In this lecture, we will focus on two very important digital building blocks: counters which can either count events or keep time information, and shift registers, which is most useful in conversion between serial and parallel data formats.
Lecture Objectives

- Understand how digital systems may be divided into datapath and control logic
- Appreciate the different ways of implementing control logic
- Understand how shift registers and counters can be used to generate arbitrary pulse sequences
- Understand the circumstances that give rise to output glitches
- Able to design various types of counters and timers
Logic circuits generally consists of circuits in three categories:

- **Datapath blocks** — these are arithmetic blocks, logic units, floating point units, digital processing blocks etc.
- **Memory blocks** — these store information for digital circuit to manipulate or store the instructions for a microprocessor
- **Control logic blocks** — these are generally clocked and they make sure that data goes to the right places at the right time for computation

Control logic can be implemented either in a microprocessor or a microcontroller, a synchronous finite state machine (probably best) or using random logic with counters and shift registers.

In this lecture, we will introduce the use of counters and shift registers for producing control circuits.
Here is the design of a 4-bit synchronous counter in schematic form. The four D-FFs store the current count value. The adder is used to compute the next count value and fed to the input of the D-FFs. On every positive edge of the signal \( \text{clk} \), the count value \( \text{count}[3:0] \) goes up by one.

The Verilog description of this counter is shown. Note how easy it is to specify the D-FFs (via the always block with \text{posedge} keyword in the sensitivity list). The add-one is specified with the ‘+’ operator.

In the actual implementation, the CAD software will NOT insert a 4-bit adder circuit. Instead the synthesis software will reduce this to an optimised gate-level implementation because one of the operand of the adder is the constant ‘1’. However, as a designer, you do not need to worry about this optimisation step (except may be in answering an examination question which asks you to simplify). The CAD software will deal with this for you. In fact, it is often much better to leave the specification as shown here – the design is more readable and therefore easier for others to understand what you are doing.

For example, one input to the adder is: \( 4’b0001 \). Imagine you have many AND and OR gates inside the adder. The AND driven by 0 or 1 can be mapped to a simple wire. Similarly simplification can be applied to OR gates.

Finally, shown here is the state transition diagram. Each bubble represents a state (labeled with the count value) and on each rising edge of \( \text{clk} \), it transits to the next state.
Here is a modification from the previous counter by adding a low-active reset signal RST (shown as !RST).

The implementation is simple in Verilog – just add a if-else statement to specify that when RST = 0, reset on the next clock rising edge.

The state transition diagram is modified to include the reset action.

The symbol for the counter is worth noting. CTR4 indicates that it is a 4-bit counter. CT means count value. C1/+ label on the clk signal indicates that this clock signal controls either the action of input 1 or will increment the counter. The 1R label indicates that it is a reset input and it is controlled by the clock signal 1. Finally the triangle at the input of 1R says that this input is low active (i.e. low to reset).
What if you want a counter that only counts from 0 to 9, then back to 0 again? This is really easy with Verilog. A simple if-else construct testing for the terminal counter of 9 will do the trick.

The actual implementation could use of 4-bit binary counter with a simple AND gate to detect when the counter value of 9 is reached, and then synchronously reset the counter back to 0.

The state transition diagram is as shown. Beware of what happens when count value is outside the range of 0 to 9.
When you use gates to combine values from a counter, beware that you will get glitches at the output of the gates.

Here is an example where Y could contain glitches. This is because the outputs Q[3:0] will not change exactly at the rising edge of the clock signal. Instead, there will be some delays and the delays are likely not the same for different Q outputs.

For example a count value transition from 7 to 8 could temporarily go to 1111 before going 1000. (i.e. 0111 -> 1111 -> 1000). Similarly transitions from 5 to 6 and 0 to 1 could result in glitches in Y.
Eliminating output glitches can be achieved by adding an extra D-FF at Y output. Z will be glitch free. However, beware that Z is one cycle delayed relative to Y.

If Z is instead of Y is to be used to control other circuits, we need to time synchronous all the signals. This may involve adding D-FFs to datapaths that are controlled by Z in order to make sure that data signals and control signals arrive at the same clock cycle.
If the one-cycle delay is not desirable and you need to avoid any glitches, one could use a special counter that counts in a special sequence that guarantees NOT to produce glitches when its count outputs are gated together. The count sequence is known as Gray code.

In a Gray code counter, successive values only have ONE bit changing. Here is a 4-bit gray code counter and it is specified as a case statement in Verilog.
Counters are good in counting events (e.g. clock cycles). We can also use counters to provide some form of time measurement.

Here is a useful timer component that uses a clock reference, and produces a pulse lasting for one cycle pulse every \( N+1 \) clock cycles.

If “enable” is low (not enabled), the clkin pulses will be ignored.

Shown below is the module interface for this circuit in Verilog.

Note that the \texttt{parameter} keyword is used to define the number of bits of the internal counter (or the count value \( N \)). This makes the module easily adaptable to different size of counter.
The actual Verilog specification for this module is shown here.

There has to be an internal counter `count` whose output is NOT visible external to this module. This is created with the `reg [N_BIT-1:0] count;` statement.

The output `tick` has to be declared as `reg` here because its value is updated inside the `always` block.

Also note that instead of adding '1' on each positive edge of the clock, this design uses a **down counter**. The counter counts from N to 0 (hence N+1 clock cycles). When that happens, it is reset to N and the tick output is high for the next clock cycle.
Using this style of designing a clock tick circuit allows us to easily connect multiple counters in series as shown here.

The **clktick** module is producing a pulse on the **tick** output every 50,000 cycles of the 50MHz clock. Therefore **tick** goes high for 20 microsecond once every 1 msec (or 1KHz).

The **clktick** module is sometimes called a **prescaler** circuit. It prescale the input clock signal (50MHz) in order for the second counter to count at a lower frequency (i.e. 1KHz).

The second counter is now counting the number of millisecond that has elapsed since the last time reset signal (1R) goes high.

The design of this circuit is left as a tutorial problem for you to do.
Here is yet another useful form of a counter. I call this a **clock divider**. Unlike the clktick module, which produces a one cycle tick signal every N+1 cycle of the clock, this produces a symmetric clock output **clkout** at a frequency which is the input clock frequency divided by 2*(K+1).

Shown here is the module interface in Verilog. Again we have used the **parameter** statement to make this design ease of modification for different internal counter size.
The Verilog specification is similar to that for clktick. This also has an internal counter that counts from K to 0, then the output clkout is toggled whenever the count value reaches 0.
Instead of producing binary signals using a counter, one could use a shift register to produce a sequence of pulses delayed relative to each other, and use gates to merge these together and produce different binary signals.

Shown here is a D-input to a shift register, producing P Q R and S, delayed from the previous signal by one clock cycle.

Using AND, NOT, OR or XOR gates, we can produce the various digital signals with different delays and pulse widths.

Beware that producing control signals this way may generate glitches.
To specify a shift register in Verilog, use the code shown here (in blue box). We use the \( \leq \) assignment to make sure that \( \text{sreg}[4:1] \) are updated only at the end of the \textbf{always} block.

On the right is a short-hand version of the four assignment statements:

\[
\text{sreg} \leq \{\text{sreg}[3:1], \text{data\_in}\}
\]

This way of specifying the right-hand side of the assignment is powerful. We use the concatenation operation \{ \ldots \} to make up four bits from \( \text{sreg}[3:1] \) and \text{data\_in} (with \text{data\_in} being the LSB) and assign it to \( \text{sreg}[4:1] \).
We can also make a shift register count in binary, but in an interesting sequence.

Consider the above circuit with an initial state of the shift register set to 4'b0001.

The sequence that this circuit goes through is shown in the table here. It is NOT counting binary. Instead it is counting in a sequence that is sort of random. This is often called a pseudo random binary sequence (PRBS).

The shift register connect this way is also known as a “Linear Feedback Shift Register” or LFSR. There is a whole area of mathematics devoted to this type of computation, known as “finite fields” which we will not consider on this course.

The circuit shown below is effective implementing a sequence defined by a polynomial shown: $1 + X^3 + X^4$. The term “1” specifies the input to the left-most D-FF. This signal is derived as an XOR function (which is the finite field ‘+’) of two signals "tapped" from stage 3 (i.e. $X^3$) and stage 4 (i.e. $X^4$) of the shift register.

For a m-stage LFSR, where m is an integer, one could always find a polynomial (i.e. tap configuration) that will provide maximal length. This means that the sequence will only repeat after $2^m-1$ cycles. Such a polynomial is known as a “primitive polynomial”.

The table here shows some of the popular primitive polynomials for different value of m.

Since the output of such a counter is pseudo-random, it is a commonly used circuit to produce random binary sequence for different applications.
Here is the Verilog specification for a 4-bit LFSR.

```verilog
module lfsr4 (data_out, clk);
    output [4:1] data_out; // four bit random output
    input  clk; // clock input
    reg [4:1] sreg; // 4 stage D-FF for this shift register
    initial sreg = 4'b1;
    always @ (posedge clk)
    assign data_out = sreg;
endmodule
```
Quiz Questions

1. If the CLOCK period is $T$, what is the range of possible time delays between a change in the DATA input of a shift register and the resultant change in the output of the first stage?
2. How do you combine the outputs of a shift register to generate a pulse for both the rising and the falling edges of its input signal?
3. In order to guarantee that a shift register will notice a pulse on its DATA input, how long must a pulse last?
4. If an AND gate is used to combine 2 of the outputs from a 4-bit counter, how many different count values will make the AND gate output go high?
5. Why do output glitches not occur when a counter counts from 6 to 7?
6. Name two ways in which output glitches may be avoided.

Answers are all in the notes.