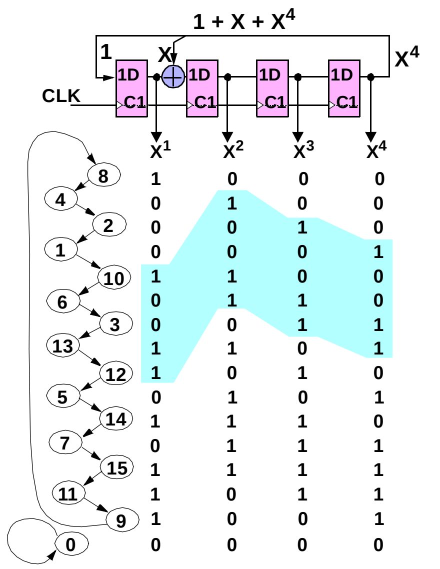
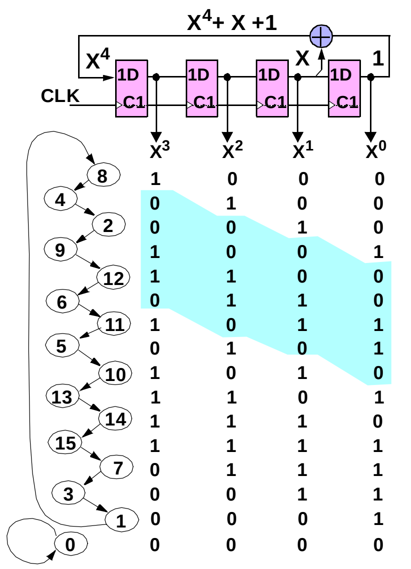
Linear Feedback Shift Registers (LFSR)

A type of circuit made from XOR ⊕ and D ff which give repeatable "random" numbers.



X^1	X^2	X^3	X^4
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	1	0	0
0	1	1	0
0	0	1	1
1	1	0	1
1	0	1	0
0	1	0	1
1	1	1	0
0	1	1	1
1	1	1	1
1	0	1	1
1	0	0	1
0	0	0	0

Internal circuit, XORs inside shift reg.



X^3	X^2	X^1	X^0
1	0	0	0
0	1	0	0
0	0	1	0
1	0	0	1
1	1	0	0
0	1	1	0
1	0	1	1
0	1	0	1
1	0	1	0
1	1	0	1
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

External circuit, XORs outside shift reg.

Linear Feedback Shift Registers

Properties

Names

Linear-Feedback Shift-Register (LFSR), *Autonomous* LFSR, *Pseudo-Random-Number Generators*, *Polynomial Sequence Generators*, *Pseudo-Random-Pattern generators*, etc.

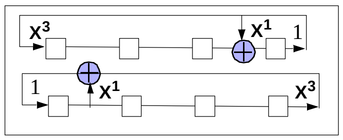
Math

The connections to the feedback loop are given placeholder names which are powers of X.
One end is always $X^0=1$, the other is always X^n. The others are X^k if there is an XOR connection at k.

The powers of X define a polynomial.
If the polynomial is what we call primitive, the circuit will sequence through 2^N-1 numbers for N flip-flops.
Else it will have several shorter sequences and will not be much use.

Circuits

There are four circuits with the same polynomial and close to the same properties:
- An internal circuit.
- An external circuit.
- An internal circuit with the connections and labelling reversed
- An external circuit with the connections and lab



The polynomials are the same. If one is primitive they both are.
However the sequencing of the numbers will be different.

Primitive Polynomials

They are listed in many books.[1]

[1] V.N. Yarmolik, and I.V Kachan, *Self Testing VLSI Design*, Elsevier 1993. It goes up to 300 flip-flops.

Linear Feedback Shift Registers

Properties of LFSR

Names

- ***Linear-Feedback Shift-Register*** **(LFSR),** ***Pseudo-Random-Number Generators, Polynomial Sequence Generators*** **etc., etc.**
- **Individual circuits have polynomial names related to their connections; i.e. $1 + X + X^4$**
- **Can deduce the properties of the circuit from its polynomial. (and a math degree)**
- **Use certain polynomials called *primitive*.**

Circuit

- **Uses only a few (1 to 3) XOR gates, and D flip-flops.**
- **Internal circuit is very fast. Max delay = (1 XOR delay) + (1 D ff delay).**
- **Primitive polynomials have 2^N-1 sequential states.**
- **The all zero state is always isolated.**
  If you reset a LFSR at the start, it locks up in the all zero state.

Randomness (primitive polynomials)

- **It obeys 15 of 25 standard statistical tests**
- **Consecutive numbers have a shift register relation (particularly external). Columns are identical but displaced (see background shading).**
- **No number is repeated until the complete sequence is done.**
- **Mean = $-1/(2^N -1) \Rightarrow$ slight dc bias**
  There is one more 1 than there are 0s in any column.

Linear-Feedback Shift-Registers (cont.)

Properties

Size

A LFSR takes less area than any other common counter except a ripple counter.

The ripple counter is not synchronous and much harder to interface reliably.

Speed

A LFSR is faster than any other common counters except the Mobius counter.

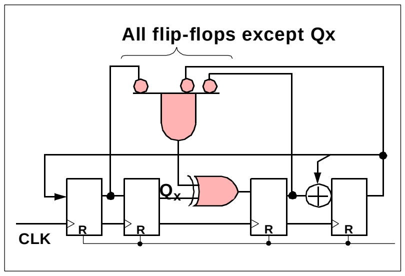
Counting

It does not count in binary. It counts modulo 2^N-1, a binary counter counts modulo 2^N.

Applications

Where one needs to count a fixed time, but do not need to need arithmetic compatible intermediate values.
Time out counts; shut off the screen if no one touched the keyboard for 2^N-1 seconds.
Send an input to the control computer every 100ms, reboot if no response.

Cycling through addresses for refreshing DRAM. T he order of refresh does not matter. Not going through address 0000 might matter.

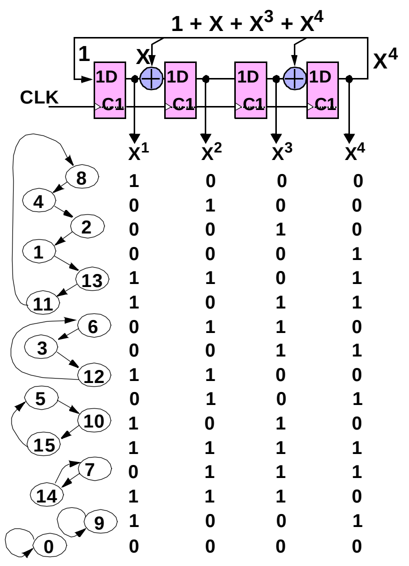


1.• PROBLEM

In any LFSR, insert the following circuit between two adjacent flip-flops which were not previously connected through an XOR. The previous state graph will have looked like that of the "Linear-Feedback Shift-Registers (cont.)," p. 220.
(a) What will the new state graph look like?
(b) What will happen if you start by resetting all the flip-flops?

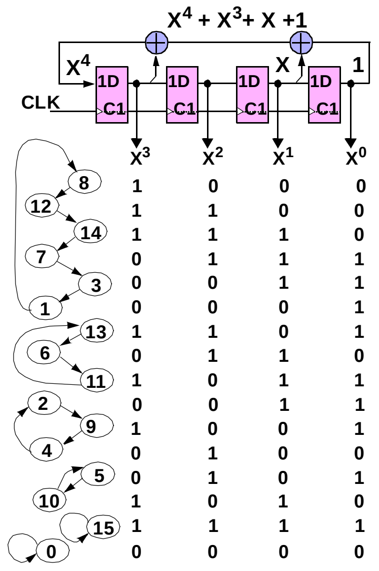
Linear-Feedback Shift-Registers (cont.)

Example of a Nonprimitive Polynomial



	X^1	X^2	X^3	X^4
8	1	0	0	0
4	0	1	0	0
2	0	0	1	0
1	0	0	0	1
13	1	1	0	1
11	1	0	1	1
6	0	1	1	0
3	0	0	1	1
12	1	1	0	0
5	0	1	0	1
10	1	0	1	0
15	1	1	1	1
7	0	1	1	1
14	1	1	1	0
9	1	0	0	1
0	0	0	0	0

Internal circuit, XORs inside shift reg.



	X^3	X^2	X^1	X^0
8	1	0	0	0
12	1	1	0	0
14	1	1	1	0
7	0	1	1	1
3	0	0	1	1
1	0	0	0	1
13	1	1	0	1
6	0	1	1	0
11	1	0	1	1
2	0	0	1	0
9	1	0	0	1
4	0	1	0	0
5	0	1	0	1
10	1	0	1	0
15	1	1	1	1
0	0	0	0	0

External circuit, XORs outside shift reg.

Minimal Hardware, Primitive Polynomials

- $1+x+x^2$ $1+x+x^3$ $1+x+x^4$ $1+x^2+x^5$ $1+x+x^6$ $1+x+x^7$
  $1+x+x^5+x^6+x^8$ $1+x^4+x^9$ $1+x^3+x^{10}$ $1+x^2+x^{11}$ $1+x^3+x^4+x^7+x^{12}$
  $1+x+x^3+x^4+x^{13}$ $1+x+x^{15}$ $1+x^2+x^3+x^5+x^{16}$ $1+x^3+x^{17}$ $1+x^7+x^{18}$ $1+x^3+x^{20}$
  $1+x^2+x^{21}$ $1+x+x^{22}$ $1+x^5+x^{23}$ $1+x+x^3+x^4+x^{24}$ $1+x^3+x^{25}$ $1+x+x^7+x^8+x^{26}$
- For N flip-flops, one seems to only need 3 or fewer XOR gates.

Starting

- All LFSR lockup in the all zero state. A very reliable circuit would check if the flip-flops are all zero and inject a one. Less reliable circuits will initialize some of the flip-flops to one.

Randomness

- The numbers as integers look random. The numbers as bit patterns show the shifting strongly.
- Some applications, like testing with random numbers, scramble the leads going to the various flip-flops. This gives a better test for multipliers or barrel shifters.

Bias

- Some communication circuits do not like inputs that have a dc bias. To make a random generator that has no bias, and includes zero, Take the output of the most significant bit of the generator though an inverter. This will change 100...0, the most negative 2’s complement number, into 000...0, and balance the plus and minus numbers

dec	2’s compl	LFSR (no zero)	LFSR with inverter
3	011	011 →	111
2	010	010 →	110
1	001	001 →	101
0	000		000
-1	111	111 →	011
-2	110	110 →	010
-3	101	101 →	001
-4	100	100	

2.• PROBLEM

Modify the Verilog code for the shift register to make a LFSR for 7 flip-flops ($1+x+x^7$).

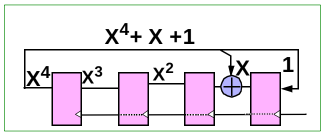
Verilog LFSR

- One can specify a subset of bits like Reg[4:2].
- One must not reset a LFSR to 0.

```
module LFSR2(Reg, clk, reset);
    parameter n=4                    // Change more than n to change LFSR length.
    output[n:1]Reg;
    reg   [n:1]Reg;                  //All procedure outputs must be registered
    input clk  reset;

        always @(posedge clk
                or posedge reset)
        if
         (reset) Reg <=1;
        else
         Reg <= {Reg[n-1:2], Reg[n]^Reg[1], Reg[n]};

endmodule //LFSR2
```



Nonblocking Assignment

The <= assignment in procedures is called *nonblocking.*

The variables on the right of <= are captured in parallel on the @ trigger.

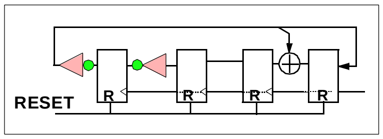
The variables on the left are always calculated from these initial values.

- It is a master-slave like operation.
- If one uses nonblocking anywhere in a procedure one must use it everywhere.

Verilog LFSR

Reset To Zero

One must never clear a LFSR. If the flip-flops have a set, use it. If not place inverters before and after the flip-flop to fake a set.



Parameters

Parameters are very handy. I would recommend using them frequently.
Here changing n will not be enough to change the LFSR. One must check that the XOR is in the right place for the larger size. However it saves having to change output, reg and other statements.

always @(posedge clk or posedge reset)

The @ is the edge triggered signal. Most counters, registers, flip-flops should start with this way.

Blocking and Nonblocking

Normal *blocking* assignment in procedures is via the "=." Such variables behave like a normal program. Thus:

```
R[1]= R[n];
```

will replace R[1]. Then:

```
R[2] = R[n]^R[1];
```

will use the revised R[1] in the calculation. This is not what is wanted!

Nonblocking assignment uses the "<=" symbol. Nonblocking assignment is more like a register of D flip-flops. The inputs are transfered across the "<=" at once in the same way that all flip-flops are clocked at once.

Use Nonblocking <= for Flip-flops, Counters and Shift Registers

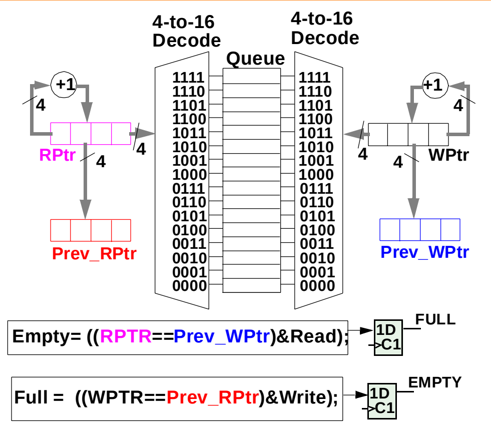
Your reflex action should be to use "<=" for all procedures containing flip-flops.

■ Linear Feedback Shift Registers ■

Application of the LFBSR:

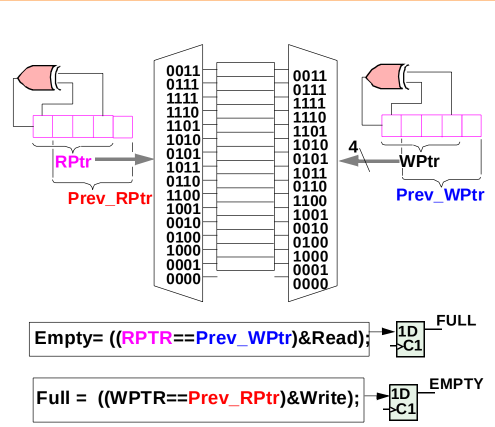
The Circular Queue (FIFO)

Locating *Queue Empty* and *Queue Full* require comparing present and previous pointers
A shift-register can remember its previous state using 1 extra ff.
LFSR is smaller and faster than a binary counter.
The strange counting order of the LFSR does not matter.



Empty= ((RPTR==Prev_WPtr)&Read);

Full = ((WPTR==Prev_RPtr)&Write);



Empty= ((RPTR==Prev_WPtr)&Read);

Full = ((WPTR==Prev_RPtr)&Write);

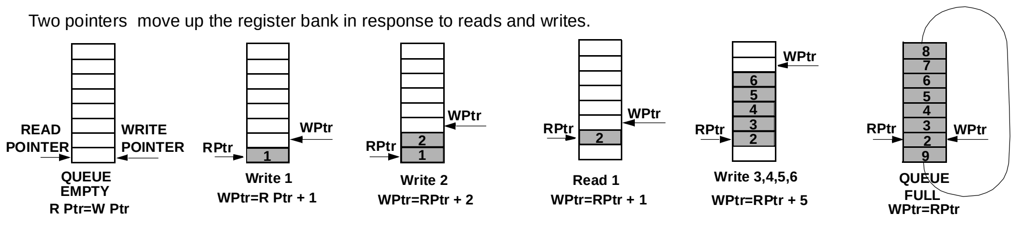
Using N-bit Binary Counters
Use 4N ff
Incrementer hardware x2
Counter carry-chain delay (next p.)

Using LFSR
Use 2N+2 ff — **2N+4 for 2^N queue**
2 XOR gates — **6 for a 2 a ^{N} queue**
1 XOR gate delay — **3 for a 2^N queue**

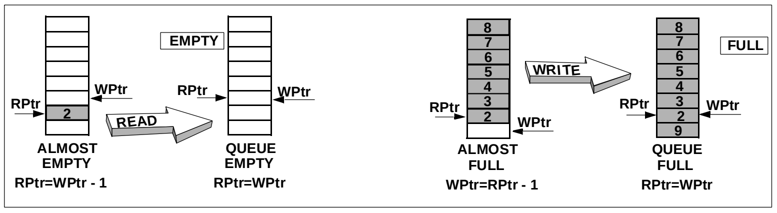
Applications of LFSR

Circular Queue or First-In-First-Out Register

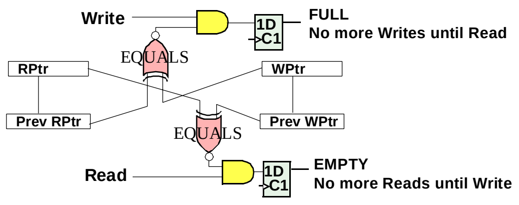
Two pointers move up the register bank in response to reads and writes.



Queue full and queue empty both have:-
  Wptr = RPtr
To tell full from empty compare the pointers. If
  WPtr =RPtr -1
and one is writing, that write will fill the queue and should set a *full* flag.



Similarly if
  RPtr =WPtr -1   i.e. RPtr = Previous WPtr
and one is reading, then that read will empty the queue. One should set an *empty* flag and allow no more reads until after a write.



3.• PROBLEM

Implement the queue with $2*2^N$ bit shift registers.

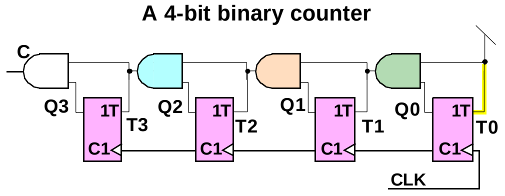
Binary Counters

Binary Up-Counter

Simplest Circuit Uses T Flip-Flops

(Simplest for People)

A 4-bit binary counter



The T table shows where bits toggle going between the present state and the next state.

The circles show how the toggle signals are defined by ANDs:

- **T1 = Q_0**
- **T2 = Q_1Q_0**
- **T3 = $Q_2Q_1Q_0$**
- **TC = $Q_3Q_2Q_1Q_0$**

Propagation Delay

Notice the ripple carry. All binary counter have this speed limiting carry.

Present State				Next State				What to Toggle			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	T_3	T_2	T_1	T_0
0	**0**	**0**	**0**	0	0	0	1				**T**
0	0	0	1	0	0	1	0			T	**T**
0	0	1	0	0	0	1	1				**T**
0	0	1	1	0	1	0	0		T	T	**T**
0	1	0	0	0	1	0	1				**T**
0	1	0	1	0	1	1	0			T	**T**
0	1	1	0	0	1	1	1				**T**
0	1	1	1	1	0	0	0	T	T	T	**T**
1	0	0	0	1	0	0	1				**T**
1	0	0	1	1	0	1	0			T	**T**
1	0	1	0	1	0	1	1				**T**
1	0	1	1	1	1	0	0		T	T	**T**
1	1	0	0	1	1	0	1				**T**
1	1	0	1	1	1	1	1			T	**T**
1	1	0	0	1	1	1	1				**T**
1	1	1	1	**0**	**0**	**0**	**0**	T	T	T	**T**

Binary Up-Counter

Design of the Counter

The Karnaugh maps for the toggle inputs are given below. Many people are more used to them, and they are much less error prone than picking bits off the state tables.

Map for T3

Q_3Q_2 \ Q_1Q_0	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	0	1	0
10	0	0	0	0

Map for T2

Q_3Q_2 \ Q_1Q_0	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	0	0	1	0
10	0	0	1	0

Map for T1

Q_3Q_2 \ Q_1Q_0	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

Map for T0

Q_3Q_2 \ Q_1Q_0	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

Simplicity

This counter looks much simpler than most textbook counters. However if one implements it with D flip-flops, converting to enabled toggle flip-flop requires an XOR for each flip-flop. Thus it is simple for the designer, but it is only apparent simplicity.

Speed

The binary counter is one form of binary adder. The slow carry chain is present. For long counts the carry chain and counter will be quite slow

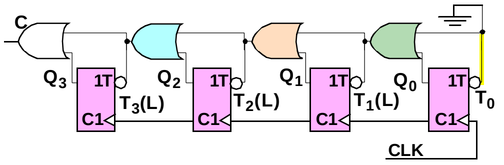
To get around this one could use a LFSR. It will count quickly, but one cannot do arithmetic on its result. One cannot easily tell if the signal is greater than some reference. One can only tell equality easily.

Binary Down-Counter

T Flip-Flops Circuit

(Simplest for People)

A 4-bit binary down counter



Present State				Next State				What to Toggle			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	T_3	T_2	T_1	T_0
0	0	0	0	1	1	1	1				
1	1	1	1	1	1	1	0				T
1	1	1	0	1	1	0	1			T	T
1	1	0	1	1	1	0	0				T
1	1	0	0	1	0	1	1		T	T	T
1	0	1	1	1	0	1	0				T
1	0	1	0	1	0	0	1			T	T
1	0	0	1	1	0	0	0				T
1	0	0	0	0	0	0	0	T	T	T	T
0	1	1	1	0	1	1	0				T
0	1	1	0	0	1	0	1			T	T
0	1	0	1	0	1	0	0				T
0	1	0	0	0	0	1	1		T	T	T
0	0	1	1	0	0	1	0				T
0	0	1	0	0	0	0	1			T	T
0	0	0	1	0	0	0	0				T

The T table shows where the bits toggle in going to the next state.

The circles show how the toggle signals are defined by zeros:

$T_1(L) = \underline{Q_0}$ $=Q_0$
$T_2(L) = \underline{Q_1 Q_0}$ $=Q_1+Q_0$
$T_3(L) = \underline{Q_2 Q_1 Q_0}$ $=Q_2+Q_1+Q_0$
$T_C(L) = Q_3 Q_2 Q_1 Q_0$ $=Q_3+Q_2+Q_1+Q_0$

T(L) means a “0” togles the flip-flop.

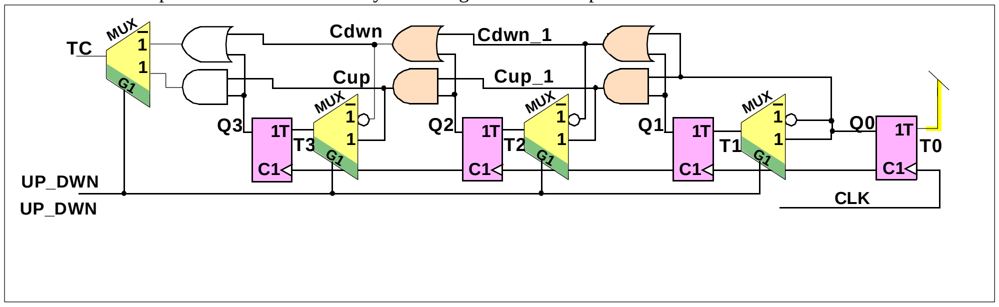
Binary Down-Counter

Design of the Counter

This counter is very much like the up-counter. The difference is the toggle signals are generated by checking for groups of zero flip-flop outputs, such as $\overline{Q_2}\overline{Q_1}\overline{Q_0}$. Using DeMorgan’s theorem changes these terms to a chain of OR gates instead of AND gates.

Up-Down Counters

A controllable up-down counter is made by switching between the up and down circuits.



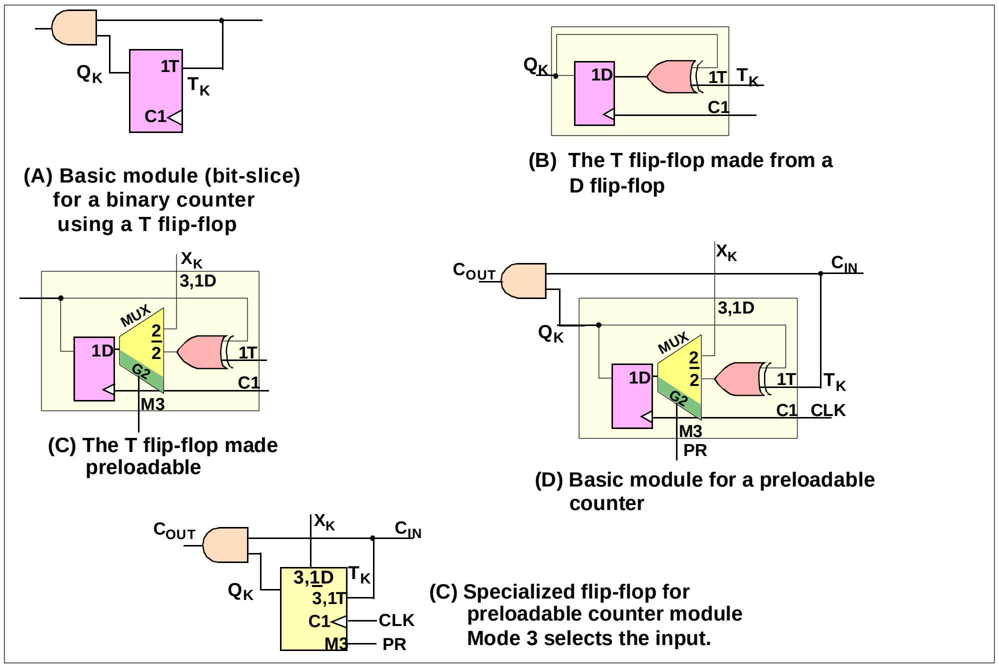
T3= (up_dwn) ? Cup : (~Cdwn);

Cup = Q_1&Cup_1

Cdwn = Q_1&Cdwn_1

Preloadable Counters

Constructed from the Basic Counter Module



(A) Basic module (bit-slice) for a binary counter using a T flip-flop

(B) The T flip-flop made from a D flip-flop

(C) The T flip-flop made preloadable

(D) Basic module for a preloadable counter

(C) Specialized flip-flop for preloadable counter module Mode 3 selects the input.

Preloadable Counters

Hiearchy

The preloadable counter is a hierarchy of modules starting with a D flip-flop.

a. D flip-flop
b. Preloadable D flip-flop
c. Preloadable T flip-flop
d. The Counter Module With AND Gate.
e. The Preloadable Up-Counter

IEEE Symbols

The PR input changes the *mode.* The number 3 was picked for the mode. In mode 3, the circuit (C) and (D) do a preload on the clock edge. The “3,1D” says that in mode 3 and (comma and) on the “1” signal edge this is a D input.

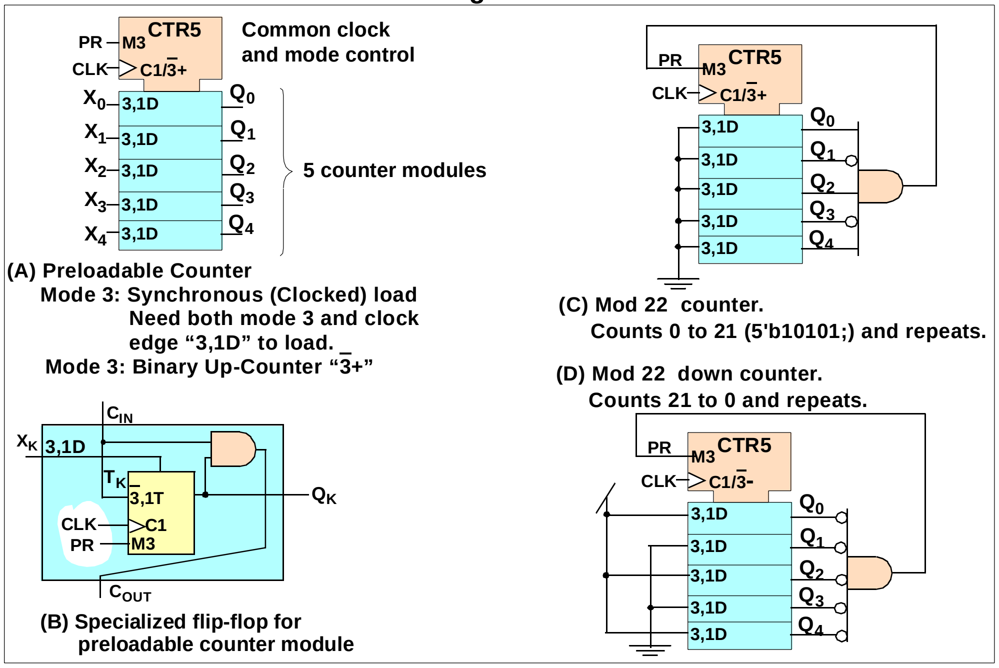
The control number in the MUX was changed to 2 because, in this symbol, 1 was already used by the clock.

Any numbers can show a connection between a control an what it controls. However the number must be the same at both sides. Thus 2 controls 2.

■ Binary Counters ■

Preloadable Counters (cont.)

Counters That Count to Something Besides 2^N



(A) Preloadable Counter
Mode 3: Synchronous (Clocked) load
Need both mode 3 and clock edge “3,1D” to load.
Mode $\overline{3}$: Binary Up-Counter “$\overline{3}$+”

(B) Specialized flip-flop for preloadable counter module

(C) Mod 22 counter.
Counts 0 to 21 (5'b10101;) and repeats.

(D) Mod 22 down counter.
Counts 21 to 0 and repeats.

Preloadable Counters (cont.)

IEEE Counter Symbol

The symbol is divided into 5 special modules and a common T shaped control block.

- The control block contains the letters CTR5 to show the circuit is basically a 5 bit counter.
- The common mode control “M3” for preloading.
- The common clock input “C1/$\overline{3}$+.” The “+” says the counting is binary and upwards unless one is in mode 3. The C1 says it functions as a clock independent of mode.
- “+” after the clock input is an up counter. “-” after the clock input is a down counter.

Counting to M≠2^N

Decode Terminal Count

Decode, with the equivalent of a many input AND gate, the final count. Use that to activate the preload and load zero. This allows the counting to go upwards.

Preload the Maximum Count and Count Down

At the start of counting, loading the maximum count and count down.
Alternately load the two’s complement of the maximum count, and count up.

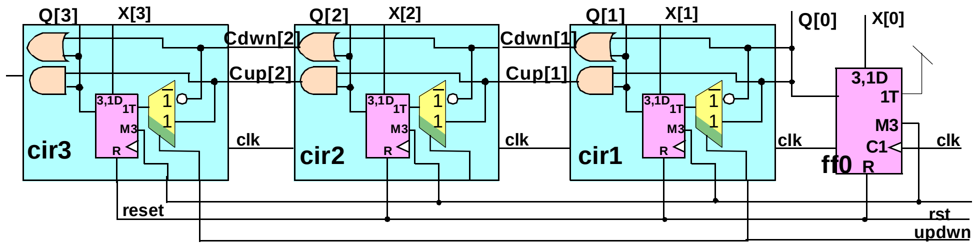
This is a better method if the desired count keeps changing. One does not need a different AND gate for each count.

A Verilog Preloadable Up/Down Counter

```
module UpDwn(Q, updwn, X, pr, clk, reset);
        parameter n=4;
        input [n-1:0] X;                        // The number to preload into counter
        input pr, updwn,                        // preset control,   up(H)/down(L)
              clk, rst;                         // clock,      asynchronous reset
        output[n-1:0] Q;
        wire   [n-1:1]  Cup, Cdwn;              // The carries for up and down counting.

           pr_T_ff    ff0(Q[0],X[0],1,pr,clk,rst);
           cntr cir1(Q[1],Cup[1],Cdwn[1],Q[0],Q[0],updwn,X[1],pr, clk,rst);
           cntr cir2(Q[2], Cup[2],Cdwn[2],Cup[1],Cdwn[1],updwn,X[2],pr,clk, rst);
           cntr cir3(Q[3],Cup[3],Cdwn[3],Cup[2],Cdwn[2],updwn,X[3],pr,clk,rst);

endmodule UpDwn
```



A Verilog Preloadable Up/Down Counter (cont)

A Counter Module

The four components on the diagram are easily spotted in the code.

- The flip-flop is the module call.
  pr_T_ff ff1(Qi, Xi,Ti, pr, clk, reset);
- The & gate
  Cup = Cup_1 & Qi,
- The OR “|” gate.
  Cdwn= Cdwn_1 | Qi,
- The MUX
  Ti=(updwn) ? Cup_1 : ~Cdwn_1;

<u>This Module is Structual</u>

The connections are described. Not the operation.

Named Module Connections (Named Ports)

The connections between a module definition and its instantiation have been made by *position*.
The shaded box shows how to make the connections by using the module port *name*.
<*port name used in the original definition*>.(<*wire/reg name connecting this particular instantiation*>)
.T(Ti)

For small modules *position* is simpler.
For large modules *name* is better. Errors in position are easy to make with 20 or more variables.

Note this has nothing to do with *call by name/call by value* described in programming courses.

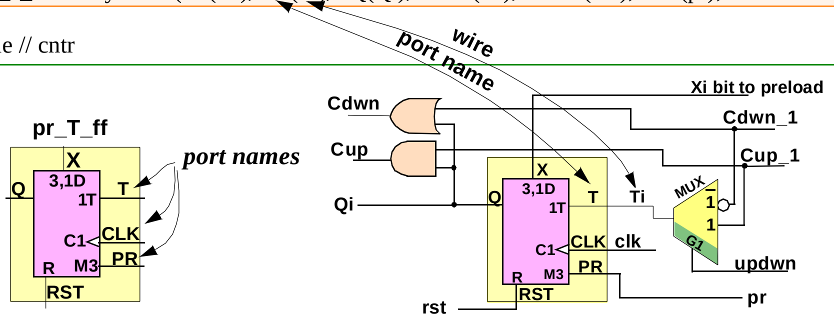
A Verilog Preloadable Up/Down Counter

```
module cntr(Qi, Cup, Cdwn, Cup_1, Cdwn_1, updwn, Xi, pr, clk, rst);
   input Cup_1, Cdwn_1, updwn, Xi, pr, clk, rst;
   output Qi, Cup, Cdwn;
   wire Ti;

   assign
        Cup = Cup_1 & Qi,
        Cdwn = Cdwn_1 | Qi,
        Ti = (updwn) ? Cup_1 : ~Cdwn_1;
        pr_T_ff     ffByPos(Qi, X, Ti, pr, clk, rst);

// Alternate connection of module ports (arguments) by name.
//        The position of the argument is no longer important.
//        pr_T_ff    ff ByName( .X(Xi),  .T(Ti),  .Q(Qi),  .RST(rst),  .CLK(clk),  .PR(pr);

endmodule // cntr
```



A Verilog Preloadable Up/Down Counter (cont)

The Flip-Flop Module

This module follows the classic flip-flop standard:

- All outputs, or variables on the right-hand-side of a procedure must be of type reg.
  reg Qi;
- The flip-flop is edge triggered, @, and has an asynchronous reset.
  always @(posedge clk or posedge reset)
  The asynchronous reset allows the system to be placed in a known state during start-up. Synchronous reset requires the co-operation of the clock. A system with clock and clock/2 for example, has to be carefully planned to reset properly using synchronous reset.
- The nonblocking assignment "<=" is used.
  Qi <= (pr) ? Xi : Qi^Ti;
  Thus flip-flop outputs, fed back into flip-flop inputs, always use the previous values for inputs.
  For example Q[2]<=Q[1]; Q[1]<=Q[2]; always exchanges Q[1] and Q[2].

The Interaction of XOR and Toggle

The logic for Qi^Ti requires a little thought:

Qi is the fed back flip-flop output.
Ti is the toggle enable.

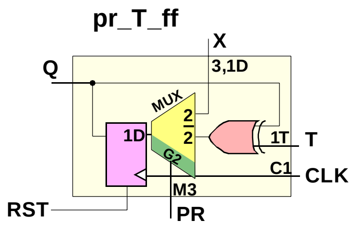
- When Ti=1, the flip-flop toggles i.e. ~Qi is fed back.
- When Ti=0, the flip-flop holds is old value, i.e. Qi is fed-back.

This reduces to Qi <= Qi^Ti.

A Verilog Preloadable Up/Down Counter (cont)

```
module   pr_T_ff(Q,  X, T, PR, CLK, RST);
    input   X, T, PR, CLK, RST;
    output   Q; reg Q;

    always @(posedge CLK or posedge RST)
    begin
       if (RST)
              Q <= 0;
      else
              Q  <=  (PR) ? X : Q^T;            //  It is ~((~Q)^T))
    end  // always
endmodule   // pr_T_ff
```



Glitches in Counters

All binary counters are glitchy

Binary is a glitchy way to count. Every second increment changes several bits at once.
If several variables change at once, one must really change first. until the second one changes, there is a glitch.

Synchronous Circuits and Glitches

Counter glitches come after the clock edge. There is a short delay for the clock to propagate to the output of the flip-flops. Then the glitches come.

However the glitches do no real damage unless:
  They are fed to some high speed output which can respond to glitches.
  They are captured by some flip-flop.

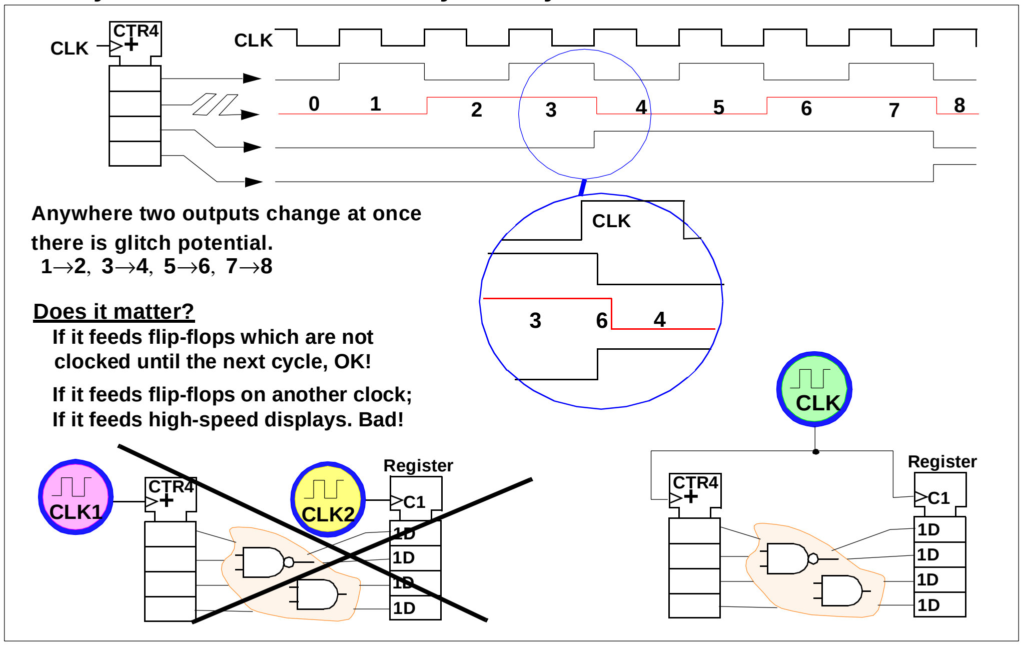
In normal synchronous circuits, all flip-flops are driven by the same clock. Glitches come to late to be captured by one clock edge, and too early to be captured by the next edge. Thus counter glitches are not problem in synchronous design.

Glitches can be Latched With Asynchronous Clocks

With two clocks running asynchronously, the second clock may come at any time with respect to the first. The register may clock just as the counter is sending out glitches. Then the glitches become a permanent erroneous signal in the system.

Glitches In Binary Counters

Binary Counters are Intrinsically Glitchy



Anywhere two outputs change at once there is glitch potential.
1→2, 3→4, 5→6, 7→8

Does it matter?
If it feeds flip-flops which are not clocked until the next cycle, OK!
If it feeds flip-flops on another clock;
If it feeds high-speed displays. Bad!

The Ripple Counter

The Glitchiest of the Glitchy Counters

Because each clock pulse must ripple through the flip-flops, the flip-flops never switch at the same time. In the binary counter one could, with well balanced flip-flops, have all the bits change at once.

The Bad Ripple Counter

- The counter is not synchronous. It has N different clocks.
  If one puts gates between the flip-flops to make an up-down or preloadable counter, any glitches on those gates may clock part of the counter.
- The ripple time through the N flip-flops is very long. It may take a long time for the final reading to settle after a clock pulse.
- The counter gives out many transient wrong counts before it reaches the final count.
- Testing people, who use scan testing, cannot tolerate anything except a true clock going to a flip-flop clock input.

The Good Ripple Counter

- Very small area.
- Very fast toggle rate. As long as the first flip-flop has flipped, one can change the clock again. The rest of the count will ripple down the counter, One can have several counts rippling down.
- Very low power. Power is used only when a flip-flop flips. Further they only flip if they are going to change and they flip only once per increment.
- If (i) the clock flips the first flip-flop, (ii) the counter finishes before the next clock edge, (iii) testing can be performed, then the counter can be contained within a synchronous circuit.

Application

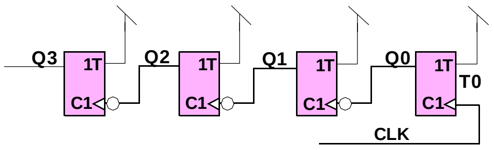
Every digital watch has a long ripple-counter chain.

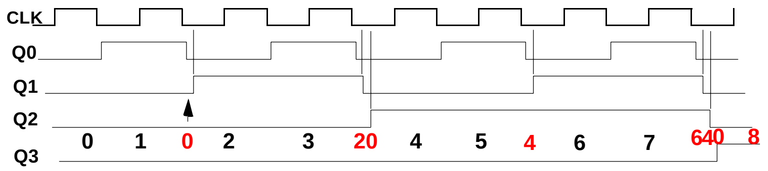
The Ripple Counter

Glitches Galore

The Ripple Up Counter

- The counter is not synchronous
  All flip-flops are not run from one clock.
- The counter gives many, wide glitches
  on half the counts.
- The carry chain is very slow.
  They are one of the slowest counters.





In Praise of Ripple Counters

- They are the simplest, smallest counter.
- They are slow to complete the count,
  but the input (clk) can run at the flip-flop toggle rate.
- They have very low power consumption.

Verilog Ripple Counter[1]

Triggering on nonclocks

The “@” statement

The @ is a “wait at this point” in the procedure until the trigger is satisfied.

One might think

```
always   @(negedge clk)
         @(negedge a[0])
         @(negedge a[1] . . .
```

would properly ripple through the counter. It will not!

The counter does not ripple the full length every time. When it only goes part way, it will ripple only until it hits the first a[i] that does not change. Then it will sit there forever waiting for the trigger.

The name change Q[i] to qi

Synthesizers usually do not like bits of an array as a clock veriable.

Parallel Constructs

The counter is coded with “parallel” always blocks. They are triggered independently.
The clk will trigger the first block. If a[0] rises that will trigger the next block.
If a[0] does not rise, nothing will happen until the clock rises again.

Delays

The delays, #0.5 delays the output of the statement by 0.5 time units. This makes it appear as though each flip-flop has a 0.5 unit propagation delay. The delays have no meaning for synthesis.

Reset

Reset is put in as a trigger. Any reset will immediately take effect on all flip-flops.

1. See also Verilog3Gotchas.fm5 cira p.108

A Verilog Ripple Counter

```
module  rip(Q, clk, reset);
    output[n:0] Q;            reg  [n:0] Q;
    input   clk,   reset;
    wire  q0,  q1, q2;
```

Fig. Com 0-1

```
        always @(negedge clk or posedge reset)     // negedge for up counter
          if (reset) Q[0]=0;                        // posedge for down counter
            else begin
            #0.5  Q[0] = ~Q[0];    // Delay will not synthesize but makes simulation clearer.
            end
         assign q0 = Q[0];    // Subscripted variables cannot be put in trigger lists as clocks.

        always @(negedge q0 or posedge reset)
          if (reset) Q[0]=0;
          else begin
            #0.5  Q[1] = (~Q[1]);
          end
        assign q1 = Q[1];

        always @(negedge q1 or posedge reset)
          if (reset) Q[2]=0;
          else begin
          #0.5  Q[2] = (~Q[2]);
          end
        assign q2 = Q[2];

    . . .
endmodule // rip
```

Gray Codes

Glitch-Free Counting

Because they change only one bit at a time, Gray code counters are inherently glitch free. This only applies if one increments by 1. Counting with increments of 2 or more will give glitches.

Types of Gray Codes

Here we define a Gray code as any code that increments with one bit-change at a time. There are thousands of Gray codes. Tracing through the Karnaugh map will show many. A Johnson Counter gives a Gray code.

The reflected Gray code is the most commonly used Gray code

a. Take the Gray code shown and drop the most significant bit temporarily.
b. Draw a line half way up the list.
c. Think of the line as a mirror. Then the numbers below the line are the reflection of those above.

Wrap-Around Gray Codes

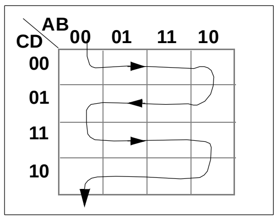
Some Gray codes may not wrap around.

Gray Code Counters

Single Bit-Change Counters

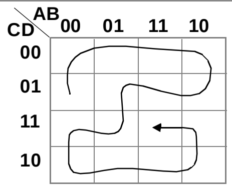
- Gray codes are binary encodings of numbers which change only one bit at a time.
- There are many of them.
- Gray codes can be read off a Karnaugh map
  .

Follow a trace through the Karnaugh map.
Write down the squares in the order you pass through them.
The common *reflected Gray code* shown(right).



0 ≈	0000
1 ≈	0001
2 ≈	0011
3 ≈	0010
4 ≈	0110
5 ≈	0111
6 ≈	0101
7 ≈	0100
8 ≈	1100
9 ≈	1101
10 ≈	1111
11 ≈	1110
12 ≈	1010
13 ≈	1011
14 ≈	1001
15 ≈	1000

Another Gray code which starts at 0100 and ends at 1111.
Follow the map trace and equate these with binary codes.
This one is not a Gray code on overflow.



0 ≈	0100	8 ≈	1101
1 ≈	0000	9 ≈	1100
2 ≈	0001	10 ≈	1000
3 ≈	0011	11 ≈	1001
4 ≈	0010	12 ≈	1011
5 ≈	0110	13 ≈	1010
6 ≈	0111	14 ≈	1110
7 ≈	0101	15 ≈	1111

Uses Of Gray Code Counters

An External Counter Feeding A Synchronous Machine

Overspeed Detector

This detector is reset every second.
Starting from zero it counts the number of slots that go by the toothed wheel.
If the number of slots goes above the setpoint, the motor is shut down.

It must be manually restarted.

Reliability

Depending on the way the counter glitches, this may erroneously shut down the motor.

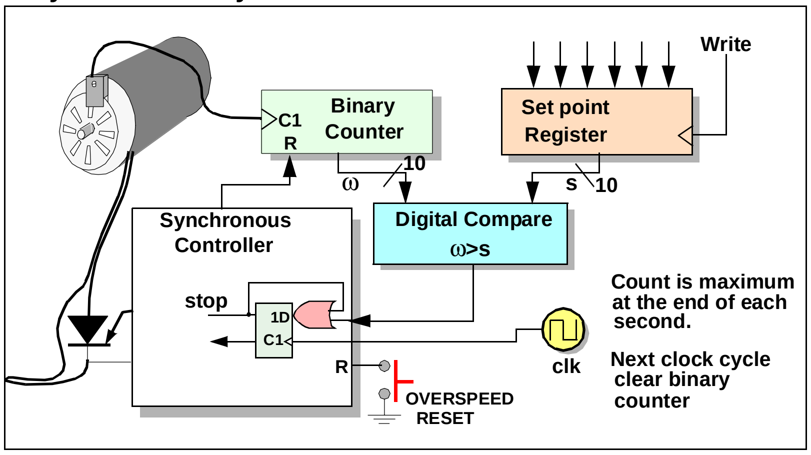
For the counter in "Binary Counters are Intrinsically Glitchy," p. 241, a set count of 6 could stop the motor when the real count was only 4.

This would not happen too often. The clock would have to rise as the counter was glitching.[1]

[1]. The ripple counter would not cause a problem here. It can never glitch to a higher number than the actual count.

Uses Of Gray Code Counters

Inputs to Synchronous Systems



Motor Overspeed Control.

The binary counter counts sensor pulses for 1 second.
The count is compared with the setpoint register. Set a MAX(pulses/sec)
If ω > s a STOP signal is sent out.

The motor stops when there is no overspeed.

Suggest some problems and their cures.

A Brute-Force Gray-Code Counter

Coding By Next-State

A Finite-state Machine As a Case Statement

a. Enter the present state as the test in a case statement
b. Enter the next state as the result.

A Finite-state Machine As an Excessive Case Statement

A 10 bit counter will have 1024 entries; a 20 bit counter will have 104857 entries. Too much!

5 or 6 flip-flops appear to be a practical limit.

The Alternative To Brute Force; Finite-State Machine (FSM) Coding

State Table For Gray Code.

State $Q_3Q_2Q_1Q_0$	Nxt St $Q_3^+Q_2^+Q_1^+Q_0$	D Inputs $D_3D_2D_1D_0$	T Inputs $T_3T_2T_1T_0$	State $Q_3Q_2Q_1Q_0$	Nxt St $Q_3^+Q_2^+Q_1^+Q_0$	D Inputs $D_3D_2D_1D_0$	T Inputs $T_3T_2T_1T_0$
0000	0001	0001	- - - t	1100	1101	0001	- - - t
0001	0011	0011	- - t -	1101	1111	0011	- - t -
0011	0010	0010	- - - t	1111	1110	0010	- - - t
0010	0110	0110	- t - -	1110	1010	0110	- t - -
0110	0111	0111	- - - t	1010	1001	0111	- - - t
0111	0101	0101	- - t -	1001	1001	0101	- - t -
0101	0100	0100	- - - t	1001	1000	0100	- - - t
0100	1100	1100	t - - -	1000	0000	1100	t - - -

Gray Code Counter

```
always @(posedge reset or posedge clock)
   if (reset)   Q <= 0;
   else
       case (Q)
      3'b000:  Q <= 3'b001;
      3'b001:  Q <= 3'b011;
      3'b011:  Q <= 3'b010;
      3'b010:  Q <= 3'b110;
      3'b110:  Q <= 3'b111;
      3'b111:  Q <= 3'b101;
      3'b101:  Q <= 3'b100;
      3'b100:  Q <= 3'b000;
      default: Q <= 3'bx;
          endcase
```

Step after Reset (in binary)	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

An O(2^N) Description

The size of this table goes up with the square of the number N of flip-flops.

- **The synthesizer may increase the circuit as O(2^N).**
- **Certainly the work to enter will increase as O(2^N).**

Avoid such descriptions if possible.

Finite-State Machine (FSM) Coding for Gray-Code Counters

Maps for D, T and enabled D flip-flops

Map of D_3 for flip-flop 3

Q3Q2\Q1Q0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	1	1
10	0	1	1	1

$D_3=Q_2\overline{Q}_1\overline{Q}_0+(Q_1+Q_0)Q_3$

Map of D_2 for flip-flop 2

Q3Q2\Q1Q0	00	01	11	10
00	0	0	0	1
01	1	1	1	1
11	1	1	1	0
10	0	0	0	0

$D_2=\overline{Q}_3Q_1\overline{Q}_0+(\overline{Q}_1+Q_0)Q_2$

Map of D_1 for flip-flop 1

Q3Q2\Q1Q0	00	01	11	10
00	0	1	1	1
01	0	0	0	1
11	0	1	1	1
10	0	0	0	1

$D_1=(\sim(Q_3\oplus Q_2))Q_0+\overline{Q}_1\overline{Q}_0$

Map of D_0 for flip-flop 0

Q3Q2\Q1Q0	00	01	11	10
00	1	1	0	0
01	0	0	1	1
11	1	1	0	0
10	0	0	1	1

$D_0=(\sim(Q_3\oplus Q_2\oplus Q_1))$

Map of T_3 for flip-flop 3

Q3Q2\Q1Q0	00	01	11	10
00	0	0	0	0
01	t	0	0	0
11	0	0	0	0
10	t	0	0	0

$T_3=(Q_3\oplus Q_2)\overline{Q}_1\overline{Q}_0$

Map of T_2 for flip-flop 2

Q3Q2\Q1Q0	00	01	11	10
00	0	0	0	t
01	0	0	0	0
11	0	0	0	t
10	0	0	0	0

$T_2=(\sim(Q_3\oplus Q_2))Q_1\overline{Q}_0$

Map of T_1 for flip-flop 1

Q3Q2\Q1Q0	00	01	11	10
00	0	t	0	0
01	0	0	t	0
11	0	t	0	0
10	0	0	t	0

$T_1=(\sim(Q_3\oplus Q_2\oplus Q_1)\overline{Q}_0)$

Map of T_0 for flip-flop 0

Q3Q2\Q1Q0	00	01	11	10
00	t	0	t	0
01	0	t	0	t
11	t	0	t	0
10	0	t	0	t

$T_0=(\sim(Q_3\oplus Q_2\oplus Q_1\oplus Q_0))$

Map of D_3 and En 3

Q3Q2\Q1Q0	00	01	11	10
00	0	d	d	d
01	1	d	d	d
11	1	d	d	d
10	0	d	d	d

$D_3=Q_2$ $En_3=\overline{Q}_1\overline{Q}_0$

Map of D_2 and En 2

Q3Q2\Q1Q0	00	01	11	10
00	d	d	d	1
01	d	d	d	1
11	d	d	d	0
10	d	d	d	0

$D_2=\overline{Q}_3$ $En_2=Q_1\overline{Q}_0$

Map of D_1 and En 1

Q3Q2\Q1Q0	00	01	11	10
00	d	1	1	d
01	d	0	0	d
11	d	1	1	d
10	d	0	0	d

$D_1=(Q_3\oplus Q_2)$ $En_1=Q_0$

Map of D_0 and En 0

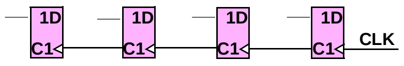
Q3Q2\Q1Q0	00	01	11	10
00	1	1	0	0
01	0	0	1	1
11	1	1	0	0
10	0	0	1	1

$D_0=(\sim(Q_3\oplus Q_2\oplus Q_1))$ $En_0=1$

Design Methods for Gray Code Counters

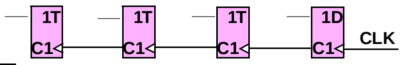
Five Design Methods

1. Lookup Table



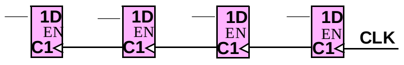
2. D Flip-Flops

$D_3=Q_2\overline{Q_1}\overline{Q_0}+(Q_1+Q_0)Q_3$ $D_2=\overline{Q_3}Q_1\overline{Q_0}+(\overline{Q_1}+Q_0)Q_2$ $D_1=(\sim(Q_3\oplus Q_2))Q_0+\overline{Q_1}\overline{Q_0}$ $D_0=(\sim(Q_3\oplus Q_2\oplus Q_1))$



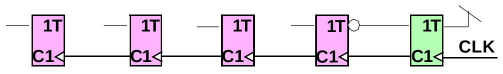
3. Toggle Flip-Flops

$T_3=(Q_3\oplus Q_2)\overline{Q_1}\overline{Q_0}$ $T_2=(\sim(Q_3\oplus Q_2))Q_1\overline{Q_0}$ $T_1=(\sim(Q_3\oplus Q_2\oplus Q_1)\overline{Q_0})$ $T_0=(\sim(Q_3\oplus Q_2\oplus Q_1\oplus Q_0))$



4. Enabled-D

$D_3=Q_2$ $En_3=\overline{Q_1}\overline{Q_0}$ $D_2=\overline{Q_3}$ $En_2=Q_1\overline{Q_0}$ $D_1=(Q_3\oplus Q_2)$ $En_1=Q_0$ $D_0=(\sim(Q_3\oplus Q_2\oplus Q_1))$ $En_0=1$



5. Extra Dummmy FF

$T_3=\overline{Q_1}\overline{Q_0}Q_D$ $T_2=Q_1\overline{Q_0}Q_D$ $T_1=(Q_0Q_D)$ $T_0=\overline{Q_D}))$ $T_D=1$

Gray Code Counter (cont.)

The Gray-Code Equations for Enabled Toggle FF

The cases follow a pattern. The 1st case is simpler if one uses a D, not a toggle, flip-flop.

$Q_1=(\sim(Q_6 \oplus Q_5 \oplus Q_4 \oplus Q_3 \oplus Q_2\oplus Q_1))$ Use a D flip-flop here and get $(\sim(Q_6 \oplus Q_5 \oplus Q_4 \oplus Q_3 \oplus Q_2))$

$Q_2=(\sim(Q_6 \oplus Q_5 \oplus Q_4 \oplus Q_3\oplus Q_2))\cdot Q_1$

$Q_3=(\sim(Q_6 \oplus Q_5 \oplus Q_4 \oplus Q_3))\cdot Q_2\cdot\overline{Q_1}$

$Q_4=(\sim(Q_6 \oplus Q_5 \oplus Q_4))Q_3\cdot\overline{Q_2}\cdot\overline{Q_1}$

$Q_5=(\sim(Q_6 \oplus Q_5))\cdot Q_4\cdot\overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}$

$Q_6=(Q_6 \oplus Q_5)\cdot\overline{Q_4}\cdot\overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}$

Complicated Gray

module Gray(Q, clk, reset);

parameter n=6; **// Number of T flip-flops**
output[n:1]Q; reg [n:1]Q; **// Register all procedure outputs.**
input clk, reset;

always @(posedge clk or posedge reset)
if (reset) Q<=0;
else begin

// $Q[1] <= (\overline{Q6} \oplus Q5 \oplus Q4 \oplus Q3 \oplus Q2)$; **// ←This one is a D -FF.**
Q[1] <= (~(^Q[n:2]));

// Toggle FF statements

// $Q2 <= IF((\overline{Q6} \oplus Q5 \oplus Q4 \oplus Q3 \oplus Q2) \& Q1)$ *THEN* $\overline{Q2}$: *ELSE* Q2;
Q[2] <= (~(^Q[n:2])&Q[1]) ? ~Q[2] : Q[2];

// $Q3 <= IF ((\overline{Q6} \oplus Q5 \oplus Q4 \oplus Q3) \& Q2 \& (\sim Q1))$ *THEN* $\overline{Q3}$: *ELSE* Q3;
Q[3] <= (~(^Q[n:3]))&Q[2]&(~|Q[1]) ? ~Q[3] : Q[3];

// $Q4 <= IF(\overline{Q6} \oplus Q5 \oplus Q4) \& Q3 \& \overline{Q2} \& \overline{Q1})$ *THEN* $\overline{Q4}$: *ELSE* Q4;
Q[4] <= (~(^Q[n:4]))&Q[n-3]&(~|Q[n-4:1]) ? ~Q[4] : Q[4];

// $Q5 <= IF((\overline{Q6} \oplus Q5) \& Q4 \& \overline{Q3} \& \overline{Q2} \& \overline{Q1})$ *THEN* $\overline{Q5}$: *ELSE* Q5;
Q[5] <= (~(^Q[n:5]))&Q[n-2]&(~|Q[n-3:1]) ? ~Q[5] : Q[5];

// $Q6 <= IF(Q6 \oplus Q5) \& \overline{Q4} \& \overline{Q3} \& \overline{Q2} \& \overline{Q1})$ *THEN* $\overline{Q6}$: *ELSE* Q6;
Q[6] <= (^Q[n:5])&(~|Q[n-2:1]) ? ~Q[6] : Q[6];

end **//** else
endmodule **//** Gray

The Gray-Code With Dummy Toggle FF

The logic can be greatly simplified by adding a dummy flip-flop, Q_D, which toggles every cycle.

Since Q_0 toggles every 2nd cycle, its toggle can be controlled directly from Q_D.

The controls for the other bit simplify as shown below..

State $Q_3Q_2Q_1Q_0$	Dum Q_D	T Inputs $T_3T_2T_1T_0$	Toggle to give next state
0 0 0 0	0	- - - t	$T_0=\overline{Q}_D$
0 0 0 1	1	- - t -	$T_1=Q_0Q_D$
0 0 1 1	0	- - - t	$T_0=\overline{Q}_D$
0 0 1 0	1	- t - -	$T_2=Q_1\overline{Q}_0Q_D$
0 1 1 0	0	- - - t	$T_0=\overline{Q}_D$
0 1 1 1	1	- - t -	$T_1=Q_0Q_D$
0 1 0 1	0	- - - t	$T_0=\overline{Q}_D$
0 1 0 0	1	t - - -	$T_3=\overline{Q}_1\overline{Q}_0Q_D$
1 1 0 0	0	- - - t	$T_0=\overline{Q}_D$
1 1 0 1	1	- - t -	$T_1=Q_1Q_D$
1 1 1 1	0	- - - t	$T_0=\overline{Q}_D$
1 1 1 0	1	- t - -	$T_2=Q_2\overline{Q}_1Q_D$
1 0 1 0	0	- - - t	$T_0=\overline{Q}_D$
1 0 1 1	1	- - t -	$T_1=Q_1Q_D$
1 0 0 1	0	- - - t	$T_0=\overline{Q}_D$
1 0 0 0	1	t - - -	$T_3=\overline{Q}_1\overline{Q}_0Q_D$

Reference:Altera Application Brief 135, Ripple-Carry Gray-Code Counters, Altera Corp, May 1994.

Extra FF Gray Code counter

T Flip-Flops Circuit

(Simplest for People)

Present State					Next State					What to Toggle				
Q_3	Q_2	Q_1	Q_0	Q_d	Q_3	Q_2	Q_1	Q_0	Q_d	T_3	T_2	T_1	T_0	T_d
0	0	0	0	0	0	0	0	1	1				T	T
0	0	0	1	1	0	0	1	1	0			T		T
0	0	1	1	0	0	0	1	0	1				T	T
0	0	1	0	1	0	1	1	0	0		T			T
0	1	1	0	0	0	1	1	1	1				T	T
0	1	1	1	1	0	1	0	1	0			T		T
0	1	0	1	0	0	1	0	0	1				T	T
0	1	0	0	1	1	1	0	0	0	T				T
1	1	0	0	0	1	1	0	1	1				T	T
1	1	0	1	1	1	1	1	1	0			T		T
1	1	1	1	0	1	1	1	0	1				T	T
1	1	1	0	1	1	0	1	0	0		T			T
1	0	1	0	0	1	0	1	1	1				T	T
1	0	1	1	1	1	0	0	1	0			T		T
1	0	0	1	0	1	0	0	0	1				T	T
1	0	0	0	1	0	0	0	0	0	T				T
0	0	0	0	0	0	0	0	1	1				T	T
0	0	0	1		0	0	0	0					T	T

The T table shows where the bits toggle between the present state and the next state.

The circles show how the toggle signals are defined by zeros:

$T_d = \underline{1}$
$T_0 = Q_d$
$T_1 = Q_0\underline{Q}_d$
$T_2 = \underline{Q}_1\underline{Q}_0Q_d$
$T_3 = Q_1Q_0Q_d$

The Most-Significant Bit

In Gray codes the most significant bit is not symmetric with the others.
This is why the expression for Q_3 on “Extra FF Gray Code counter” on page 257 does not match "Verilog Extra Flip-Flop Gray-Code Counter," p. 259

Fake Gray Code

A Binary counter can be converted to output Gray code by placing an XOR between each pair of its bits:
... $G_2=B_3 \oplus B_2$, $G_1=B_2 \oplus B_1$, $G_0=B_1 \oplus B_0$.
Unfortunately the glitches in the original binary counter will come through and give a “glitchy Gray Code” output.

Verilog Extra Flip-Flop Gray-Code Counter

```
module SimplGray(Q, clk, reset);
      parameter n=6;                // Number of T flip-flops
        output[n:1]Q;               reg [n:1]Q;          // Register all procedure outputs.
        input clk, reset;
       reg Qd;

      always @(posedge clk or posedge reset)
        begin
        if (reset) Begin Q<=0;   Qd<0;  end

        else begin
    //   Toggle FF statements
        Qd <=                                            ~Qd;
           Q[0] <= (~Qd)                             ?  ~Q[0] : Q[0];
           Q[1] <= ({Q[0],Qd}= =2'b11)               ?  ~Q[1] : Q[1];
           Q[2] <= ({Q[1:0],Qd}= =3'b101)            ?  ~Q[2] : Q[2];
           Q[3] <= ({Q[1:0],Qd}= =4'b1001)           ?  ~Q[3] : Q[3];
           Q[4] <= ({Q[3:0],Qd}= =5'b10001)          ?  ~Q[4] : Q[4];
           Q[5] <= ({Q[4:0],Qd}= =6'b100001)         ?  ~Q[5] : Q[5];    WRONG FOR END
        end // else

        end //begin for always
    endmodule //SimplGray
```

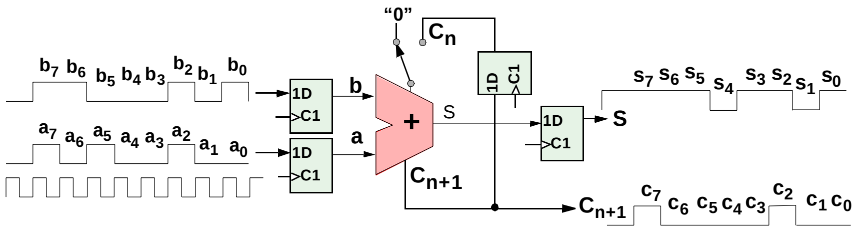
The Bit-Serial Adder

- This adder adds one bit per clock cycle.
  It adds strings of bits coming in serially and sends out the result as a serial stream.
- An 8-bit ripple-carry adder takes 8 carry propagation delays to add.
- An 8-bit serial adder need only wait for
  max(carry output, sum output)

  in each clock cycle.
  However it takes eight clock cycles.
- The 8-bit, bit-serial adder is slower than an 8-bit parallel adder, but not 8 times slower since (or if) the clock can be made faster.

The Bit-Serial Adder

The Smallest, Lowest-Power, And Slowest Adder



Words a[7:0] and b[7:0] come in serially. S[7:0] feeds out serially.
The initial carry-in C_0=0.
Subsequent carry-ins are the carry-out from the last cycle.

It takes 8 clock cycles to add 8 bits.
However the clock can run much faster than it can for a parallel adder.

The circuit is very small.

The power used is small.
The sum bits do not reverse as carries propagate through.