CS132 Coursework 1

Alen Buhanec

Contents

Question 1

(a) The truth table is show in table 1.

$A2$	$A1$	$A0$	$D7$	$D6$	$D5$	$D4$	$D3$	$D2$	$D1$	$D0$
0	0	0	0	0	0	0	0	0	0	**1**
0	0	**1**	0	0	0	0	0	0	**1**	0
0	**1**	0	0	0	0	0	0	**1**	0	0
0	**1**	**1**	0	0	0	0	**1**	0	0	0
1	0	0	0	0	0	**1**	0	0	0	0
1	0	**1**	0	0	**1**	0	0	0	0	0
1	**1**	0	0	**1**	0	0	0	0	0	0
1	**1**	**1**	**1**	0	0	0	0	0	0	0

Table 1: Truth table for a 3-to-8 active high decoder
.

(b) As we can see in table 1, each row corresponds to one and only one of the outputs being active. Therefore we can rewrite each row as an expression for the condition of that output being active.

$$
\begin{aligned}
D0 &= \overline{A2} \cdot \overline{A1} \cdot \overline{A0} \\
D1 &= \overline{A2} \cdot \overline{A1} \cdot A0 \\
D2 &= \overline{A2} \cdot A1 \cdot \overline{A0} \\
D3 &= \overline{A2} \cdot A1 \cdot A0 \\
D4 &= A2 \cdot \overline{A1} \cdot \overline{A0} \\
D5 &= A2 \cdot \overline{A1} \cdot A0 \\
D6 &= A2 \cdot A1 \cdot \overline{A0} \\
D7 &= A2 \cdot A1 \cdot A0
\end{aligned}
$$

(c) The design can be seen in appendix A and was reached with the following through process:

(i) Each of the outputs must accept input from all three inputs

(ii) According to item b these inputs must go through a triple input AND gate (a triple input AND gate can be constructed from two double input AND gates, with one of the gates running its output to the other's input, and the remaining three inputs being used as a substitute for the triple input AND input and the remaning output being used as the triple AND output)

(iii) 6 input lines should run along the 8 outputs, each line being either an active high or active low for one of the 3 inputs, with each input having one active high and one active low line

(iv) These 6 input lines should connect to the AND gates of the ouputs where appropriate, making sure each input only connects to an output once, regardless of which line is being used

(v) This design can be tested by checking where the three inputs of every ouput lead, and verify the correctness with a given truth table or set of expressions

(vi) Alternatively, if the circuit is assembled, we can think of the three inputs of a 3 digit binary number and go from 0 to 7, checking if the corresponding Dx output will be enabled, where x is the number inputted. Furthermore since we have 8 possible combinations with 8 possible outputs, we should never encouter a duplicate output. We can also run the output into a 8-to-3 encoder, and we should get our initial input back

Question 2

(a) The design can be found in appendix B. The rectangular components are simplified representations of D-type flip-flops. For clarity's sake I will refer to the flip-flop with output O_1 as the first flip-flop and O_4 as the last flip-flop.

(b) We need to assume that the circuit is constantly powered on during operation, otherwise flip-flops lose their states. Additionally we need to assume that the flip-flops are all clocked in a synchronised fashion, since once the flip-flops update the inputs to the flip-flops may change.

The core of the design rests on the combination of two AND gates leading into an OR gate connected to the input of a flip-flop. Each instance of this "combination" has one AND gate connected with one input to the active high line of the I_{dir} input, and the other AND gate connected with one input to the active low line of the I_{dir} input. This allows the design to toggle between the remaining inputs of the two AND gates as the input sent to the flip-flop through the OR gate.

The instances of the "combination" have their remaining inputs (the ones being toggled between by I_{dir}) connected to the next or previous flip-flop ouput, where possible. Since the first flip-flop has no previous one, the input used instead is the input I_D. Likewise the last flip-flop has no next flip-flop, so the input I_D is used in its place. Furthermore, since there is no previous or next flip-flop for the outputs of the first or last flip-flop to be led to in some cases, the output is led to O_D using the same "combination" to decide between using the output of the first or last flip-flop. Note that due to the layout of the circuit, the AND gates of the O_D "combination" have their positions inverted compared to the rest of the circuit.

To make sure the shifting is performed consistently across the entire register, the I_dir active high line connects to the AND gate with the previous flip-flop's output as its input, and the active low line connects to the AND gate with the next flip-flop's output as its input (substituting I_D and O_D where applicable for the first and last flip-flop).

To more clearly illustrate the design, a few consecutive steps will be explained, given the initial state of "1 0 1 0" for "D_1 D_2 D_3 D_4". The expressions used to determine the input to flip-flops when performing these steps is the following, where D_x is flip-flop x's input and Q_x is x's output. The expression for O_D is repeated twice too, to clarify the "shifting" overflow leading to the output:

$$
\begin{gathered}
O_D = D_4 \cdot I_{dir} + Q_1 \cdot \overline{I_{dir}} \\
D_1 = I_D \cdot I_{dir} + Q_2 \cdot \overline{I_{dir}} \\
D_2 = D_1 \cdot I_{dir} + Q_3 \cdot \overline{I_{dir}} \\
D_3 = D_2 \cdot I_{dir} + Q_4 \cdot \overline{I_{dir}} \\
D_4 = D_3 \cdot I_{dir} + I_D \cdot \overline{I_{dir}} \\
O_D = D_4 \cdot I_{dir} + Q_1 \cdot \overline{I_{dir}}
\end{gathered}
$$

(a) Shifting downwards (position increasing) with input 0

   i. I_D is set to 0 (input 0) and I_{dir} is set to 1 (position increasing)
      - The inputs for flip-flops 1-4 are now "0 1 0 1"
      - The input for the O_D flip flop is "0"

   ii. clock changes state, updating the flip-flops
      - The flip-flops are now set to "0 1 0 1", output is "0"
      - Essentially what happened was:

$$0\ (I_D) \rightarrow 1\ 0\ 1\ 0\ \rightarrow O_D \xrightarrow{\text{clock state change}} I_D \rightarrow 0\ 1\ 0\ 1 \rightarrow 0\ (O_D)$$

(b) Shifting downwards (position increasing) with input 1

   i. I_D is set to 1 (input 1) and I_{dir} is set to 1 (position increasing)
      - The inputs for flip-flops 1-4 are now "1 0 1 0"
      - The input for the O_D flip flop is "1"

   ii. clock changes state, updating the flip-flops
      - The flip-flops are now set to "1 0 1 0", outpit is "1"
      - Essentially what happened was:

$$1\ (I_D) \rightarrow 0\ 1\ 0\ 1\ \rightarrow O_D \xrightarrow{\text{clock state change}} I_D \rightarrow 1\ 0\ 1\ 0 \rightarrow 1\ (O_D)$$

(c) Shifting upwards (position decreasing) with input 1

   i. I_D is set to 0 (input 0) and I_{dir} is set to 0 (position decreasing)
      - The inputs for flip-flops 1-4 are now "0 1 0 1"
      - The input for the O_D flip flop is "1"

   ii. clock changes state, updating the flip-flops
      - The flip-flops are now set to "0 1 0 1", output is "1"
      - Essentially what happened was:

$$O_D \leftarrow 1\ 0\ 1\ 0\ \leftarrow 1\ (I_D) \xrightarrow{\text{clock state change}} 1\ (O_D) \leftarrow 0\ 1\ 0\ 1 \leftarrow I_D$$

Question 3

(a) The function can be found in appendix C, and is defined as the C function `int binomialCoefficient(int n, int r)`. It accepts two integers as arguments and ouputs the resulting integer. As it works recursively I defined some termination cases where the binomial coefficient always takes a specific value. The function that draws a row of Pascal's triangle based on this function is defined as `void drawPascalsRow(int n)` and accepts the row number as an argument.

(b) The function is defined as `int fibonacci(int n)` and uses `binomialCoeficient`.

(c)
- Lines 1-4: I included the standard input/output library to output results and verify my functions. Lines 3 and 4 include forward declarations of functions, so that they can be used in `main` whilst being properly defined later.
- `int main()` - this is going to get executed when the program runs, and contains calls to other functions to verify functionality of code. An example of using the functions is giving for parts (a) and (b). For part (a) I draw part of Pascal's triangle by using a for loop to draw rows 1-10, making sure to put each row in a new line by printing the newline and return carriage after each iteration. For part (b) I print out the first 10 Fibonacci in a similar manner. According to convention I return 0 at the end, indicating success (as opposed to -1, which would indicate failure).
- `int binomialCoefficient(int n, int r)` - function corresponding to $\binom{n}{r}$. As the function was defined recursively, I had to define some cases where the recursion would terminate. These cases are covered by the first two if statements, returning 0 if $n < r \lor n < 0 \lor r < 0$ or returning 1 if $n = r \lor r = 0$ where $\binom{n}{r}$. In the case that the recursion has not been terminated we call the same function, but with the second argument equal to $r - 1$. We then multiply the returned value by $\frac{n+1-r}{r}$ as defined by the coursework.
- `void drawPascalsRow(int n)` - we simply use a `for` loop to print out each element in the row, knowing that the element is defined as $\binom{\text{row number}-1}{\text{position in row}}$, if the first position in a row is defined as 0. The function only requires one argument as we know that each row number corresponds to that row's number of elements.
- `int fibonacci(int n)` - In this function we implement the mathematical sum as a `for` loop, using `r` in place of `k`. Instead of flooring the final number of repetitions as written in the coursework (that thing above sigma), I simply used integers to perform the calculation, leading to the stripping of any decimals and effectively flooring the value for me already. Each time we go through the loop we add the calculated binomial coefficient to a sum storing variable that was initially 0, and then after we are done looping we return this sum.

Question 4

(a) The program can be found in appendix D.

(b) The program can be found in appendix E.

(c) (a) i. Lines 2-3: we define the constants used in division

ii. Lines 6-7: we allocate memory for the results in the main store

iii. Lines 13-15: we set our data registers to their initial values, as labeled in the code - D0 is the divisor[1], D1 the quotient, D2 the dividend (which will become the remainder)

iv. Lines 18-20: we increment the quotient and subtract the divisor from the dividend. Using `bgt` we repeat this loop until we hit 0 or a negative number.

v. Lines 21-23: we check if we hit exactly 0 by using `beq`, and if so we jump over the logic for not hitting exactly 0. If we did not hit exactly 0 we "step back" our loop once in lines 18-20 by adding the divisor to the remainder and decrementing the quotient. This way we are left with the remainder and actual quotient.

vi. Lines 26-27: we move the quotient and remainder to the memory we allocated in the main store

(b) i. Line 2: we define the constant to check for primality

ii. Line 5: we allocate memory for the result in the main store

iii. Lines 10-11: we set our data registers to their initial values, as labeled in the code - D0 is the dividend, D1 the divisor

iv. Lines 14-16: "common" code with task (a), we keep subtracting our divisor from the dividend until we reach 0 or a negative number. If we hit exactly 0 we jump to the end of the program and store the result in memory (which would be 0, or "not prime"). This is because we successfully divided the numer without a remainder.

v. Lines 17-20: we increment divisor by 1 and move the constant dividend to D0. We then subtract the two and check if the result is 0. If it is we have reached the end of our program[2] without finding a factor and we jump to line 25, which sets the value of D0 to "prime" (1).

vi. Lines 21-22: we move the constant divident to D0 again since we subtracted the divisor just 2 lines before and then jump to the beginning of the division loop.

vii. Line 25: the only way to reach this line is to jump to it from line 16, which happens on determining that the number is a prime. It sets the value of D1 to 1.

viii. Line 28: stores the value of D1 to the memory allocated in the main store. D1 should either be "prime" (1) or "not prime" (0)

[1]We could use the `divisor` constant instead of a register throughout the program, but in anticipation of the follow up task, I implemented division using the data register instead

[2]We have checked every possible number up to the number itself, although inefficient this is a sure way to check all factors.

A 3-To-8 Decoder

Figure 1: A 3-to-8 active high decoder

B 4-bit Bidirectional Shift Register

Figure 2: A 4-bit bidirectional shift register with input I_D, output O_D, direction control I_dir and bit status indicators O_1 to O_4

C C Functions Relating to Pascal's Triangle

D MicSim Division

E MicSim Primality Test