Imperial College London

Lecture 9

Counters & Shift Registers

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

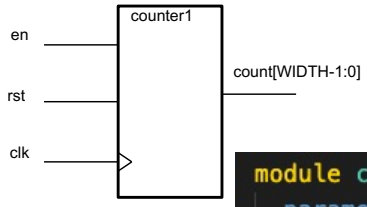
In this lecture, we will focus on two very important digital building blocks: **counters** which can either count events or keep time information, and shift **registers**, which is most useful in conversion between serial and parallel data formats. We will also learn about a special type shift register known as Linear Feedback Shift Registers, which are widely used to generate random digital numbers.

Learning outcomes

- How to specify a simple binary counter?
- How to convert from binary to BCD format?
- How the generate various clock signals with different periods?
- How to specify shift registers?
- How to design a Linear Feedback Shift Register (LFSR) that produces pseudo-random binary sequence (PRBS)?
- How to specify ROM and RAM in SystemVerilog

Here are a list of learning outcome for this lecture. It is also tightly coupled with Lab 2, which will take you through the steps in designing with ROM, RAM and counter, to produce a variable frequency sinewave generator.

Example: Simple Counter




```
module counter #(
  parameter WIDTH = 4
)(                              // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // counter enable
  output logic [WIDTH-1:0] count     // count output
);
  always_ff @ (posedge clk)
    if (rst) count <= {WIDTH{1'b0}};
    else     count <= count + {{WIDTH-1{1'b0}}, en};
endmodule
```


In Lab 4, you are to design an 4-bit counter as shown here. The counter has three inputs:

1. clk – the clock signal (positive edge triggered)
2. rst – the reset signal (high reset), synchronous to clk
3. en – the enable signal, i.e. counting only if en = '1'

The counter has output count[3:0].

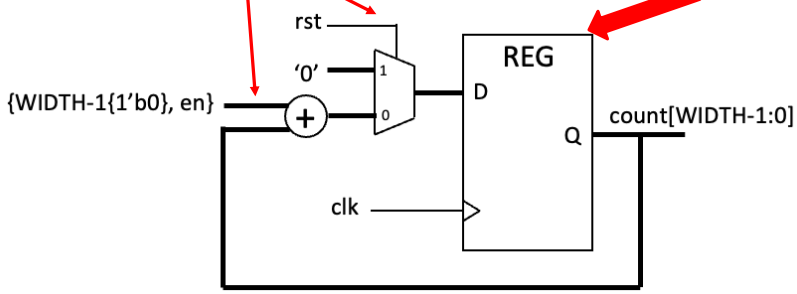
Note the following:

1. We use parameter to define the width of the counter to be 4-bit. The use of parameter allows the same module to be used with different counter width (covered in next lecture).
2. The use of concatenation {.} to create 8-bit value with the LSB = en signal.

Mapping from SV to hardware

```
if (rst) count <= {WIDTH{1'b0}};
else     count <= count + {{WIDTH-1{1'b0}}, en};
```

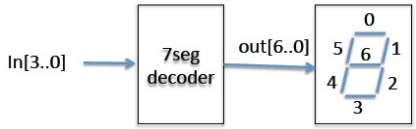
```
8    output reg  [WIDTH-1:0] count
11   always_ff @ (posedge clk)
```



This is how the SV code is mapped to the actual hardware synthesized by Verilator.

The if-else statement is mapped to the MUX. The couting action is achieved via the adder on the feedback path of the register.

Displaying a binary number as decimal



- In Lab 4 Task 2, you are required to display the counter value as binary coded decimal number instead of hexadecimal. A SystemVerilog component **bin2bcd_16.sv** is provided.
- Hex numbers are difficult to interpret. Often we would like to see the binary value displayed as decimal. For that we need to design a combinational circuit to converter from binary to binary-coded decimal. For example, the value 8'hff or 8'b11111111 is converted to 8'd255 in decimal.

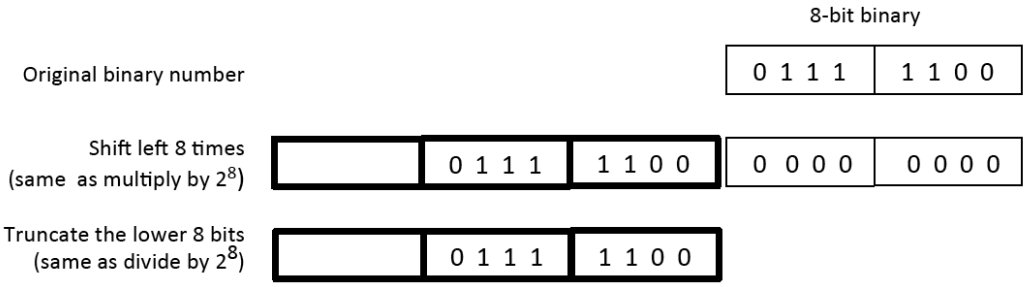
We now take another example of a relative complex combinational circuit, and see how we can specify our design in SystemVerilog.

The goal is to design a circuit that converts an 8-bit binary number into three x 4-bit binary coded decimal values (i.e. 12 bit).

There is a well-known algorithm called "**shift-and-add-3**" algorithm to do this conversion. For example, if we take 8-bit hexadecimal number 8'hff (i.e. all 1's), it has two hex digits. Once converted to binary coded decimal (BCD) it becomes 255 (3 BCD digits).

Shift and Add 3 algorithm [1] – shifting operation

- Let us consider converting hexadecimal number 8'h7C (which is decimal 8'd124)
- Shift the 8-bit binary number left by 1 bit = multiply number by 2
- Shifting the number left 8 times = multiply number by 2^8
- Now truncate the number by dropping the bottom 8 bits = divide number by 2^8
- So far we have done nothing to the number – it has the same value
- The idea is that, as we shift the number left into the BCD digit "bins", we make the necessary adjustment to the hex number so that it conforms to the BCD rule (i.e. falls within 0 to 9, instead of 0 to 15)



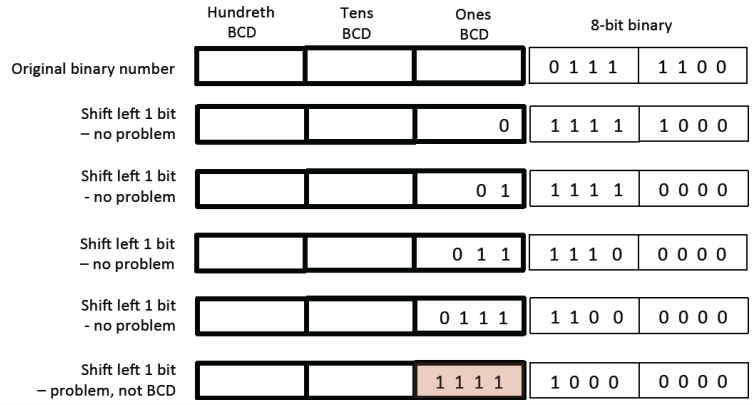
Before we examine this algorithm in detail, let us consider the arithmetic operation of shifting left by one bit. This is the same as a x 2 operation.

If we do it 8 times, then we have multiplied the original number by 256 or 2^8.

Now if you ignore the bottom 8-bit through a truncation process, you effectively divide the number by 256. In other words, we get back to the original number in binary (or in hexadecimal).

Shift and Add 3 algorithm [2] – shift left with problem

- If we take the original 8-bit binary number and shift this three times into the BCD digit positions. After 3 shifts we are still OK, because the **ones digit** has a value of 3 (which is OK as a BCD digit).
- If we shift again (4th time), the digit now has a value of 7. This is still OK. However, no matter what the next bit it, another shift will make this digit illegal (either as hexadecimal "e" or "f", both not BCD).
- In our case, this will be a "F"!



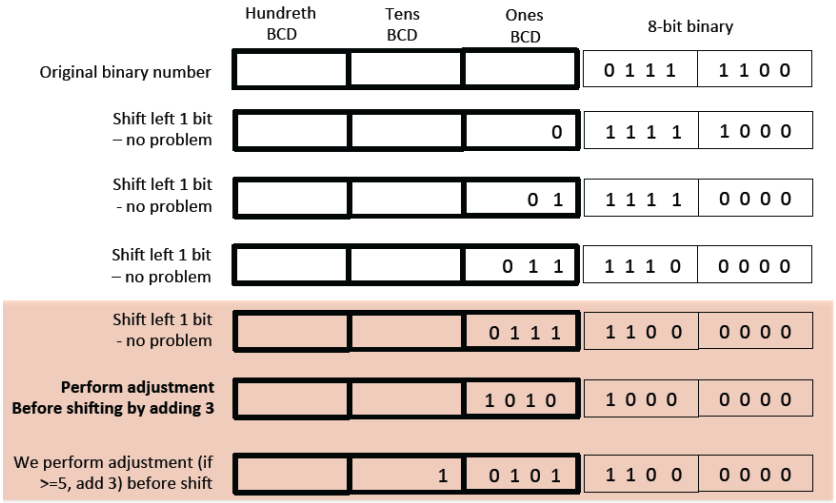
Our conversion algorithms works by shift the number left 8 times, but each time make an adjustment (or correction) if it is NOT a valid BCD digit.

Let us consider this example. We can shift the number four time left, and it will give a valid BCD digit of 7.

However, if we shift left again, then 7 becomes hex F, which is NOT valid. Therefore the algorithm demands that 3 is added to 7 (7 is larger or equal to 5) before we do the shift.

Shift and Add 3 algorithm [3] – shift and adjust

- So on the fourth shift, we detect that the value is > or = 5, then we adjust this number by adding 3 before the next shift.
- In that way, after the shift, we move a 1 into the tens BCD digit as shown here.



The rationale of this algorithm is the following. If the number is 5 or larger, after shift left, we will get 10 or larger, which cannot fit into a BCD digit. Therefore if the number 5 (or larger) we add 3 to it (after shifting is adding 6), which measure we carry forward a 1 to the next BCD digit.

Shift and Add 3 algorithm [4] – full conversion

- In summary, the basic idea is to shift the binary number left, one bit at a time, into locations reserved for the BCD results.
- Let us take the example of the binary number 8'h7C. This is being shifted into a 12-bit/3 digital BCD result of 12'd124 as shown below.

	Hundreth BCD	Tens BCD	Ones BCD	8-bit binary	
Original binary number				0 1 1 1	1 1 0 0
Shift left three times no adjust			0 1 1	1 1 1 0	0
Shift left Ones = 7, ≥5			0 1 1 1	1 1 0 0	
Add 3			1 0 1 0	1 1 0 0	
Shift left Ones = 5		1	0 1 0 1	1 0 0	
Add 3		1	1 0 0 0	1 0 0	
Shift left 2 times Tens = 6, ≥5		1 1 0	0 0 1 0	0	
Add 3		1 0 0 1	0 0 1 0	0	
Shift left BCD value is correct	1	0 0 1 0	0 1 0 0		

To recap: the basic idea is to shift the binary number left, one bit at a time, into locations reserved for the BCD results. Let us take the example of the binary number 8'h7C. This is being shifted into a 12-bit/3 digital BCD result as shown above.

After 8 shift operations, the three BCD digits contain respectively: hundredth digit = 4'b0001, tens digit = 4'b0010 and ones digit = 4'b0100, thus representing the BCD value of 124.

The key idea behind the algorithm can be understood as follow (see the diagram in the slide):

1. Each time the number is shifted left, it is multiplied by 2 as it is shifted to the BCD locations;
2. The values in the BCD digits are the same as as binary if its value is 9 or lower. However if it is 10 or above, the number is wrong for BCD. Instead, it should carry over to the next digit. A correction must be made by adding 6 to this digit value.
3. The easiest way to do this is to detect if the value in the BCD digit locations are 5 or above BEFORE the shift (i.e. X2). If it is ≥5, then add 3 to the value (i.e. adjust by +6 after the shift).

SystemVerilog implementation - bin2bcd_8.sv

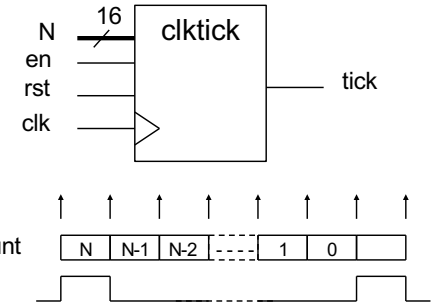
```
14  module bin2bcd_8 (
15     input  logic [7:0]   x,       // value ot be converted
16     output logic [11:0]  BCD     // BCD digits
17  );
18     // Concatenation of input and output
19     logic  [19:0] result;  // = bit in x + 4 * no of digits
20     integer i;
21
22     always_comb
23     begin
24        result[19:0] = 0;
25        result[7:0] = x;     // bottom 8 bits has input value
26
27        for (i=0; i<8; i=i+1) begin
28           // Check if unit digit >= 5
29           if (result[11:8] >= 5)
30              result[11:8] = result[11:8] + 4'd3;
31
32           // Check if ten digit >= 5
33           if (result[15:12] >= 5)
34              result[15:12] = result[15:12] + 4'd3;
35
36           // Shift everything left
37           result = result << 1;
38        end
39
40        // Decode output from result
41        BCD = result[19:8];
42     end
43
44  endmodule
```


Here is the SystemVerilog implementation of the binary to BCD algorithm. You are invited to examine how the algorithm described in previous slides are implemented in this behavioural description in SV.

Note that although this description looks like a software function, synthesis program will produce hardware implementation of it, say, in FPGA.

A Flexible Timer – clktick.sv

- Instead of having a counter that count events, we often want a counter to provide a measure of **time**. We call this a **timer**.
- Here is a useful **timer** component that uses a clock reference, and produces a pulse lasting for one cycle every N+1 clock cycles.
- If "en" signal is low (not enabled), the clkin pulses are ignored.




```
module clktick #(
    parameter WIDTH = 16
)(
  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // enable signal
  input  logic [WIDTH-1:0] N,        // clock divided by N+1
  output logic             tick      // tick output
);

logic [WIDTH-1:0] count;
```


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 component called a "clock tick" circuit. We are not interested in the actual count value. What is needed, however, is that the circuit generates a **single clock pulse** (i.e. lasting for one clock period) for every **N+1 rising edge** of the clock input signal **clk**.

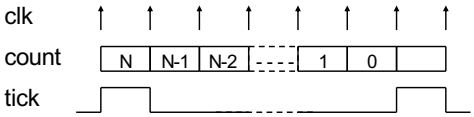
We also add an enable signal **en**, which must be set to '1' in order to enable the internal counting circuit.

Shown here is the module interface for this circuit in SystemVerilog.

Note that the **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.

clktick.sv explained

- "count" is an internal counter with WIDTH bits
- We use this as a down (instead of up) counter
- The counter value goes from N to 0, hence there are N+1 clock cycles for each tick pulse




```
always_ff @ (posedge clk)
    if (rst) begin
        tick <= 1'b0;
        count <= N;
        end
    else if (en) begin
      if (count == 0) begin
        tick <= 1'b1;
        count <= N;
          end
      else begin
        tick <= 1'b0;
        count <= count - 1'b1;
          end
        end
endmodule
```


The actual SystemVerilog specification for this module is shown here.

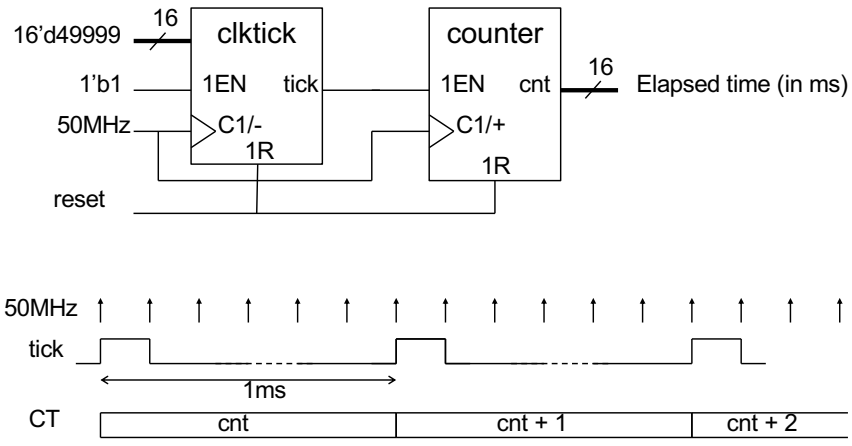
There has to be an internal counter **count** whose output is NOT visible external to this module. This is created with the **reg [N_BIT-1:0] count;** statement.

The output **tick** has to be declared as **reg** 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.

Cascading counters

- By connecting **clktick** module in series with a counter module, we can produce a counter that counts the number of millisecond elapsed as shown below.



Using this style to design 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 a 50MHz clock. Therefore **tick** goes high for 20 nanosecond 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).

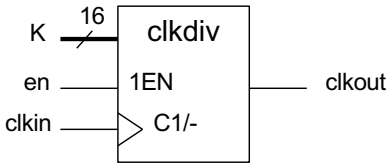
The second counter is now counting the number of millisecond that has elapsed since the last time reset signal (1R) goes high.

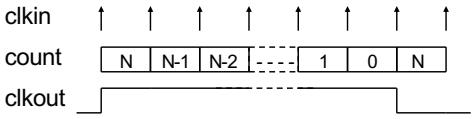
The design of this circuit is left as a Laboratory task for you to do.

In case you are not familiar with the schematic notation here (which is a IEEE standard), C1/- indicates that the clock input is synchronized to the enable and reset input (1EN and 1R), and it results in circuit counting DOWN ('-' sign).

Clock divider (clkdiv.sv)

- Another useful module is a **clock divider circuit**.
- This produces a **symmetrical** clock output, dividing the input clock frequency by a factor of 2*(K+1).



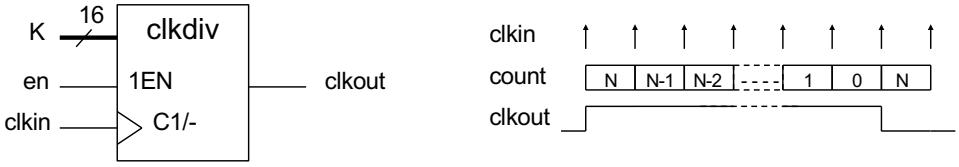



```
module clkdiv #(
    parameter WIDTH = 16
)(
  input  logic             clkin,  // Clock input signal to be divided
  input  logic             en,     // enable clk divider when high
  input  logic [WIDTH-1:0] K,      // half clock period counts K+1 clkin cycles
  output logic             clkout  // symmetric clock output Fout = Fin / 2*(K+1)
);          //  End of port list

logic [WIDTH-1:0] count;        // internal counter
```


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 SystemVerilog. Again we have used the **parameter** statement to make this design ease of modification for different internal counter size.

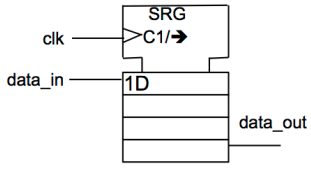
clkdiv.v explained

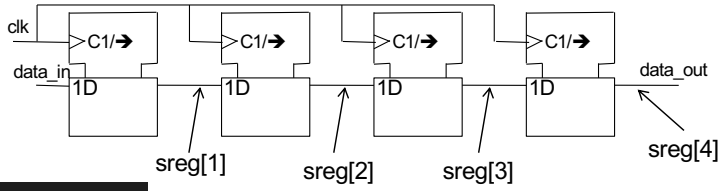



```
initial clkout = 1'b0;          // alternative way to initialise logic
initial count = {WIDTH{1'b0}}   // ... or FF (Good for FPGA designs)

//------------ Main Body of the module -------------------------

    always_ff @ (posedge clkin)
        if (en == 1'b1)
            if (count == {WIDTH{1'b0}}  begin
                clkout <= ~clkout;              // toggle the clock output
                count <= K;  // shift right one bit
                end
            else
                count <= count - 1'b1;
endmodule // End of Module clkdiv
```

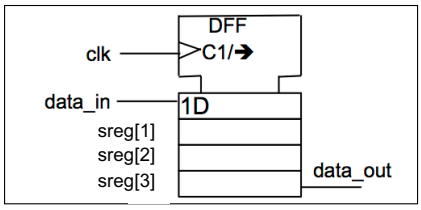

The Verilog specification is similar to that for **clktick**. This also has an internal counter that counts from K to 0, then the output **clkout** is toggled whenever the count value reaches 0.

Shift Register specification in SystemVerilog






```
module sreg4 (
  input  logic      clk,        // input clock
  input  logic      rst,        // reset
  input  logic      data_in,    // serial data in
  output logic      data_out    // serial data out
);          //  End of port list

   logic [4:1]   sreg;         // shift register

   always_ff @ (posedge clk)
       if (rst)
           sreg <= 4'b0;
       else  begin
           sreg[4] <= sreg[3];
           sreg[3] <= sreg[2];
           sreg[2] <= sreg[1];
           sreg[1] <= data_in;
       end
   assign data_out = sreg[4];
endmodule
```




```
sreg <= {sreg[3:1], data_in};
```

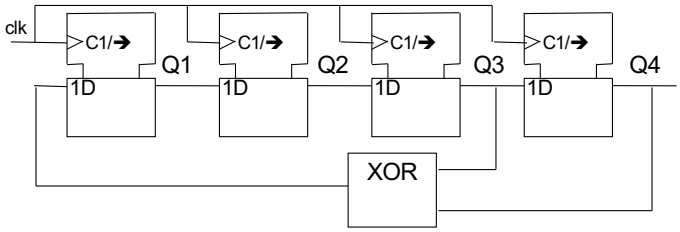

To specify a shift register in SystemVerilog, use the code shown here. We use the **<=** assignment to make sure that **sreg[4:1]** are updated only at the end of the **always** block.

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

sreg <= {sreg[3:1], data_in}

This way of specifying the right-hand side of the assignment is powerful. We use the **concatenation operation** { …. } to make up four bits from **sreg[3:1]** and **data_in** (with **data_in** being the LSB) and assign it to **sreg[4:1]**.

Linear Feedback Shift Register (LFSR) (1)



- Assuming that the initial value is 4'b0001.
- This shift register counts through the sequence as shown in the table here.
- This is now acting as a 4-bit counter, whose count value appears somewhat random.
- This type of shift register circuit is called "**Linear Feedback Shift Register**" or LFSR.
- Its value is sort of random, but repeat very 2^N-1 cycles (where N = no of bits).
- The "taps" from the shift register feeding the XOR gate(s) is defined by a polynomial as shown above.

Q4	Q3	Q2	Q1	count
0	0	0	1	1
0	0	1	0	2
0	1	0	0	4
1	0	0	1	9
0	0	1	1	3
0	1	1	0	6
1	1	0	1	13
1	0	1	0	10
0	1	0	1	5
1	0	1	1	11
0	1	1	1	7
1	1	1	1	15
1	1	1	0	14
1	1	0	0	12
1	0	0	0	8
0	0	0	1	1

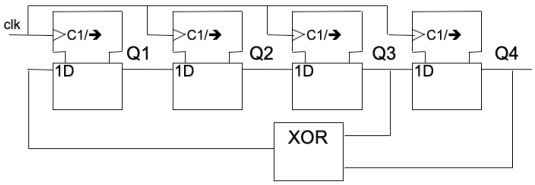
We can also make a shift register count in binary, but in an interesting sequence.

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.

Primitive Polynomial



Primitive polynomial: $1 + X^3 + X^4$

- This circuit implements the LFSR based on this **primitive polynomial**:
- The polynomial is of order 4 (highest power of x)
- This produces a **pseudo random binary sequence** (PRBS) of length $2^4 - 1 = 15$
- Here is a table showing primitive polynomials at different sizes (or orders)

m	
3	$1+X+X^3$
4	$1+X+X^4$
5	$1+X^2+X^5$
6	$1+X+X^6$
7	$1+X^3+X^7$
8	$1+X^2+X^3+X^4+X^8$
9	$1+X^4+X^9$
10	$1+X^3+X^{10}$
11	$1+X^2+X^{11}$
12	$1+X+X^4+X^6+X^{12}$
13	$1+X+X^3+X^4+X^{13}$

m	
14	$1+X+X^6+X^{10}+X^{14}$
15	$1+X+X^{15}$
16	$1+X+X^3+X^{12}+X^{16}$
17	$1+X^3+X^{17}$
18	$1+X^7+X^{18}$
19	$1+X+X^{2+}X^5+X^{19}$
20	$1+X^3+X^{20}$
21	$1+X^2+X^{21}$
22	$1+X+X^{22}$
23	$1+X^5+X^{23}$
24	$1+X+X^{2+}X^7+X^{24}$

PYKC 12 Nov 2024 EE2 Circuits and Systems Lecture 9 Slide 18

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 shown in the slide has primitive polynomial at various order. For example, a 15-bit LFSR that produces a maximal length PRBS can be achieved by implementing the primitive polynomial:

$$1 + X + X^{15}$$

This result in a 15-bit shift register with one XOR gate from Q1 and Q15, feeding back to input of FF1 (which we would label Q0). This effective implements the equation:

$$X^0 = X + X^{15}$$

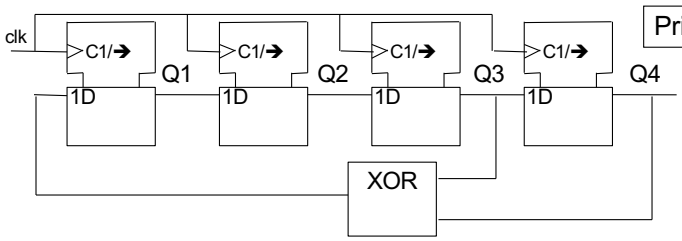
Note that for a given order, the primitive polynomial shown here is NOT unique.

For example, for order 4, the table shows an alternative primitive polynomial:

$$1 + X + X^4$$

This will produce a pseudo-random sequence which is also maximal length different from that using the polynomial in the slide ($1 + X^3 + X^4$).

lfsr4.sv



Primitive polynomial: $1 + X^3 + X^4$

```
module lfsr4 (
    input logic         clk,        // clock
    input logic         rst,        // reset
    output logic [4:1]  data_out    // pseudo-random output
);

always_ff @ (posedge clk, posedge rst)
    if (rst)
        sreg <= 4'b1;
    else
        sreg <= {sreg[3:1], sreg[4] ^ sreg[3]};

assign data_out = sreg;
endmodule
```

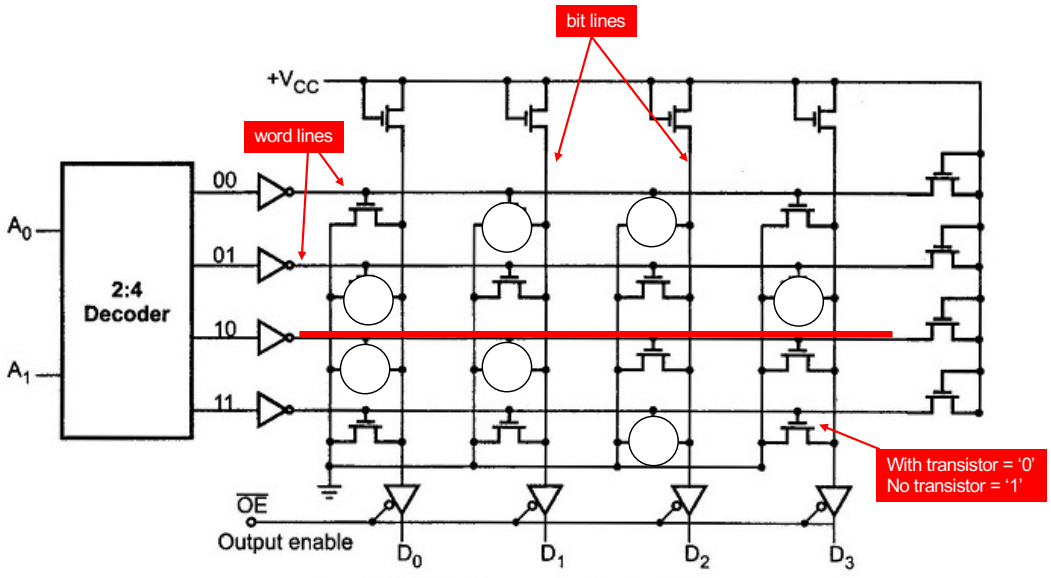

Here is the implementation of a 4-bit LFSR of the primitive polynomial

$$1 + X^3 + X^4$$

This is essentially a shift register with data_in feed from an XOR gate with Q3 and Q4. Note that we MUST initialize the shift register to a value other than 4'b0000 (e.g. 4'b0001 will do).

This module has not been parameterized because for different WIDTH, we need different primitive polynomial!

Simplified 4 x 4 ROM array



This is a simplified internal circuit of a 4 words x 4 bits Read-Only Memory (ROM) component.

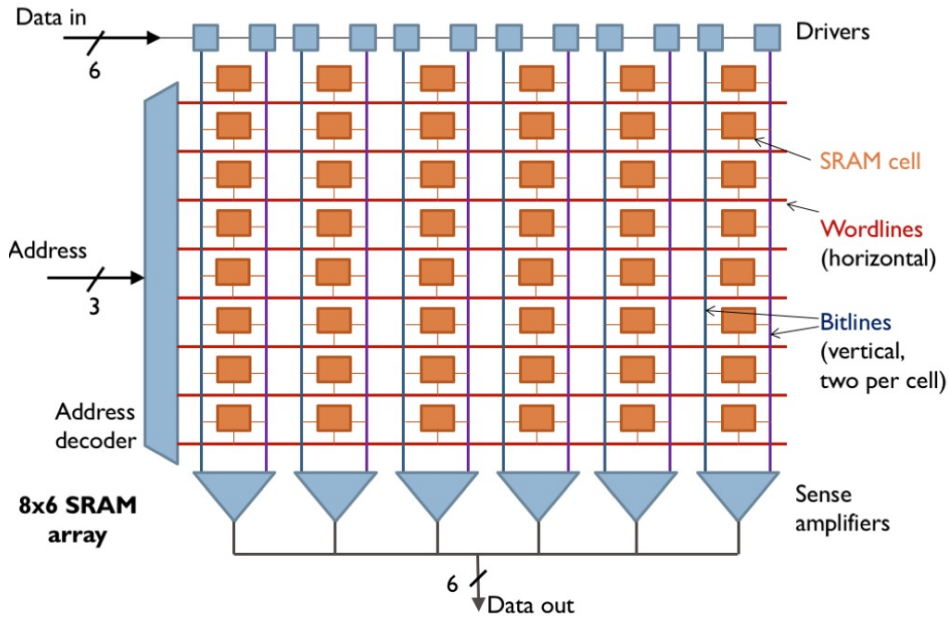
It consists of a 2-D array of transistors, which turns ON when their gate terminals are asserted (high). A '0' is stored if a transistor is present. A '1' is stored if the transistor is omitted.

The two bit address is decoded by the 2:4 decoder into one-hot code, 1110, 1101, 1011 or 0111, which brings one of the four WORD lines high.

Wherever a transistor is present, the vertical BIT line is pull down to zero, otherwise the the BIT line is low.

For example, if A1:A0 is 2'b10, then the third word line is high. The output will be D[3:0] = 4'b0011.

Simplified 8 x 6 Static RAM array



This slide shows a typical organisation inside a RAM chip. Memory cells are usually organised in the form of a 2-D array of RAM cells. In this case, the address is 3 bits, therefore there are 8 words in this memory. Only ONE ROW will be enable at any one time (hence one-hot).

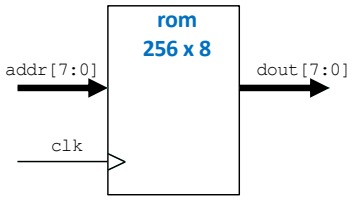
Similar to the last slide, each bit is a memory cell. In this case, each cell is more complex than a single transistor. Instead, a static memory cell is usually a simple cross-coupled inverter with read/write transistors – normally 6 transistor cell. There are now two bit lines per cell (Q and Qbar).

The output buffer is called a sense amplifier. It sense the DIFFERENCE between the two complementary bit lines. Detail of a SRAM cell design is outside the scope of this module.

System Verilog specification of 256 x 8 ROM

```
1  module rom  #(
2      parameter   ADDRESS_WIDTH = 8,
3                  DATA_WIDTH = 8
4  )(
5      input logic                      clk,
6      input logic  [ADDRESS_WIDTH-1:0] addr,
7      output logic  [DATA_WIDTH-1:0]    dout
8  );
9
10 logic [DATA_WIDTH-1:0] rom_array [2**ADDRESS_WIDTH-1:0];
11
12 initial begin
13         $display("Loading rom.");
14         $readmemh("sinerom.mem", rom_array);
15 end;
16
17 always_ff @(posedge clk)
18     // output is synchronous
19     dout <= rom_array [addr];
20
21 endmodule
22
```

Verilator gives a warning unless you add an extra line!!!!!

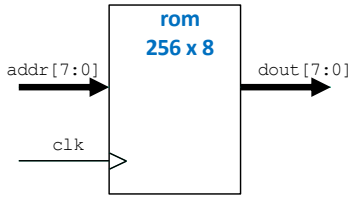


For this module, we will not worry about the physical implementation of a ROM or RAM component. Instead, we will specify them behaviourally. This allows digital system to be modelled, simulated and verified. While counters, shift registers and other circuits are synthesized to produce transistors and gates, memories are mapped to pre-designed blocks. For example, memory in FPGAs are usually explicitly instantiated as embedded RAM. This is because synthesized memory cells are synthesized into D-FF, and are large and expensive in resources.

Shown here is a 256 x 8 bit ROM model in SystemVerilog. This is specified as a synchronous ROM. The ROM output data only appears on **dout** on positive edge of **clk**. Here we also omit the output enable (OE) control signal.

Initialization of the ROM

```
initial begin
        $display("Loading rom.");
        $readmemh("sinerom.mem", rom_array);
end;
```



sinegen.py

```
import math
import string
f = open("sinerom.mem","w")
for i in range(256):
    v = int(math.cos(2*3.1416*i/256)*127+127)
    if (i+1)%16 == 0:
        s = "{hex:2X}\n"
    else:
        s = "{hex:2X} "
    f.write(s.format(hex=v))

f.close()
```

sinerom.mem

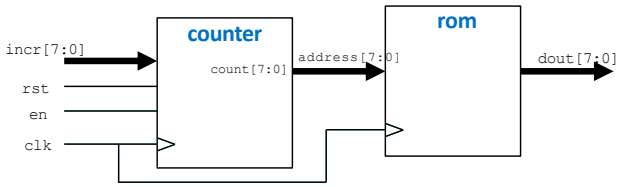
```
FE FD FD FD FD FC FC FB FA FA F9 F8 F7 F6 F5
F4 F3 F1 F0 EF ED EB EA E8 E6 E5 E3 E1 DF DD DA
D8 D6 D4 D1 CF CD CA C8 C5 C2 C0 BD BA B8 B5 B2
AF AC A9 A6 A3 A0 9D 9A 97 94 91 8E 8B 88 85 82
7E 7B 78 75 72 6F 6C 69 66 63 60 5D 5A 57 54 51
4E 4B 48 45 43 40 3D 3B 38 35 33 30 2E 2C 29 27
25 23 20 1E 1C 1A 18 17 15 13 12 10  E  D  C  A
 9  8  7  6  5  4  3  3  2  1  1  0  0  0  0  0
 0  0  0  0  0  0  1  1  2  3  3  4  5  6  7  8
 9  A  C  D  E 10 12 13 15 17 18 1A 1C 1E 20 23
25 27 29 2C 2E 30 33 35 38 3B 3D 40 43 45 48 4B
4E 51 54 57 5A 5D 60 63 66 69 6C 6F 72 75 78 7B
7F 82 85 88 8B 8E 91 94 97 9A 9D A0 A3 A6 A9 AC
AF B2 B5 B8 BA BD C0 C2 C5 C8 CA CD CF D1 D4 D6
D8 DA DD DF E1 E3 E5 E6 E8 EA EB ED EF F0 F1 F3
F4 F5 F6 F7 F8 F9 FA FA FB FC FC FD FD FD FD FD
```


ROM needs to “programmed” or configured with original contents. In SystemVerilog, the **$readmemh(.)** function allows the ROM to be loaded with the contents stored in a file with numbers stored as hexadecimal code as shown in the slide.

How is the text file **sinerom.mem** generated? For Lab 2, Task 1, this file contains 256 samples of a single cycle cosine values with a number ranging from 8’h00 to 8’hFF.

Sinerom.mem is generated with a simple Python script shown here. You don’t need to know Python. You could use any tools to produce this file, e.g. C++ or Matlab.

Simple Sinewave Generator




```
1  module sinegen #(
2      parameter   A_WIDTH = 8,
3                  D_WIDTH = 8
4  )(
5    // interface signals
6    input  logic               clk,      // clock
7    input  logic               rst,      // reset
8    input  logic               en,       // enable
9    input  logic [D_WIDTH-1:0] incr,     // increment for addr counter
10   output logic [D_WIDTH-1:0] dout      // output data
11 );
12
13   logic  [A_WIDTH-1:0]       address;  // interconnect wire
14
15 counter addrCounter (
16   .clk (clk),
17   .rst (rst),
18   .en (en),
19   .incr (incr),
20   .count (address)
21 );
22
23 rom sineRom (
24   .clk (clk),
25   .addr (address),
26   .dout (dout)
27 );
28
29 endmodule
```

Instantiate counter module called addrCounter

external signal name

Internal signal name

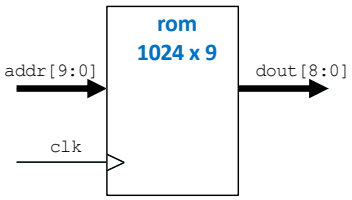
A simple sinewave generator can be designed with combining **counter.sv** and **rom.sv**. The counter produces the address of the ROM, and the output is the sine (or cosine) values. The frequency of the output sinewave is determined by **incr[7:0]**. If incr = 1, then the sinewave period is 256 x clock period. In general, the output sinewave frequency is:

$$f_{out} = f_{clk} * incr/256$$

Note how this top-level module **sinegen.sv** instantiate the two components: **counter** and **rom**.

Parameterised ROM:

```
1  module rom  #(                          counter
2      parameter   ADDRESS_WIDTH = 8,
3                  DATA_WIDTH = 8
4  )(
5      input logic                         clk,
6      input logic  [ADDRESS_WIDTH-1:0] addr,
7      output logic  [DATA_WIDTH-1:0]    dout
8  );
9
10 logic [DATA_WIDTH-1:0] rom_array [2**ADDRESS_WIDTH-1:0];
11
12 initial begin
13         $display("Loading rom.");
14         $readmemh("sinerom.mem", rom_array);
15 end;
16
17 always_ff @(posedge clk)
18     // output is synchronous
19     dout <= rom_array [addr];
20
21 endmodule
22
```

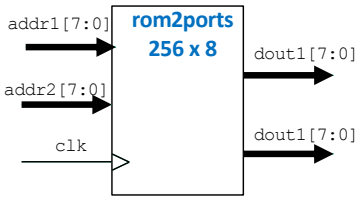



```
rom #(10, 9) sineRom_1024x9 (.....)
```


Note that **rom.sv** is defined with two parameters: **ADDRESS_WIDTH** and **DATA_WIDTH**. These are given default values. However, if you need a ROM that is 1024 x 9 bit instead of 256 x 8 bit, you can simple specifiy these parameter as shown here when instantiating the ROM component. The order of the parameters is important!

Dual-port ROM

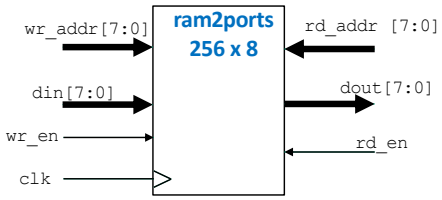
```
1   module rom2ports  #(
2       parameter   ADDRESS_WIDTH = 8,
3                   DATA_WIDTH = 8
4   )(
5       input logic                         clk,
6       input logic  [ADDRESS_WIDTH-1:0]    addr1,
7       input logic  [ADDRESS_WIDTH-1:0]    addr2,
8       output logic  [DATA_WIDTH-1:0]      dout1,
9       output logic  [DATA_WIDTH-1:0]      dout2
10  );
11
12  logic [DATA_WIDTH-1:0] rom_array [2**ADDRESS_WIDTH-1:0];
13
14  initial begin
15          $display("Loading rom.");
16          $readmemh("sinerom.mem", rom_array);
17  end;
18
19  always_ff @(posedge clk) begin
20      // output is synchronous
21      dout1 <= rom_array [addr1];
22      dout2 <= rom_array [addr2];
23      end
24
25  endmodule
```



In designing on-chip memory for microprocessors, we often need to perform more than one access operations simultenously to the same memory. Here is a specification for a dual-port ROM. The actual SystemVerilog code is very simple and obvious. Now a user can read from two separate memory location at the same time.

Dual-port RAM

```
1  module ram2ports  #(
2      parameter   ADDRESS_WIDTH = 8,
3                  DATA_WIDTH = 8
4  )(
5      input logic                     clk,
6      input logic                     wr_en,
7      input logic                     rd_en,
8      input logic  [ADDRESS_WIDTH-1:0] wr_addr,
9      input logic  [ADDRESS_WIDTH-1:0] rd_addr,
10     input logic  [DATA_WIDTH-1:0]    din,
11     output logic [DATA_WIDTH-1:0]    dout
12 );
13
14 logic [DATA_WIDTH-1:0] ram_array [2**ADDRESS_WIDTH-1:0];
15
16 always_ff @(posedge clk) begin
17     if (wr_en == 1'b1)
18         ram_array[wr_addr] <= din;
19     if (rd_en == 1'b1)
20         // output is synchronous
21         dout <= ram_array [rd_addr];
22 end
23 endmodule
```



Here is the design of a dual-port RAM. We need more control signals: the specify whether we are reading or writing to the RAM.

Such a component is extremely important in any digital system design because we often need to perform both read and write operations at the same time.

What if the read and write addresses are identical? For example, if memory location **8'hA2** of the RAM stores a value **8'h33**, and you want to write a new value **8'h44** to the same address location, what do you think the value of **dout** is? Why?