CARLETON UNIVERSITY

Deparment of Electronics

97.350 **Digital Electronics** **November 1, 2002**

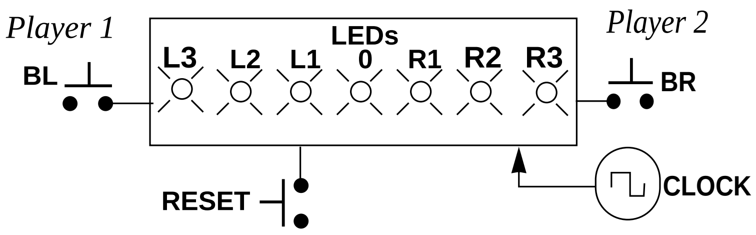
The Tug-of-War Game

Rev 11.

1.0 The Game

The players of this game see a row of 7 LEDs represented by the numbers L2 through R2 in FIGURE 1

FIGURE 1. **The tug of war machine. It contains 3 push-buttons, 7 light-emitting diodes (LEDs) and 1 field-programmable gate array (FPGA)**



After pushing reset, all the LEDs should come on for a second and then go off. This is the get ready signal. After a random time the middle LED comes on again. Then each player will try to push his button before the other player does. The position of the lit LED will move toward the fastest button pusher.

Thus if player 2 is fastest, LED 0 will go out and LED R1 will come on. This is the end of **round** one. After a second the LEDs will all go out. This is the get ready signal for round two. After a random time LED R1 will come on (assuming player 2 won the first round). Again each player will try to push his button first, and again the light will shift toward the fastest button pusher. The game is won when one player makes the position of the lit LED move off the end of the display.

If a player jumps the light, i.e. he pushes his button while the LED is off, then the light will come on immediately and it will be shifted one position away from him.

The FAVOR-THE-LOSER circuit gives a slow starter a chance. When a player is about to win (the R3 or L3) position) the light will jump back Three spaces (to position 0) if he loses, but only one position if he jumps-the-light.

The circuit must be able to distinguish the winner within two gate delays (a few nanoseconds). Further the design must be fair. For example, the design should not assign ties to one of the players.

2.0 The Field Programmable Gate Array (FPGA)

These are a collection of several hundred flip-flops, and several thousand gates, all collected in one integrated circuit. Also in the integrated circuit are a large number of wires which run across different parts of the integrated circuit. There are also many electrically controlled switches which connect the flip-flops and gates to the wires, and the wires to each other. By opening and closing the switches, one can build almost any digital circuit unless the circuit needs more gates or flip-flops than there are in the FPGA.

Setting theses connection switches might be quite a job, Fortunately there is a computer program that does all the work. Just connect the FPGA to the serial port of a PC, run the program, and in about five seconds, your circuit is built. It sure beats stripping wires!

Your assignment is to design a Field-Programmable Gate Array (FPGA) to perform the tug-of-war game or an alternate game approved by the professor. All the logic will be on one IC. Only the push buttons, LEDs, LED drivers and the oscillator will be external.

3.0 Good Design Practice

The following practices shall be followed in your design:

a. Do not gate the clock. The gate will cause the flip-flops controlled by the gated clock to flip later than the ones directly connected. Use enabled flip-flops, and gate the enable instead of the clock.

b. Use only ONE clock for all timing. Divide its output to get slower timing signals. Then use a pulse fed into the clock-enable (CE) flip-flop input to give a lower effective clock rate.

c. Two related (passing through common logic) asynchronous input signals must be stable near the clock edge on which they are read.

d. Do not allow one asynchronous input to change two state variables (flip-flops) together. If the clock changes just as the input changes you get a race. Instead clock the input into a single D flip-flop. This makes the input into a synchronous signal which can safely initiate multiple-variable state changes.

e. Do not use asynchronous preset or clear in counters and shift registers. The clear signal may clear itself before clearing all the flip-flops. Asynchronous clear may, and should, be used to clear all flip-flops during power-on re-start.

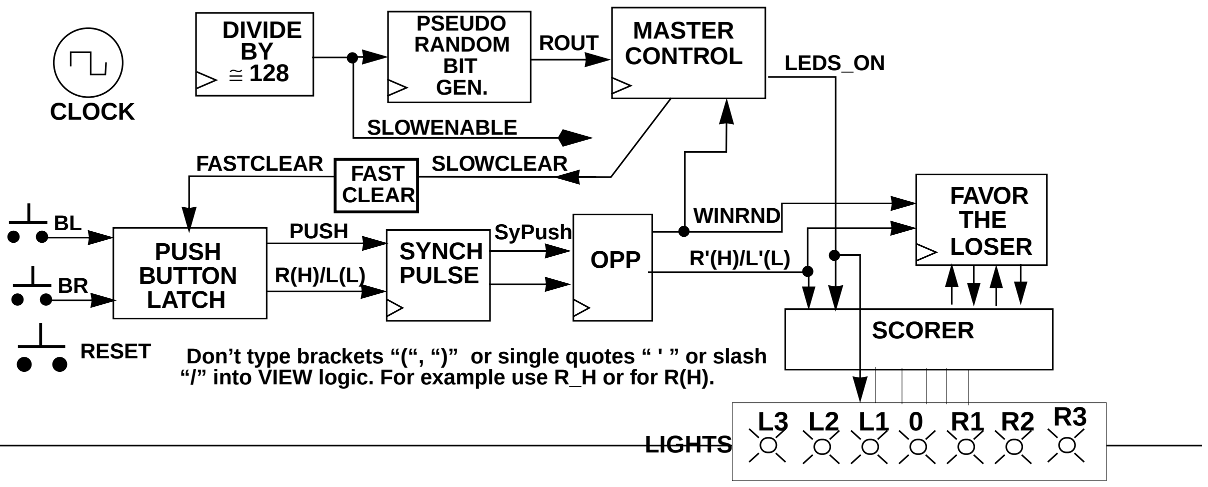
f. Any asynchronous circuit which latches, i.e. your push button circuit, must be checked for hazards and races.

g. Any unclocked latches in your circuit are asynchronous. If you place cross coupled NAND gates in the synchronous part of your circuit, you have created an asynchronous island in a supposedly synchronous circuit. You must then check for hazards, races and essential hazards at all latch inputs.[1]

4.0 Initial Planning

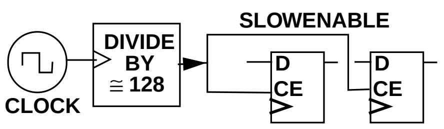
As with most specifications, there are a lot of subspecifications buried in the design document. You should read through the complete document and list these before you start serious design. See the section on deliverables.

FIGURE 2. A possible block diagram for the game. The blocks are completely synchronous and run from a common clock, except for the PUSH BUTTON LATCH which is asynchronous.



1. There is another reason for not placing any of the above asynchronous concepts into a synchronous circuit. The testing method (Scan Testing) used for modern digital circuits will not work with embedded asynchronous elements.

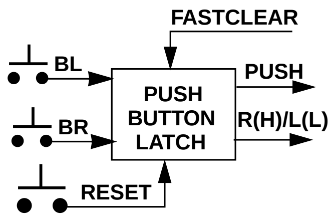
4.1 Clock and Clock Divider



The master clock should be about 500 Hz. This is fast enough so humans will not see appreciable delay, but it is slow enough so a divide by 256 can give the 1/2 second timing humans can see. The divide by 256 gives a pseudo-clock, a pulse one-clock-cycle wide every 2^8 or 256 clock cycles., called SLOWENABLE. . This pulse can enable/disable the flip-flops so they can only change at a 1/2 sec rate, even though they are clocked at 500 Hz.

You may use the laboratory square wave generator for the 500 Hz clock if one is not built onto the LCA board.

4.2 Push Button Latch



This must be asynchronous since it must tell which button was pushed first within a couple of gate delays. It debounces the buttons by merely latching on the first contact of the first button pushed. It should hold the signal until reset by the FASTCLEAR input. It has three inputs: BL, BR and FASTCLEAR.

4.3 The FAVOR-THE-LOSER circuit

When a player is in the L3 or R3 LED positions and the other player wins, the lights will jump back two positions to LED position 0. The block diagram above seems to suggest that this is a separate block. Another, probably better, way is to build it into the scorer. Note it only shifts back one position if the winner jumps-the-light.

4.4 The SCORER

It is suggested that the SCORER have three inputs:

WinRnd............. a one clock-cycle pulse which comes after a button push. It tells that somebody has won the round.

R(H)/L(L) ≡ R/L. tells which button was pushed first. Right if high , left if low.

LEDS_ON........... tells if a LED is lit and thus can be used to tell if a player pushes his button ahead of time (jumps the light) The score will move left or right according to the formula

Right = WinRnd·MR.

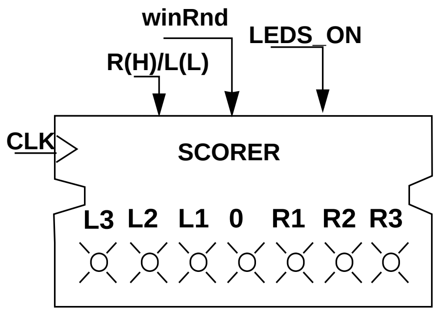
Left = WinRnd·$\overline{\text{MR}}$

Where MR...........move right = 1; if R/L,LEDS_ON =1, 1, or 0,0

0; if R/L,LEDS_ON =0, 1, or 1,0.

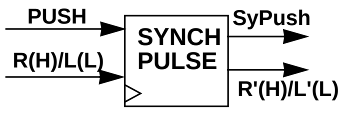
Thus MR signifies a right move
if (the right button was pressed) and (the player did not jump the light),
or (the left button was pressed) and (the player jumped the light).

You may want to separate the the scorer into two parts as in FIGURE 2.



4.5 The Synchronizer Circuit

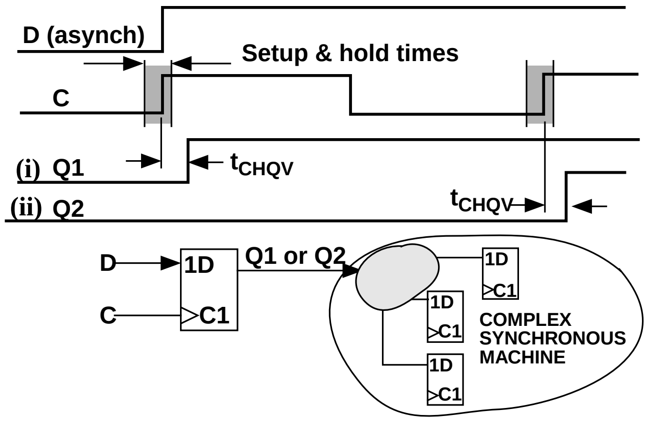
The output from the PUSH BUTTON LATCH is asynchronous, that is it may change at any time including on the clock edge. The SCORER may **not** count correctly if its inputs change very near the clock edge.



As you will learn in the lectures, inputs to an FSM's D flip-flops must never change near the clock edge. Near means inside the fli-flops setup and hold times. If the D inputs changes near the clock edge, one does not know whether the flip-flops will capture the old D value, the new D value, or some will capture the old and some the new. For much more detail, see Appendix A.

The PBL output must be synchronized, so it cannot change on the clock edge. This is done by passing it through a D flip-flop. The clock-to-output delay, t_{CHQV}, inside the flip-flop will delay Q till after the clock edge. Thus Q acts as a slightly delayed synchronous D. FIGURE 3 shows how this happens.

FIGURE 3. How a D flip-flop synchronizes a signal. If the D input changes inside the flip-flop setup or hold times, the signal may:
i) be captured, in which case it appears as the Q_1 output, rising a safe distance after the clock edge.
ii) have the capture delayed until the next clock edge. Then it will appear as Q_2, which is delayed, but still a safe distance from the active clock edge. Either Q1 or Q2 is a safe, synchronized signal to feed to multiple flip-flops in a synchronous machine.
iii) go metastable which is not a problem with very slow clocks like this one.



The tentative design suggests two input signals:

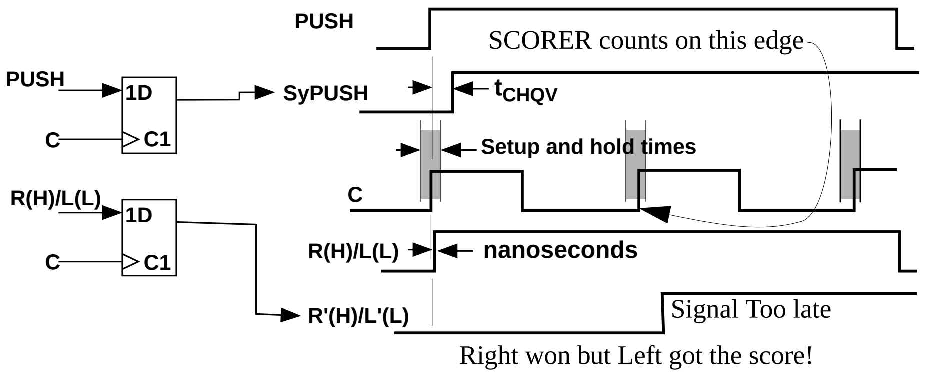
PUSH(H)....is latched true as soon as any button is pushed.

R(H)/L(L)...tells which button was pushed first, the right button or the left. .

While one asynchronous signal can be synchronized by passing it through a D flip-flop, two logically related asynchronous signals cannot. They will appear to be properly synchronized **until they both** change on the same clock edge. Then one may be captured by its D flip-flop but the other may not.

The problems can be seen by looking at PUSH and R(H)/L(L) in FIGURE 4 The two input signals PUSH and R(H)/L(L) are logically related and change at about the same time. Suppose PUSH and R(H)/L(L) both change inside the setup and hold time. Then the Synchronizer circuit might latch the correct PUSH and old incorrect value of R(H)/L(L). On the next clock cycle R'(H)/L'(L) would be correct but it is too late! The SCORER has already moved to the wrong light.

FIGURE 4. The signals PUSH and R(H)/L(L) are logically related and occur only nanoseconds apart. Suppose PUSH is captured on one cycle, but R(H)/L(L) is not captured until the next cycle. Then left will appear to have won because R/L = 0 on the next clock edge when SCORER is enabled.



The cure is simple. Latch only PUSH; do not latch R(H)/L(L). See FIGURE 6 on page 5. We know when PUSH goes high, R(H)/L(L) is at worst nanoseconds behind it. The SCORER reacts, not to the clock edge where PUSH is captured, but on the next (2nd) clock edge. By that time R(H)/L(L) will be correct for sure

Alternately, PUSH might not be captured and then SyPush would be delayed a clock cycle. See FIGURE 5. Then the SCORER would not react until the 3rd clock edge. By that time R(H)/L(L) will have been correct for two cycles..

FIGURE 5. This time PUSH was too late for the 1st clock edge and was captured on the 2nd as SyPush.

R(H)/L(L) does not go through a flip-flop and rises within nanoseconds of PUSH rising.remains correct until cleared.

The SCORER reacts to SyPush on the 3rd clock cycle. It counts in a direction given by R(H)/L(L). Both signals are stable at the 3rd clock edge.

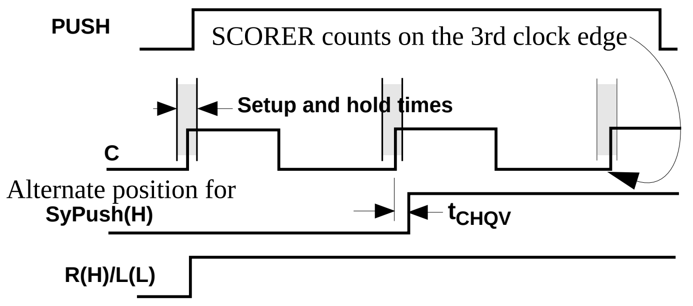
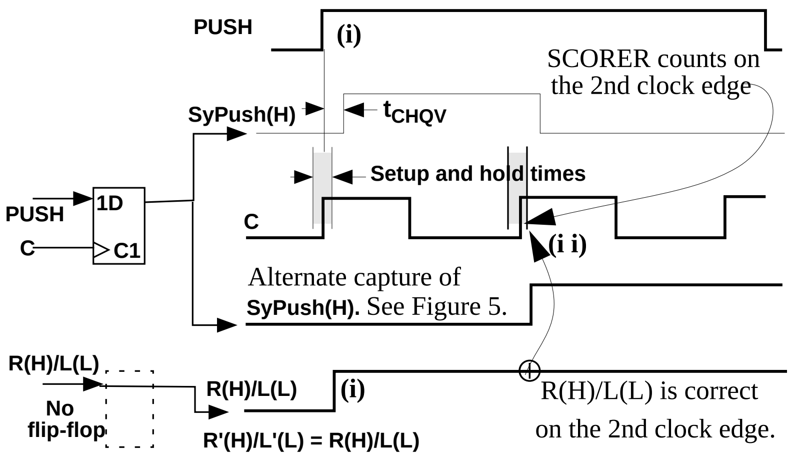


FIGURE 6. The signals PUSH and R(H)/L(L) are logically related and occur only nanoseconds apart.

(i) Suppose PUSH rises very near the clock edge, SyPush will rise either on the same clock edge or the next one. However, R(H)/L(L) will be correct within nanoseconds of PUSH rising, and will remain correct until cleared.

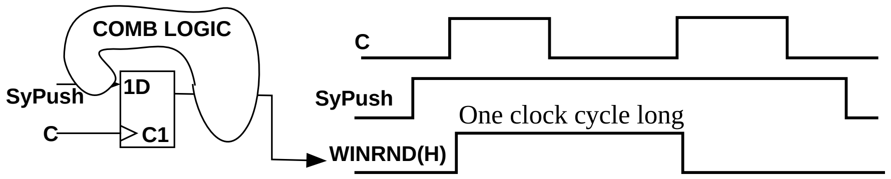
(ii) The SCORER reacts to SyPush on the 2nd cycle. It counts up or down according to R(H)/L(L) which is stable long before the 2nd clock edge.


PUSH
(i)
SCORER counts on the 2nd clock edge
SyPush(H)
t_{CHQV}
Setup and hold times
PUSH
1D
C
C1
C
(i i)
Alternate capture of SyPush(H). See Figure 5.
R(H)/L(L)
No flip-flop
R(H)/L(L)
(i)
R'(H)/L'(L) = R(H)/L(L)
R(H)/L(L) is correct on the 2nd clock edge.

4.6 The OPP (One-Pulse-Per-Push) Circuit

The SCORER must count only once, be it up or down, for each round played on the game. Otherwise each push will count for many wins. The OPP circuit insures this by giving out one-pulse-per-push (round). The synchronized push signal, SyPush, will initiate an output signal WINRND which will last exactly one clock cycle.

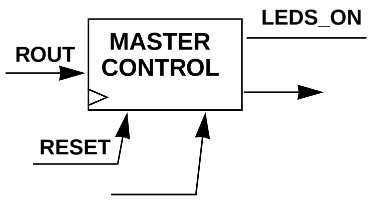
FIGURE 7. The WINRND signal lasts a single clock cycle so one round of the game will only move the SCORER once.



4.7 The Pseudorandom Counter

This circuit creates ROUT, a string of 1s and 0s which is unpredictable to the players. The time between bits should be about 1 second.

4.8 The Master Controller



The controller times the action of the game as follows:

a. Someone pushes a button either as a win or a jumped light.
b. The LEDs will come on immediately after a button is pushed.
c. The LEDs stay on for 1/2 to 1 second so it is clear who won; under 1/2 is too short, longer would be boring
d. Then all the LEDs go out for a random time, minimum of 1/2 second.
e. A single LEDs come back on after this random time.

4.8.1 Rising Edge Detection

The pseudorandom bit generator gives random 0 and 1 strings. You may wish to check bit pairs of ROUT instead of just checking for a “1.” For example checking for the pair 01 will find rising edges. Thus the ROUT string -

```
0111001100101110        would become
010000100010100         after a rising edge detector.
```

In the top sequence, there is a 50% chance of any bit being 1. If the lights came back on in response to ROUT=1, there is a 50% chance of the lights coming back on in 1/2 second. With the lower sequence there is only a 25% chance of getting a 1 in the first half second. This makes the game more interesting.
Also the second sequence gives a simpler controller design.

4.8.2 Slow Players and Gloating Players

If the players are slow, the controller circuit should keep the lights on until a push occurs. It also holds the light on an additional 1/2 to 1 second after a push so the movement of the lights can be observed and the winner can gloat.

4.8.3 Jumping the Light

If a player “jumps the light”, the light should change immediately. However the timing is the same as for a legitimate push. Also the controller should wait the “gloat period” and start the next round.

4.8.4 FastClear

After the LEDs go off, one must clear the push button latch. The clear circuit send out a FASTCLEAR pulse to do this. This arms the push buttons quickly and allows detecting any player “jumping the light” at anytime except the first clock cycle of the dark period (about 1/250 of a second).

5.0 More Detailed Synchronous Design

5.1 The SCORER

The state diagram for the basic SCORER is as shown.

State	Next State		
	EN=0	EN=1	
		MR=0	MR=1
0	0	L1	R1
R1	R1	0	R2
R2	R2	1	W
W	W	W	W
L1	L1	L2	0
L2	L2	W	L1

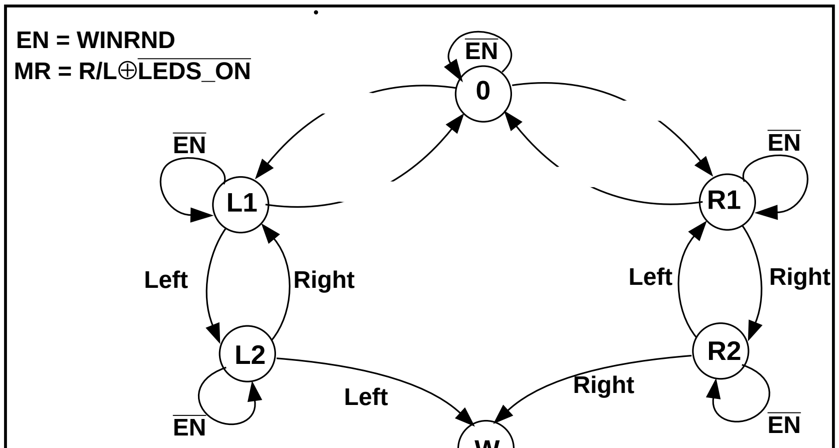


FIGURE 8

5.1.1 State Encoding

In asynchronous circuits state encoding is critical to avoid races. In synchronous circuits state encoding is used to reduce logic. There are some rules of thumb you can try to use. They usually work better than random state assignment, but not ask well as an experienced designer using symmetry. The rules are:

1. Parents of the same child, for the same input, should be adjacent states on the Karnaugh map (differ by only one bit). (L2, R2) have child W.
2. Children of the same parent should be adjacent. (L1, R1) (0, R2) (R2, W) (L2, 0) (W,L1)(W, R1).

One possible state assignment obtained from these guidelines is shown in FIGURE 9
If these states are filled in on Karnaugh maps one obtains the following tables. The bits in the state will be called a, b, and c.

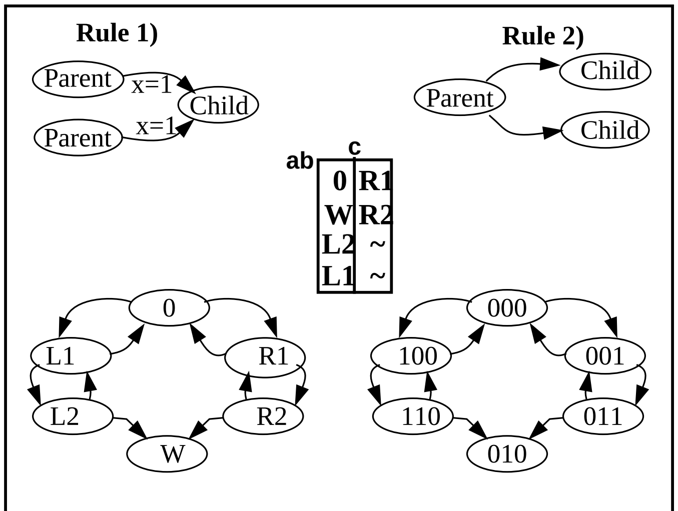


FIGURE 9 SCORER state assignment

You will not be able to satisfy all the guidelines. However a suboptimal state assignment in a synchrous circuit only adds a few extra gates.

ab\c	0	1
00	0	R1
01	W	R2
11	L2	~
10	L1	~

Present state
also
Next state for WinRnd=0

ab\c	0	1
00	L1	0
01	W	R1
11	W	~
10	L2	~

Next state for MR=0,WinRnd=1

ab\c	0	1
00	R1	R2
01	W	W
11	L1	~
10	0	~

Next state for MR=1,WinRnd=1

a,b \ c,MR	00	01	11	10
00	0	0	R1	R1
01	W	W	R2	R2
11	L2	L2	~	~
10	L1	L1	~	~

Next States, WinRnd=0

a,b \ c,MR	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	~	~
10	1	1	~	~

Map of a+

a,b \ c,MR	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	~	~
10	0	0	~	~

Map of b+

a,b \ c,MR		01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	0	~	~
10	0	0	~	~

Map of c+

a,b \ c,MR	00	01	11	10
00	L1	R1	R2	0
01	W	W	W	R1
11	W	L1	~	~
10	L2	0	~	~

Next States, WinRnd=1

a,b \ c,MR	00	01	11	10	
00	1	0	0	0	
01	0	0	0	0	0
11	0	1	~	~	
10	1	0	~	~	

Map of a+

a,b \ c,MR	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	1	0	~	~
10	1	0	~	~

Map of b+

a,b \ c,MR		01	11	10
00	0	1	1	0
01	0	0	0	1
11	0	0	~	~
10	0	0	~	~

Map of c+

Verilog users normally do state assignments, but logic mimimization is done on the computer, not with maps.

5.1.2 Unused states in the SCORER

Eventually the don't care states are filled in. Then one must check what happens if noise puts one into either of the unused states, 111 or 101. These will be called states 7 and 5 respectively.

What should the next states be for states that never happen? There are several possibilities:

a) Choose the states to minimize the logic. This was done in '367 or '267.

b) Have the bad states indicates an error to the users. Then let the user decide what to do. A lot of software errors are like that.

c) Have the bad states indicate an error and then automatically go to state 0.

d) Try to correct the error. This can be done by considering the most likely faults. Consider state 7 (111). If only one flip-flop went wrong to enter this state, then the previous state was either L2 (110) R2 (011) or 5 (101). Assuming only one error, it could not have started from the error state 101. Unfortunately one cannot tell whether it did start from 110 or 011, so this is not too useful here.

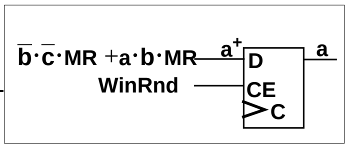
State	Next State		
	Win Rnd =0	WinRnd=1	
		MR=0	MR=1
0	0	L1	R1
R1	L1	0	R2
R2	R2	1	W
W	W3	W	W
L1	L1	L2	0
L2	L2	W	L1
5			
7			

e) What one does not want to do is have the machine lock up in some state with no indication of error.

Choose a philosophy to deal with the unused states. State what it is. For example, one might say, “On error, the machine will display an error light and require a reset push.” That way no large bets will be lost because of the error.”

In your report, explain what happens when you enter a bad state.

5.1.3 Final SCORER Design.



Schematic capture designers should use the CE (chip enable) pin on the flip-flops to simplify their work. The drawing shows how the signal connected to CE must, when low, hold the flip-flop unchanging.

$a+ = \overline{\text{WinRnd}}\cdot a + \text{WinRnd}\cdot(\bar{b}\cdot\bar{c}\cdot MR + - - -$

$b+ = \overline{\text{WinRnd}}\cdot b + - - - -$

$c+ = \overline{\text{WinRnd}}\cdot c + - - -$

CE is automatically taken care of for Verilog designers if the state equations are correct.

Karnaugh map designers can minimize logic by using don’t cares in their maps. Verilog designers can do the same by using **casex** for their next_state determining logic, and making the final case:

default: state =3’bxxx;

5.2 One-Hot State Encoding

An alternative scheme is to encode the state so that only one bit is “1” per state. This gives a shift-register like structure instead of a shift register like circuit instead of a counter. This uses more flip-flops, but it has simpler logic, and it does not require decoding logic to drive the LEDs. The shorter logic is usually faster so it is favored for circuits which need a high clock speed.

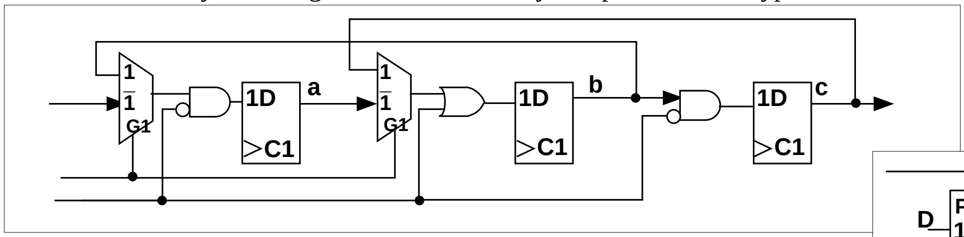
		Next State			State Equations	State Equations
	State	WinRnd = 0	WinRnd=1			with bits
abcde			**MR=0**	MR=1	WinRnd = 1	WinRnd = 1
00100	C	C	B	D	$C = D\cdot\overline{MR} + B\cdot MR$	$c = \bar{a}\cdot\bar{b}\cdot\bar{c}\cdot d\cdot\bar{e}\cdot\overline{MR} + \bar{a}\cdot b\cdot\bar{c}\cdot\bar{d}\cdot\bar{e}\cdot MR$
00010	D	D	C	E	$D = E\cdot\overline{MR} + C\cdot MR$	$d = \bar{a}\cdot\bar{b}\cdot\bar{c}\cdot\bar{d}\cdot e\cdot\overline{MR} + \bar{a}\cdot\bar{b}\cdot c\cdot\bar{d}\cdot\bar{e}\cdot MR$
00001	E	E	D	W	$E = D\cdot MR$	$e = \bar{a}\cdot\bar{b}\cdot\bar{c}\cdot d\cdot\bar{e}\cdot MR$
01000	B	B	A	C	$B = C\cdot\overline{MR} + A\cdot MR$	$b = \bar{a}\cdot\bar{b}\cdot c\cdot\bar{d}\cdot\bar{e}\cdot\overline{MR} + a\cdot\bar{b}\cdot\bar{c}\cdot\bar{d}\cdot\bar{e}\cdot MR$
10000	A	A	W	B	$A = B\cdot\overline{MR}$	etc.
00000	W	W	W	W	$W = A\cdot\overline{MR} + E\cdot MR + W$	

Drawing a 7 variable Karnaugh map for this circuit is messy and unnecessary. Since we know only one bit is “1” at a time, the “State Equations with bits” reduce to:

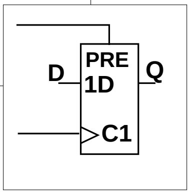
$c = d\cdot\overline{MR} + b\cdot MR$

$d = e\cdot\overline{MR} + c\cdot MR$ etc.

a. These type of equations give a left-or-right shifting register using flip-flops and muxs. A 3-bit example is shown. Unfortunately, checking for bad states is very complex with this type of machine.



b. Having 00100 as the centre state is convenient for lighting the centre LED, so use a centre flip-flop with an asyncronous preset (PRE) instead of a clear (CLR).



Verilog designers can easily encode one-hot states into FSMs using a **casex** statement. Be sure to make the last case:

default: next_state = 5'bxxxxx;

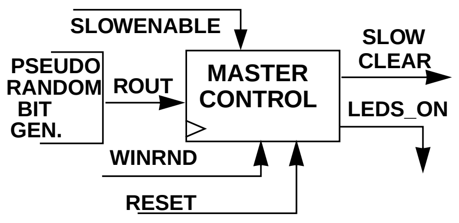
Verilog will automatically insert the centre preset and reset flip-flops in response to:

```
always @(posedge clk or posedge clear) begin
        if (clear)  state = 5'b00100;
```

5.2.1 The SCORER Circuit, Adding Extras

If you decide to add extras, like separate win states, flashing lights, music, or a siren, design them into your state machine now. Do not stick patches on at the end!

5.3 The Master Controller



The inputs signals shown for this subcircuit are:

ROUT............. the bit string from the pseudorandom bit generator.
WINRND........goes high for one clock cycle after a button is pushed.
SLOWENABLE.... which acts like a slow clock

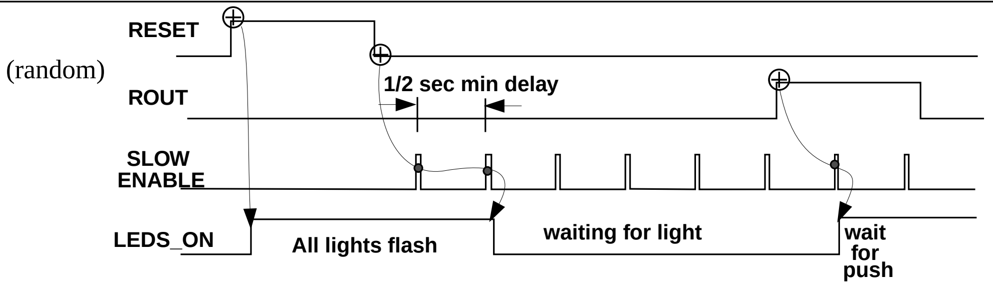
The output signals might be:

LEDS_ON....to say one or more LEDs are on.
SLOWCLEAR..which generates FASTCLEAR to clear the PUSH-BUTTON LATCH.

5.3.1 Initial Lamp Check

On RESET, all the LEDs should immediately come on, and stay on for 1 sec after RESET is released. This shows that all the lights work. All LEDs should then go out and the center LED should come back on after the next rising edge of ROUT.

FIGURE 10. The game after the initial reset. Here LEDS_ON comes on again, a Moore machine delay after the rising edge of ROUT. Note SLOW ENABLE does not run during RESET.



5.3.2 The Mid-Game

Consider midgame play as in FIGURE 11.

(a) When the ROUT signal goes high it raises LEDS_ON on the next SLOWENABLE. This turns on a LED and signals the players to push. Note ROUT rises just after pulse hence it cannot cause a reaction until pulse B.

(b) Pushes generate a WINRND signal which moves the light and will eventually turn off LEDS_ON. However LEDS_ON stays on for at least 1 second after WINRND, so the players can see who won, and remove their fingers.

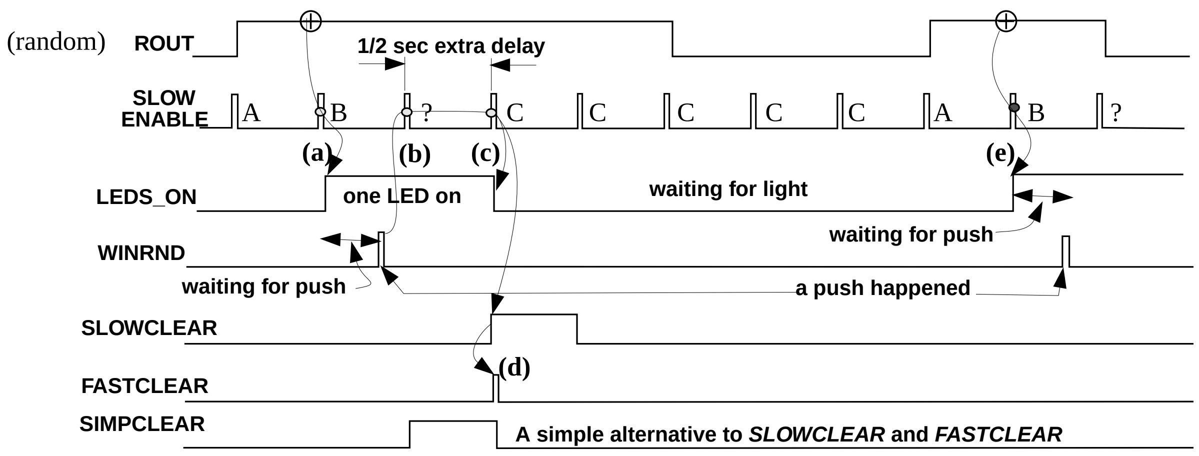
LEDS_ON will stay on indefinitely if no one pushes a button. This is a feature for absent minded players.

(c) After the 1 sec extra delay, LEDS_ON will go off even though ROUT may stay high.

(d) The Master controller generates SLOWCLEAR and from that FASTCLEAR which clears the push-button latch.

(e) LEDS_ON should wait a short time before coming on again. If ROUT stays high it would come back on the next SLOW ENABLE. Waiting until ROUT has a rising edge gives a better delay.

FIGURE 11. .The midgame after one or more rounds.
(a) ROUT rising turns on the LEDs, the signal to the players to push. (b) A push raises the WINRND pulse, which will, after 1/2 to 1 sec, (c) lower LEDS_ON and pulse SLOWCLEAR. (d) SLOWCLEAR in turn will pulse FASTCLEAR. (e) The next LEDS_ON will wait till the next rising edge of ROUT.



5.3.3 Alternatives for The Rising Edge Detector Part of The Controller

The controller of Sect. 5.3.4 allows LEDS_ON to come on after a rising edge from the random-bit generator. One could remove state A and use a pulse generated from the rising edge of ROUT to signal a simpler controller when LEDS_ON should go high. This should give a simpler overall design, especially since you will design an edge-detector is an OPP circuit. Further the OPP can share one of the flip-flops in the random generator.

5.3.4 A Possible, But Not Very Good, Controller

These tables define a controller with timing close to FIGURE 10 and FIGURE 11. The letters within the SLOWENABLE signal in FIGURE 11. correspond to the states as closely as possible.

Note this controller does not hold the lights lit for 1/2 sec. after a push. This 1/2 sec corresponds to the “?” state in FIGURE 11. Also it does not turn on all the lights immediately after RESET.

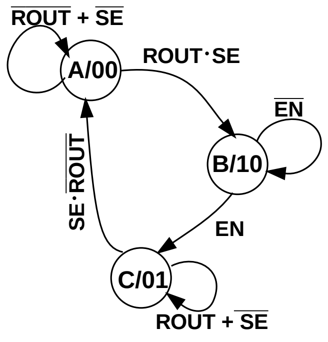
State	Next State SE·ROUT 00	01	11	10	Output
A	A	B	B	~	0,0
B	B	B	C	C	1,0
C	A	C	C	A	0,1

Outputs = **LEDS_ON, SLOWCLEAR**

State	Next State SE·ROUT 00	01	11	10	Outputs
00	00	01	01	~	0,0
01	01	01	11	11	1,0
11	00	11	11	00	0,1
10	~	~	~	~	~

Outputs = **LEDS_ON, SLOWCLEAR**

FIGURE 13 The states of the poor quality controller. SE = SLOWENABLE



5.3.5 Mealy and More in Master Controller Design

This the hardest part of the design. You can use either a Mealy or a Moore outputs. However:
(i) Mealy outputs usually reduce the number of states needed.
(ii) Moore output are easier to understand.
(iii) You may use SLOW ENABLE to step the controller in 1/2 second steps. In Verilog one might might imply enabled flip-flops by using:

```
always @(posedge clk or posedge clr)
        begin if (clr) state=reset;
        else if (clk & slowenable)  state=next_state;
```

(iv) A Moore output may make one to wait 1/2 sec between a jumping the light and a LEDS_ON. With Mealy outputs one can can react quickly. Hint: There is a better input than WINROUND for the controller.

This is a summary of some points about the master controller:

a. Turn all the lights on during reset and for 1 second therafter.

b. Leave the light on for ΔT, 1/2 < ΔT < 1 sec, after a push.

c. Move the light right away after a jump-the-light, and leave it on in the new position for ΔT, 1/2 < ΔT < 1 sec. The controller can treat a jump exactly like a valid push. Let the the scoring circuit decide which LED goes on.

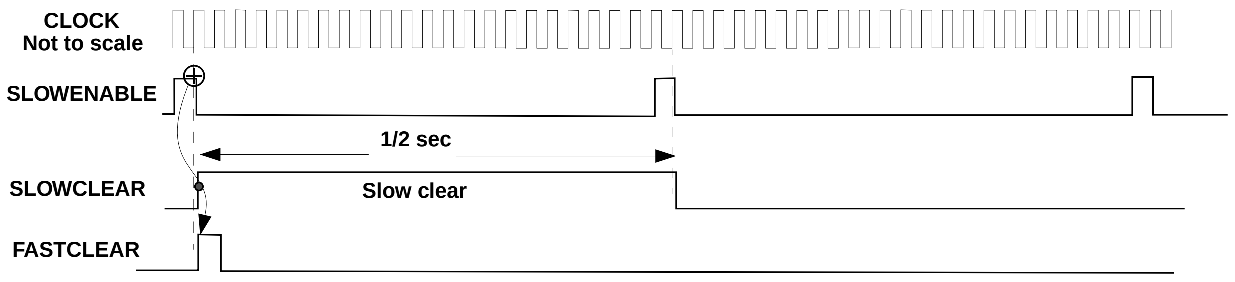
d. Capture a WINROUND signal even if thought it comes and goes between SLOWENABLE pulses.

e. Record a jumped light even in the first half second after the light goes off.

5.3.6 SLOWCLEAR and FASTCLEAR

SLOWCLEAR comes on as LEDS_ON goes off. It is slow because it is generated by the SLOWENABLE. SLOWCLEAR is not fast enough to clear the push-button latch. Someone might jump the light during the half second it is high. The FASTCLEAR is a pulse one CLOCK cycle wide starting at the rising edge of SLOWCLEAR. It will clear the latch quickly..

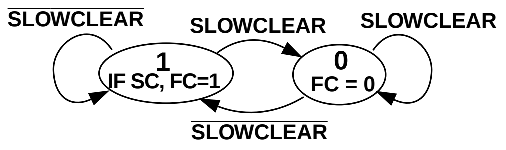
FIGURE 14. Slow clear has to go between two SLOWENABLE pulses. It is 1/2 sec. long. We prefer to clear the bush-button latches with FASTCLEAR which only takes them out of service for 4 ms.



5.3.7 The Fast/Slow Clear State Machine

This circuit takes SLOWCLEAR which lasts about 0.5 sec, and initiates a fast pulse FASTCLEAR which lasts only one clock cycle. This circuit is called an edge detector, or a “one-and-only-one” pulse circuit. It generates a pulse only when the state is 1 and there is a SLOWCLEAR input from the controller. This condition only lasts one clock cycle (1/250 sec.). The circuit uses only one flip-flop and one gate.

FIGURE 15b. The Mealy machine used to generate FASTCLEAR (FC) from SLOWCLEAR (SC).



5.4 Pseudorandom Sequence Generators

FIGURE 17 on page 12 shows a widely used circuit which generates a sequence of bits. The output may be taken from any of Q1,Q2.Q3 or Q4. While the sequence is periodic, inside of its 15 cycle period, it passes many of the statistical tests for randomness. Notice it is a 15, not a 16,cycle FSM. The 16th state is all zeros. If it gets into that state it will never come out.

These pseudorandom shift registers can be made in any length but the most efficient ones have N flip-flops and have sequence length 2^N-1. The final state is all zeros and not used.

Connections for sequences with periods up to 511 bits are shown in FIGURE 16 and the table on page 12

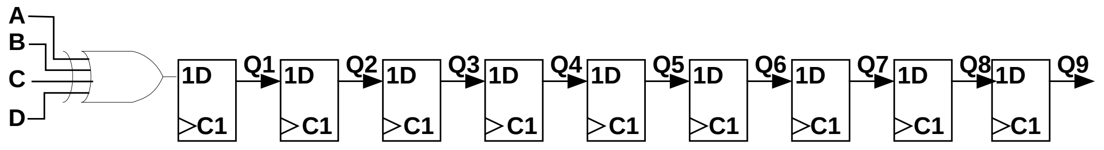


FIGURE 16 A general pseudorandom bit stream generator. The number of flip-flops used depends on the length of sequence desired. Thus, for a 15 bit sequence remove all but 4 flip-flops. For a 31 bit sequence, remove all but 5 flip-flops, etc. See FIGURE 17 and FIGURE 18

SEQUENCE LENGTHS FOR FIGURE 16

Link	15	31	63	127	255	511
A to	Q3	Q3	Q5	Q6	Q2	Q5
B to	Q4	Q5	Q6	Q7	Q3	Q9
C to	Omit	Omit	Omit	Omit	Q4	Omit
D to	Omit	Omit	Omit	Omit	Q8	Omit

Examples using this table are shown in FIGURE 17 and especially FIGURE 18.

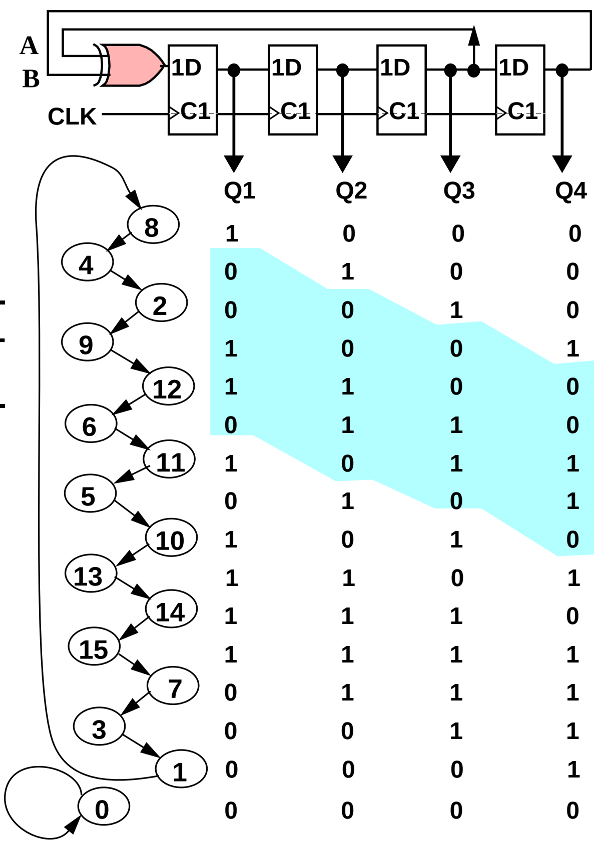
Note that the generator must not be all zeros initially or it will not start, i.e. this is a state machine which will lock up in the 0000 state!

Note cycle 0 and 15 are the same The sequence is always of length 2^N-1 because the flip-flops can never be all zeros!

Verilog code to imply the random number generator on the right, is given below.

```
reg [4:1] Q;
always @(posedge clk or posedge clr)
   begin if (clr) Q<=4'd8;
        // Must not reset state to zero.
        else if (clk)
            begin
            Q[4:2]<=Q[3:1];        //(line 7)
              Q[1] <= Q[1]^Q[4];  //(line 8)
            end
/*      The "<=" is called a nonblocking
assignment. The Qs on the right all use
the values they had at the begin.
     Thus line 8 uses the original Q[4] and
not the revised value from line 7.*/
```

IGURE 17 A circuit made from XOR and FFs which gives repeatable "random" number sequences.
This example can be constructed from the general example of FIGURE 16 by connecting A to Q3 and B to Q4



Q1	Q2	Q3	Q4
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

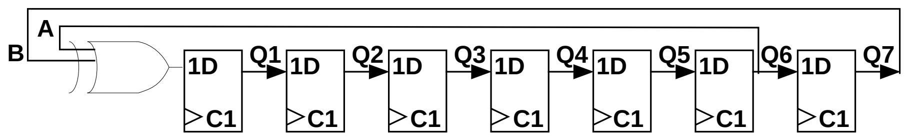


FIGURE 18. A 127 state pseudorandom bit generator constructed from the general circuit of FIGURE 16. Use the connections for the 127 sequence length, and remove the two unused flip-flops.

5.4.1 Alternative Random Bit Stream Generators

a. One alternative is to have a square-wave oscillator running asynchronously at a high speed and sample it every clock cycle.

b. Previous students have used a counter which was preloaded with the least significant bits of the clock divider every time a button was pushed. With this number in preloaded, the counter was then clocked down to zero to give a random delay. Since buttons tend to get pushed a reaction time after a slow clock edge, the numbers tend to be close together. If you use this method, reverse the bit order and/or use the pseudorandom generator as your divider.

5.5 Generating the SLOWENABLE

This circuit must generate a single pulse every 128 clock pulses. One needs a 7 bit counter which gives a carry pulse (TC) out every time it rolls over from say 1111111 -> 0000000. Counters are trivial to generate in Verilog.

The 7 bit pseudorandom number generator counts 1 to 127, which is close enough to 128 for this game. The order of the count is pseudorandom, but that does not matter for this application, and it is the simplest counter you can build. Can you see how to use it to generate SLOWENABLE? (In the past, using this circuit has been a mark generator for the design review)

6.0 The Design of the Asynchronous Circuits

6.1 The Basis of the Design of the Push Button Circuit

The push button circuit shall be as fair as modern technology can make it.

i) It must tell who pushed first within a time equal to the propagation delay of two inverters.

ii) It shall have no theoretical bias toward one player. Because of the difference in delays between gates and/or lead lengths, it is impossible not to have some bias in the circuit after construction.

iii) If your circuit goes into a tie state, it shall either:

a. have equal theoretical probability of exiting toward either player.

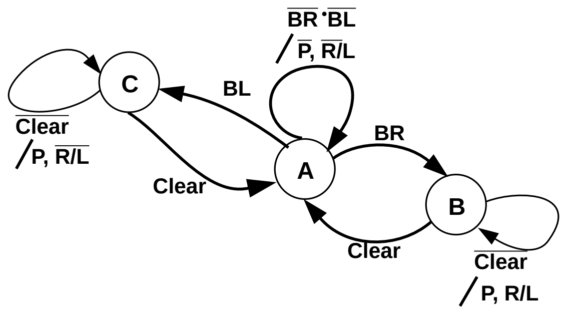
b. leave the light stationary for that round.

iv) It should not depend on which player releases their button first. In a poor design, this can happen in the case of a tie, which causes the winner to be determined by the bounce properties of the push buttons.

6.1.1 The State Diagram for the Latch.

FIGURE 20 The Push Button Latch

State A........	No button pushed since Clear.
State B........	Button 2 (right) was pushed first.
State C........	Button 1 (left) was pushed first.
Input BL.......	Left Button input.
Input BR.......	Right Button input.
Input Clear...	FASTCLEAR
Output P........	PUSH; some button pushed.
Output R/L....	R(H)/L(L), High if BR came before BL; low if BL preceded BR



6.1.2 State Table for the State Diagram

Asynchronous circuits must have a race free state assignment. This means state variables should change by only one bit at a time. Two possible assignment are shown in FIGURE 21The state table for an assignment of FIGURE 21 is shown in TABLE 9. The two state variables needed will be named G and H

The equations governing G+, H+ and hence the circuits to generate them, are obtained from the Karnaugh maps, which in turn are obtained from the state table. Because of the don't care states, one appears to have a choice of several equations for G+ and H+. However behavior and/or reliability of the circuit will suffer if the wrong choice is made, as will be shown in Sect 5.1.4.

Pres State	Next State					Output
	Inputs= **BL, BR, Clear**					
	000	010	110	100	- -1	P, R/L
A	A	B	-	C	A	0 -
B	B	B	-	B	A	1 1
C	C	C	-	C	A	1 0

Circled states are stable. - is don't care

FIGURE 21. Two possible state assignments for the latch, one has a race, the other is race free.

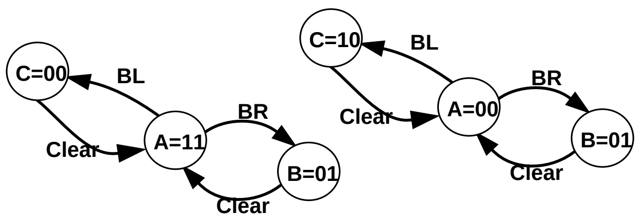


TABLE 9.

Pres State	Next State **G+ H+**					Output
	Inputs= **BL, BR, Clear**					
G H	000	010	110	100	- -1	P, R/L
0 0	0 0	0 1	-	1 0	0 0	0 -
0 1	0 1	0 1	-	0 1	0 0	1 1
-	-	-	-	-	-	-
1 0	1 0	1 0	-	1 0	0 0	1 0

A possible set of equations and the way they complete the state table, is shown below

.

$$G+ = (G + BL)\overline{H}\ \overline{Clear} \tag{EQ 1}$$

.

$$H+ =(H + BR)\overline{G}\ \overline{Clear} \tag{EQ 2}$$

MAP of G+

Clear=0

GH \ BL,RL	00	01	11	10
00	0	0	~	1
01	0	0	~	0
11	~	~	~	~
10	1	1	~	1

Clear=1

GH \ BL,RL	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	~	~	~	~
10	0	0	0	0

MAP of H+

Clear=0

GH \ BL,RL	00	01	11	10
00	0	1	~	0
01	1	1	~	1
11	~	~	~	~
10	0	0	~	0

Clear=1

GH \ BL,RL	00	01	11
00	0	0	0
01	0	0	0
11	~	~	~
10	0	0	0

TABLE 10. Table 9 with "d" filled in

Pres State	Next State **G+ H+**					Output
	Inputs= **BL, BR, Clear**					
G H	000	010	110	100	- -1	P, R/L
0 0	0 0	0 1	1 1	1 0	0 0	0 -
0 1	0 1	0 1	0 1	0 1	0 0	1 1
1 1	0 0	0 0	0 0	0 0	0 0	-
1 0	1 0	1 0	1 0	1 0	0 0	1 0

TABLE 11. Table 10 with letter names.

Pres State	Next State **G+ H+**					Output
	Inputs= **BL, BR, Clear**					
G H	000	010	110	100	- -1	P, R/L
A=0 0	A	B	D	C	A	0 -
B=0 1	B	B1	B	B	A	1 1
D=1 1	A	A	A	A	A	-
C=1 0	1 0	1 0	C	C	A	1 0

cycle

6.1.3 Side Effects of the Complete State Assignment

As an example look at equations 1 & 2 above. Circling the maps gave definite values to the don't care next states. These fill ins are shown in TABLE 10. TABLE 11.shows the states converted back to letters. The tie state (both buttons pushed at once) is unstable. Pushing both buttons at once will send one to state D, which will return one to state A, which will send one back to D, which --. Actually, instead of changing 00->11->00, after a few cycles one of F or G will change before the other, sending one to either state B or C. This is an example of metastability (Sect. A.2).

6.1.4 Other Considerations Based on State Assignment

i) If both buttons are pushed at once:

- Make sure a tie does not have a bias toward either side?
- Understand that pushing two buttons together is not a normal race. It is true that the state change A=00 ->D=11, may instead go 00 -> 10 or 00 -> 01, but this is not an avoidable race. It is just one side or the other winning. It is normal circuit operation.
- When there is a tie, if one stays in state A=00, the push button circuit is still able to latch toward one side or the other. When a switch breaks contact due to bouncing, the person pushing that switch will lose. This "lose on first bounce" is an avoidable unfairness.

6.1.5 Examples of Low-Grade PB Latches

Table 12:(a) Suppose B won the push in this machine. When C pushes, the machine will come out of state B, go to the tie state D and stay there.

Table 12: (a)

Pres State	Next State **G+ H+**				
	Inputs= **BL, BR, Clear**				
G H	000	010	110	100	- -1
A=0 0	**A**	**B**	**D**	**C**	**A**
B=0 1	**B**	**B**	**D**	**B**	**A**
D=1 1	**A**	**A**	**D**	**A**	**A**
C=1 0	**C**	**C**	**D**	**C**	**A**

Table 12: (b)

Pres State	Next State **G+ H+**				
	Inputs= **BL, BR, Clear**				
G H	000	010	110	100	- -1
A=0 0	**A**	**B**	**A**	**C**	**A**
B=0 1	**B**	**B**	**B**	**B**	**A**
D=1 1	**A**	**A**	**A**	**A**	**A**
C=1 0	**C**	**C**	**C**	**C**	**A**

Table 12:(b) If there is a tie, this machine stays in state A.(inputs 110). However if one button is released, the other person wins. This is a lose on first bounce" circuit.

Table 13:(a) This is another "lose on first bounce" circuit. The machine will not stay in the tie state D when one button is released.

Table 13: (a)

Pres State	Next State **G+ H+**				
	Inputs= **BL, BR, Clear**				
G H	000	010	110	100	- -1
A=0 0	**A**	**B**	**D**	**C**	**A**
B=0 1	**B**	**B**	**B**	**B**	**A**
D=1 1	**A**	**B**	**D**	**C**	**A**
C=1 0	**C**	**C**	**C**	**C**	**A**

Table 13: (b)

Pres State	Next State **G+ H+**				
	Inputs= **BL, BR, Clear**				
G H	000	010	110	100	- -1
A=0 0	**A**	**B**	**B**	**C**	**A**
B=0 1	**B**	**B**	**B**	**B**	**A**
D=1 1	**A**	**A**	**A**	**A**	**A**
C=1 0	**C**	**C**	**C**	**C**	**A**

Table 13:(b) A circuit based on this table will never go to state D. However it has a clear bias toward B.

You should choose a state table for your latch. You should analyze its behavior under tie conditions. And you must explain why your choice was theoretically fair. This should be in your report.

6.1.6 Reliability Based on Races and Hazards

No asynchronous state table can be guaranteed to work without checking the following:
a) Are there hazards in the circuit.
b) Are there any critical races in the state table?
c) Are there essential hazards in the state table?

6.1.7 Checking for Hazards

Test Summarized

1. Expand the logic equations for your state table.
2. Look for variables x, where x and $\bar{x}$ are both in the same equations. If x and $\bar{x}$ do not both appear in the any equation there are no hazards.
3. If x and x do both appear, make all the other variables 1 and 0 in all possible combinations. Try to get the equation to reduce to either $x\bar{x}$, $x + \bar{x}$, $x\bar{x}$ + , or $(x +\bar{x})x$. If one cannot, there are no hazards.

Example 1

Equ. 1 & 2 will be checked for hazards.

$G+ = (G + BL)\bar{H}.\overline{Clear}$ $H+ =(H + BR)\bar{G}.\overline{Clear}$ (1 & 2)

Since no variable appears in both complemented and uncomplemented form, there are no static or dynamic hazards.

Example 2

$G+ = (GH + BL)\bar{H}.\overline{Clear}$

This equation has a hazard, because G=1, BL=0 and Clear=0, reduce G+ to $G+ = H\bar{H}$

Most of the PUSH-BUTTON LATCHEs have no hazards. However it is a major offense not to show in your report how you checked for hazards. **In general it is very important to check for hazards in asynchronous circuits!**

6.1.8 When You Must Check for Hazards

1. Any asynchronous latch must be checked for hazards. The state table cannot be trusted if there are hazards.
2. Synchronous circuits normally do not need to be checked for hazards. All changes in synchronous circuits come soon after a clock edge. One need only make the clock cycle long enough so that all glitches die out before the next clock edge. Then they cannot propagate to the next state.
3. Asynchronous circuits feeding clocked logic should be hazard free. If a glitch occurred near a clock edge, it could be captured as a false signal.

A common error in the PUSH-BUTTON LATCH, is to use $G\oplus H$ to generate the PUSH signal. G and H will both be high on a tie, so $G\oplus H$ should not generate a PUSH signal on a tie. Unfortunately, there is a built-in two-variable hazard in the exclusive-or. If a tie occurs on a clock edge the hazard may generate a glitch which may be caught in the SYNCHRONIZER circuit. The cure is to do your X-OR after the SYNCHRONIZER.

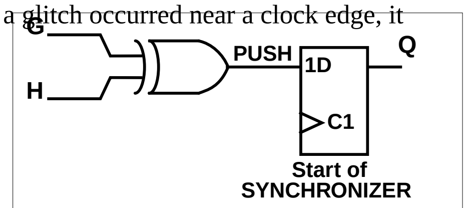
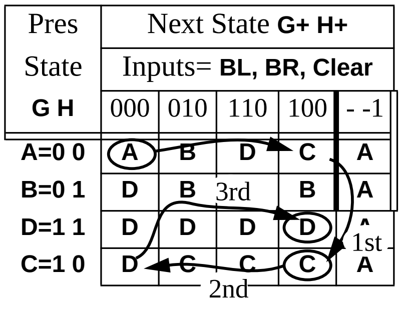


FIGURE 22 common hazard in the push button latch circuit

6.1.9 Checking for Essential Hazards

Races and essential hazards and hazards are equally likely. Unfortunately many people only check for the one they understand, which is races. Checking for only races is like checking only addition in arithmetic, and pretending that multiplication and division do not need checking.

An essential hazard is a race between an input and a state change. It is a property of the state table, and unlike races, cannot be removed by a new state assignment.



Pres State	Next State G+ H+				
	Inputs= BL, BR, Clear				
G H	000	010	110	100	- -1
A=0 0	A	B	D	C	A
B=0 1	D	B		B	A
D=1 1	D	D	D	D	^
C=1 0	D	C	C	C	A

The test for essential hazards is to see if three changes in an input variable, i.e. 1 -> 0 ->1 send the machine to a different state from a single change 1 ->0.

In this example:

Starting at state A	
The single change BL	0 -> 1
cause state change	A -> C
The triple change in BL	0 -> 1 -> 0
causes state changes	A -> C -> D

Here the final states are different, and this state table has an essential hazard.

6.1.10 Asynchronous Circuits in Verilog

Verilog is not very good at synthesizing asynchronous circuits.
You will want to turn off any optimization.
You may have trouble with **always** blocks, and have to use **assign**. If that also gives trouble build the feedback elements with basic gates, **and** and **or**. For example:

$G+ = (G + BL)\overline{H}.\overline{Clear}$ becomes:

```
and  and1(G, temp, ~H, ~Clear);
or     or2(temp, G, BL;)
```

6.2 The Design of the Synchronizer Circuit

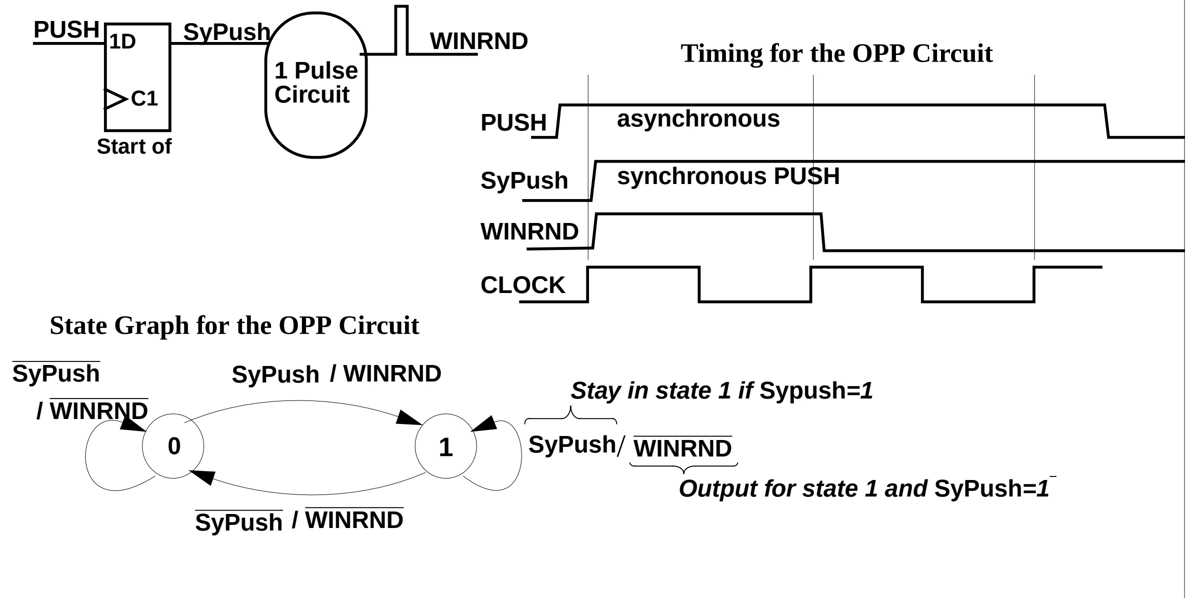
The synchronizer circuit takes the asynchronous PUSH input which may change anytime and makes it into SyPush which cannot change near the clock edge. That is during the setup or hold times of the flip-flops.

Logically independent asynchronous input signals should be sent through a single D flip-flop before they enter the rest of the synchronous machine. But this will not work for two logically-dependent signals. See Sect 4.5.

6.3 The Design of the OPP (One-Pulse-per-Push) Circuit

It gives out a single synchronous pulse one clock cycle long. It always gives one pulse, and only one pulse.

FIGURE 23 The sychronizer D flip-flop and a the OPP circuit. The OPP circuit is simpler to implement as a Mealy machine, as shown by its state graph.



Appendix A : Timing Properties of Flip-Flops

A.1 Normal D flip-flop operation.

FIGURE 24. The normal operation of the rising-edge edge-triggered D flip-flop. The input, as sampled a short time *before* and/or *during* the rising clock edge, is stored in the flip-flop. The stored data output appears a short time *after* the clock edge.

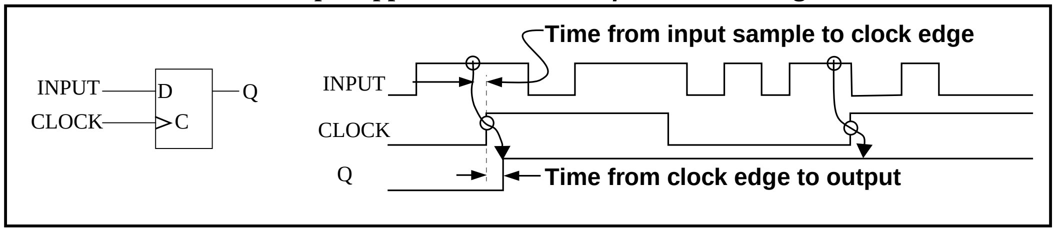
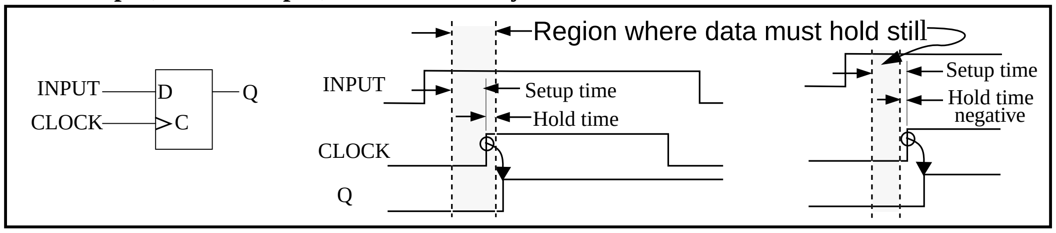

INPUT
CLOCK
D
C
Q
INPUT
CLOCK
Q
Time from input sample to clock edge
Time from clock edge to output

FIGURE 25. If the D input of the flip-flop changes during this final sample time, the flip-flop cannot tell whether it should store the old or the new D value. Thus the manufacturer places a time range around the clock edge where the input should not be changed. If the input is changed inside this region, the output may be derived from either the old input, the new input, or even half-way in between.



A.1.0.1 Setup, Propagation and Hold Times

There are two time intervals associated with the active clock edge in D flip-flops:

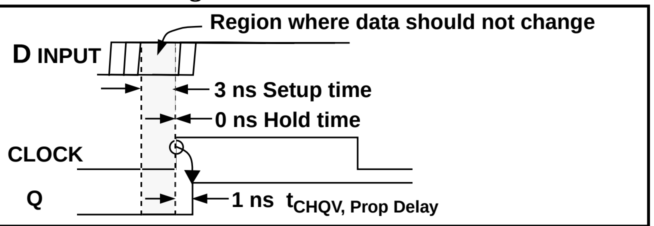
- The *setup time,* which is the interval before the clock where the data must be held stable.
- The *hold time,* which is the interval after the clock where the data must be held stable.

Most modern flip-flops have a zero or a negative hold time. A negative hold time means the data can change slightly before the clock edge without risk of not being captured.

FIGURE 26. Example of the setup time, hold time, and propagation delay for a 74AC74 CMOS flip-flop. The INPUT signal shows several legal transitions, but avoids showing one in the shaded area where data should not change.

74AC74

Setup time	Hold time	Prop Delay t_{CHQV}	
3 ns	0 ns	1 ns	min
		6.5	max
1.5 ns	-1.5ns		typ

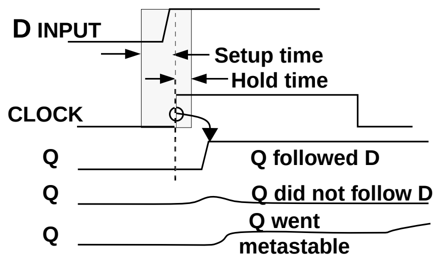

Region where data should not change
D INPUT
3 ns Setup time
0 ns Hold time
CLOCK
Q
1 ns $t_{CHQV, Prop Delay}$

One other useful specification is the flip-flop *clock to output propagation delay.* It is given the abbreviation $\mathbf{t}_{CHQV}$ (**t**ime from Clock High to Q Valid).

The minimum delays are the ones one uses in most designs. The manufacturer will guarantee those. The typical delay is not very rigorously defined.

A.1.0.2 Summary of the Restricted Region

The *restricted* region is the time interval round the active clock edge where a flip-flop's input signal should not change. If it does change in that region the flip-flop's output, after the clock edge, may:
1) follow the change in D.
2) not follow D.
3) follow it halfway (go metastable).



A.2 Metastability,

Metastability is when a flip-flop balances at "1/2" trying to make up its mind whether to go to a 1 or a 0.

The flip-flop output will eventually go to either a 1 or a 0. Usually this will happen in less than a clock cycle.

Metastability is a problem only for flip-flops running at close to their maximum speed. Not for the Tug-of-War.

The Push-Button-Latch under tie conditions is metastable in the state table of Sect. 6.1.4. Many metastable circuits oscillate for a cycle or so.

A.3 Flip-Flops With Inputs Near the Clock Edge

A.3.1 Synchronous and Asynchronous Signals

A *synchronous* signal will be defined as one which is constrained so it cannot change in the restricted region near the clock edge. An *asynchronous signal* can and will change anywhere. See FIGURE 27..

Any signal which passes through a flip-flop is synchronous. The propagation delay inside the flip-flop, $\mathbf{t_{CHQV}}$, will give enough delay to move the signal edges out of the restricted region. Thus the Q signal in FIGURE 28. is synchronous and results from sending the upper signal through a D flip-flop.

FIGURE 27. Three different D inputs to a flip-flop are shown. The upper two are synchronous because they do not change in the restricted region (inside the setup and hold times). The lower one is asynchronous because it has one transition inside the restricted region.

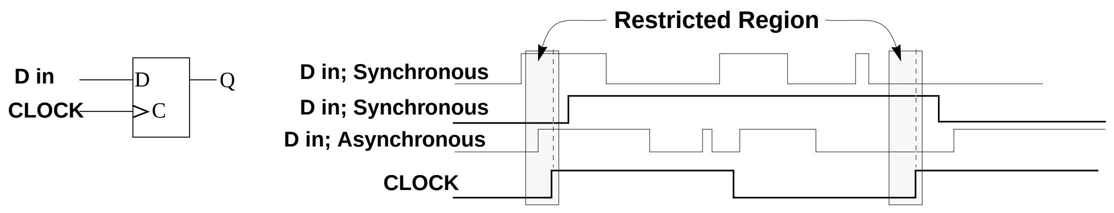
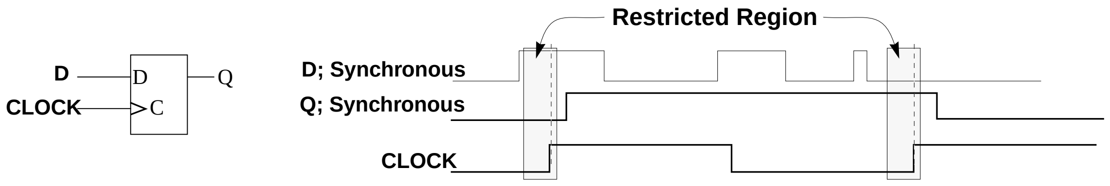


FIGURE 28. **The output of a flip-flop is always synchronous. The only exceptions are unlikely and are mentioned in the text.**



A.3.1.1 Making an Asynchronous Signal Synchronous

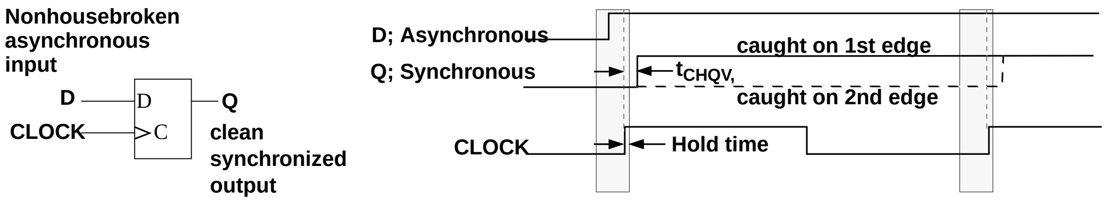
Sending an asynchronous signal through a D flip-flop will sychronize it provided:

a) $t_{(CHQV)Min} < t_{HOLD}$ This should not happen in any flip-flop designed since 1975.

b) Metastability is not a problem. This is usually the case unless the flip-flops are running close to maximum clock speed.

When the asynchronous signal changes in the restricted region, it may or may not be captured. However, provided the asynchronous signal stays stable for at least a clock cycle, the signal will be captured on the next clock edges. If one needs to catch superfast pulses, which rise and fall inside a clock cycle, one must use an asynchronous pulse-catching circuit.

FIGURE 29. **Synchronizing an asynchronous input. Under the conditions above, the change in D inside the restricted region will either be caught on the edge where it changed, or on the next edge.**



A single asynchronous input may be synchronized by sending it through a D flip-flop before allowing it into a synchronous circuit.