will learn how to design digital logic circuits using AHDL.

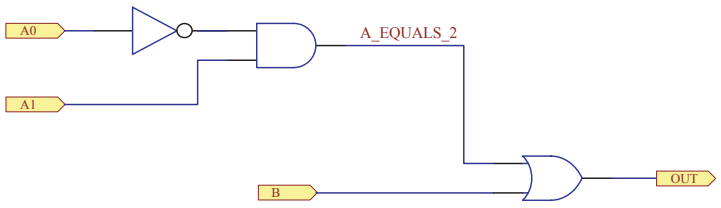
With AHDL, there are two methods for entering designs – a text design file (.tdf) or a graphical design file (.gdf). Using the graphical design method you enter circuits as you would draw them on a schematic. This method is quite easy to learn and use for simple circuits, but becomes very cumbersome for large designs. The text design method is flexible, and much easier to use for large designs. We will show several examples using the graphical design method, but will concentrate mainly on the text design method.

This lab deals with designing combinational circuits. This brief tutorial will give you the information you need to design combinational circuits using the text design method with AHDL.

Here is a sample text design file which illustrates many features of combinational AHDL, boolean.tdf (taken from "MAX+PLUS II AHDL" manual from Altera):

```
SUBDESIGN boolean
(
    a0, a1, b       : INPUT;
    out             : OUTPUT;
)
VARIABLE
    a_equals_2  :   NODE;
BEGIN
    a_equals_2 = a1 & !a0;
    out = a_equals_2 # b;
END;
```

This implements the following logic `OUT = ((A1 AND NOT A0) OR B)`:



Key things to note about this:

- A TDF file must have a line `SUBDESIGN`. The subdesign name must be the same as the file name.
- There is a section which describes the inputs and outputs.
- There is an optional variable section which describes variables (other than inputs and outputs) used in the code which follows. In this example, the variable `a_equals_2` is declared to be a `NODE`, which is used to store the value of an intermediate expression, and requires no extra logic.
- The code is surrounded by a `BEGIN` and an `END`.
- Each line of code ends with a semicolon.
- Outputs must be on the left hand side of an equal sign.
- Inputs must be on the right hand side of an equal sign.
- Nodes can be on either side of an equal sign.
- Boolean expressions are written using ands (`&` or `AND`), ors (`#` or `OR`), exclusive ors (`$` or XOR). nots (`!` or `NOT`), nands (`!&` or NAND), nors (`!#` or NOR), and exclusive nor (`!$` or XNOR).

Here is an example of a more TDF file which implements Boolean logic using a truth table (also taken from the "MAX+PLUS II AHDL" manual):

```
TITLE "EE 231 Example TDF File";
SUBDESIGN example1
(
    i[3..0]             : INPUT;
    ascii_code[7..0]    : OUTPUT;
)
BEGIN
    DEFAULTS
        ascii_code[7..0] = B"00111111";   % ASCII question mark %
    END DEFAULTS;

    TABLE
        i[3..0]         =>  ascii_code[];
        B"1000"     =>  B"01100001";  % "a" %
        B"0100"     =>  B"01100010";  % "b" %
        B"0010"     =>  B"01100011";  % "c" %
        B"0001"     =>  B"01100100";  % "d" %
    END TABLE;
END;
```

This shows several features of AHDL:

1. An optional `TITLE` statement can be used. This puts the specified title on the output files generated by the Altera compiler.
2. Multiple inputs and outputs can be grouped together into a group. One can refer to a subset of the group be using brackets – `ascii_code[1]` or `ascii_code[1..0]`. All elements of the array can be referred to as `ascii_code[]` or `ascii_code[7..0]`.
3. A `DEFAULTS` section indicates default values to use for a variable. For this program, the inputs `i[]` can have one of 16 values. The program specifies the output `ascii_code[]` for 4 of the possible 16 values. The `DEFAULTS` section tells the Altera compiler that, if the inputs are in one of the other possible states, the output should be an ASCII question mark.
4. Comments are entered by surrounding them with percent (`%`) signs.
5. A binary number can be entered with the following notation: `B"1100"`. Other ways to enter this number are: decimal (`12`), octal (`O"14"` or `Q"14"`) and hexadecimal (`X"C"` or `H"C"`).

A third way to enter a Boolean expression is with a `CASE` statement. Here is code to implement the above logic with a `CASE` statement:

```
TITLE "EE 231 Example TDF File";
SUBDESIGN example2
(
    i[3..0]             : INPUT;
    ascii_code[7..0]    : OUTPUT;
)
BEGIN
    CASE i[] is
        WHEN B"1000" =>
            ascii_code[] = B"01100001";  % "a" %
        WHEN B"0100" =>
            ascii_code[] = B"01100010";  % "b" %
        WHEN B"0010" =>
            ascii_code[] = B"01100011";  % "c" %
        WHEN B"0001" =>
            ascii_code[] = B"01100100";  % "d" %
        WHEN OTHERS =>
            ascii_code[] = B"00111111";   % ASCII question mark %
    END CASE;
END;
```

- The `CASE` statement is similar to the `switch` statement in C.
- Quartus II will use less resources if you use a `DEFAULTS` block rather than a `WHEN OTHERS` block to specify the default values.

A fourth way to enter a Boolean expression is with an `IF THEN ELSE` syntax. Here is code to implement the above logic with `IF THEN ELSE` syntax:

```
TITLE "EE 231 Example TDF File";
SUBDESIGN example3
(
    i[3..0]             : INPUT;
    ascii_code[7..0]    : OUTPUT;
)
BEGIN
    IF (i[] == B"1000") THEN
        ascii_code[] = B"01100001";
    ELSIF (i[] == B"0100") THEN
            ascii_code[] = B"01100010";
    ELSIF (i[] == B"0010") THEN
           ascii_code[] = B"01100011";
    ELSIF (i[] == B"0001") THEN
            ascii_code[] = B"01100100";
    ELSE ascii_code[] = B"00111111";   % ASCII question mark %
    END IF;

END;
```

- As in C, `==` is used for comparison — if the left-hand and right-hand side of the expression equal each other, the result is `TRUE`, and the `THEN` statement will be implemented.
- Using the `ELSIF` clause may cause Altera to generate unnecessarily complex logic. This is because the logic not only has to check to see if the current `ELSIF` statement is true, but also has to verifty that the `IF` and all preceeding `ELSIF` statements are false. It is usually better to avoid using the `ELSIF` clause if possible. Use the `TABLE` or `CASE` statements instead.
- Quartus II will use less resources if you use a `DEFAULTS` block rather than a final `ELSE` block to specify the default values.