MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY

****

**REPORT ON: Analog Hardware Description Language (AHDL)**

**COURSE CODE: CSE-315**

**COURSE TITLE: DIGITAL SYSTEM DESIGN SESSIONAL**

**SUBMITTED TO: SUBMITTED BY:**

**1.**[**Sharifa Rania Mahmud**](http://cse.mist.ac.bd/employees/sharifa-rania-mahmud/) **GROUP-4**

Lecturer, CSE DEPT 1. Jannatul Maowa Shaila 201514013

**2.Anika Binte Islam** 2. Fatema Binte Alam 201514029

Lecturer, CSE DEPT 3. Maria Afnan 201514030

4. Mohoshina Akter Toma 201514032

5. Nuzhat Tabassum Promi 201514039

6. Syeda Nusraht Khaledur 201514046

1. **INTRODUCTION:**

In [electronics](https://en.wikipedia.org/wiki/Electronics), a **hardware description language** (**HDL**) is a specialized [computer language](https://en.wikipedia.org/wiki/Computer_language) used to describe the structure and behavior of [electronic circuits](https://en.wikipedia.org/wiki/Electronic_circuit), and most commonly, [digital logic](https://en.wikipedia.org/wiki/Digital_logic) circuits.

A hardware description language enables a precise, [formal](https://en.wikipedia.org/wiki/Formal_language) description of an electronic circuit that allows for the automated analysis and [simulation](https://en.wikipedia.org/wiki/Electronic_circuit_simulation) of an electronic circuit. It also allows for the [synthesis](https://en.wikipedia.org/wiki/Logic_synthesis) of a HDL description into a [netlist](https://en.wikipedia.org/wiki/Netlist) (a specification of physical electronic components and how they are connected together), which can then be [placed and routed](https://en.wikipedia.org/wiki/Place_and_route) to produce the [set of masks](https://en.wikipedia.org/wiki/Mask_set) used to create an [integrated circuit](https://en.wikipedia.org/wiki/Integrated_circuit).

A hardware description language looks much like a [programming language](https://en.wikipedia.org/wiki/Programming_language) such as [C](https://en.wikipedia.org/wiki/C_(programming_language)); it is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time.

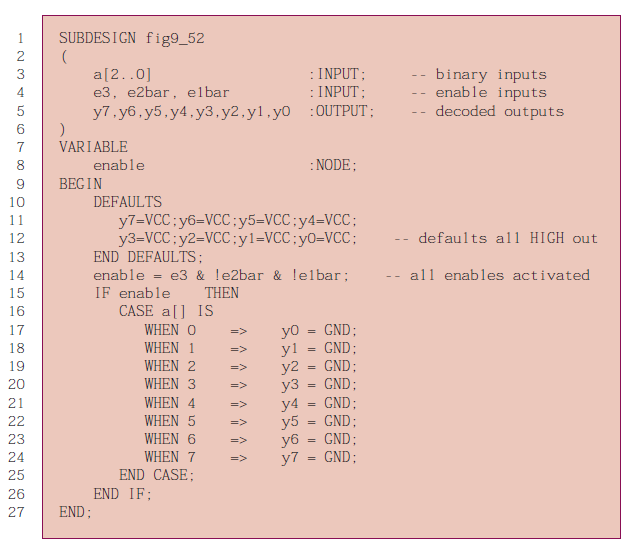
HDLs form an integral part of [electronic design automation](https://en.wikipedia.org/wiki/Electronic_design_automation) (EDA) systems, especially for complex circuits, such as [application-specific integrated circuits](https://en.wikipedia.org/wiki/Application-specific_integrated_circuit), [microprocessors](https://en.wikipedia.org/wiki/Microprocessor), and [programmable logic devices](https://en.wikipedia.org/wiki/Programmable_logic_device).

1. **AHDL:**

A hardware AHDL stands for Analog Hardware Description Language. The HDLs used for analog circuits include Analog Hardware Description Language (AHDL). AHDL is most commonly used as a HDL language for analog circuits.

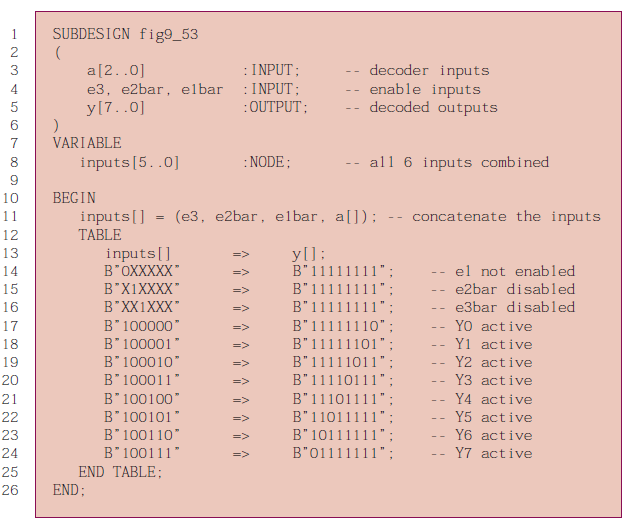
1. **AHDL DECODERS:**

The first illustration of an AHDL decoder, shown in Figure 1, is intended to demonstrate the use of a CASE construct that is evaluated only under the condition that all enables are active. The outputs must all revert back to HIGH as soon as any enable is deactivated. This example also illustrates a way to accomplish this without explicitly assigning a value to each output for each case, and it uses individually named output bits. Line 3 defines the three-bit binary number that will be decoded. Line 4 defines the three enable inputs, and line 5 specifically names each output. The unique property of this solution is the use of the **DEFAULTS** keyword in AHDL (lines 10 to 13) to establish a value for variables that are not specified elsewhere in the code. This maneuver allows each case to force one bit LOW without specifically stating that the others must go HIGH.

**

**FIGURE 1:** AHDL equivalent to the 74138 decoder.

The next illustration, in Figure 2, is intended to demonstrate the same decoder using the truth table approach. Notice that the outputs are defined as bit arrays but are still numbered *y[7]* down to *y[0].* The unique aspect of this code is the use of the don’t-care values in the truth table. Line 11 is used to concatenate the six input bits into a single variable (bit array) named *inputs[ ]*.Notice that in lines 14, 15, and 16 of the table, only one bit value is specified as 1 or 0.The others are all in the don’t-care state (X). Line 14 says, “As long as *e3* is *not* enabled, it does not matter what the other inputs are doing; the outputs will be HIGH.” Lines 15 and 16 do the same thing, making sure that if *e2bar* or *e1bar* is HIGH (disabled), the outputs will be HIGH. Lines 17 through 24 state that as long as the first three bits (enables) are “100,” the proper decoder output will be activated to correspond with the lower three bits of *inputs[ ].*

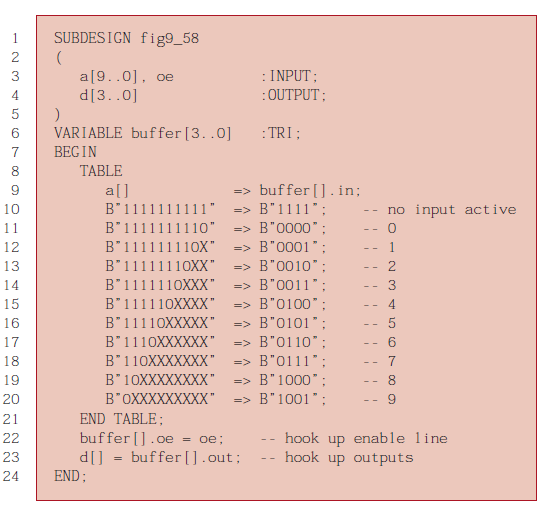


**FIGURE 2:** AHDL decoder using a TABLE.

1. **AHDL ENCODER:**

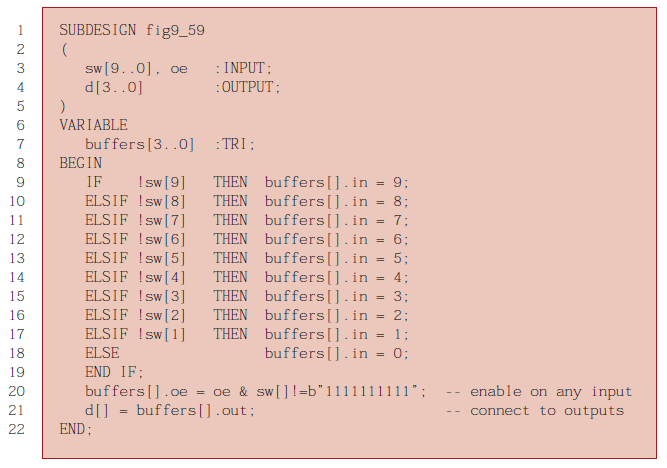
The most important point to be made from Figure 3 is the method of establishing priority, but also note the I/O assignments. The AHDL input/output descriptions do not provide a separate type for integers but allow a bit array to be referred to as an integer. Consequently, line 4 describes the outputs as a bit array. In this illustration, a TABLE is used that is very similar to the tables often found in data books describing this circuit’s operation. The key to this table is the use of the don’t-care state (X) on inputs. The priority is described by the way we position these don’t-care states in the truth table.

Reading line 15, for instance, we see that as soon as we encounter an active input (LOW on input *a[4]*), the lower order input bits do not matter. The output has been determined to be 4. The tri-state outputs are made possible by using the built-in primitive function: TRI on line 6. This line assigns the attributes of a tri-state buffer to the variable that has been named *buffer.* Recall that this is the same way a flip-flop is described in AHDL. The ports of a tri-state buffer are quite straightforward. They represent the input (*in*), the output (*out*), and the tri-state output enable (*oe*).



**FIGURE 3:** AHDL priority encoder with tri-state outputs

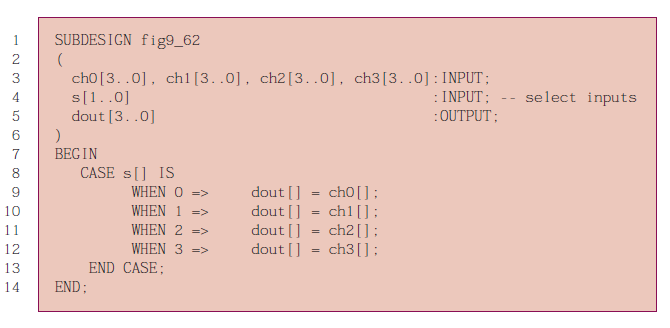
The next illustration (Figure 4) uses the IF/ELSE construct to establish priority, very much like the method demonstrated in the 7-segment decoder example. The first IF condition that evaluates TRUE will THEN cause the corresponding value to be applied to the tri--state buffer inputs. The priority is established by the order in which we list the IF conditions. Notice that they start with input 9, the highest-order input. This illustration adds another feature of putting the outputs into the high-impedance state when no input is being activated. Line 20 shows that the output enables will be activated only when the *oe* pin is activated and one of the inputs is activated. Another item of interest in this illustration is the use of bit array notation to describe individual inputs. For example, line 9 states that IF switch input 9 is activated (LOW), THEN the inputs to the tri-state buffer will be assigned the value 9 (in binary, of course).



**FIGURE 4:** AHDL priority encoder using IF/ELSE.

1. **AHDL MUX :**

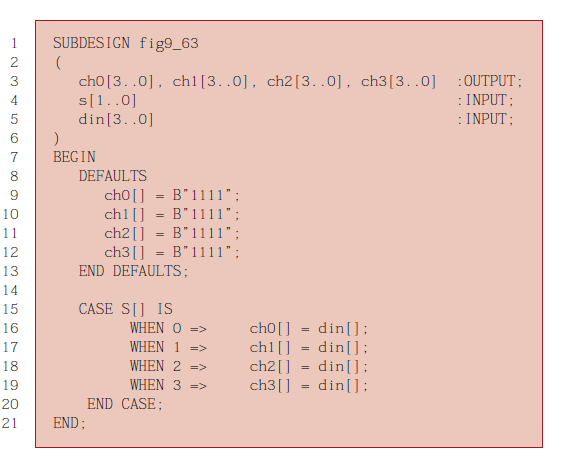
We will implement the multiplexer first. Figure 5 describes a multiplexer with four inputs of four bits each. Each input channel is named in a way that identifies its channel number. In this figure, each input is described as a four-bit array. The select input (*s[ ])* requires two bits to specify the four channel numbers (0–3). A CASE construct is used here to assign an input channel conditionally to the output pins. Line 9, for example, states that in the case when the select inputs (*s[ ])* are set to 0 (that is, binary 00), the circuit should connect the channel 0 input to the data output. Notice that when assigning connections, the destination (output) of the signal is on the left of the = sign and the source (input) is on the right.



**FIGURE 5:** Four-bit x four channelMUXin AHDL

1. **AHDL DEMUX:**

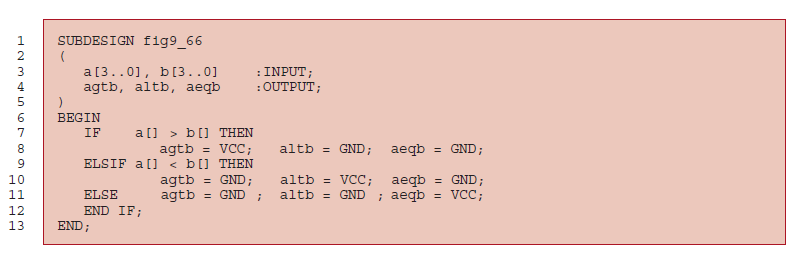
The demultiplexer code works in a similar way but has only one input channel and four output channels. It must also ensure that the outputs are all HIGH when they are not selected. In Figure 6, the inputs and outputs are declared as usual on lines 3–5.The default condition for each channel is specified after the keyword DEFAULTS, which tells the compiler to generate a circuit that will have HIGHs on the outputs unless specifically assigned a value elsewhere in the code. If this default section is not specified, the output values would default automatically to all LOW. Notice on lines 16–19 that the input signal is assigned conditionally to one of the output channels. Consequently, the output channel is on the left of the = sign and the input signal is on the right.



**FIGURE 6:** Four-bit x Four channel DEMUX in AHDL.

1. **AHDL COMPARATOR:**

The AHDL code in Figure 7 follows the algorithm we have described using IF/ELSE constructs. Notice in line 3 that the data values are declared as four-bit numbers. Also note in lines 8, 10, and 11 that several statements can be used to specify the circuit’s operation when the IF clause is true. Each statement is used to set the level on one of the outputs. These three statements are considered concurrent, and the order in which they are listed makes no difference. For example, in line 8, when *A* is greater than *B*, the *agtb* output will go HIGH at the same time the other two outputs (*altb, aeqb*) go LOW.



**FIGURE 7:** Magnitude comparator in AHDL

1. **AHDL BCD-TO-BINARY CODE CONVERTER:**

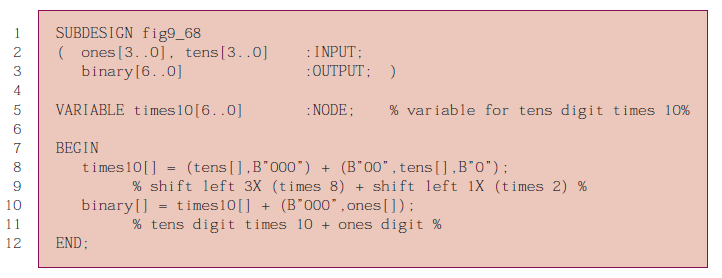
The key to this strategy is in being able to multiply by 10. AHDL does not offer a multiplication operator, so in order to use this overall strategy, we need some math tricks. We will use the shifting of bits to perform multiplication and then employ the distributive property from algebra to multiply by 10. In the same way that we can shift a decimal number left by one digit, thus multiplying it by 10, we can likewise shift a binary number one place to the left and multiply it by 2. Shifting two places multiplies a binary number by 4, and shifting three places multiplies by 8.The distributive property tells us that:

num \* 10 = num \* (8 + 2) = (num \* 8) + (num \* 2)

If we can take the BCD tens digit and shift it left three bit positions (i.e., multiply it by 8), then take the same number and shift it left one place (i.e., multiply it by 2), and then add them together, the result will be the same as multiplying the BCD digit by 10. This value is then added to the BCD ones digit to produce the binary equivalent of the two-digit BCD input. The next challenge is to shift the BCD digit left using AHDL. Because AHDL allows us to make up sets of variables, we can shift the bits by appending zeros to the right end of the array. For example, if we have the number 5 in BCD (0101) and we want to shift it three places, we can concatenate the number 0101 with the number 000 in a set, as follows:

(B“0101”, B“000”) = B“0101000”

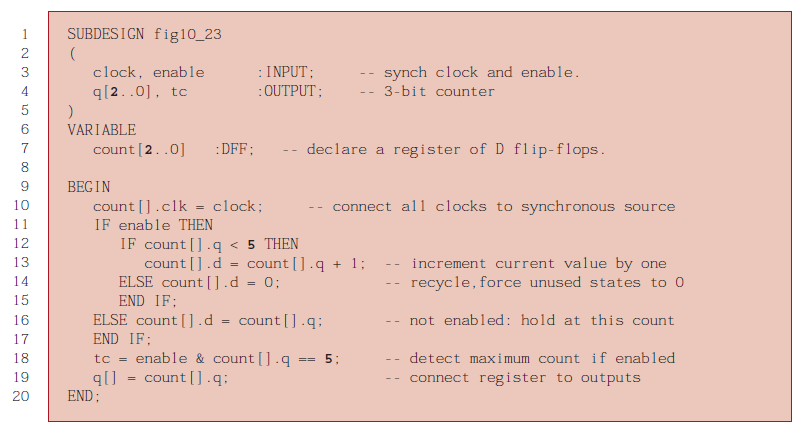
The AHDL code in Figure 8 begins by declaring inputs for the BCD ones and tens digits. The binary output must be able to represent 9910, which requires seven bits. We also need a variable to hold the product of the BCD digit multiplied by 10. Line 5 declares this variable as a seven-bit number. Line 8 performs the shifting of the *tens[ ]* array three times and adds it to the *tens[ ]* array shifted one place to the left. Notice that this latter set must have seven bits in order to be added to the first set, thus the need to concatenate B“00” on the left end. Finally, in line 10, the result from line 8 is added to the BCD ones digit with leading zero extensions (to make seven bits) to form the binary output.



**FIGURE 8:** BCD-to-binary code converter in AHDL

1. **AHDL MOD-6 COUNTER:**

The only additional features that this design needs are the count *enable* input and terminal count (*tc*) output shown in Figure 9. Notice that the extra input (*enable,* line 3) and output (*tc*, line 4) are included in the I/O definition. A new line (line 11) in the architecture description tests *enable* before deciding how to update the value of *count* (lines 12–15). If *enable* is LOW, the same value is held on *count* at every clock edge by the ELSE branch (line 16). Remember always to match an IF with an END IF, as we did on lines 15 and 17.Terminal count (*tc*, line 18) will be HIGH when it is *true* that *count* = = 5 AND *enable* is active. Notice the use of double equal signs (= =) to evaluate equality in AHDL.



**FIGURE 9:** The MOD-6 design in AHDL.

1. **AHDL MOD-10 COUNTER:**

The MOD-10 counter varies only slightly from the MOD-6 counter that

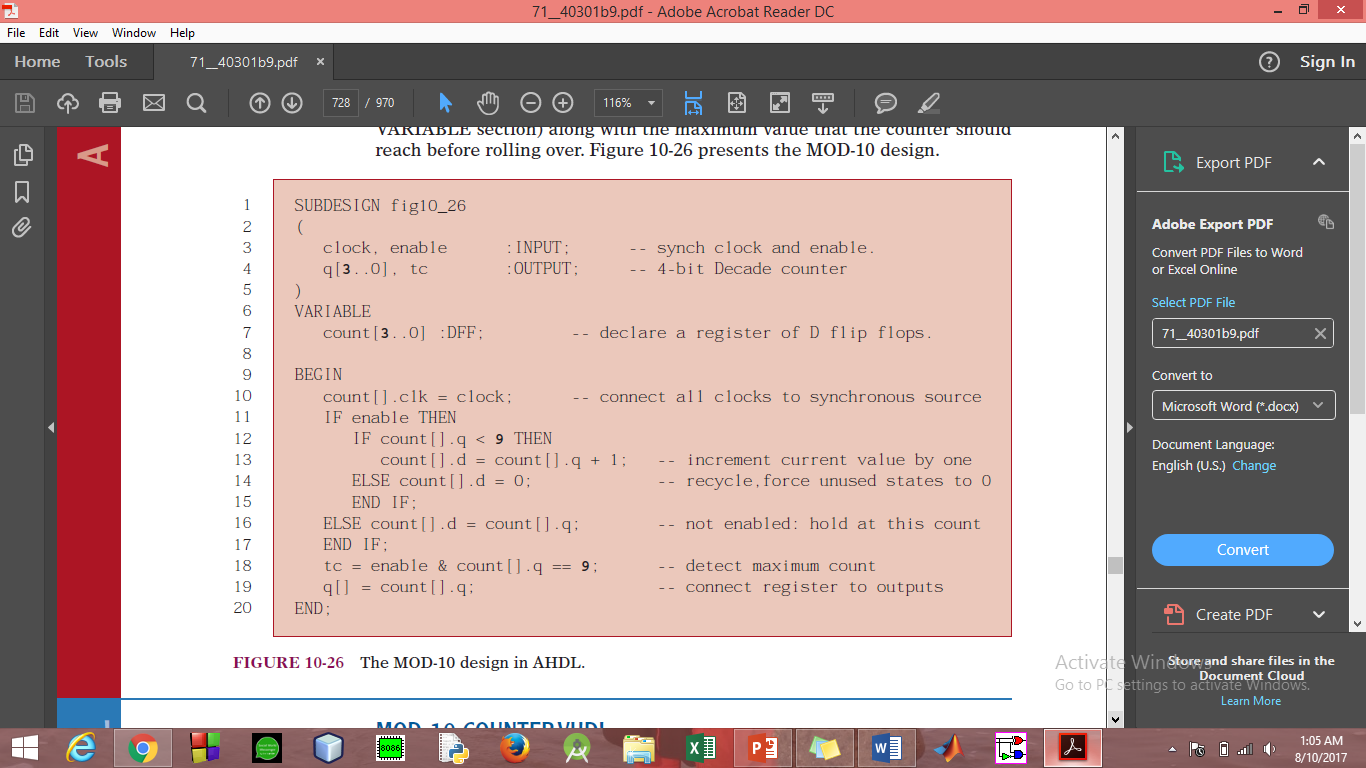
was described in Figure 10-23. The only changes that are necessary

involve changing the number of bits in the output port and the register

(in the VARIABLE section) along with the maximum value that the

counter should reach before rolling over. Figure 10 presents the

MOD-10 design.



**FIGURE 10:** The MOD-10 design in AHDL

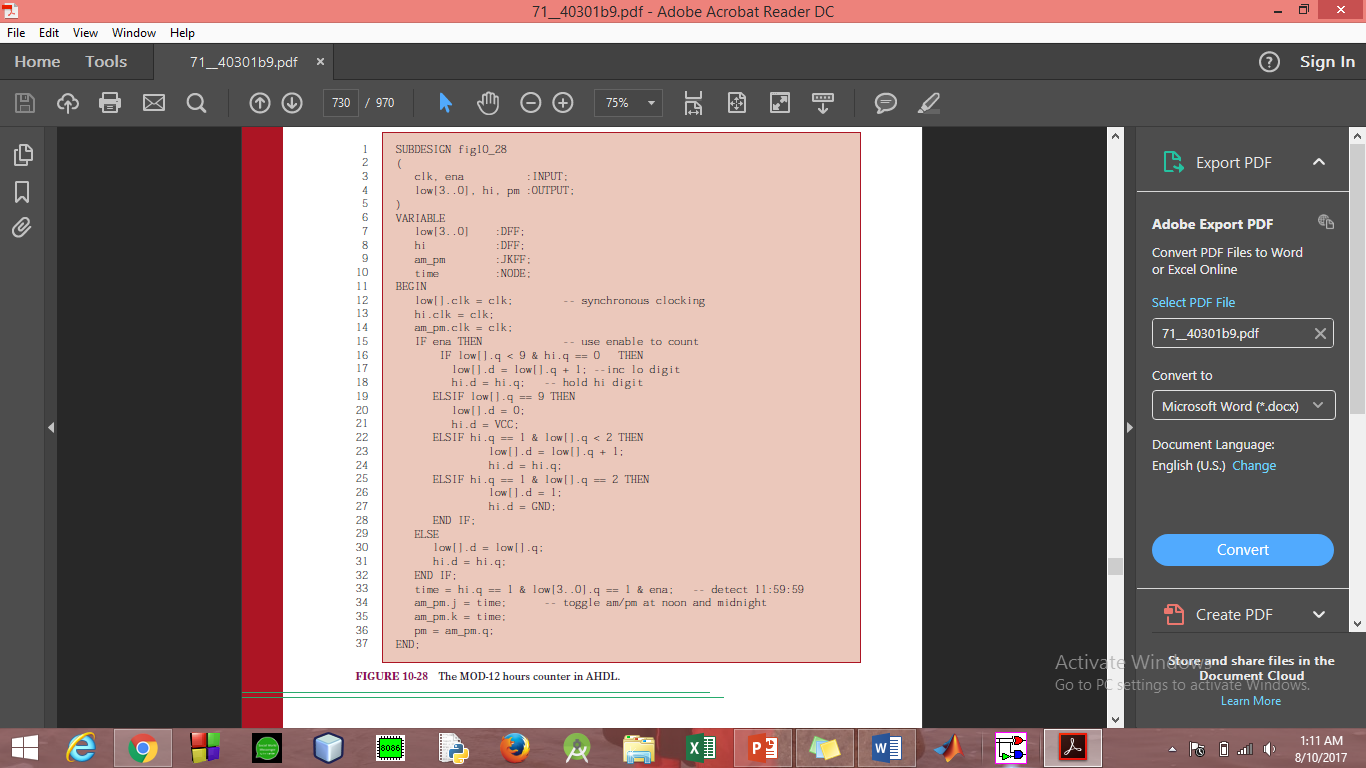
1. **AHDL MOD-12 COUNTER:**

The AHDL counter needs a bank of four D flip-flops for the low-order BCD

digit and only a single D flip-flop for the high-order BCD digit because its

value will always be 0 or 1. A flip-flop is also needed to keep track of A.M. and P.M. These primitives are declared on lines 7–9 of Figure 11. Also note that in this design, the same names are used for the output ports. This is a convenient feature of AHDL. When the enable input (*ena)* is active, the circuit evaluates the IF/ELSE statements of lines 16–28 and performs the proper

operation on the high and low nibble of the BCD number. Whenever the enable input is LOW, the value remains the same, as shown on lines 30 and 31. Line 33 detects when the count reaches 11 while the counter is enabled. This signal is applied to the *J* and *K* inputs of the am pm flip-flop to cause it to toggle at 11:59:59.



**FIGURE 11:** The MOD-12 design in AHDL

1. **AHDL MODULE INTEGRATION:**

Let’s go back to the two AHDL files for the MOD-6 and MOD-10 counters.

we combine these files into a MOD-60 counter using only text-based

AHDL. The method is really very similar to that of graphic integration.

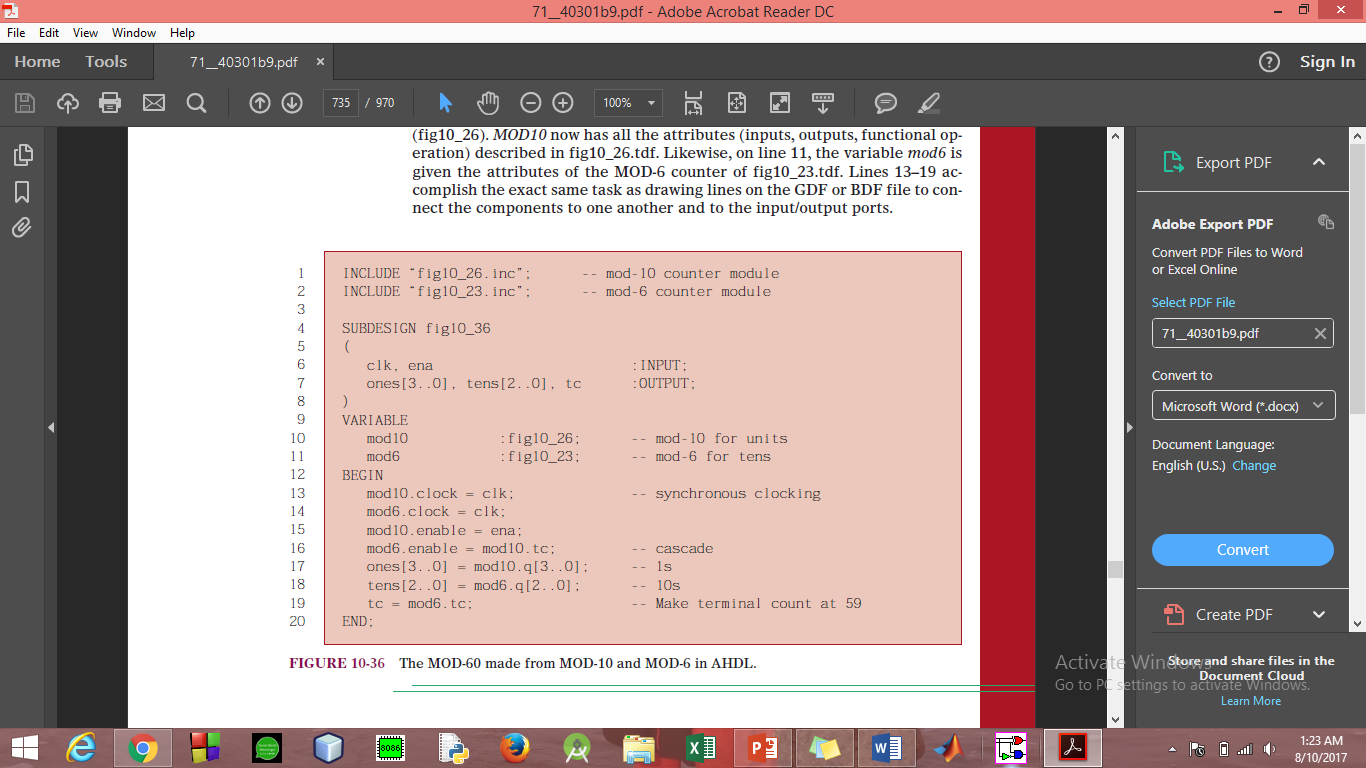
Instead of creating a “symbol” representation of the MOD-6 and MOD-10

files, a new kind of file called an “INCLUDE” file is created. It contains all

the important information about the AHDL file it represents. To describe a

MOD-60 counter, a new TDF file, shown in Figure 12, is opened. The building block files are “included” at the top, as shown on lines 1 and 2. Next, the names that were used for the building blocks are used like library components or primitives to define the nature of a variable. On line 10, the variable *mod10* is now used to represent the MOD-10 counter in the other module.*MOD10* now has all the attributes (inputs, outputs, functional operation) described already.

Likewise, on line 11, the variable *mod6* is given the attributes of the MOD-6 counter of Lines 13–19 accomplish the exact same task as drawing lines on the GDF or BDF file to connect the components to one another and to the input/output ports. This file can be translated into an “include” file by the compiler and then used in another tdf file that describes the interconnection of major sections to make up the system. Each level of the hierarchy refers back to the constituent modules of the lower levels.



**FIGURE 12:** The MOD-60 made from MOD-10 and MOD-6 in AHDL.

1. **CONCLUSION:**

• **Analog hardware design languages are staged languages**. A hardware description is a staged program. The first stage (the meta-level stage) constructs the hardware description algorithmically. The second stage (the object-level stage) interprets the hardware description to give it meaning.

**• Analog hardware descriptions are data structures**. A description is an intentional structure that can be observed, taken apart, compared for equality, etc. Description data-structures are designed to be given multiple interpretations. For example, the same description could be interpreted as a simulation or a net-list.

•**Separate meta-level recursion from object-level feedback**. Constructing large circuits with repeated patterns from smaller ones is described by writing recursive meta-level functions. Feedback loops in sequential circuits are described by recursive equations at the object-level. These two kinds of recursion should be separated. The first is implemented by iteration in the first stage, and the second has much in common with object-language descriptions for languages with variable-binding occurrences.

1. **References**:
2. Digital Systems-Principles and Applications by Tocci, Widmer & Moss.
3. <https://en.wikipedia.org/wiki/Hardware_description_language>.
4. <https://pdfs.semanticscholar.org/b79f/0e605db197c2c1a5d62fdc6363685486d9da.pdf>.