Experiment 28 - Digital Electronics with Altera

Part I: The Pre-Lab Test

Department of Electrical Engineering & Electronics

April 2022, Ver. 8.1

Experiment Specifications

Module(s)	ELEC211
Experiment code	28
Semester	2
Level	5
Subject(s) of relevance	Digital Logic Design, FPGAs and Verilog
Assessment method	Pre-lab online test in Canvas, which is worth 30%
Submission deadline	**See Canvas for the formal Pre-Lab Test deadline**

Instructions: Read this document carefully and **attempt the online pre-lab test available on Canvas** (which is worth 30%).

Regardless of the Pre-Lab Test deadline, you should read this document and attempt the Pre-Lab Test questions before your allocated experiment day (even if you do not actually submit the test yet).

You are also encouraged to download and install Quartus II at home (see instructions in main script) to get some practice in advance of the lab.

1 Introduction

This experiment will take you through a tutorial on how to program your design project in the Altera ***Quartus II*** FPGA (Field Programmable Gate Array) development package, and an awareness of the **DE1** development board (Figure 1) which incorporates the Cyclone II FPGA.

The Experiment script and supporting information will guide you step-by-step through design processes such as schematic capture, simulation and planning pins on the board itself. The purpose of this Prelab is different: to get you used to the **Verilog** 'hardware description language' (HDL), an alternative design method to schematic capture. You will use Verilog in the middle and later stages of the Experiment script.

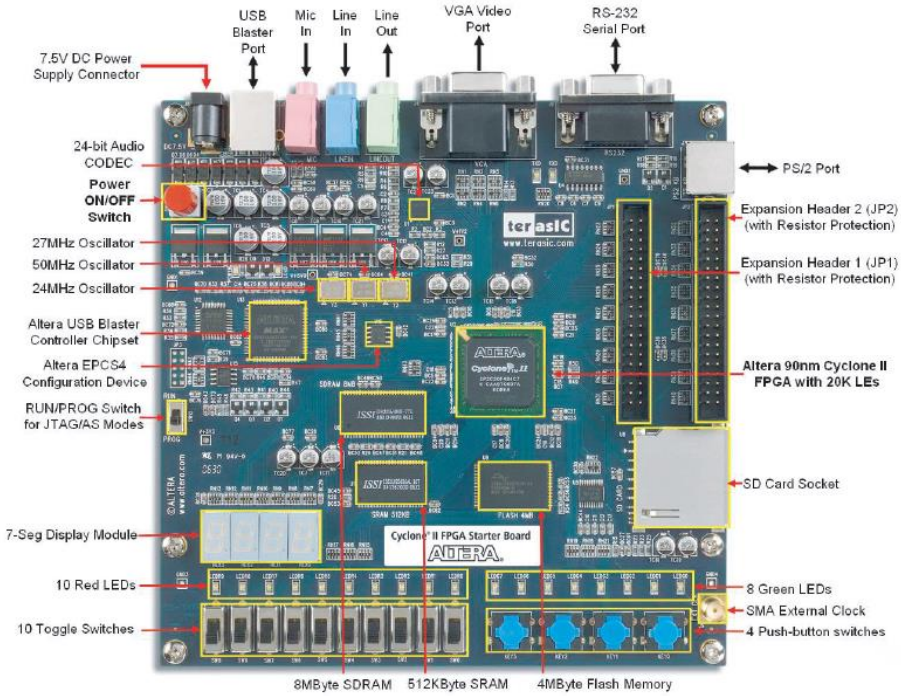

Figure 1. Altera DE1 Board

1.1 Schematic Capture

Schematic capture is one of the common methods for entering a design. This is available in Quartus II. It involves using a design tool to draw a schematic of the desired circuit. This is probably the easiest method of design capture to understand, but it is not necessarily the most efficient. Imagine your aim is to design the AND gate with two inputs A and B and one output Z, as shown in Figure 2.

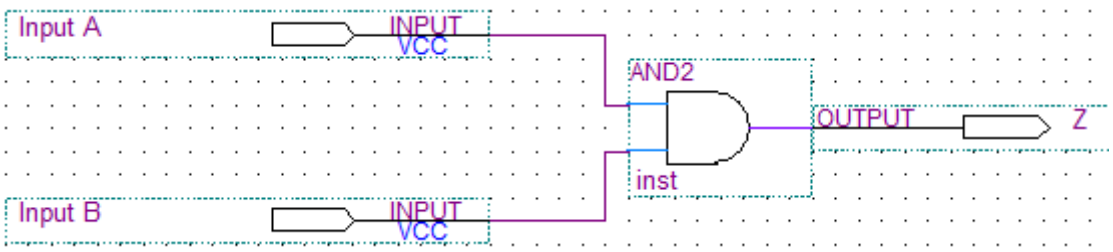

Figure 2. AND gate

We could perhaps summarise the operation of the above circuit as follows:

input A, B;
output Z;
Z = A & B;

where **&** represents the Boolean **AND** operator.

1.2 Verilog combinational design – Verilog modules

A Hardware Description Language (HDL) is an efficient language with which to program your design objectives – a schematic description in text. We cover an IEEE industry standard HDL called Verilog in this experiment. We can create Verilog HDL files (**.v**) in Quartus II. Both combinational and sequential systems can be entered. Starting with combinational systems, if we want to create an AND gate similar to the one in Figure 2 from scratch, one way to write the Verilog code is as follows:

```
module ANDAB(Z, A, B);
      input A,B;
      output Z;
      assign Z = A & B;
endmodule
```

In Verilog, any circuit is thought of as a "**module**". Above, we named our module "ANDAB", and in the module declaration in brackets, we specified circuit 'ports', which we defined as inputs and an output in the two lines below the declaration. We then used the **&** operator (logical AND) to **assign** a value to the output wire (Z) based on the inputs (A, B). Finally, we ended our module with the keyword "**endmodule**".

We could achieve the same thing using the Verilog primitive structure "**and**" as follows:

```
module ANDAB(Z, A, B);
      input A,B;
      output Z;
      and(Z, A, B);
endmodule
```

Looking at the bolded line above, notice that to use the built-in primitive "and", we must specify the output wire, Z, first in the brackets, followed by the input wires A, B.[1] We

[1] In Quartus II, if you are working directly on a Block Diagram File (BDF – for Schematic Capture) instead of a Verilog HDL file, you don't need to create your 'own' 2-input AND gate in Verilog as seen here. The 2-input AND gate is available in the list of on-board Quartus II primitive symbols. Double-click anywhere in your BDF to enter a symbol, then navigate to primitives → logic → **and2**.

could then save our Verilog module file by a convenient name, e.g. **ANDAB.v**. *Note, though, that if we are designing the overall project from a given Verilog (.v) or schematic (.bdf) file, it's usually safest to save this file – known as the 'top-level design entity' – by the same name as the project. More on this later and in the main script.*

Finally, a more compact way to define the ports and their directions is to use inline definition at the point of declaration, as follows:

```
module ANDAB(output Z, input A, B);
      and(Z, A, B);
endmodule
```

This module achieves exactly the same as the previous example. By the way, the port signal direction can be input, output or 'inout' (bi-directional port). This Pre-Lab script uses the more compact mode of port definition, but you can use whichever convention you are comfortable with – the line by line approach may be slightly less error-prone.

1.3 Implementing Boolean expressions and equations

Here are some of the logical operators and primitive structures used for operations and gates in Verilog. **In this list, assume that Z is the output, and A, B are inputs:**

Structure/ operator:	**Example:**	**Description:**
-	assign Z = -A;	two's complement (negative/sign change)
~	assign Z = ~A;	one's complement (prefix inverter; NOT)
not()	not(Z, A);	inverter (NOT)
buf	buf(Z, A);	buffer
&	assign Z = A & B;	AND
and()	and(Z, A, B);	AND
~(&)	assign Z = ~(A & B);	AND inverter (NAND)
nand()	nand(Z, A, B);	NAND
|	assign Z = A | B;	OR
or()	or(Z, A, B);	OR
~(|)	assign Z = ~(A | B);	OR inverter (NOR)
nor()	nor(Z, A, B);	NOR
^	assign Z = (A ^ B);	XOR (exclusive OR)
xor()	xor(Z, A, B);	XOR (exclusive OR)
~(^)	assign Z = ~(A ^ B);	XOR inverter (XNOR)
xnor	xnor(Z, A, B);	XNOR (exclusive NOR)

Table 1. Some gates and operators in Verilog

You can use either the worded structure, or the operator symbol, as shown above.

Each primitive or operator above represents a multi-input logic gate (two or more inputs), except:

- NOT [~ or not()], which is a prefix inverter on a single node
- the 'negative' symbol (-)

The above operations need not be on a single bit. For example, if Z and A are defined as 4-bit buses, as follows, then every bit in A would be inverted, bitwise (e.g. if A = 0101 then Z = 1010):

```
module INVERTBUS (output[3:0]Z, input[3:0]A);
      assign Z = ~A;
endmodule
```

Z is defined above as a 4-bit output bus, by means of "output[3:0] Z;". The outputs can be referred to individually using Z[3], Z[2], Z[1] and Z[0].

Note: we haven't mentioned the data type so far. In Verilog, not mentioning the data type means that the data is assumed to be of type 'wire' – more on this below.

Here is another example which implements a simple Boolean expression:

```
module ANDINVERSE (output Z, input A, B);
      assign Z = A & ~B;
endmodule
```

This module describes a circuit performing the operation A AND $\overline{B}$ (Figure 3):

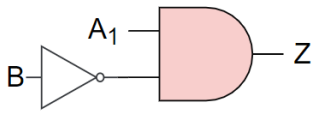

Figure 3. Basic logic circuit

If desired, we could implement this 'bitwise' as follows:

```
module ANDINVERSEBW (output[3:0]Z, input[3:0]A, B);
      assign Z = A & ~B;
endmodule
```

The circuit described by this code would perform $Z_n = A_n . \overline{B_n}$ bitwise, for n = 0 to 3. So if we have two 4-bit input buses, A, B, with A = 0110 and B = 1100, then Z = 0010.

We could, alternatively, use the worded forms of the primitives (see Table 1), if we want to instantiate our gates more obviously and explicitly in the code:

```
module ANDINVERSEBW (output[3:0] Z, input[3:0] A, B);
      not(w,B);
      and(Z,A,w);
endmodule
```

This describes the same circuit as the previous code.

Note: you may be wondering what 'w' is in the above example. It's a wire – a connecting signal between gates which is not a port to the whole circuit (it's not an input or output, for example.) The above code shows that we actually don't have to declare wires – any component we didn't declare is automatically assumed to be a wire. So we didn't have to declare w; we just used it.

In fact, our circuit outputs and inputs are implicitly wires, too. This is an example of 'data type' as discussed in the previous note; more on this later. (By the way, you don't have to use the letter 'w' for a wire!) In the example above, 'w' is used to refer to the signal leaving the NOT gate, which is then ANDed with A to produce the circuit output Z.

Of course, we are not limited to only having one circuit output:

```
module ANDINVERSE (output Y, Z, input A, B);
      not(w,B);
      and(Z,A,w);
      or(Y,A,B);
endmodule
```

This code describes the circuit in Figure 4, with two input ports and two output ports:

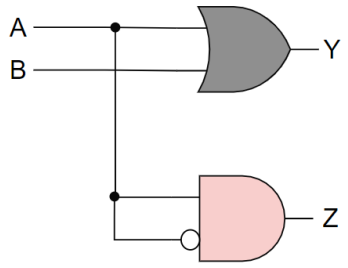

Figure 4. Logic circuit with two outputs

(Recall from lectures that the inversion bubble is just shorthand for a NOT gate.)

To describe what is happening in words: The **Z** output is driven by the logical AND of A with the inverse of B, and the **Y** output is driven by the logical OR of A with B. Since these equations are evaluated concurrently, their order in the file is not important.

1.4 More complex designs

1.4.1 Wires

The purpose of a HDL is to describe a logic circuit in code. Wires are an important part of this. Having declared the module interface (inputs / outputs) and instantiated the gates, you will need internal wires to connect the gates in more complex circuits than the above. Most wires are thus effectively internal signals in the circuit.

Take this simple 'majority circuit' as an example (Figure 5). A majority circuit outputs a logic HIGH if more than two of the input signals are at logic HIGH:

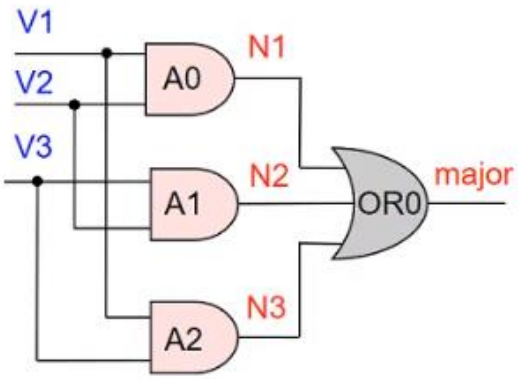

Figure 5. Majority circuit

Now let's describe this circuit in Verilog code, making our gates and wires very explicit:

```
module majority(output major, input V1, V2, V3);
      wire N1, N2, N3;
      and A0(N1, V1, V2);
      and A1(N2, V2, V3);
      and A2(N3, V1, V3);
      or OR0(major, N1, N2, N3);
endmodule
```

Notice how we declared three wires (N1, N2, N3) to represent internal, intermediate signals. We defined what these wires connect with what, by:

- entering them as outputs of the AND gates (remember that with primitives, the outputs **always** come first in the brackets)
- entering them as inputs of the OR gate (same comment applies)

As we mentioned above, the order of the statement lines which describe the gates is not important, since we are building a circuit, not defining a sequential algorithm.

There's also a really handy simplification. As noted previously, **you don't actually need to declare wires.** Identifiers (names) which have not been explicitly declared are assumed by default to be wires. So the following achieves the same as before:

```
module majority(output major, input V1, V2, V3);
      and A0(N1, V1, V2);
      and A1(N2, V2, V3);
      and A2(N3, V1, V3);
      or OR0(major, N1, N2, N3);
endmodule
```

1.4.2 Identifiers – naming gates, wires, ports, modules

Notice, in the code above, how we have now started **naming our gates** (A0, A1, OR0, etc) before the brackets mentioning the wires they operate on. This is useful for keeping track of all our components while designing, and for debugging. Component names ('identifiers') can only include digits 0-9, characters a-z, _ and $. However, **names must not start with digits or $**, and can't include spaces.

To avoid problems in Quartus II, **apply the above rules to module names too.**

Input/output ('port') names are separated by commas in the module declaration brackets. (As noted before, there is a third kind of port which is bi-directional, declarable with the keyword 'inout'; we won't be considering this type of port in Experiment 28.) Similarly, if/when declared, wire names are separated by commas. In Verilog, all statements finish with a semicolon – except '**endmodule**'.

1.4.3 Modules as symbol files

Figure 6 shows what our module code above creates, if we insert it in a Quartus II schematic (a schematic is also known as a BDF or Block Diagram File). In Experiment 28, you will learn how to achieve this by creating a symbol file for any module you create. This generates a block, or 'black box', showing the module name, inputs and outputs but not showing the logic and components within:

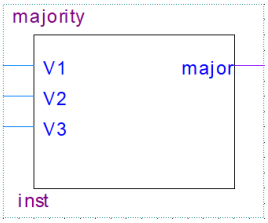

Figure 6. Majority circuit symbol created from Verilog module

We could complete our circuit by adding some (as yet un-named) inputs and outputs as shown in Figure 7:

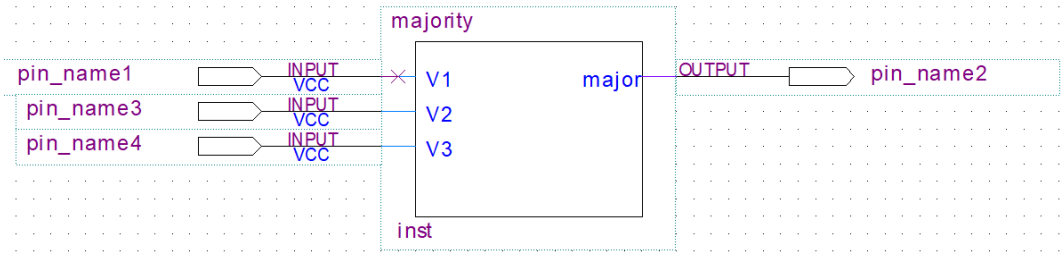

Figure 7. Majority circuit created from Verilog module

We could then go one step further and define what physical pins on the Cyclone II FPGA we wish to correspond to the inputs and outputs of the above circuit (e.g. push-buttons or other nodes). Once again, there is the opportunity to learn this through completing the Experiment.

Finally, we can also inspect the signal data flow in a more detailed schematic, by means of the Quartus II RTL Viewer[2], as seen in Figure 8. Caution should be exercised, as what is seen using the RTL Viewer is not always quite what we would expect on a logic circuit, although in this case the similarities with Figure 5 are obvious:

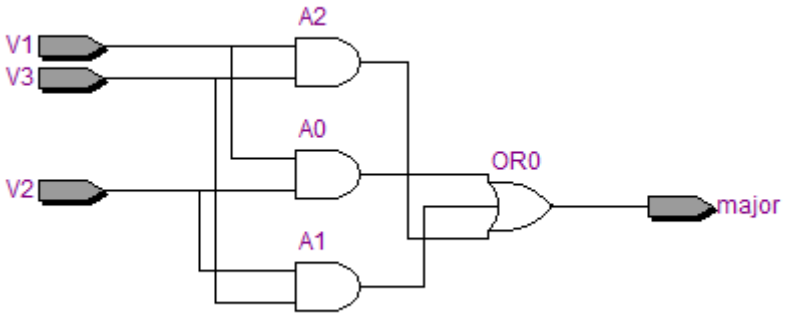

Figure 8: Quartus II RTL view of Majority circuit created from Verilog module

One thing to be aware of in the Quartus II RTL Viewer is that intersection nodes are not clearly signified (e.g. with dots as they might be on a circuit diagram) – you need to click on an individual data path to find out what it truly connects to. This functionality is not shown in Figure 8.

1.4.4 Numbers in Verilog

Numbers are represented by **<size>'<base><number>** , where <size> is the number of bits. Note the apostrophe ' after <size>. Here are some example representations:

Number	Means	(Decimal equivalent)	In binary
2'd3	Decimal 3 (2 bits)	3	11
5'd2	Decimal 2 (5 bits)	2	00010
4'd14	Decimal 14 (4 bits)	14	1110
7'b110	Binary 110 (7 bits)	6	0000110
8'b0110_0101	Binary 01100101 (8 bits)	101	01100101
8'h9	Hex 9 (8 bits)	9	00001001
6'hc	Hex c (6 bits)	12	001100
-8d'4	Decimal -4 (8 bits)	-4	11111100

Notice how, if you do not specify all the bits, leading zeroes are added to the binary form. Secondly, it's good to 'break up' binary numbers with an underscore _ for readability (see the 5th row).

[2] "Register Transfer Level" Viewer (this does not necessarily mean registers in the sense of arrays of flipflops). After compiling a project, you can find this tool in Tools → Netlist Viewers → RTL Viewer. However, it will show only what is in the top-level design entity – see Section 5 below.

1.5 Reusing modules

Let's look at the code for a half adder. Here's the circuit (Figure 9):

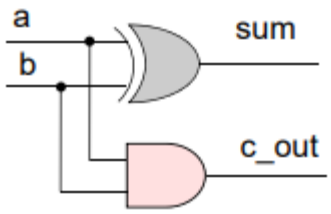

Figure 9. Half Adder circuit

Here is the Verilog code:

```
module add_half (output sum, c_out, input a, b);
      xor (sum, a, b);
      and (c_out, a, b);
endmodule
```

Now that we've created this module, we can reuse it in another module. For example, let's say we want to create a **full adder** out of two half adders as follows (Figure 10):

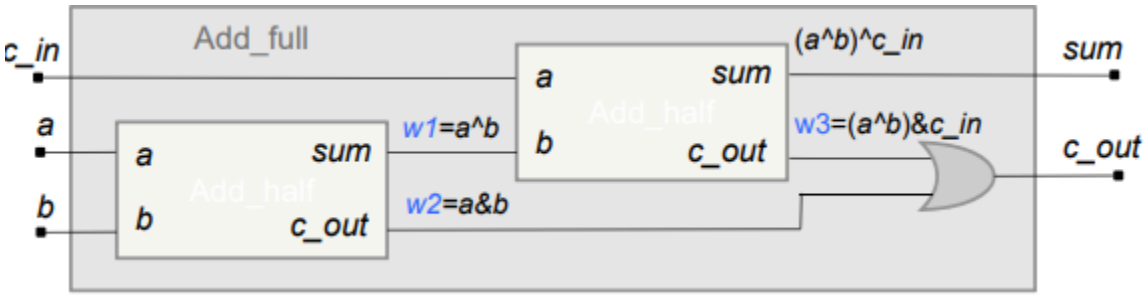

Figure 10. Full Adder from two Half Adders.

Remember, the **^** symbol denotes XOR; the **&** symbol denotes AND. (You can check that the sum and c_out logic here matches up with the logic of the circuitry shown in ELEC211 Lecture 14 Part 1, slide 24.)

To describe this system, we can use the following Verilog code:

```
module add_full (output sum, c_out, input a, b, c_in);
      wire w1, w2, w3;
      add_half M1 (w1, w2, a ,b);
      add_half M2 (sum, w3, c_in , w1);
      or (c_out, w2, w3);
endmodule
```

In the bolded lines, we have created instances of the **add_half** module we designed previously. M1 and M2 are names we've given to each instance of the module. Note that on this occasion we have chosen to explicitly declare the wires w1, w2 and w3 – but as discussed before, we didn't have to do this in order to use them.

1.6 Behavioural modelling in Verilog

Instantiating gates in words is a very clear way to code, but it takes a bit longer***. Let's return to using operators*** such as &, |, etc., along with the "assign" keyword.

Even using such operators, the more complex the circuit is, the longer it takes to code. So far we have been using **'structural modelling'** – coding at a very low level in which we specify all gates, etc. We can simplify our code greatly by using **'behavioural modelling'**. This approach still ultimately describes hardware, but it's a bit like programming with a compiler instead of programming in assembly code. Instead of explicitly mentioning the required hardware components, we describe what behaviour we want to be achieved in terms of circuit inputs and outputs – and rely on our synthesis engine (such as the Quartus II synthesiser) to implement this efficiently. So we are 'zooming out' to a higher level of abstraction by describing the functionality (behaviour) but leaving the structure as a 'black box' for the synthesis engine to work out. In practice, modern designs tend to use a mixture of structural and behavioural modelling.

1.6.1 Example: 4-1 MUX

We can describe a 4-1 multiplexer with the structural modelling techniques discussed already (but now using operators instead of explicit gate instantiations in words)[3]:

```
module mux421(output out, input i0, i1, i2, i3, s0, s1);

      assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) |
                   (s1 & ~s0 & i2) |(s1 & s0 & i3);

endmodule
```

Above, we have two select inputs (s0, s1), three data inputs (i0, i1, i2) and an output (out) – altogether, a 4 to 1 MUX. If the signals at ports s0 and s1 are both zero, i0 is selected; if they are both 1, i3 is selected; if they're different, i1 or i2 are selected. Figure 11 shows how Quartus II visualises this data flow in its RTL Viewer mentioned above. This should remind you of 4-1 MUX circuit diagrams seen in lectures:

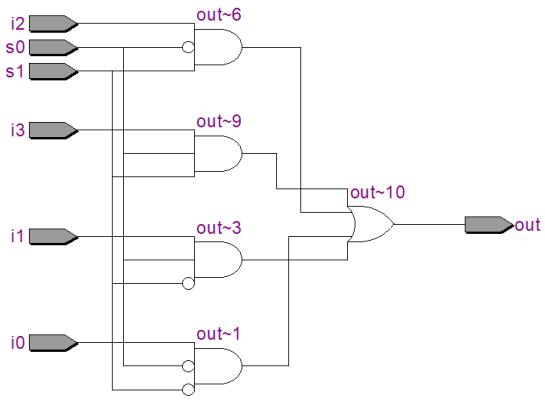

Figure 11. Quartus II RTL view of 4-1 MUX

[3] Two lines are used here for the "assign out" equation due to space constraints in this document – not necessary in itself, although splitting code into multiple lines can be helpful for readability/organisation.

1.6.2 4-1 MUX with behavioural modelling (conditional operator)

Now, let's recreate the same multiplexer but without considering the individual gates at all – let's just say what we want the multiplexer to achieve, using behavioural modelling. The code below uses the ? conditional operator. The syntax is:

```
<condition expression> ? <true expression> : <false expression>;
```

In the module code which follows[4], we create a **nested** statement of conditional operators: in line 3, each bracket of the overall conditional statement is also, itself, a conditional statement:

```
module mux421cond(output out, input i0, i1, i2, i3, s0, s1);

      //use nested conditional operator
      assign out = s1 ? (s0 ? i3:i2) : (s0? i1:i0);

endmodule
```

In the above code, s1 is a condition expression:

- if s1 is true, the red bracket is enacted;
- if s1 is false, the orange bracket is enacted.

Our other MUX select input, s0, is also a condition expression:

- In the red bracket (s1 true): if s0 is true, i3 is selected; otherwise, i2 is selected;
- in the orange bracket (s1 false): if s0 is true, i1 is selected; else, i0 is selected.

By 'selected', we mean that the relevant signal is assigned to the circuit output, 'out'.

Figure 12 shows how Quartus II visualises the data flow in the 4-1 MUX as described in the above way. This looks different from Figure 11, because we have now used logical statements which do not explicitly refer to low-level logic gates. The software has determined that we could achieve our logical objectives with two 2-1 multiplexers:

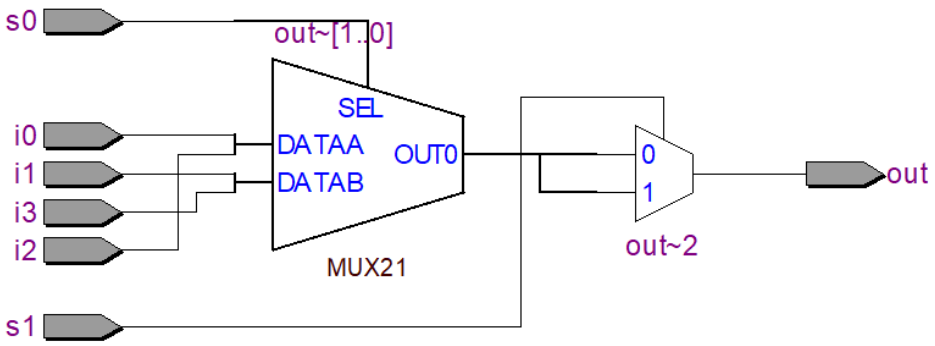

Figure 12. Quartus II RTL view – alternative – of 4-1 MUX

This RTL view is not exactly what we might draw for a logic circuit cascading two 2-1 MUX, but you should be able to spot some clear similarities.

[4] The above code uses Verilog commenting syntax which is the same as C/C++: // for a single line comment; /* … */ for a multiple-line comment.

1.7 More syntax! 'always', reg and parameters

1.7.1 Procedural blocks: 'always' and 'initial'

In Verilog, activity flows run in parallel. Each activity flow starts at simulation time 0. So far, we've written simple combinational code without a sense of time. In behavioural modelling we can use **'always'** blocks to handle time and sequence. Each 'always' block represents a separate activity flow, but statements **inside** the block are "procedural" – meaning they are executed sequentially, a bit like a C program, whenever the block is triggered. Order matters inside the block!

The **always** keyword is used to model a block of activity that is repeated continuously in a digital circuit. This could be combinational or sequential logic, as we will see below. An **always** block starts at time 0 and executes the statements in a loop, triggered by a signal change in the 'sensitivity list' after "**always@**" (for example, a change in input value, or a clock edge).

By contrast, an **initial** statement executes only **once** during a simulation. These blocks are typically used for initialisation, monitoring waveforms and other processes that must be executed a single time during the entire simulation run. Although the block itself is only executed once, statements **inside** an '**initial**' block are, again, executed sequentially. For synthesis with an actual board, initialisation is often done via a reset.

1.7.2 'Registers' (reg) and parameters as variables

Up to now, all our inputs and outputs have been wires ('nets'). Wires are physical entities, but **variables** are used for containing values. Wires cannot store values that are capable of being used in a procedural block such as an **always** or **initial** block.

A common variable type in sequential Verilog design is the 'register' (**reg**). These are particularly useful for storing output values. In Verilog, **registers** are not necessarily the hardware registers we have looked at in lectures, built of groups of edge-triggered flipflops. In Verilog, the term **register** refers to a variable that can hold a value until another value is placed onto it. In the example below, we will consider how **reg** must be used for outputs when we use '**always**' blocks.

Parameter is another kind of variable; this is used to create constants in a module.

1.7.3 Example: priority encoder (always, reg, if-else)

The **priorityenc** module shown below shows a priority encoder that converts the level of the highest-priority active input into a binary value. It generates a 2-bit code that indicates the highest-priority input driven by V_{CC}. This example introduces the use of the **always** block with **begin**/**end**, as well as **registers**, and the syntax of **if-else** statements in Verilog:

```
module priorityenc(output reg [1:0]A, input [3:0]Y);

always@(Y)
      begin
            if(Y[3])
                  A = 2'b11;
            else if(Y[2])
```

```
            A = 2'b10;
        else if(Y[1])
            A = 2'b01;
        else if(Y[0])
            A = 2'b00;
        else
            A = 2'b00;
    end

endmodule
```

Above, the **always** block is triggered by any change in the value of Y. This is depicted by **always@(Y)**. We surround the **always** block with **begin** and **end** keywords, because our block contains more than one statement. These points of syntax move us into a more typical style of Verilog code, and also prepare us for designing sequential systems (see p.16).

The second main difference compared with previous code we have looked at is that the output, A, is given as a **register** (**reg [1:0]**). This is necessary for outputs when we use a procedural block such as **always** or **initial**; by virtue of being 'registers', such outputs can be treated as 'variables' (not just 'nets' or wires) as mentioned in 1.7.2.

> **Note:** Above, A is a 2-bit register. For declaring **single-bit outputs**, you can use **output reg** without the square-bracketed number e.g. output reg A. However, if you have multiple outputs of different lengths, you must repeat the 'output' keyword for each one. E.g. if we have two outputs A, B, this will work: "**output reg [1:0]A, output reg B**" but, for example, this wouldn't work: "output reg[1:0] A, reg B".

The other main difference with our previous examples is that we've used an **if-else statement**. Inputs Y[3], Y[2], Y[1] and Y[0] are evaluated to determine whether they are driven by V_{CC}. The "**if**" statement activates the equations that follow the active **if**, **else if** or **else** clause. E.g. if Y[3] is driven by V_{CC}, A is binary 11 (decimal 3).

If more than one input is driven by V_{CC}, the **if** statement evaluates the priority of the inputs, which is determined by the order of the **if** and **else if** clauses (the first clause has the highest priority).

In the priorityenc module, Y[3] has the highest priority, Y[2] has the next highest priority, Y[1] has the third highest priority and Y[0] has the lowest priority. The **if** statement activates the equations that follow the highest-priority **if** or **else if** clause that is true.

If none of the inputs are driven by V_{CC}, the equation following the **else** keyword is activated.

Figure 13, generated in Quartus II, illustrates the behaviour of this priority encoder by simulating the output for various input combinations:

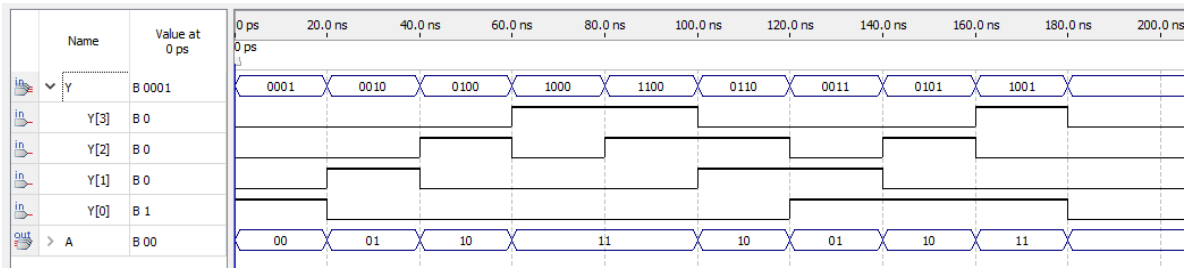

Figure 13. Functional simulation (timing diagram) for priority encoder

1.8 Case Statement Logic

The **decoder** module shown below is a 2-bit to 4-bit decoder. It converts two binary code inputs into a "one-hot" code. (It does the opposite of an encoder.) This example introduces the syntax of case statements in Verilog:

```
module decoder(output reg [3:0]A, input [1:0]Y);
always@(Y)
      begin
            case(Y)
                  2'b00: A = 4'b0001;
                  2'b01: A = 4'b0010;
                  2'b10: A = 4'b0100;
                  2'b11: A = 4'b1000;
            endcase
      end
endmodule
```

In this example, the input group code, Y, is the binary equivalent of the decimal value 0, 1, 2, or 3. Each equation in the case statement is activated if Y is equal to the number shown on that line. For example, if Y = 10_2, A is set to 0100_2. Since the values of the expressions are all different, only one clause can be active at one time. The simulation in Figure 14 shows the behaviour of this decoder:

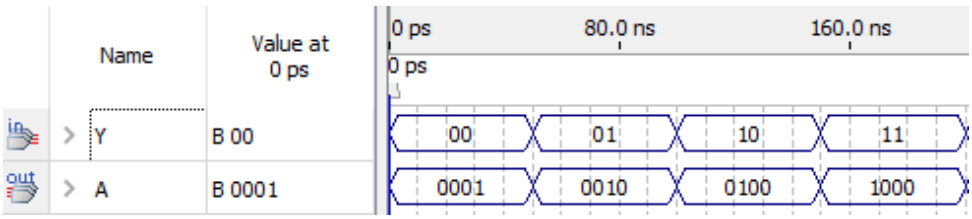

Figure 14. Functional simulation for 2-4 decoder

1.9 7-segment decoder using a case statement

A decoder contains combinatorial logic that interprets input patterns and converts them to output values. In Verilog, as we have seen, you can use a case statement to describe a decoder. Effectively, a Verilog case statement describes a truth table for the required combinatorial logic. [This is similar to the idea of a **look-up table** (LUT).]

The **seven_segment** module shown below is a 7-segment HEX decoder that specifies logic for patterns of light-emitting diodes (LEDs). The LEDs are illuminated in a seven-segment display to show the hexadecimal numbers 0 to 9 and the letters A to F. To remind you about seven-segment displays from lectures, the commented code at the top (shown below in light blue) first illustrates the working of such a display:

```
//   -a-
// f|   |b
//   -g-
// e|   |c
//   -d-
//
// 0 1 2 3 4 5 6 7 8 9 A b C d E F
```

```
module seven_segment (output reg[6:0] sevenseg, input[3:0] in);
always @(in)
     begin
           case (in)
              // abcd                abcdefg
                 4'h0: sevenseg = 7'b0000001;
                 4'h1: sevenseg = 7'b1001111;
                 4'h2: sevenseg = 7'b0010010;
                 4'h3: sevenseg = 7'b0000110;
                 4'h4: sevenseg = 7'b1001100;
                 4'h5: sevenseg = 7'b0100100;
                 4'h6: sevenseg = 7'b0100000;
                 4'h7: sevenseg = 7'b0001111;
                 4'h8: sevenseg = 7'b0000000;
                 4'h9: sevenseg = 7'b0000100;
                 4'hA: sevenseg = 7'b0001000;
                 4'hB: sevenseg = 7'b1100000;
                 4'hC: sevenseg = 7'b0110001;
                 4'hD: sevenseg = 7'b1000010;
                 4'hE: sevenseg = 7'b0110000;
                 4'hF: sevenseg = 7'b0111000;
           endcase
     end
endmodule
```

The output pattern for all 16 possible input patterns of in[3..0] is described in the case statement. Although the code initially defines the input as 4-bit binary, the inputs are referred to in the case statement by their hex values to emphasise what the decoder is doing. Notice also that zeroes are used to activate the correct segments, because they are active-LOW as explained in Supporting Information Section 2.1.

2 Sequential design in Verilog

For any sequential system, some form of memory is required, implying the need for flipflops and, if the system is to be synchronous, a clock. “D” type flipflops (DFFs) are, in practice, the most commonly used. (Other types of flipflops are generally configured within the FPGA using DFFs.) Quartus II will interpret sequential systems you design in Verilog and if designed correctly, will insert DFFs as necessary.

2.1 Oscillator

All synchronous devices need a clock. For simulation purposes, we can set the clock manually as if it was any other input (as you will in the experiment.) For physical implementation, the DE1 board has a choice of oscillators as indicated in Table 1.

Signal Name	FPGA Pin	Description
CLOCK_27	PIN_D12	27 MHz clock input
CLOCK_50	PIN_L1	50 MHz clock input
CLOCK_24	PIN_B12	24 MHz clock input from USB Blaster
EXT_CLOCK	PIN_M21	External (SMA) clock input

Table 1 DE1 Clocks

2.2 Counter design in Verilog

Perhaps the most straightforward sequential system to implement is a simple counter:

```
/* Below: declare module and declare/define ports. Circuit inputs: clock, clear and
two enables. Define count as a 4-bit output reg (counter output); define rco as a
single-bit output (ripple-carry-out signal). Note: as rco is being assigned outside
a procedural block, it has to be a net (wire), not a variable, which is why it doesn't
have the reg keyword.*/

module dec_count (output reg[3:0] count, output rco, input clk, clr, enc, ent);

/* Below: a 'continuous assignment' of rco so that it goes high virtually immediately
when the count is 9 and ent is true (i.e. rco goes high in a combinational way, not
waiting for the active clock edge). This makes rco a conditional output.*/

      assign rco = ((count == 4'd9) && ent);  // Generate the rco when
                                              // the count is 9 and ent is true.


// Below - this 'always' block is triggered by the rising clock edge
// It is executed every time the clock transitions from 0 to 1

always @ (posedge clk)
      begin
            if (clr)                                  // If clr = 1,
                  count <= 0;                         // reset 'count' to zero.
            else if (enc && ent && count != 4'd9)     // Else, if enc & ent = 1,
                                                      // and count isn't 9,
                  count <= count + 1;                 // increment count by 1

            else if (enc && ent && count == 4'd9)     // Else, if both enables
                                                      // are true, and count = 9,
                  count <= 0;                         // return count to 0.

            else                          // Else (if one or both enables false),
                  count <= count;         // do NOT increment.
      end
endmodule
```

In the **dec_count** module a 4 bit counter is defined. There are four inputs consisting of the clock, two enables (ent & enc) and a clear. The outputs are the four count bits and a Ripple Carry out (rco). This code will cause the synthesis of four D type flipflops for the four output count bits. The comments in the file explain the code's operation.

Notice how this time we introduced some **combinational** logic (before the always block with clock-edge sensitivity), for the ripple-carry-out signal. Remember from ELEC211 lectures that sequential systems usually have two components, combinational logic and sequential logic. Splitting your code into a **sequential block** and a **combinational block** is a good practice which can avoid timing problems. (Although, in the above example, we only have a sequential 'always' block, preceded by a combinational 'assign' statement, in the examples below we will use both sequential and combinational 'always' blocks.) We use combinational logic for the rco because we want it to go high as soon as possible after the counter value becomes 9, and remain high only while it is 9. This avoids the rco signal going through a DFF and waiting one clock cycle before going high (which would cause it to go high late, when count == zero).

*Notice also how, following the module declaration, we used the **output** keyword twice for two circuit outputs of different lengths and types. Because rco is a continuous assignment (not part of an 'always' block) then it isn't defined as type reg, and certainly not as type reg[3:0].*

2.2.1 Logical equality (==), logical AND (&&) and friends…

The line beginning with "assign rco = …" introduces the use of two more Verilog operators. These are logical equality and logical 'and' (similar to a number of programming languages). They are for the purpose of evaluation: rco goes true if count is equal to (**==**) 4'd9, and (**&&**) ent is true. The operator **!=** denotes logical inequality, while || denotes logical OR.

2.2.2 Assignment operators (=, <=) in combinational/sequential 'always' blocks

You may have noticed that in the dec_count module, we mostly used **<=** to assign values, where previously we would have just used **=**. This brings us to our next topic. As noted above, we tend to use **two** types of always blocks, for combinational and sequential logic respectively, and we use a different type of assignment in each one.

In **combinational** logic blocks, such as the earlier always blocks in Sections 1.7 to 1.9, we tend to use '**blocking**' assignments (**=**). In this kind of assignment (e.g. x = y), the statements following after are made to wait, or 'blocked' from executing, until the current assignment is complete. So for, say, "x = y", followed by "z = x", z gets the value of y. This allows a signal to trickle through a combinational circuit, with the earlier operations in the circuit taking effect marginally before the next ones.

In **sequential** logic blocks, such as the always statement in the counter just considered, we tend to use **nonblocking** assignments (<=). This allows all the assignments to be executed at once. So for 'x <= y' followed by 'z <= x', z would get the **original** value of x (before y was assigned to it). This reflects the fact, for example, that in a given clock cycle, all relevant inputs to an ASM are evaluated simultaneously.

The use of these operators in combinational and sequential blocks might, at first, seem back-to-front intuitively – but it does work. More examples of this when we look at State Machines (where we will use both sequential and combinational logic blocks).

3 State Machine implementation

This section will run through the implementation of state machines using Verilog and also cover how the state machine is implemented in hardware. For a reminder on Mealy and Moore machines, see ELEC211 Lecture 10 (or Section 4 below.)

3.1 Mealy machine example

The ASM in Figure 15 shows a three state system with 1 test input and two outputs.

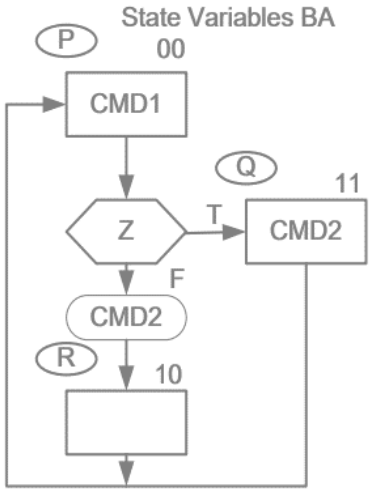

Figure 15. ASM chart (Mealy Machine)[5]

Below is shown the Verilog code to implement this Algorithmic State Machine, which is a Mealy Machine. The module opens by declaring the outputs (cmd1, cmd2), and the inputs (clock, reset, z).

The line **reg[1:0] p_state, n_state;** declares two extra 2-bit variables to be used for storing the present and next state addresses respectively. By the way, if we wanted to use the 'one-hot' addressing scheme, we could have used 3-bit variables in this case (see ELEC211 Lecture 11), since we have three states.

As we said before, the '**parameter**' is a kind of variable which is used to create constants in a module. In this case, the parameters S_P, S_Q and S_R define the actual flipflop value combinations (called 'state variables BA' in Figure 11) for states P, Q and R respectively. Later in the code, we assign these to n_state (the next state address).

The first '**always**' block in the code below sets the sequential part of the system – what will happen on the rising clock edge (**posedge clk**) – and also what will happen when **reset** is pressed and becomes true. If reset is true, the present state (p_state) returns to S_P. Otherwise, on the rising edge of the clock, the next state value becomes the present state.

```
// Declare module and declare/define ports
module statemachine1 (output reg cmd1, cmd2, input clk, reset, z);

      reg[1:0] p_state, n_state;                // Present and next states

      parameter S_P = 2'b00, S_Q = 2'b11, S_R = 2'b10;// State variables
                                                      // (BA)

always @ (posedge clk, posedge reset)           // Sequential block
      if (reset == 1) p_state <= S_P;           // (.. setting the effects
                                                // of the clock edge
      else p_state <= n_state;                  // and reset on the state)
```

[5] Remember, the input hexagon has the same meaning as an input 'diamond' in ELEC211 lecture slides

```
always @ (p_state, z)                              // Combinational block
      begin                                        // (.. setting the effects
                                                   // of the present state and
                                                   // input on the next state
                                                   // and the output)
            cmd1 = 1'b0;
            cmd2 = 1'b0;
            n_state = p_state;            // Default outputs and next state

            case (p_state) // Case stmt: for present state P, R or Q
                  S_P:
                    begin                          // State P: see below!
                        cmd1 = 1'b1;
                        if(z)
                              n_state = S_Q;
                        else
                          begin
                              cmd2 = 1'b1;
                              n_state = S_R;
                          end
                    end
                  S_R:
                    begin                          // State Q:
                        n_state = S_P;             // Next state = P
                    end
                  S_Q:
                    begin                          // State Q:
                        cmd2 = 1'b1;               //cmd2 is the op
                        n_state = S_P;             //Next state = P
                    end
            endcase
      end
endmodule
```

The second '**always**' block (above) sets the combinational part of the system – what will happen to the next state or the present output, as a result of changes in the present state or present input. (Think back to Mealy Machines in ELEC211 Lecture 10.) Continuing in the second '**always**' block, we begin by setting some default values: **cmd1** and **cmd2** are both zero by default (unless instructed otherwise); the next state defaults to be equal to the present state (unless instructed otherwise).

Next, we use a **case statement** to evaluate the present state: does p_state equal **S_P**, **S_R** or **S_Q**? If we're in state P, cmd1 is true. Also, while we're in state P then (if input **z** is true) our next state will be Q. Alternatively (if **z** is false) we'll get a conditional output (**cmd2** will go true) and the next state will be state R.

See the comments in the code for what happens to the output and next state for the other states, R and Q. By the way, the input doesn't have any influence in these states (no input hexagon/diamond in either state in the ASM chart).

3.1.1 'Begin' and 'end'

When you compare the above code to the code for our **seven_segment** module, the eagle-eyed amongst you might be wondering why we used the 'begin' and 'end'

keywords inside the case statement. These aren't just used to open and close 'always' blocks; they're used to group together statements whenever necessary.

In the **statemachine1** module, there are multiple statements within each overall case. (In the seven_segment module, there was only a single statement per case so it wasn't necessary.) We didn't actually need the 'begin' and 'end' around the case statement for S_R though - it was just included for readability and consistency, since the other cases contained multiple statements.

(We used 'begin' and 'end' inside the if/else statement too, for the same reason.)

3.2 Moore Machine example

The ASM in Figure 16 is almost identical to the one in Figure 15, but this time it's a Moore machine – there is no longer any conditional output in state P:

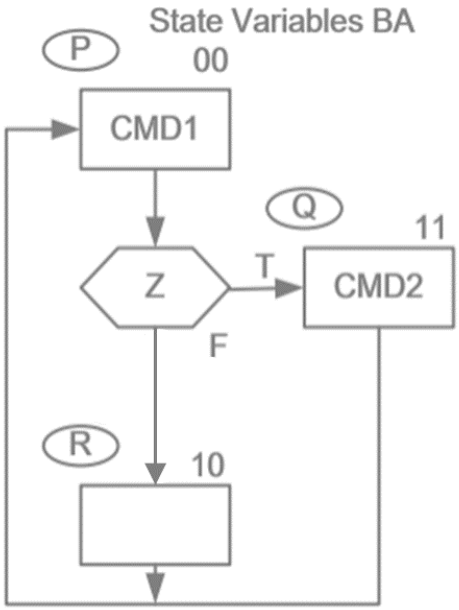

Figure 16. ASM chart (Moore Machine)

Below is shown the Verilog code to implement this Algorithmic State Machine. Needless to say, it's very similar to the previous one, but as it's a Moore Machine there are no conditional outputs. See if you can work through it and verify all the stages.

```
module statemachine2 (output reg cmd1, cmd2, input clk, reset, z);

      reg[1:0] p_state, n_state;

      parameter S_P = 2'b00, S_Q = 2'b11, S_R = 2'b10;

always @ (posedge clk, posedge reset)
      if (reset == 1) p_state <= S_P;
      else p_state <= n_state;

always @ (p_state, z)
      begin
            cmd1 = 1'b0;
            cmd2 = 1'b0;
            n_state = p_state;
```

```
            case (p_state)
                  S_P:
                    begin
                        cmd1 = 1'b1;
                        if(z)
                              n_state = S_Q;
                        else
                              n_state = S_R;
                    end
                  S_R:
                    begin
                        n_state = S_P;
                    end
                  S_Q:
                    begin
                        cmd2 = 1'b1;
                        n_state = S_P;
                    end
            endcase
      end
endmodule
```

See the next two pages for a reminder about Mealy and Moore networks.

Following this, you can find sections on 'Assessment and marks weighting' and 'Plagiarism and Collusion'.

4 Mealy and Moore Networks - a reminder!

This section reminds you about the difference between Mealy and Moore Networks; they are also referred to as Mealy State Machines and Moore State Machines; you have learned about these in ELEC211 Lecture 10.

4.1 Mealy networks

For a Mealy network the outputs, which correspond to a given input(s), can change immediately (immediate combinational effect of inputs) as shown in Figure 17.

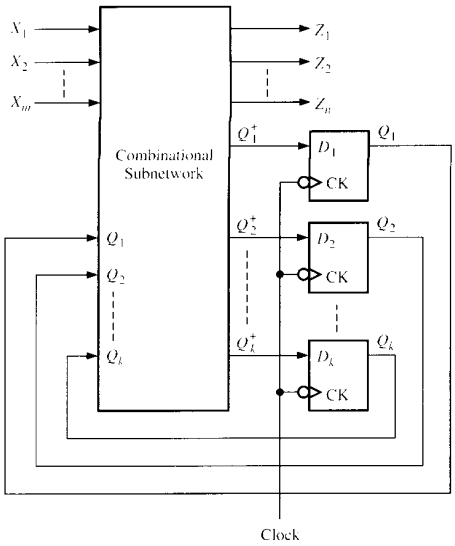

Figure 17. Mealy Network

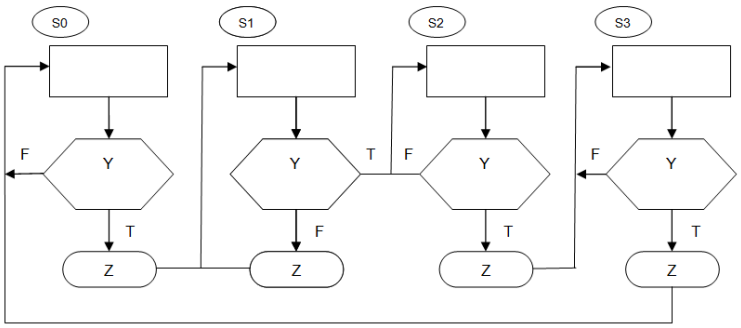

Figure 18. ASM for some other Mealy Network

This is equivalent to having **conditional** outputs in ASM charts as shown in Figure 18. In this example the conditional "Z" output causes the ASM to describe a Mealy network.

4.1.1 Simulation Waveform for Mealy

Figure 19 shows a simulation of the Mealy network from Figure 18, with a reset. Notice how no input (including the reset) changes on the active edge of the clock – which is the rising edge in this simulation – and that all inputs are stable for at least one full clock period. These features are important for reliable detection of the signal.

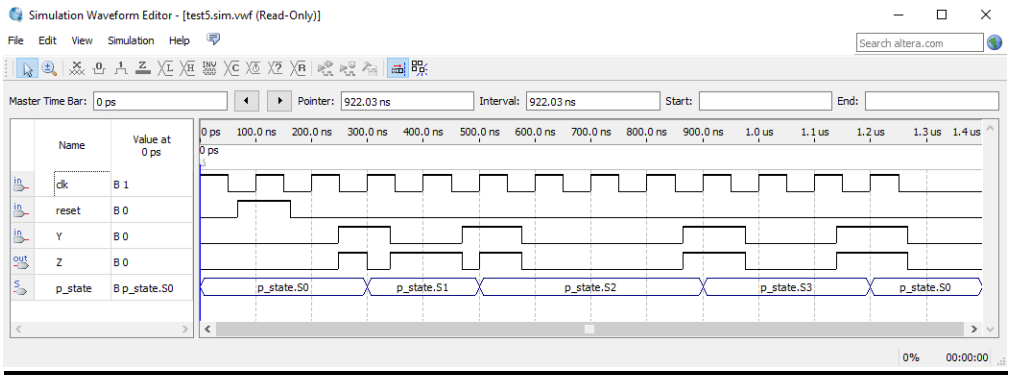

Figure 19. Simulation of a Mealy ASM

4.2 Moore networks

For a Moore network the outputs are only a function of the current (i.e. present) state and not of the value of any inputs, as shown in Figure 21 (input signals only affect outputs via flip-flops).

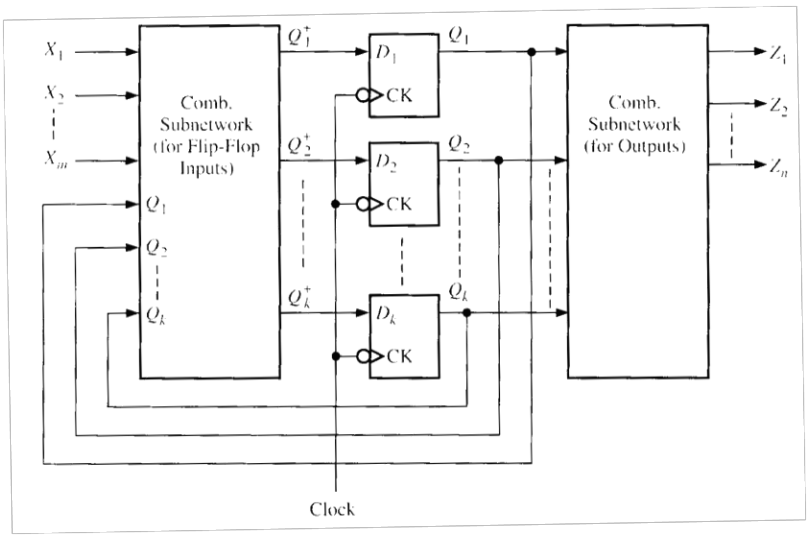

Figure 20. Moore Network

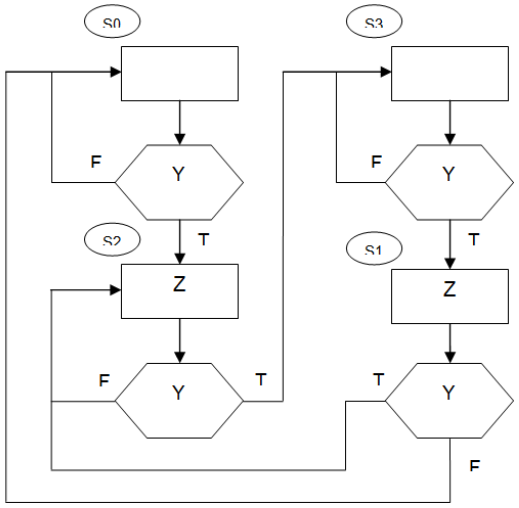

Figure 21. ASM for a Moore Network

Therefore, the outputs of a Moore network can change only when the flip-flops change state and not when the inputs change. This is equivalent to only having unconditional outputs in an ASM. The ASM in Figure 21 is a Moore network as the outputs only depend on the value of the current state.

5 A little note about RTL Viewer and top-level design entities

As noted on p.9 and in footnote 2, once you're up and running with Quartus II you can inspect the signal data flow by means of the Quartus II RTL Viewer, using Tools → Netlist Viewers → RTL Viewer. However, RTL Viewer will display only what is contained in the 'top-level design entity' after compilation. In Quartus II, the top-level design entity is the root of your design hierarchy[6] i.e. it is the parent design file in your project, which can contain any relevant sub-designs from the project. In Quartus II, you can see the design hierarchy of your project in the Project Navigator (on the left). By default, Quartus II expects that the name of the project itself will also be the name of the top-level design entity – be that a block diagram/schematic file (.bdf) or a Verilog module described in a Verilog HDL file (.v). Therefore, if, early in a project, you want some entity to be the top-level design entity (provided that something else hasn't been set as it) a simple shortcut is often to set its name to be the same as the project name. In the case of a block diagram/schematic, this means naming the .bdf file after the project. In the case of a Verilog module, this means naming the module – the word after the "module" keyword – after the project (it's probably safest to give the Verilog file itself the same name too.) You can also reassign the top-level design entity of a project (one way is to select the desired file, then follow Project → Set as Top-Level Entity) which alters the naming of said entity from the default; you must then re-compile the project for this to take effect. After compiling your project, you can use RTL Viewer for a peek at the signal flow contained in your top-level design entity.

[6] https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/reference/glossary/def_top_level.htm

6 Assessment and marks weighting

This experiment is assessed by means of a pre-lab test (Canvas) and practical work. Read this document carefully and **attempt the online Pre-Lab Test. The deadline is given on Canvas.**

The marks weighting for Experiment 28 is as follows:

- The Pre-Lab Test: 30 Marks
- The Practical work Report: 70 Marks (check the script for Part II: The Practical Part plus 'Supporting Material' and the Submission Template)

7 Plagiarism and Collusion

Plagiarism and collusion or fabrication of data is always treated seriously, and action appropriate to the circumstances is always taken. The procedure followed by the University in all cases where plagiarism, collusion or fabrication is suspected is detailed in the University's Policy for Dealing with Plagiarism, Collusion and Fabrication of Data, Code of Practice on Assessment, Category C, available on http://www.liv.ac.uk/tqsd/pol_strat_cop/cop_assess/appendix_L_cop_assess.pdf.

Follow these guidelines to avoid any problems:

(1) Do your work yourself.

(2) Acknowledge all your sources.

(3) Present your results as they are.

(4) Restrict access to your work.

Version history

Name	Date	Version
Dr D G McIntosh	April 2022	Ver. 8.1
Dr D G McIntosh	April 2021	Ver. 8 (Verilog)
Dr D G McIntosh	April 2020	Ver. 7 (remote)
Dr D G McIntosh	March 2020	Ver. 6.5
Dr M López-Benítez	September 2019	Ver. 6.4
Dr M López-Benítez	December 2017	Ver. 6.3
Dr M López-Benítez	November 2017	Ver. 6.2
Dr A Al-Ataby	October 2014	Ver. 6.1
Dr A Al-Ataby, F Davoodi Samirmi and Prof J Smith	October 2013	Ver. 6
Dr A Al-Ataby	October 2012	Ver. 5.8
Prof J S Smith	October 2011	Ver. 5.7
Dr S Amin-Nejad	July 2011	Ver. 5.6

Feedback:

If you have any feedback on your laboratory experience for this experiment (e.g. timing, difficulty, clarity of script, demonstration...etc) and suggestions to how the experiment may be improved in the future, please write them down in the space below. This feedback is important for future versions of this script and to enhance the laboratory process, and will not be assessed. If you wish to provide this feedback anonymously, you may do so by detaching this page and submitting it to the Student Support Centre (fifth floor office).

Script re-writing award
If you think that this experiment could do with enhancement or changes and you have some ideas that you'd like to share, why not re-write this script yourself and you may get an award from the lab organisers with an official letter of thanks, and your name will be added to the version history list in future versions of the script. Something good for your CV. Contact one of the lab organisers for more details.