ENCS 533 – Advanced Digital Design
Lecture 9
More about RTL coding (Sequential VHDL) and Memories

1 Introduction

In this lecture we will look at two separate topics that will be necessary for you to begin the assignment. The first is how to code systems that use flip-flops, registers, and latches. It is often not necessary to explicitly instantiate flip-flops in our design. Instead, it is usually possible to write behavioural code that specifies the required timing behaviour, and the synthesis tool will then infer where the flip-flops should go when the behaviour is converted into a hardware realisation. The second issue is the description of *memory* devices.

2 Defining clocks

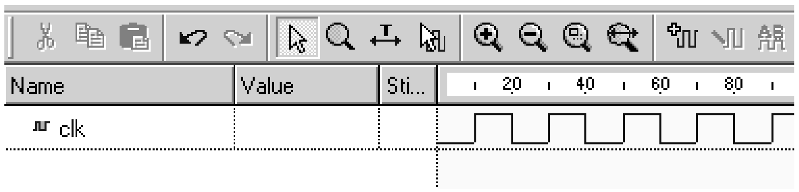
A clock is essentially a signal that alternates between 1 and 0. Here is one way to define such a signal:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mytest IS
END ENTITY mytest;

ARCHITECTURE simple OF mytest IS
    SIGNAL clk: STD_LOGIC:='0';
BEGIN
    clk <= NOT clk AFTER 10 NS;
END ARCHITECTURE simple;
```

The basic idea is this. The statement shown in bold recomputes its right hand side every time its left hand side changes. Each time the statement runs, it queues a transition to take place 10 ns later. When this transition takes effect, the resulting change in the RHS value forces the statement to run again. And so on, ad infinitum.



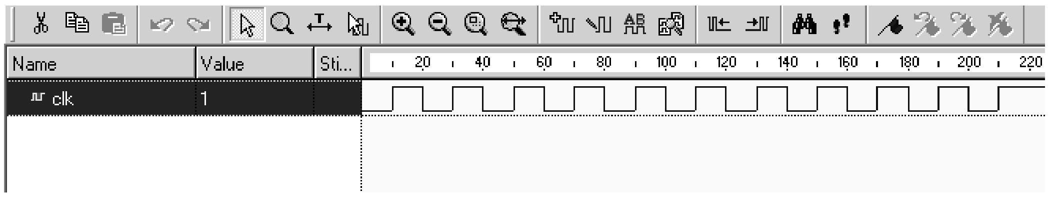
In order to make this work, we had to initialise the value of clk to ‘0’ or ‘1’. If we had left clk uninitialized, then at the start of the simulation clk would have the value ‘U’. This is a form of garbage value, and in VHDL any logical operation on a garbage input produces a garbage output. Specifically, clk <= NOT ‘U’ would result in clk receiving the value ‘U’. So clk would be permanently stuck at ‘U’.

2.1 Another clock definition

Here is another way that we could define a clock signal.

```
ARCHITECTURE number2 OF mytest IS
    SIGNAL clk: STD_LOGIC;
BEGIN
    PROCESS
    BEGIN
        FOR i IN 0 TO 10 LOOP
            clk <= TRANSPORT '0' AFTER i * 20 NS;
            clk <= TRANSPORT '1' AFTER i * 20 NS + 10 NS;
        END LOOP;
        WAIT;
    END PROCESS;
END ARCHITECTURE number2;
```

The PROCESS has no sensitivity list, so it runs as soon as simulation starts. The loop queues 20 transitions on clk to take place at times 0, 10, 20, … ns. If we had omitted TRANSPORT keyword, the VHDL would by default have used INERTIAL delay, which means that all the transitions being placed on the queue would have overwritten each other, and clk would never have changed. Here is a simulation of the resulting output:



2.2 Detecting the clock edge

In order to write descriptions of edge triggered devices, we need some way in VHDL to represent the edge of a clock signal. There are two ways we can do this:

2.2.1 Signal attributes

Signals within VHDL have various attributes that can be used. The syntax for referring to the value of an attribute is `SignalName'Attribute`. The apostrophe, used to separate the name from the attribute is pronounced "tick". Here are some examples of useful attributes:

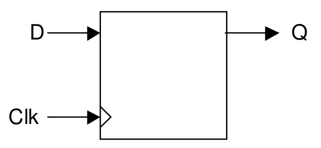
- clk'EVENT is TRUE if clk has changed its value in the last delta, and FALSE otherwise
- clk'STABLE is TRUE if clk has not changed its value in the last delta, and FALSE otherwise.
- clk'STABLE (5ns) is TRUE if clk has not changed its value for the last 5 ns, and FALSE otherwise.

2.2.2. The rising_edge function

The *rising_edge* function is contained in the `STD_LOGIC_1164` package, and returns `TRUE` when clk has changed from 0 to 1 during the last *delta*.

3 The D-type flip-flop

The basic device that is used to accomplish synchronous operation is the D-type flip flop.



Here is the ENTITY definition for this device

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dff IS
    PORT ( d, clk  : IN  STD_LOGIC; Q  : OUT STD_LOGIC);
END ENTITY dff;
```

The behaviour of this device is as follows. When the clock is stable, Q simply holds its value constant. When a rising clock edge occurs, the output Q takes on the value that D has at the moment when the clock edge occurred. It then holds that value constant until the next rising clock edge occurs, at which time it updates itself again.

Note that the output q does *not* update its value whenever d changes. So to write this

```
ARCHITECTURE wrong OF dff IS
BEGIN
    q <= d;
END ARCHITECTURE wrong;
```

would give completely the wrong behaviour. Here is an architecture that correctly describes the behaviour of the device:

```
ARCHITECTURE correct OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF ( rising_edge(clk) ) THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE correct;
```

Whenever *clk* changes its value, the process will run. However, clk might have changed due to a *falling* edge of the clock (which should not trigger an update to q) so we need to insert an IF statement which causes q to be updated only on the *rising* edge of clk.

Here is another way to express the D-type behaviour

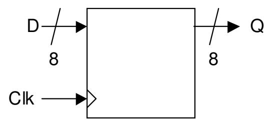
```
ARCHITECTURE number2 OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF ( clk'EVENT and clk='1' ) THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE number2;
```

The rising edge is recognised by the fact that clk'EVENT is TRUE (so clk must have just changed) and clk='1' (so it must have been a rising edge). Here is yet another approach:

```
ARCHITECTURE number3 OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF ( NOT clk'STABLE and clk='1' ) THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE number3;
```

3.1 Registers

A register is a group of D-type flip-flops, all governed by the same clock. So here is an 8-bit register:



When there is a rising edge of the clock

The value of D(7 downto 0) *at that precise instant* is read into the device

That value appears at Q(7 downto 0) a moment (i.e. one delta) *later*.

At other times

Q(7 downto 0) holds it old value

4 More about types and declarations

4.1 Types

All signals have a *type*. The type tells VHDL what sort of values a signal can assume. So far we have met type `STD_LOGIC` (which takes values '0', '1', 'X' and 'U') and type `STD_LOGIC_VECTOR`, which is an array of `STD_LOGIC`.

There are also several other *types* available, such as `INTEGER` (which describes signals whose value is a whole number, e.g. 1 or 9 or -50) and `CHARACTER` (which describes signals whose value is a character, e.g. 'a' or 'B' or 'z'). Here is an example that shows the declaration of a character signal, an integer signal and a `STD_LOGIC` signal.

```
ARCHITECTURE simple OF example IS
    SIGNAL a: CHARACTER;
    SIGNAL b: INTEGER;
    SIGNAL c: STD_LOGIC;
BEGIN
END ARCHITECTURE simple;
```

4.2 Initialization during declaration

When we declare a signal we can also give it an initial value, using the := operator. So here are examples of declaration of signals with initialisation:

```
ARCHITECTURE initialised OF example IS
    SIGNAL a: CHARACTER :='H';
    SIGNAL b: INTEGER :=5;
    SIGNAL c: STD_LOGIC :='X';
BEGIN
END ARCHITECTURE initialised;
```

4.3 Arrays

You will have met arrays in other languages, such as C. VHDL also has arrays. An array is a list of items, all of which have the same type, which are indexed by a number.

Suppose, for example, that I frequently use lists of 11 characters. I could set up a *type* which describes this sort of data. This would be done as follows:

```
TYPE list11 IS ARRAY (0 TO 10) OF CHARACTER;
```

I have used *list11* as the name of this type. Now, when I declare a signal, I can use my new type. Suppose I want to create two 11-item lists, one called a the other called b. I could declare them like this:

```
SIGNAL a, b: LIST11;
```

Now suppose I want to initialise these signals as I declare them. The first will contain the characters *Hello there* and the second will contain *How are you*
As before, we introduce an initialisation using the symbol :=. This time the initialisation isn’t a single item, but a list. We write each member of the list separated by commas. The entire list is contained in brackets:

```
SIGNAL a: LIST11 :=(‘H’,’e’,’l’,’l‘,’o’,’ ’,’t’,’h‘,’e’,’r’,’e’);
SIGNAL b: LIST11 :=(‘H’,’o’,’w’,’ ‘,’a’,’r’,’e’,’ ‘,’y’,’o’,’u’);
```

So here is a complete example showing the declaration and initialisation of a and b:

```
ARCHITECTURE simple OF example2 IS
    TYPE list11 IS ARRAY (0 TO 10) OF CHARACTER;
    SIGNAL a: LIST11 :=('H','e','l','l','o',' ','t','h','e','r','e');
    SIGNAL b: LIST11 :=('H','o','w',' ','a','r','e',' ','y','o','u');
BEGIN
END ARCHITECTURE simple;
```

So in this example a(0) has the value ’H’, a(1) is ‘e’, a(11) is ‘e’ and so on.

4.4 Type conversion

VHDL is a *strongly typed* language. This means that if a and b are of different types, you can’t just write

a <= b;

Instead you need to use a conversion function, which tells VHDL how to convert between two incompatible types. For example, suppose we have

```
ARCHITECTURE wrong OF example3 IS
   SIGNAL a: INTEGER;
   SIGNAL b: STD_LOGIC_VECTOR(7 DOWNTO 0) := X"FF";
BEGIN
   a <= b;   --WRONG !!
END ARCHITECTURE wrong;
```

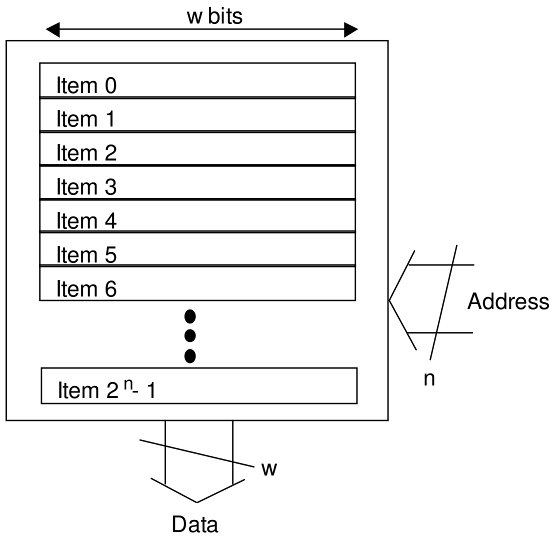
This would not compile correctly. The basic reason for this is that a and b are of different types. One of the reasons why VHDL won't let you directly assign unalike types is in order to protect you from error. As we saw in lecture 1, it is not clear whether FF represents 255 or –1. To convert between integer and std_logic, there is a function called `CONV_INTEGER`. There are two different versions of this, one held in the sub-library[1] `IEEE.STD_LOGIC_UNSIGNED`, and the other in `STD.LOGIC_SIGNED`. You have to open one of these up with a `USE` statement before you can use the `CONV_INTEGER` function. So here is a correct listing:

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ARCHITECTURE correct OF example3 IS
   SIGNAL a: INTEGER;
   SIGNAL b: STD_LOGIC_VECTOR(7 DOWNTO 0) := X"FF";
BEGIN
   a <= CONV_INTEGER(b);
END ARCHITECTURE correct;
```

5 Memories

A very simple example of a read-only memory (ROM) looks like this



The memory contains a series of 2^n items, each of which is w bits wide. An n-bit address input selects out one of the items, which is steered to the data output.

5.1 An example

As a specific example, imagine that we wanted to store the marks for 15 students in a ROM. Here are the marks:

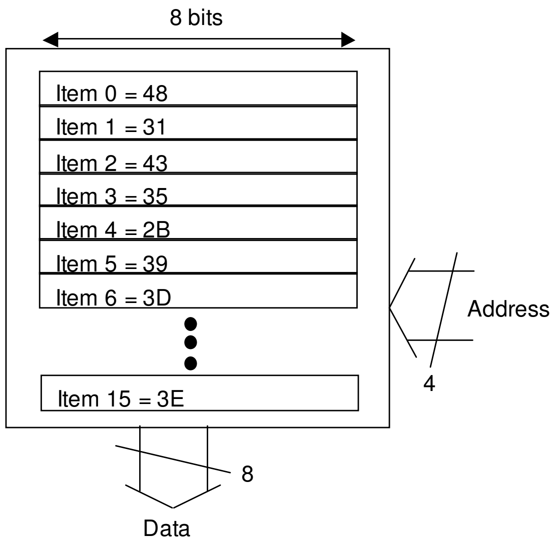
[1] In the jargon of VHDL, a sub-library is called a *package*.

Student	Mark
0	72
1	49
2	67
3	53
4	43
5	57
6	61
7	37
8	48
9	55
10	79
11	51
12	40
13	61
14	58
15	62

Things become much easier if we express this data in hexadecimal rather than denary, so here is the table translated to hex:

Student	Mark
0	48
1	31
2	43
3	35
4	2B
5	39
6	3D
7	25
8	30
9	37
A	4F
B	33
C	28
D	3D
E	3A
F	3E

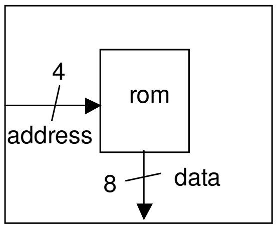
We now store each of these items in the storage locations inside the ROM:



The index of the list runs from 0 to 15 denary (0 to F hex), which needs four binary bits (i.e.one hex digit) to represent. To represent the number from 0 to 100 denary (0 to 64 hex) could be done in 7 bits, but it's often easier to keep the width of our signal as a power of 2, so we choose 8 bits to represent the marks.

5.2 The ENTITY of the ROM example

Firstly we need to write the ENTITY declaration. Here are the inputs and outputs of the ROM.



So here is an ENTITY declaration:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY rom IS
      PORT ( address: IN   STD_LOGIC_VECTOR(3 DOWNTO 0);
             data   : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0) );
END ENTITY rom;
```

5.3 The ARCHITECTURE of the ROM example

The contents of the ROM is essentially a list of 8-bit data items, which are indexed by the address. This corresponds to an *array*. We will create an array called *rom_data*. The various items in this list will be the marks of the students, stored as 8-bit binary numbers (i.e. 2-digit hex numbers). So for example:
rom_data(2) = 43
rom_data(6) = 3D
and so on.

So here is the architecture description for the ROM:

```
ARCHITECTURE simple OF rom IS
    TYPE rom_array IS ARRAY ( 0 TO 15 ) OF STD_LOGIC_VECTOR ( 7 DOWNTO 0 );
    SIGNAL rom_data: rom_array := ( X"48",  X"31",  X"43",  X"35",
                                    X"2B",  X"39",  X"3D",  X"25",
                                    X"30",  X"37",  X"4F",  X"33",
                                    X"28",  X"3D",  X"3A",  X"3E" );
BEGIN
      data <= rom_data ( CONV_INTEGER (address) );
END ARCHITECTURE simple;
```

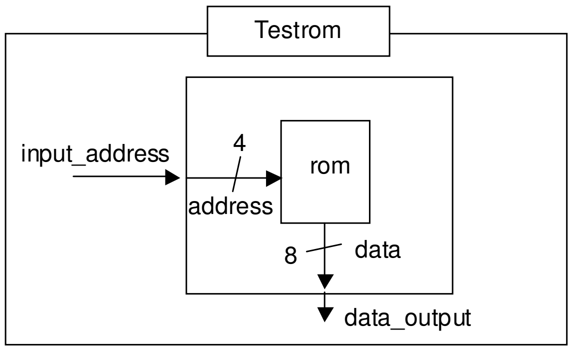
The statement

```
      data <= rom_data ( CONV_INTEGER (address) );
```

takes the member of the list that is indexed by *address* and copies into the *data* output (after converting it from `STD_LOGIC_VECTOR` to integer, so as to be a valid index for the array).

5.4 A testbench for the ROM example

The test bench looks like this:



We will generate addresses and apply them to the ROM, and look at the data that emerges to see if it corresponds to the table of students' marks. Here is a possible testbench:

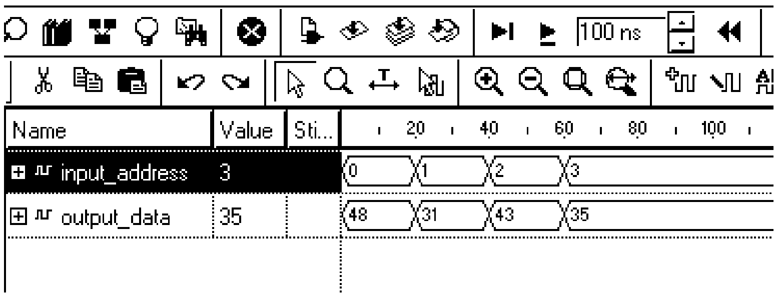
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY rom_test IS
END ENTITY rom_test;

ARCHITECTURE simple OF rom_test IS
    SIGNAL input_address: STD_LOGIC_VECTOR (3 DOWNTO 0);
    SIGNAL output_data: STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
    g1: ENTITY work.rom(simple)
        PORT MAP ( address=>input_address, data=>output_data);

    input_address <= X"0",
                     X"1" AFTER 20 NS,
                     X"2" AFTER 40 NS,
                    X"3" AFTER 60 NS;
END ARCHITECTURE simple;
```

And here is the result of running the testbench through the simulator:



Name	Value	Waveform
input_address	3	0, 1, 2, 3
output_data	35	48, 31, 43, 35

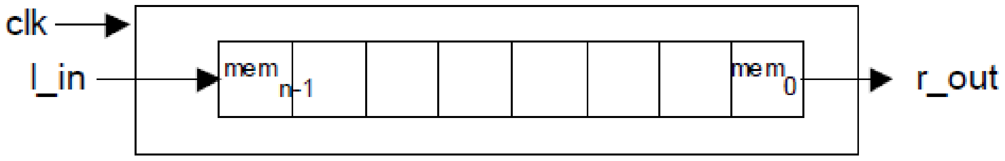
The results give us confidence that the design is working correctly; as we apply the ID of the student to the input, we have that student's mark returned at the output.

6 Shift registers

Another important class of devices is the shift register. Here are a few typical "standard patterns" of VHDL description for these devices.

6.1 Basic shift register

The basic shift register contains n memory bits.



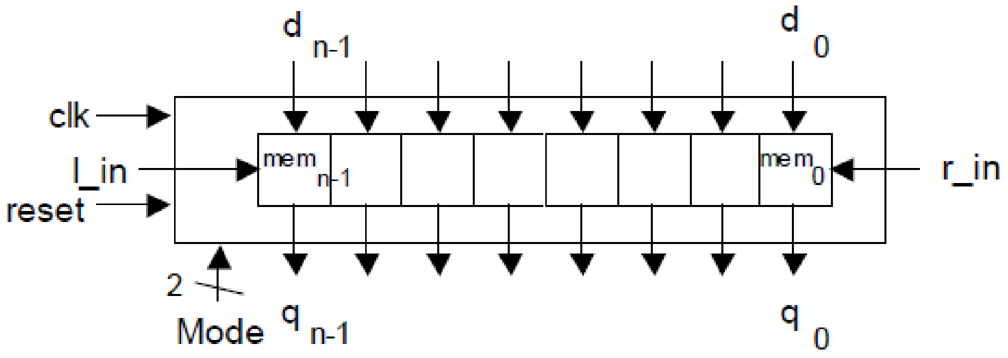
At each clock cycle, the content of the memory shifts one stage to the right. This can be coded as follows

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;

ENTITY shift IS
     PORT ( clk: IN STD_LOGIC;
          l_in : IN STD_LOGIC;
          r_out : OUT STD_LOGIC );
          CONSTANT n: INTEGER:=8;
END ENTITY shift;
```

```
ARCHITECTURE simple Of shift IS
SIGNAL mem : STD_LOGIC_VECTOR (n-1 DOWNTO 0);
BEGIN
     PROCESS ( clk )
     BEGIN
          IF rising_edge (clk) THEN
               r_out <= mem(0);
               mem (n-2 DownTo 0) <= mem (n-1 DownTo 1);
               mem (n-1) <= l_in;
          END IF;
     END PROCESS;
END ARCHITECTURE simple;
```

6.2 A more elaborate shift register
Now we extend the example to a more elaborate and fully featured shift register.



The contents in the shift register are held in the memory array called *mem*. If the reset input goes high, then the device is asynchronously reset. In order to extend the flexibility of the device, we make n a generic parameter. The function of the shift register is established by the mode input. The device has four modes:

00: hold. The memory array is preserved unchanged.
01: shift left: The memory array shifts one position to the left . Mem0 receives its value from r_in, the right input.
10: shift right: The memory array shifts one position to the right . Memn-1 receives its value from l_in, the left input.
11: load: The memory array undergoes a parallel load from the d input.

Here is the corresponding VHDL code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY shifter IS
     GENERIC ( n: POSITIVE:=8);
     PORT ( clk, reset: IN STD_LOGIC;
          l_in, r_in: IN STD_LOGIC;
          mode: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
          d: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
          q: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
END ENTITY shifter;
```

```
ARCHITECTURE simple OF shifter IS
SIGNAL mem: STD_LOGIC_VECTOR(n-1 downto 0);
BEGIN
     PROCESS (clk,reset)
     BEGIN
          IF (reset='1') THEN
               mem <= (others => '0');
          ELSIF rising_edge(clk) THEN
               CASE mode IS
                    WHEN "00" => NULL;

                    WHEN "01" => mem(n-1 DOWNTO 0) <=
               mem(n-2 DOWNTO 0) & r_in;

                    WHEN "10" => mem(n-1 DOWNTO 0) <=
               l_in & mem(n-1 DOWNTO 1);

                    WHEN "11" => mem(n-1 DOWNTO 0) <=
               d(n-1 DOWNTO 0);

                    WHEN OTHERS => NULL;
               END CASE;
          END IF;
     END PROCESS;
     q <= mem;
END ARCHITECTURE simple;
```

Null is a keyword of VHDL, which means “do nothing”.

6.3 Clarifying the code by using constants to provide names

It may make the code easier to understand if we give each of the modes a name, like this:

```
ARCHITECTURE nicer OF shifter IS
SIGNAL mem: STD_LOGIC_VECTOR(n-1 downto 0);
CONSTANT hold: STD_LOGIC_VECTOR(1 downto 0):= "00";
CONSTANT sh_left: STD_LOGIC_VECTOR(1 downto 0):= "01";
CONSTANT sh_right: STD_LOGIC_VECTOR(1 downto 0):= "10";
CONSTANT load: STD_LOGIC_VECTOR(1 downto 0):= "11";
BEGIN
     PROCESS (clk,reset)
     BEGIN
          IF (reset='1') THEN
               mem <= (others => '0');
          ELSIF rising_edge(clk) THEN
               CASE mode IS
                    WHEN hold => NULL;
                    WHEN sh_left => mem(n-1 DOWNTO 0) <=
               mem(n-2 DOWNTO 0) & r_in;
```

```
                    WHEN sh_right => mem(n-1 DOWNTO 0) <=
               l_in & mem(n-1 DOWNTO 1);
                    WHEN load => mem(n-1 DOWNTO 0) <=
               d(n-1 DOWNTO 0);
                    WHEN OTHERS => NULL;
               END CASE;
          END IF;
     END PROCESS;
     q <= mem;
END ARCHITECTURE nicer;
```

Summary
In this lecture we have looked in more depth at different types in VHDL. We have looked at how ARRAYs can be used to respresent memory structures. We also looked at clock generation and RTL coding

You should now know...

How to form simple memory structures using ARRAYs.
Memories in VHDL
Clock generation for VHDL simulation
Shift registers