# Chapter 13
# Synthesis of Digital Circuits

## 13.1 A brief Introduction

When designing a new computing system, the computer engineer is faced with a number of choices. These include the programming *paradigm*[1], the programming language, and the hardware on which the system will run. Hardware design often starts with the choice of the CPU or MPU; do we need something like a Pentium or alternatively like an ATMega328 MPU (used on the Arduino) or a Cortex MPU used on more powerful Arduino variants? Or should we choose a Digital Signal Processing (DSP) chip, which is usually the case if our system is to handle audio or video data. Across the huge range of choices we have, we find different characteristics such as clock speed, bus width, functionality and cost.

At this point a couple of questions spring to mind. First, how is the digital electronic circuitry on each of these chips *designed*? Second, how can we *design* and *synthesise* a completely new chip for a bespoke application? In the early days of electronic engineering, a computer system was designed and developed 'on paper' then 'by hand' using discrete chips and breadboards. This is how I created the 8-bit data-processing computer 'Condac-85' some years ago. This is no longer the case. Today's engineers have CAD tools where the electronic circuitry design is done in software which specifies how digital functional blocks are connected using signals (which run along wires).

A typical workflow is shown in the diagram below. On the left is a block diagram of a simple CPU which we need

[1] 'Paradigms' are not languages! They are a higher-level description of the *style* of programming. The one you are used to is *imperative* (C, Java, C#). The *functional* paradigm has no assignments, functions are applied to arguments (Scheme). The *logical* paradigm specifies facts and inference rules, and asks if something is true, the program goal (Prolog).

to create using digital circuitry; there is a lot of functionality behind this block diagram which we capture in some sort of specification document (or hold in our heads or in conversations). The first stage in synthesis is to convert this

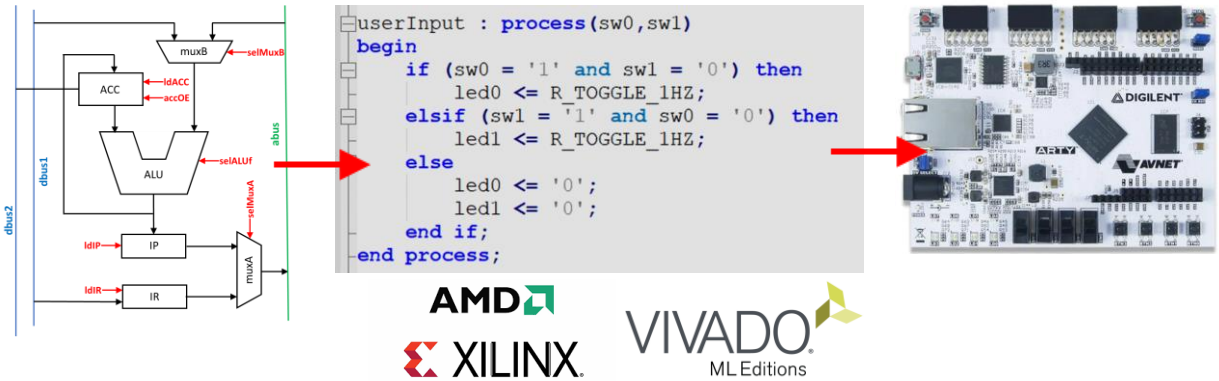



diagram and its functionality into a computer language which has been designed to support digital circuitry. This is a 'Hardware Description Language' (HDL). When the computer program describing the block diagram is complete, we call the compiler, which synthesizes the design into a 'net' list.

This net list describes how various electronic building blocks are wired up to create our desired functionality. The final stage is to upload a representation of this list to our target device, which is a 'Field Programmable Gate Array' (FPGA). The one we shall use is the Arty-A7 board shown on the right, the software is 'Vivaldo' from Xilinx, now a part of AMD.

The takeaway point is that when we code in VHDL, we are **not** programming a CPU, but we are physically wiring up an electronic circuit. This happens inside the FPGA, so let's see what an FPGA actually is. To do this let's compare the hardware architecture of the MPU in a typical Arduino shown in Fig.1 (top) with the architecture of a FPGA shown in Fig.1 (bottom).

The MPU has a clear and specific architecture with various functional blocks connected via signal wires and

signal buses. There are also signal wires connected to the external environment which ultimately appear as Arduino input and output pins. Now consider the architecture of the FPGA. At first glance it appears that there is no architecture at all! All we have appears to be an array of 'blobs' connected by wires. Well, the technical term for these 'blobs' is 'Configurable Logic Blocks' (CLBs). Each CLB will contain some look-up tables (implemented using RAM) and some flip-flops which can provide sequences of operations.

Running horizontally and vertically between the CLBs are loads of wires, and each CLB can connect to any of these wires which run next to the CLB. Also, at the intersections of the horizontal and vertical wires are a load of switches which can connect horizontal with vertical, a 'matrix' of switches.

The point is that all of these switches and connexions are *programmable*, and this is how it works. The HDL captures the digital design, and then when it uploads to the FPGA chip, it makes the correct CLB-wire connexions and switch connexions to create 'synthesize' the digital design. So at this point, our desired CPU runs on full-speed hardware, the design has become hardware. One may say on a theological level, that our hardware design has become *incarnate* in the FPGA.

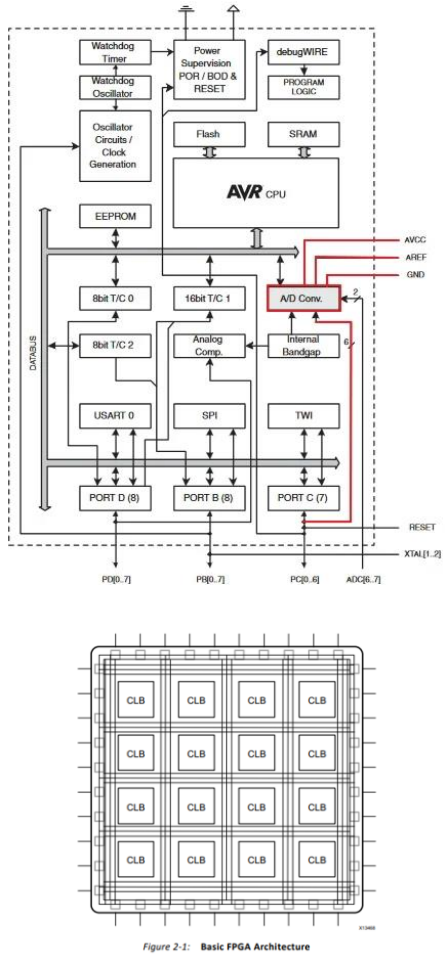



*Figure 1* Top shows the architecture of a Mega328 MPU chip, bottom shows the architecture of a FPGA.

## 13.2 Introduction to VHDL

Our job will be to use a HDL to create a number of digital circuits from a simple door-lock to a simple CPU (the one shown above). The specific language we shall use is 'VHDL' which stands for **V**HSIC **H**ardware **D**escription **L**anguage where the acronym VHSIC in turn stands for **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit. Just call me 'VHDL'.

There is one fundamental difference between the basic code object we use in conventional programming and those used in VHDL. The languages we have used so far manipulate

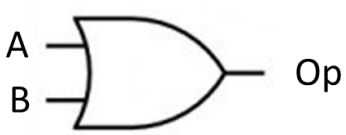
basic objects called *variables*. They may be assigned a value, and they retain (remember) that value, e.g., the statement

```
dist = distArray[index];
```

grabs the value in the array distArray at the index, and assigns it to the variable dist. From this point on dist will have this value unless it is changed. In VHDL the basic object is a *signal* which is a voltage (LOW or HIGH) which resides on a wire. So we could have the statement

```
output <= input;
```

This states that the signal on the output wire is identical to the signal on the input wire for all time. So if the input signal changes the output signal will change too. There is no memory implied here. Note the <= operator which reminds us that we are dealing with *signals* rather than procedural assignments = .




| A | B | Op |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

*Figure 2* OR-gate. Top shows the structure and bottom shows functionality.

### 13.2.1 VHDL code for the Or-gate

Let's consider just a simple OR-gate. Sure, we'd never use VHDL to actually model this, but it's a good place to learn some basic syntax.

Let's review the structure and functionality of the OR-gate; see Fig.2. Here the structure is shown at the top. There are two inputs A and B and an output Op. Of course this structure is not unique to the OR-gate, all two-input logic gates have the same structure. It's the functionality which is unique to each gate type; this is represented as a truth table. The code for the OR-gate reflects this structure and functionality, separating them into two separate blocks of code. Here's the entire code, where I've retained the VHDL coloring.

```
-----------------------------------
-- OR_gate.vhd
-----------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- structure
entity or2 is
    Port ( A : in std_logic;
           B : in std_logic;
          Op : out std_logic);
end or2;

-- functionality
architecture Behavioral of or2 is
begin

    Op <= A or B;

end Behavioral;
```

The structure is given in the the **entity** block which is named "or2". Inside the block we have the details of the structure of the OR-gate, the inputs and outputs which are captured in the **Port** statement. Ports are the means which entities can communicate with other entities using signals. Like maritime ports which have incoming and outgoing ships, VHDL ports have incoming and outgoing signals.

So the **Port** for the OR-gate has two inputs A and B and one output Op, which agrees with Fig.2. The inputs and outputs are assigned a type. So both inputs are assigned as in and their type is std_logic which means they are a single wire with standard logic levels. The output Op is assigned as out and it is also a standard logic type. This is rather like

declaration of variables in conventional programming, except that here we must specify both type and direction.

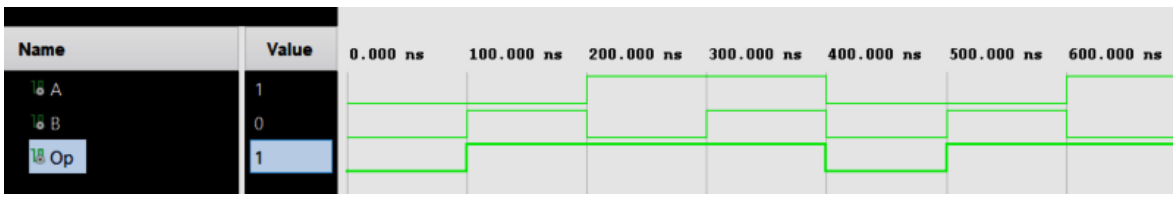
The functionality of the gate is specified in the architecture block of code. VHDL supports several ways of coding, we shall always use the Behavioural architecture. Here we have a single line of code `Op <= A or B` which simply states the Op signal is signal A or signal B, which expresses the truth table.

It will come as no surprise that there is an alternative way of coding the architecture. Here's how to be a little more explicit,

```
architecture Behavioral of or2 is
begin

Op <= '0' when A = '0' and B = '0' else
      '1' when A = '1' or  B = '1';

end Behavioral;
```

Expressed in plain English, this code reads "When A is 0 and B is 0 then the Op signal becomes 0. Else when A is 1 or B is 1 the Op signal becomes 1".

We can test that the code is working correctly by running a simulation. We drive the circuit with all four possible combinations of A and B and monitor the output. Vivado let's us use a separate VHDL 'TestBench' file. Since electronic engineers need to understand the time response of their circuits, test bench files change inputs over time. Here't the simulation for our OR-gate



The values of A and B cycle through the four rows of the truth table, (0,0), (0,1), (1,0) and (1,1) and you can see that the output is high for three of these. Note that the time scale is accurate for the Artix-7 FPGA device we are using, the input signals are changed after 100 nano-seconds which corresponds to a frequency of 10MHz.

### 13.2.2 An Example from Logic and Language

In the previous chapter Logic and Language we looked at how language problems could be mapped onto digital logic, and in particular how Boolean expressions could be constructed to solve logical problems. Let's revisit one problem here, and see how to code it using VHDL and synthesise a digital circuit.

Consider the following Boolean expression expressed in 'sum-of-products' form, comprising two miniterms.

**L = A . ~B + A . B**

Here's one way of coding this using VHDL,

```
----------------------------------------------------
-- NoC_Logic_Problem (from logic in language Unit)
-- L = A . ~B + A . B
----------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity NoC_Logic_Problem is
    Port ( A : in std_logic;
           B : in std_logic;
          Op : out std_logic);
end NoC_Logic_Problem;

architecture Behavioral of NoC_Logic_Problem is
signal miniTerm1, miniterm2 : std_logic;
begin
miniTerm1 <= A and not(B);
miniTerm2 <= A and B;
Op <= miniTerm1 or miniTerm2;

end Behavioral;
```

The entity block of code is the same since we have the same problem structure, two inputs and a single output. The architecture section introduces a new concept. Note that we declare two internal signals, 'miniterm1' and 'miniterm2', and they are of type std_logic, thus single wires. Think of these like 'local' variables in conventional programming. This has to be done before the architecture begin statement. Then we can use these. First we have

```
miniTerm1 <= A and not(B);
```

which tells us that the signal miniterm1 is produced by A and not-B (which is the first miniterm). Then we produce the second mini-term signal

```
miniTerm2 <= A and B;
```

and then we create the 'sum-of-products' signal as the Op signal,

```
Op <= miniTerm1 or miniTerm2;
```

That's the problem solved. One important point to understand in the above code. The three statements that create then combine the mini-terms are ***not sequential***, they could be written in any order! Remember what VHDL code is doing, it is wiring up signals, so it doesn't matter which ones you wire up first; the circuit will only function when they are all wired up. These statements are known as *concurrent signal assignment statements* (CSAs)

Of course, there is a simpler way of solving this problem without using internal signals; we could code the Boolean expression directly as

```
Op <= (A and not(B)) OR (A and B);
```

which shows how powerful VHDL is at solving any combinatorial logic problem expressed as a Boolean expression.

## 13.3 The Process Construct

The modelling of more complex components such as memory and CPUs requires a different approach. We highlighted that signals, unlike conventional variables, have no memory. So how can we construct a memory chip. In conventional programming we would use an array of numbers, each cell would store a value with the array index as the cell address. Wouldn't it be nice if we could use some conventional constructs in our VHDL code. Well, that's the purpose of the **Process** construct.

### 13.3.1 The D Flip-Flop

We shall write some VHDL for the D Flip-Flop. Let's first remind ourselves of what this is and how it works. Fig.3 shows the circuit symbol for a D flip-flop. There is an input D and output Q. The reset input when raised high (1) will reset the flip-flop so the output becomes low, Q = 0. The detailed behaviour is shown below.

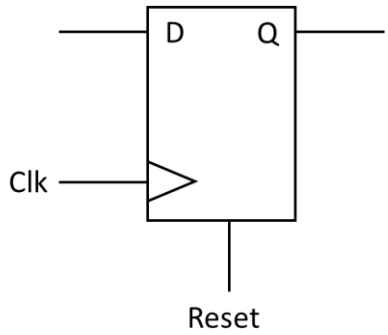
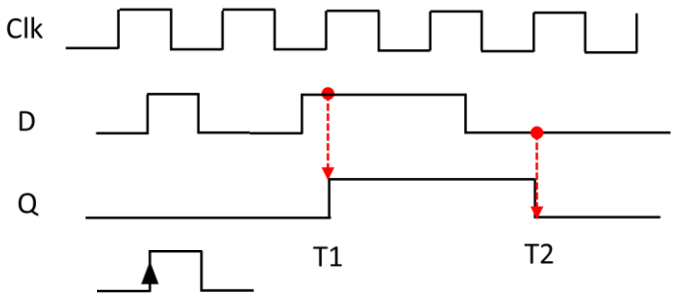



*Figure 3 D- flip flop with input D, output Q, clock input Clk and reset input*




The input on D only passes to the output on a rising edge of the clock. So at T1 D=1 so this value is transferred to the output Q. Note later, when D becomes 0, Q remains at 1. Only

at T2 is the value of D=0 clocked through to the output which then drops to zero, Q = 0. So the flip-flop remembers the value of D until the next rising clock edge, in other words the D flip-flop is *one bit of memory*. In actual circuits, we expect that the input D changes less frequent than the clock pulse.

So how do we synthesize a single D flop in VHDL. Let's start off ignoring the reset input. Here's some code.

```
----------------------------------------
-- single dFlop without reset
----------------------------------------
Library IEEE;
USE IEEE.Std_logic_1164.all;

entity d_flipflop_1 is
port(
    Q : out std_logic;
    Clk :in std_logic;
    D :in  std_logic
);
end d_flipflop_1;

architecture Behavioral of d_flipflop_1 is
begin

process(Clk)
begin
    if(rising_edge(Clk)) then
        Q <= D;
    end if;
end process;

end Behavioral;
```

First check out the Port declaration in the entity block. There are three signals of type std_logic, D and Clk are inputs and there is an output Q.

Now look at the new construct, the **process**. Note it has its own begin and end. What do we find inside the process block, well lookie here, we have an if-then statement from

conventional programming! Hey, that's fantastic, so we *can* use conventional programming inside the process. So what's going on here? Well the statement

```
if(rising_edge(Clk)) then
    Q <= D;
end if;
```

is simply saying "if there is a rising Clk edge, then the input signal is transferred to the output signal which is what we want. The function call to **rising_edge(Clk)** is built in to VHDL. Now look at the start of the process block and you will see we have **process(Clk)**. Here, Clk is ***not a parameter***. Process is not a conventional *function*, we can only use conventional programming *within* the process block. So if this is not a parameter, what is it? Well any signal in those brackets is part of the *sensitivity list* which is a list of signals which cause the code in the process block to execute, *when they change*. So in this case, when Clk changes, the process block will execute as described above.

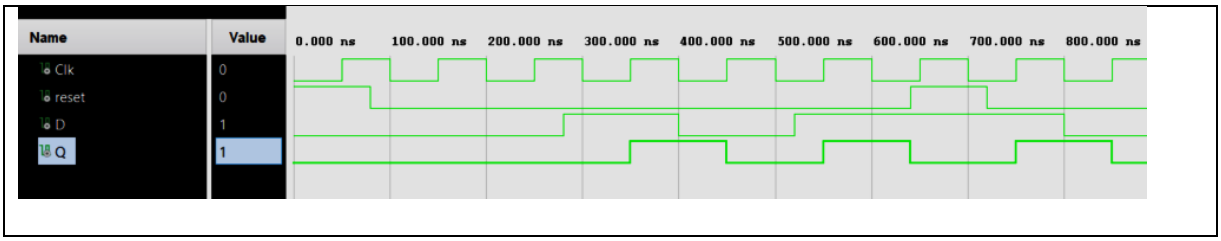
Now we can extend this code to deal with the reset input, let's see how the process block has changed.

```
architecture Behavioral of d_flipflop_1 is
begin
process(Clk,reset)
begin
    if(reset='1') then
        Q <= '0';
    elsif(rising_edge(Clk)) then
        Q <= D;
    end if;
end process;
end Behavioral;
```

We now have a full if-then-else inside the process. We first test for reset and if this is 1 then we set the output signal to 0. Note the syntax, the single quotes means we are setting a single wire or bit. Then we test for a rising clock edge as

before. Glance at the sensitivity list, we have now added reset to this, so that the process will respond to changes in the reset signal.

Before we move on, let's have a look at the testbench waveform for the D flip-flop with reset.




There are two things to note. First, when reset is high the output goes low Q = 0. Second look at the rising clock edges and check that the value of D at that time is transferred to Q. Check that Q does not change except at certain rising clock edges.

### 13.3.2 VHDL for RAM

Let's now turn to synthesizing memory. We mentioned above that a D flop is one bit of memory, so we could assemble a load of D flops. Sure that would work, but it would be hard work for us, why not let the VHDL compiler do this work. First let's look at the signals (port) into and out of memory, see Fig.4. The thick arrows represent *buses*, collections of wires. We need a way of getting data into and out of memory, so we have two data buses. We choose to have these 16 bits wide. These buses feed into an ***array*** of memory cells, each one with an address. So we need an address bus which is an input. We choose to have this 3 bits wide which will give us 8 memory cells. Finally we need two control signals to indicate whether we are reading (mR) from memory or writing (mW) to it. If mR = '1' then we are reading and if mW = '1' then we are writing.

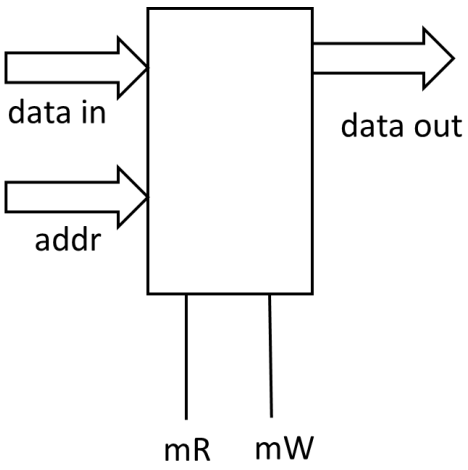



*Figure 4RAM showing input and output data buses, input memory address and read / write control signals*

So how do we go about writing our VHDL? I gave you a hint above when I referred to an ***array*** of memory cells, and we know that we can use conventional code constructs inside a process, so let's use a conventional array!

Let's have a look at the complete code and first hunt down this array. On line 21 we declare a type "ram_type" and state that this **is** an **array**. We state the indices go from **(0 to 7)** so we have our 8 array elements, and they are **of** type std_logic_vector (which is a bunch of signals) and they are 16 bits wide, with indices going from **(15 downto 0).** Then on line 22 we declare a new (local) signal "RAM" of ram_type, and we initialize it with values (lines 23-30). Note the **:=** operator. This means we are assigning a conventional programming variable, and not a signal where we use <=. The syntax := first appeared in Pascal (I think). The contents of each memory row is initialized as a hexadecimal number hence the X in X"1000". We have a 16 bit number, and each hexadecimal digit (called a nibble – half a byte) is 4 bits wide. So we need 4 hex digits to represent each 16 bit data value.

Now let's check out the entity declaration and we see (i) the address bus of 3 bits, (ii) the 16-bit read_data bus (output), (iii) the 16-bit write_data bus (input) and two single signals mRead and mWrite. All of this should make sense.

Now back to the architecture section, and inside the process we find an if-then-else conventional construct. If mRead = '1' then we must read out data. So (line 38) we get the value of the address and use this as the index into our RAM array, and output the result to the signal bus read_data. Else if mWrite = '1' we are to write data into memory (line 40). The function **conv_integer(address)** converts the address (which is a signal) into an integer which is used as index into the array.

```
1  ----------------------------------------------------------------------------
2  -- RAM.vhd
3  ----------------------------------------------------------------------------
4
5    library IEEE;
6    use IEEE.STD_LOGIC_1164.ALL;
7    use IEEE.std_logic_arith.all;
8    use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10   entity RAM is
11     Port ( address : in std_logic_vector(2 downto 0);
12            read_data  : out std_logic_vector (15 downto 0);
13            write_data : in std_logic_vector (15 downto 0);
14            mRead : in std_logic;
15            mWrite : in std_logic
16          );
17   end RAM;
18
19   architecture Behavioral of RAM is
20
21   type ram_type is array (0 to 7) of std_logic_vector (15 downto 0);
22   signal RAM : ram_type := (
23   X"1000",
24   X"1001",
25   X"1010",
26   X"1011",
27   X"1100",
28   X"1101",
29   X"1110",
30   X"1111");
31
32   begin
33
34   Memory: process(address,write_data)
35
36     begin
37       if(mRead = '1') then
38          read_data <= RAM(conv_integer(address));
39       elsif(mWrite = '1') then
40          RAM(conv_integer(address)) <= write_data;
41       end if;
42     end process;
43
44   end Behavioral;
```

## 13.4 Finite State Machines

Imagine we have been commissioned to design and develop a digital system to open a door when a correct 4-digit pin code is entered. How would we approach this? Using a finite state machine (FSM) is a great design choice. Let's say we have four buttons and we must press them in this order B3 B1 B2 B4. If we get any button out of order, then the FSM needs to return to the starting state. Fig.5 shows the FSM structure. The starting state is **stateA** and you can see how the machine progresses through the states as the correct button is pressed in sequence, any incorrect button will send the machine straight back to **stateA**.

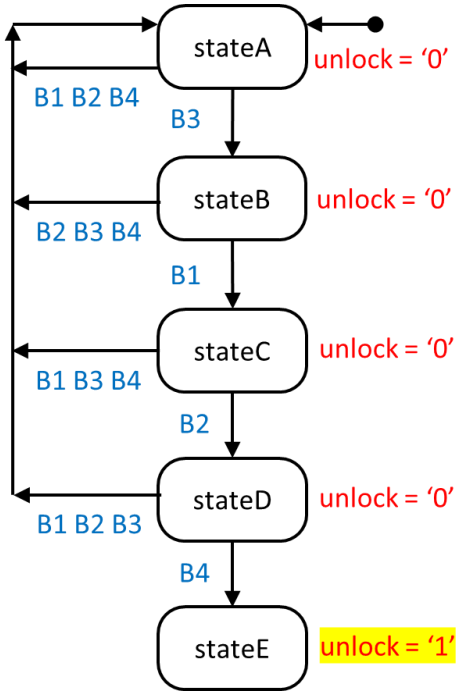



*Figure 5 FSM for the 4-button pin-code lock.*

So how to approach coding this in VHDL? Well, if we were programming this conventionally, we could easily use a **switch(…)** construct like this

```
switch(state) {

    case stateA :
    if(button = B3)
        state = stateB;
    else
        state = stateA;
    break;

    case stateB :
    if(button = B1)
        state = stateC;
    else
        state = stateA;
    break;

    ...

}
```

but we know we can use conventional programming constructs inside a **process** and the switch is no exception, Si here is some VHDL code to do this. You will see that we use two process blocks, one to change state, and the second to provide the unlock signal based on the state.

Here's the VHDL for the next state logic. You can clearly see the FSM case statements, and correct button testing using if-then-else construct. Also a reset button has been included. Look at the sensitivity list, any signal that changes and needs to be used within the process must appear here.

```
next_state_logic: process(button_reset,button_1,button_2,button_3,button_4)
begin
    if( button_reset = '1' ) then
        state <= stateA;
    end if;

        case state is
            when stateA =>
                if ( button_3 = '1' ) then
                    state <= stateB;
                elsif( button_1 = '1' or button_2 = '1' or button_4 = '1') then
                  state <= stateA;
                end if;

            when stateB =>
                if ( button_1 = '1' ) then
                    state <= stateC;
                elsif( button_2 = '1' or button_3 = '1' or button_4 = '1') then
                  state <= stateA;
                end if;


        end case;

end process;
```

The second process creates the unlock signal based on the current state. This is a list of conditionals using the keyword **when**.

```
output_logic: process(state)
begin
    case state is
    when stateA =>
        unlock_out <= '0';
    when stateB =>
        unlock_out <= '0';
    when stateC =>
        unlock_out <= '0';
    when stateD =>
        unlock_out <= '0';
    when stateE =>
        unlock_out <= '1';
    end case;
end process;
```

So we have two processes. How should we view their operation. Remember that VHDL is describing how to wire things up. Within each process there can be selections or waits going on. So a process is sequential. But all processes

operate concurrently. Each process produces its own associated electronic hardware which will usually contain sequential components. But the two chunks of hardware produced by two processes will execute concurrently.

## 13.5. Further Examples

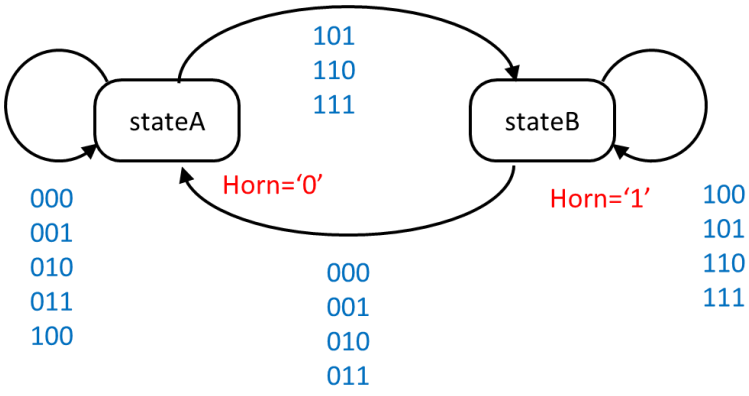
### 13.5.1 Car Alarm

Let's say we are designing a car alarm. There is a door sensor, D=0 when the door is closed and D=1 when the door is open. There is an ultrasonic detector U=0 when there is no motion in the car and U=1 when it detects motion. Finally, the alarm is armed with a key when K=1, and the alarm is turned off when K=0. The truth table that describes this scenario is shown in Fig.6 where the final column indicates if the horn H is off H=0, or on H=1.

| K | D | U | H |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

*Figure 6 Truth table for car alarm problem.*

Of course this is fine, but it won't really work. Imagine the system is armed, K=1, then the door is opened, D=1, so the horn sounds, H=1. But as soon as the door is closed, D = 0, then the horn will stop sounding, not exactly a useful alarm. So we have to remember when the alarm has been activated and keep the horn sounding until the alarm is deactivated, K = 0. This is clearly a job for a FSM. Here's a possible solution. The triples are values of KDU in that order.




These will translate to a VDHL type std_logic_vector (2 downto 0). In stateA the horn is off, and in stateB it is on. Starting in stateA if the alarm is armed (K=1) then we transit

to stateB is the door is open, KDU = “110”, or if the ultrasonic sensor is activated, KDU=”101”, or both KDU=”111”. You can see that the only way back from stateB to stateA is when K=’0’, i.e. the triples KDU = “000”, “001”, “010”, “011”.

There are several ways to code this FSM using VHDL, of course we always strive for optimality. So here’s one good solution. There are two process blocks, one to make the state transitions and the other to send the correct signal to the Horn. Remember both synthesized circuits will run in parallel. Before the FSM, we need to declare a type statetype and initialize the starting state,

```
type statetype is (stateA,stateB);
signal state: statetype := stateA;
```

Here's the next_state_logic process

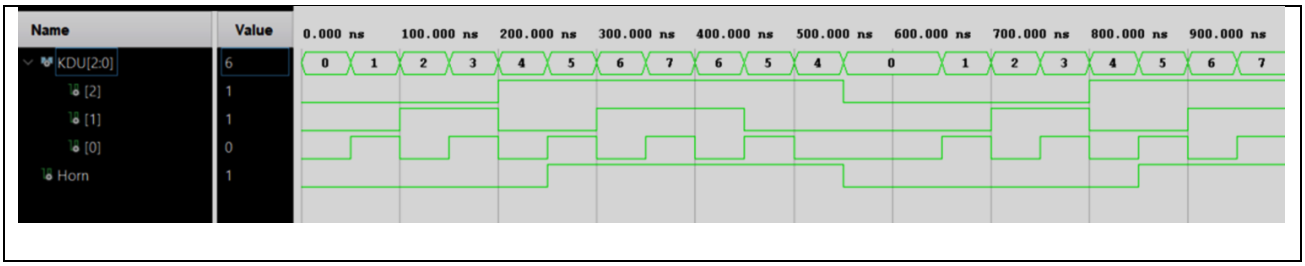
```
next_state_logic: process(KDU)
begin
case state is
    WHEN stateA =>
        IF(KDU = "101") THEN
            state <= stateB;
        ELSIF (KDU = "110") THEN
            state <= stateB;
        ELSIF (KDU = "111") THEN
            state <= stateB;
        ELSE
            state <= stateA;
        END IF;

    WHEN stateB =>
        if(KDU(2) = '0') THEN
            state <= stateA;
        else
            state <= stateB;
        end if;
END CASE;
END PROCESS;
```

The two states are used in the case statement. The three possible transitions into stateB are coded explicitly using the

three possible values of KDU. This is straightforward. To transit from stateB to stateA we use a trick. Since this transit will occur only if K = '0', then we test explicitly for this bit condition if(KDU(2) = '0') then. The second process block is straightforward

```
output_logic: process(state)
begin
case state is
    WHEN stateA =>
        Horn <= '0';
    WHEN stateB =>
        Horn <= '1';
end case;
end process;
```

Finally, a couple of points. Note the contents of the sensitivity lists of both processes and remember these are the things the process is sensitive to, i.e., those things that will make the process run. Second note that I have written some very *sloppy* code with a mix of upper and lower case for VHDL keywords. At least this tells you one thing, that for these words VHDL is not case-sensitive! The testbench simulation produces the following showing the expected transitions.




The horn starts sounding when the lowest bit of KDU (which is U) goes high. Note also that the horn remains on when D and U have dropped to zero, and only turns off shortly after 500 ns when K is dropped to zero.

### 13.5.2 Counters

Counters are very useful things, e.g., you could store a musical melody in a RAM and use a counter to provide the address to select the melody notes. A simple counter is an n-bit binary counter which starts at 0 and ends at the number $2^n - 1$. So a 4-but counter would count from 0 to 15 (or 0 to F in hex).

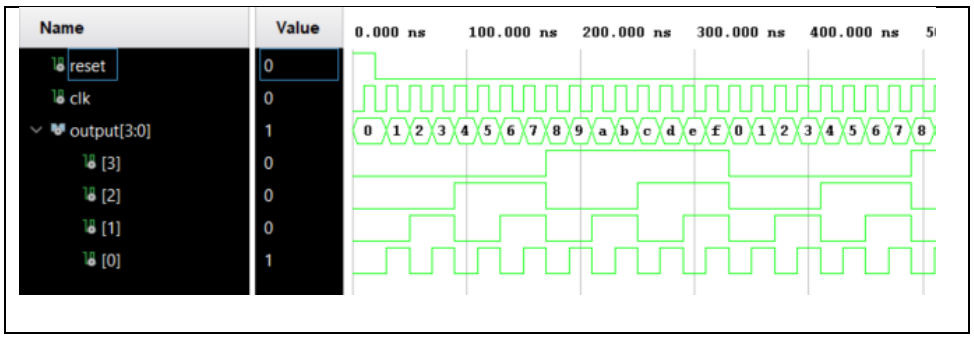
Counters are simple to code, if we use a process block and inside that conventional coding. Here's an example.

```
1  ----------------------------------------------------
2  -- up_counter.vhd
3  -- CBP 03-01-23
4  ----------------------------------------------------
5
6  library IEEE;
7  use IEEE.STD_LOGIC_1164.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10 entity up_counter is
11     Port ( clk: in std_logic; -- clock input
12            reset: in std_logic; -- reset input
13            output: out std_logic_vector(3 downto 0
14 end up_counter;
15
16 architecture Behavioral of up_counter is
17 signal count_up: std_logic_vector(3 downto 0);
18
19 begin
20
21 process(clk,reset)
22 begin
23     if(reset = '1') then
24       count_up <= "0000";
25     elsif(rising_edge(clk)) then
26       count_up <= count_up + 1;
27     end if;
28 end process;
29
30 output <= count_up;
31
32 end Behavioral;
```

The entity tells us that the counter is being driven by a clock and that it has a reset input. In the architecture section we declare a signal variable **count_up.** The process(clk, reset) is sensitive to changes in clk and reset. The conventional code is straightforward, we use an if-then-else to test for a reset whence we set count_up to zero, else look for a rising clock

edge, and if we have one then we increment the value of count_up by a conventional addition. Outside of the process block, we send the current value of count_up to the output signal. Remember this line and the entire process block are concurrent, the circuitry for both runs at the same time.

What happens when the count reaches $2^n - 1$ and we do a further addition. A peek at the testbench waveform will help. When the output reaches this value (hex f) then it returns




to zero because adding a 1 to f (binary 1111) will produce a 1 on the overflow bit, and the other bits revert to 0.

## 13.6 Structural Architecture

So far we have been using the Behavioural architecture which is particularly suited to the synthesis of small, often complex devices. But often we are faced with a situation where the device we wish to synthesize is complex, with many components such as an entire CPU. In such cases we retain the behavioural architecture for each component, but we connect the components together through their Ports. This is the structural architecture, which operates at a higher level.

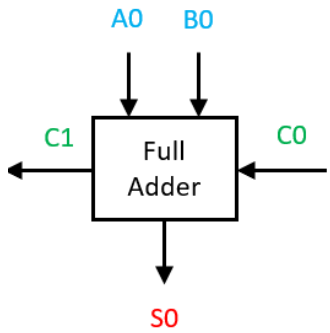



*Figure 7 Single bit-slice full adder*

As an example we shall consider a 4-bit adder which sums the values of two 4-bit numbers A and B. The building block is a bit-slice full adder shown in Fig.7. This takes two single-bit inputs A0 and B0 and outputs the sum S0 and any carry C1. So if A=0, B=1 then S=1 and C=0. If A=1, B=1 then S=0 and C1=1. There is also a carry-in C0. Clearly such bit-slices can be combined to add multi-bit numbers, we shall

see this shortly, but first we must understand a single bit slice. The truth table and associated circuit made from gates is shown in Fig.8. From this it is straightforward to write the behavioural VHDL shown below where we specify the behaviour using standard logic.

| Cin | B | A | S | Cout |
|---|---|---|---|---|
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | |
| 0 | 1 | 1 | | 1 |
| 1 | 0 | 0 | 1 | |
| 1 | 0 | 1 | | 1 |
| 1 | 1 | 0 | | 1 |
| 1 | 1 | 1 | 1 | 1 |

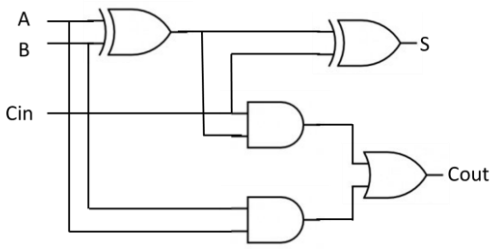
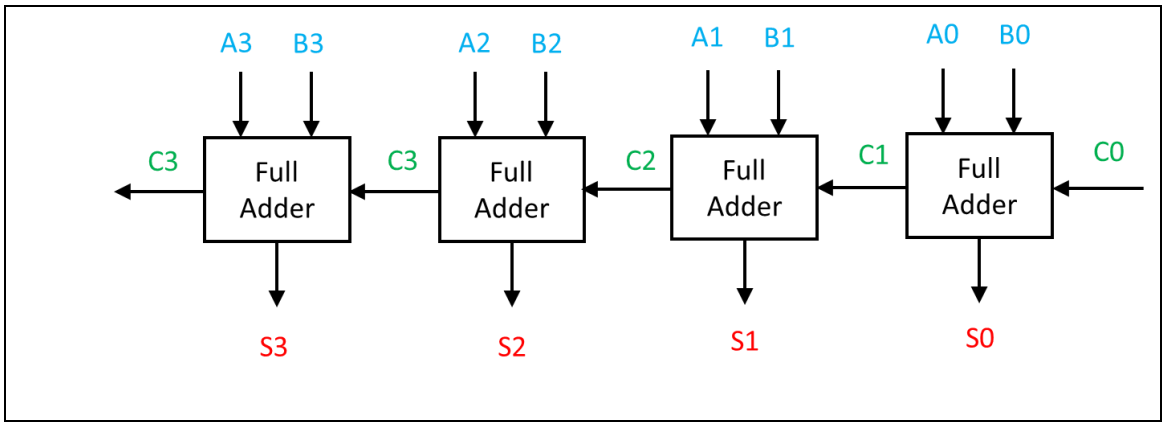



*Figure 8 Truth table and gates forming a single bit full adder.*

```
----------------------------------------------------------
-- fullAdderSlice.vhd
-- single bit full adder
-- CBP 04-01-23
----------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fullAdderSlice is
  Port (A : in std_logic;
        B : in std_logic;
        Cin : in std_logic;
        S : out std_logic;
        Cout : out std_logic);
end fullAdderSlice;

architecture Behavioral of fullAdderSlice is

begin

S <= A xor B xor Cin;
Cout <= (A and B) or (A and Cin) or (B and Cin);

end Behavioral;
```

Now to create a 4-bit adder we must chain four of these bit-slices together like this.



Each slice handles one bit of each number A and B. These numbers will have range ‘0000’ to ‘1111’. The lowest-order bit is shown on the right and the highest on the left. If there is a carry on any bit, then this is passed to the left, just as a carry when we add numbers on paper.

This architecture is interesting since we use the same component four times, and this shouts out Structural architecture. The required VHDL code is shown below.

```
1  ----------------------------------------------------------------------
2  -- FourBit_Full_Adder.vhd
3  -- Structual Architecture for a 4-bit adder
4  -- CBP 04-01-23
5  ----------------------------------------------------------------------
6
7  LIBRARY ieee ;
8  USE ieee.std_logic_1164.all ;
9
10 ENTITY FourBit_Full_Adder IS
11 PORT (  Cin : IN    STD_LOGIC ;
12         X   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
13         Y   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
14         S   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
15         Cout: OUT   STD_LOGIC ) ;
16 end entity FourBit_Full_Adder;
17
18 ARCHITECTURE Structural OF FourBit_Full_Adder  IS
19 SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
20
21 COMPONENT fullAdderSlice
22   Port (A : in std_logic;
23         B : in std_logic;
24         Cin : in std_logic;
25         S : out std_logic;
26         Cout : out std_logic);
27 END COMPONENT;
28
29 BEGIN
30 stage0: fullAdderSlice PORT MAP ( X(0), Y(0), Cin, S(0), C(1) ) ;
31 stage1: fullAdderSlice PORT MAP ( X(1), Y(1), C(1), S(1), C(2) )
32 stage2: fullAdderSlice PORT MAP ( X(2), Y(2), C(2), S(2), C(3) )
33 stage3: fullAdderSlice PORT MAP ( X(3), Y(3), C(3), S(3), Cout )
34
35 END Structural ;
```

We declare an entity as usual with its Port (lines 11-15). Then on line 18 we state that we are using the Structural architecture. We must now declare any components we are going to connect together, in this case there is only one (lines 21-27) although we shall use it 4 times.

The actual circuit construction takes place lines 30-33 where we create four examples of a fullAdderSlice component and label them (stage0 to stage3) following the block diagram above. Here we map the port values for each

component, they are given in lines 22-26. Let's start with stage0. The first port value is input A. We assign this to X(0) which is the lowest bit of the first number we input into our 4-bit adder (see line 12). The next port value, input B comes from Y(0), the lowest bit of the second number we input into this adder (line 13). Then we assign the next port value Cin from the adder Cin (line11). The next port value is the Sum out, so we assign this to S(0) the lowest bit of our added (line 14). Finally we capture C(0) the carry output from stage0. This local variable is declared on line 19.

Now we proceed left-to-right along the chain. Glance at the block diagram above. At stage1 we input the next adder bit X(1) for the first number and the next bit Y(1) for the second number, then we input any carry C(1) from stage0. Finally we send the sum to our adder sum S(1) and store any carry from this bit in C(2) ready to go into stage2. And so on to complete the chain.

A testbench waveform is shown below where we have added the following pairs of numbers: A = "0110" B = "0001", then A = "0110" B = "0010", then A = "0110" B = "0100", then A = "0110" B = "0110" and finally A = "0110" B = "0111". You can see the adder is working correctly.