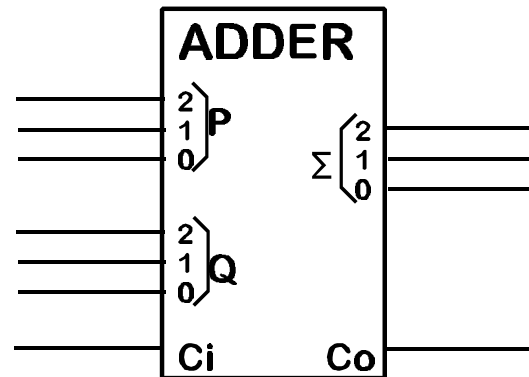


A REVIEW OF COMBINATION LOGIC AND AN INTRODUCTION TO VHDL AND TO GALAXY AND NOVA (WARP2 SOFTWARE)

In CPE219 you covered various types of combinational logic circuits. One of the most common examples is that of the adder. Computers wouldn't work if it were not for adders. We shall start these notes by designing a 3-bit binary adder. Why three and not four? Well, we want to delay some problems until later on.

How would you describe such an adder? According to IEEE Std. 91, it should be described as below. It consists of 7 inputs and four outputs. Again, to save some problems for now, we are going to simplify this by omitting the output carry.

Actually, this is a complete specification of the device because the IEEE standardized this function with this standard. You might, however, give a word description of it. This might take the form below.



This adder adds two three-bit binary numbers, P and Q , along with an input carry and produces the three bit sum, Σ . It should also produce a carry when the sum exceeds 7 (bit won't).

There are other ways this might be described. You could make a truth table, a set of Karnaugh maps, and write any number of Boolean equations. It is assumed that you could do so now. Now, we are going to introduce a new way of describing this adder. We will describe it in VHDL.

Before we look at the description, a few comments are in order. VHDL is a rather primitive language, just as are C, Pascal, and other languages. These languages only become powerful and useful when a set of libraries are included. This is also true of VHDL. A factor which may be very frustrating in learning VHDL is that there are usually many ways a device may be specified - which do you use?

Now, let's look at our 3-bit adder.

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity adder3 is
    port (
        ci:      in  std_logic_vector (0 downto 0);
        p, q:    in  std_logic_vector (2 downto 0);
        s:      out std_logic_vector (2 downto 0)
    );
end adder3;

architecture first_adder of adder3 is
begin
    s <= p + q + ci;
end first_adder;

```

While it is generally stated that a VHDL entity consists of two parts, practically, there are three. We shall look at each of these separately.

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

```

These three lines are used to indicate what libraries (other than the basic VHDL library) are to be used in this device or **entity**. The first line indicates where the compiler should look for the libraries. The other two lines define specific libraries to be used. The `ieee.std_logic_1164` library will be used in most designs and will be described in another set of notes. The `work.std.arith` library will be discussed below. The `.all` at the end is a reserved word which indicates that all of the functions in the named library are to be available (not all will generally be used). Normally you will always use the `.all`. The exception would be if you were to use two libraries which contained a function with the same name.

```

entity adder3 is
    port (
        ci:      in  std_logic_vector (0 downto 0);
        p, q:    in  std_logic_vector (2 downto 0);
        s:      out std_logic_vector (2 downto 0)
    );
end adder3;

```

This is the first of the two main parts of the description. It provides the same information as was provided on in the diagram on the previous page. It specifies the names of the input and output signals. It also describes them. The may in one of four modes: **IN**, **OUT**, **INOUT**, and **BUFFER**. Technically, all signals could be labeled is INOUT but IN and OUT are more descriptive (unless a signal in bidirectional). BUFFER is used for outputs that are also used

internally as inputs. Finally, as in any programming language, you must list the type of “variables” these are. There are a number of different types of variables used in VHDL. Three types of somewhat similar and you may be using them interchangeably (although doing so may sometimes cause an error). These are **bit**, **std_logic**, and **std_ulogic**. There are also three multi bit equivalents: **bit_vector**, **std_logic_vector**, and **std_ulogic_vector**. We shall not attempt to describe the difference between **std_logic** and **std_ulogic**, it is subtle.

The difference between bit and std_ulogic is in the values the signals may assume. **Bit** may be only 0 or 1. You say, for binary variables, what other value is possible? With **std_ulogic** a signal may be in any one of nine states:

'U'	Uninitialized
'X'	Forcing - Unknown
'0'	Forcing - 0
'1'	Forcing - 1
'Z'	High Impedance
'W'	Weak - Unknown
'L'	Weak - 0
'H'	Weak - 1
'-'	Don't care

Weak refers to signals such as those used with passive pull-up (or -down) whereas Forcing refers to totem-pole type signals. We shall not here need to worry about these different types. At this point we could use bit or std_ulogic.

In conclusion, the **entity** part of this module contains the same basic information as the block diagram. It describes the signals going into and coming out of the box. It says nothing about what goes on inside.

```
architecture first_adder of adder3 is
begin
    s <= p + q + ci;
end first_adder;
```

This is the second major component. It is identified by the term **architecture**. Note that the name used here (adder3 and first_adder) are not part of VHDL, you may specify any name you wish. Well, gee whiz! It sure isn't very hard to describe what goes on inside the box. S is the sum of P and Q and the input carry. VHDL must be pretty smart! Not so.

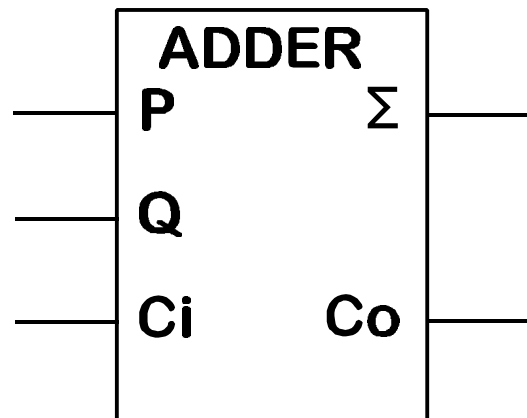
Remember the **use work.std_arith.all;** line above? The std_arith library contains all of the information on how to do addition (and subtraction and multiplication and division).

The VHDL language itself would not be able to make sense of this otherwise.

Well, now we have described this adder, what do we do with the VHDL description? You use the GALAXY compiler to compile the description. In this case we will instruct the compiler to reduce it and to **FIT** it into a 22V10 PAL. The results may be obtained, if you wish, by copying the above file and compiling it your self. We will look at report files a bit later.

Now, let's get a little more serious. We didn't include an output carry and we implemented only three bits. It is possible to implement a four-bit binary adder in a 22V10 if you do it right.

In CPE219, the one of the first things you should have learned about binary addition is the design of a FULL ADDER. Such a device is shown in the block diagram to the right. By cascading four of these (i.e., connecting co of one to ci of the next) it is possible to make a four-bit adder. It is not a very fast adder but it will fit in a 22V10.



In order to implement this in VHDL (in a practical and informative way) we will need to write two modules. The first describes the FULL ADDER.

```
library ieee;
use ieee.std_logic_1164.all;
use work.all;
entity full_adder is
    port (ci:    in bit;
          p, q:  in bit;
          sum, co: out bit);
end full_adder;

architecture full_adder of full_adder is
    signal s1, c1, c2: bit;
begin
    sum <= p xor q xor ci;
    co <= (p and q) or ((p or q) and ci);
end;
```

The library section of the above is the same as before except that the arithmetic library is no longer needed. The **entity** section is similar to the previous example except here we used the simple BIT type (either of the three types could have been used). The **architecture** section describes the full adder in a form which should be (more or less) familiar to you. Note that VHDL uses no operator precedence, hence the AND terms must have parenthesis around them. This may be compiled with GALAXY but this time we instruct the compiler to save the result in

the WORK library rather than to program a device with it.

Having described the full adder, we may write the following module.

```
library ieee;
use ieee.std_logic_1164.all;
use work.all;
entity adder_4 is
    port (ci:    in bit;
          p, q:  in bit_vector (3 downto 0);
          sum:   out bit_vector (3 downto 0);
          co:    out bit);

end adder_4;

architecture adder_4 of adder_4 is
    component full_adder
        port ( ci:    in bit;
              p, q:   in bit;
              sum, co: out bit);
    end component;
    signal c1, c2, c3: bit;

begin
    u1: full_adder port map (ci, p(0), q(0), sum(0), c1);
    u2: full_adder port map (c1, p(1), q(1), sum(1), c2);
    u3: full_adder port map (c2, p(2), q(2), sum(2), c3);
    u4: full_adder port map (c3, p(3), q(3), sum(3), co);

end;
```

In the library section of this module, we have included the WORK library where our full adder has been placed. The **entity** section is similar to our first adder except that the vectors are four bits in length rather than three, the Co signal has been added, and we are again using the BIT type.

In the **architecture** section there is a significant change. First, we have indicated that we are using multiple instantiation of our full adder, describing the signals as named in this module (regardless of what they might have been named in the FULL ADDER module). This is indicated by the **component** declaration which is very similar to the **entity** declaration of the adder.

Next, there is a **signal** declaration. This declares variables which are used internally in the module but not inputs or outputs (which would have been described in the **entity** section). The signals described are the three intermediate carries (i.e., the co from one adder to the ci of the next). Finally, we get to the architectural description. This is in an altogether different form than our first example. There are a number of architectural types in VHDL. The type we saw in the first example is called a **BEHAVIORAL** description - it described the way the module "behaved."

The description above is a **STRUCTURAL** description - it describes the actual structure we wish to use. Another type we shall see later is the **DATAFLOW** description.

```
u1: full_adder port map (ci, p(0), q(0), sum(0), c1);
u2: full_adder port map (c1, p(1), q(1), sum(1), c2);
u3: full_adder port map (c2, p(2), q(2), sum(2), c3);
u4: full_adder port map (c3, p(3), q(3), sum(3), co);
```

In this example, we have described four full-adders (labeled U1, U2, U3, and U4 for lack of any better names). The inputs to the first full-adder are Ci, p(0), and q(0), and the two outputs are sum(0) and c1 (an intermediate carry). Note that c1 is one of the inputs to U2.

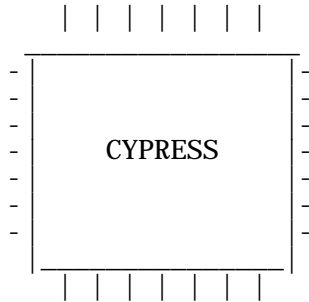
We may compile the above module, this time to be programmed into a 22V10 PAL. When we attempt this, we get error messages! Why? Because, left to do its own thing, VHDL converts this to the same type of adder as in the first example (except this is a four bit adder with carry) and in this case, it won't fit! The process is called **FLATTENING** and applies to the use of VHDL in the programming of CPLD's (and PALS). If we were to put this in an FPGA, this flattening process would not occur.

Well, we still don't have our binary adder that we wanted. We have to have some way to stop the flattening process on the intermediate carries. While this is a subject that occurs late in the VHDL text, we will note it here in order to complete the example. The process is to add three lines to the architecture section.

```
signal c1, c2, c3: bit;
attribute synthesis off of c1: signal is true;
attribute synthesis off of c2: signal is true;
attribute synthesis off of c3: signal is true;
```

begin

These three lines direct the compiler not to eliminate these three internal signals. Using this modification, we may compile our design into a 22V10. So now, let's look at the report file. Better, let's just look at part of the many pages of the report!



Warp VHDL Synthesis Compiler: Version 4 IR x77
 Copyright (C) 1991, 1992, 1993,
 1994, 1995, 1996, 1997 Cypress Semiconductor

```
=====  

Compiling:  adder4.vhd  

Options:    -q -yv2 -e10 -w100 -o2 -yga -fP -fL -v10 -dc22v10 -pPALCE22V10-5PC  

adder4.vhd  

=====
```

f:\warp\bin\vhdlfe.exe V4 IR x77: VHDL parser
 Wed Oct 22 15:37:38 1997

Library 'work' => directory 'lc22v10'
 Linking 'f:\warp\lib\common\work\cypress.vif'.
 Library 'ieee' => directory 'f:\warp\lib\ieee\work'
 Linking 'f:\warp\lib\ieee\work\stdlogic.vif'.
 Linking 'F:\proj03\lc22v10\adder.vif'.

f:\warp\bin\vhdlfe.exe: No errors.

f:\warp\bin\tovif.exe V4 IR x77: High-level synthesis
 Wed Oct 22 15:37:40 1997

Linking 'f:\warp\lib\common\work\cypress.vif'.
 Linking 'f:\warp\lib\ieee\work\stdlogic.vif'.
 Linking 'F:\proj03\lc22v10\adder.vif'.
 Note: Removing wires from arch. 'adder_4' of entity 'adder_4'.

f:\warp\bin\tovif.exe: No errors.

f:\warp\bin\topld.exe V4 IR x77: Synthesis and optimization
 Wed Oct 22 15:37:43 1997

Linking 'f:\warp\lib\common\work\cypress.vif'.
 Linking 'f:\warp\lib\ieee\work\stdlogic.vif'.
 Linking 'F:\proj03\lc22v10\adder.vif'.
 Linking 'f:\warp\lib\lc22v10\stdlogic\c22v10.vif'.

f:\warp\bin\topld.exe: No errors.

LOGIC MINIMIZATION
 DESIGN EQUATIONS (15:37:52)

$$\begin{aligned} \text{sum}_3 &= p_3 * /q_3 * /c3 + /p_3 * q_3 * \\ & /c3 + /p_3 * /q_3 * c3 + p_3 * q_3 * c3 \\ \text{sum}_2 &= p_2 * /q_2 * /c2 + /p_2 * q_2 * /c2 \\ & + /p_2 * /q_2 * c2 + p_2 * q_2 * c2 \\ \text{sum}_1 &= p_1 * /q_1 * /c1 + /p_1 * q_1 * /c1 \\ & + /p_1 * /q_1 * c1 + p_1 * q_1 * c1 \end{aligned}$$

```

sum_0 = ci * /p_0 * /q_0 + /ci * p_0 * /q_0
        + /ci * /p_0 * q_0 + ci * p_0 * q_0
co = q_3 * c3 + p_3 * c3 + p_3 * q_3
c2 = q_1 * c1 + p_1 * c1 + p_1 * q_1
c1 = p_0 * q_0 + ci * q_0 + ci * p_0
c3 = q_2 * c2 + p_2 * c2 + p_2 * q_2

```

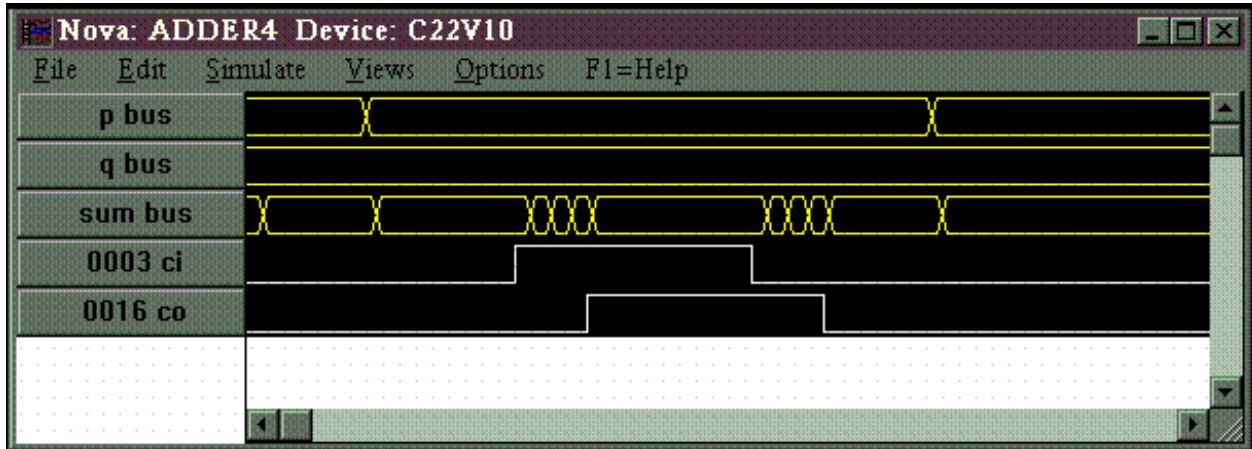
Completed Successfully

C22V10

q_0 =	1	24	* not used
p_0 =	2	23	= sum_1
ci =	3	22	= sum_3
q_2 =	4	21	= (c2)
p_2 =	5	20	= (c3)
q_1 =	6	19	* not used
p_1 =	7	18	* not used
q_3 =	8	17	= (c1)
p_3 =	9	16	= co
not used *	10	15	= sum_2
not used *	11	14	= sum_0
not used *	12	13	* not used

There you have it. The equations are what we should have expected them to be. VHDL assigned input and output pins for the PAL. Since a PAL has no internal nodes, three of the output pins were used for the internal carries. In addition to this report file, a JEDEC file was created which might have been downloaded into a device programmer and a 22V10 programmed.

SIMULATION



The NOVA simulator may be then used to simulate this device. This type of simulator obtains its simulation information from the JEDEC file and thus represents the actual timing parameters of the device (22V10) being used. Note how the SUM output changes several times as the input carry (ci) ripples through the four bits.

In case you do not have the ability to capture the screen, as above, the simulator will also produce a text file as output. This is shown (in part) below.

NOVA Simulation Printout
 File: ADDER4.psd - View: PINS and REGS
 Printout produced: Wed Oct 22 16:39:57 1997

		1	1	2	2									1	2	1	2	
		3	6	7	0	1	2	7	5	9	1	6	4	8	4	3	5	2
	c	c	j	j	j	p	p	p	p	q	q	q	q	s	s	s	s	
	i	o	e	e	e	$\bar{0}$	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{0}$	$\bar{1}$	$\bar{2}$	$\bar{3}$	u	u	u	u	
			d	d	d									m	m	m	m	
			-	-	-									$\bar{0}$	$\bar{1}$	$\bar{2}$	$\bar{3}$	
			n	n	n													
			o	o	o													
			d	d	d													
			e	e	e													
			1	2	2													
			7	0	1													
^^																		
0:	0	L	L	L	L	1	1	1	0	0	0	0	0	L	L	L	L	
1:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
2:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
3:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
4:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
5:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
6:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
7:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
8:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
9:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
10:	0	L	L	L	L	1	1	1	0	0	0	0	0	H	H	H	L	
11:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	L	
12:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
13:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
14:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
15:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
16:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
17:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
18:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
19:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
20:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
21:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	
22:	0	L	L	L	L	1	1	1	1	0	0	0	0	H	H	H	H	

This concludes this set of notes. From it you should have had some review of adders and combinational logic and a fairly good introduction to VHDL. More will follow.