Laboratory 4 A MIDI Interface

Overview

MIDI is the name for a digital interface used in electronic music. It might be used between a digital keyboard and a synthesizer or between a keyboard and a PC running a music composition program, like CakeWalk™. MIDI is a serial interface, the bits come in one after the other. The high-level view of a MIDI interface is shown below.[1]

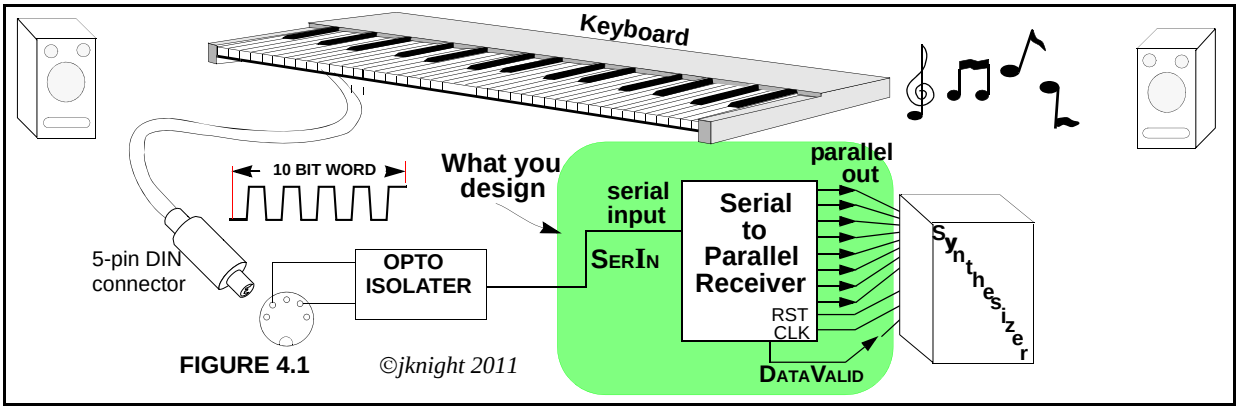
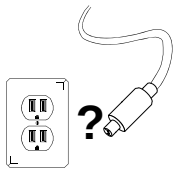


FIGURE 4.1 ©jknight 2011

The ***opto-isolator*** changes the electrical signal to light and back again. It keeps the rest of the interface from burning out if the MIDI input gets plugged into 120 V by someone with strong wrists and limited technical knowledge.



The ***keyboard*** sends out a 10 bit word, containing 8 bits of data called a *data byte*, and two control bits. The signal is sent out serially, one bit following another.

The ***Serial-to-Parallel Receiver*** changes the 10-bit serial word from the keyboard into an 8-bit parallel data byte which is then sent to the computer. This is what you will design. The 10 bit serial word has a start and stop bit which are removed for the 8-bit parallel output (See Fig. 4.2).

This *receiver* captures 10 bits as they come in serially one after the other. A group of bits may start at any time. The receiver waits until all 8 bits are collected and stable. Then it sends them out in parallel. to some other instrument like a mixer or ***synthesizer.*** It makes the DATAVALID line true whenever the 8 data bits are stable and available for the synthesizer to read.

Brief Higher-Level Description of the MIDI Signal (Background, not part of lab)

These 8-bit data bytes have a musical meaning. The first data byte is called the STATUS byte. It is followed by 0, 1, or 2 bytes called DATA-1 and DATA-2 bytes, as appropriate for the STATUS byte.

1. See for example: http://en.wikipedia.org/wiki/Musical_Instrument_Digital_Interface

Table. 4.1 shows some commands. The bytes STATUS, DATA-1 and DATA-2 are interpreted in software. However this lab will look only at the hardware part of the interface which changes each 8-bit byte sent serially, into 8 bits sent out in parallel on 8 separate wires

Table 4.1Some MIDI commands applicable to keyboard interfaces.

STATUS	DATA-1	DATA-2	Description
FF	-none-	-none-	Reset the MIDI system
80	*Note*	-none-	*Note* off
90	*Note*	*Velocity*	Turn on *Note* in DATA-1 with key *Velocity* in DATA-2
D0	*Pressure*	-none-	Change key pressure of the current note, in mid-note.

(End of background)

The Serial Input (SerIn) Signal

The keyboard will send out a *word* made up of 10 *bits* sent serially one after the other. It contains a **START** bit, 8 data bits, and a **STOP** bit. There is an idle time between words of indefinite length.

FIGURE 4.2 A typical MIDI signal showing three words.

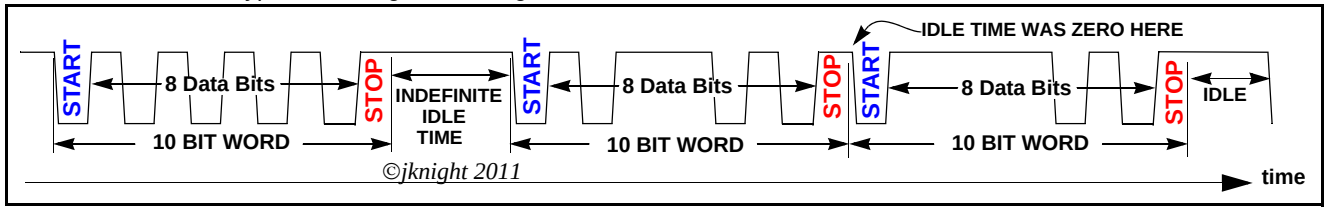
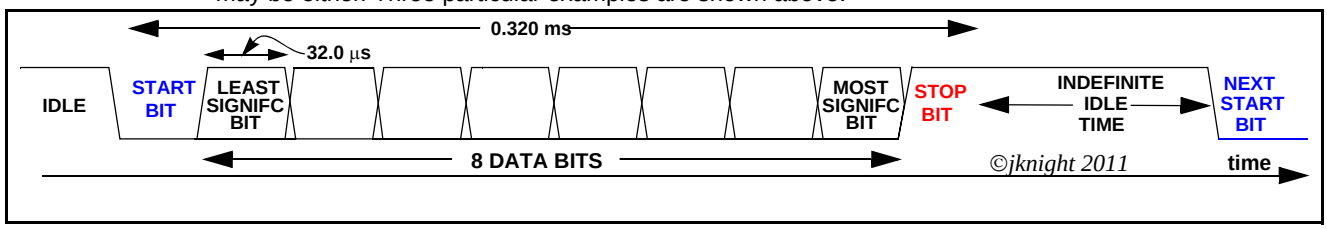


FIGURE 4.3 The *protocol* used by MIDI equipment. The data bits are drawn as both 1 and 0 since in general they may be either. Three particular examples are shown above.



IDLE: The signal is always “1” when idle.

START BIT: The signal always has a low (0) bit before the 8 data bits start. This tells the interface to start listening.

DATA BITS: 8 data bits follow the start bit. Any individual bit may be “1” or “0.” ”

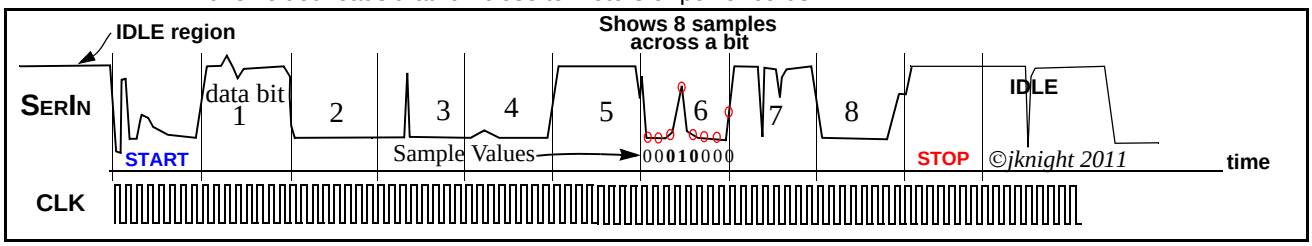
STOP BIT: The STOP BIT follows the last data bit and is always “1”. It is really the start of the next idle period. The next start bit may come immediately after the stop bit, or the signal may be idle for years.

Noise on the Serial Input (SerIn) Line

Signals coming in from a long serial line may have noise on them. This may confuse the receive circuitry and cause an incorrect input bit to be read. To try to avoid this, the receiver takes several samples inside each

midi bit, one sample at each rising clock edge. Fig. 4.4 shows the samples taken for a typical bit (data bit 6) in a noisy midi word. The digital gates will round the analog input to the nearest 0 or 1.

FIGURE 4.4 A midi Serial Input (**SERIN)** with some noise on it. Noise may be picked up by long leads, particularly unshielded leads that run close to motors or power cords.



Digital circuitry stores these samples temporarily in flip flops. The value stored is the rounded value of SERIN just before the active clock edge. Your circuit will take the majority of 3 sample values near the middle of each midi bit as the value for that bit. Thus you might use MAJORITY(0,1,0) = 0, as the value for data bit 6 in Fig. 4.4. See the prelab, question. .

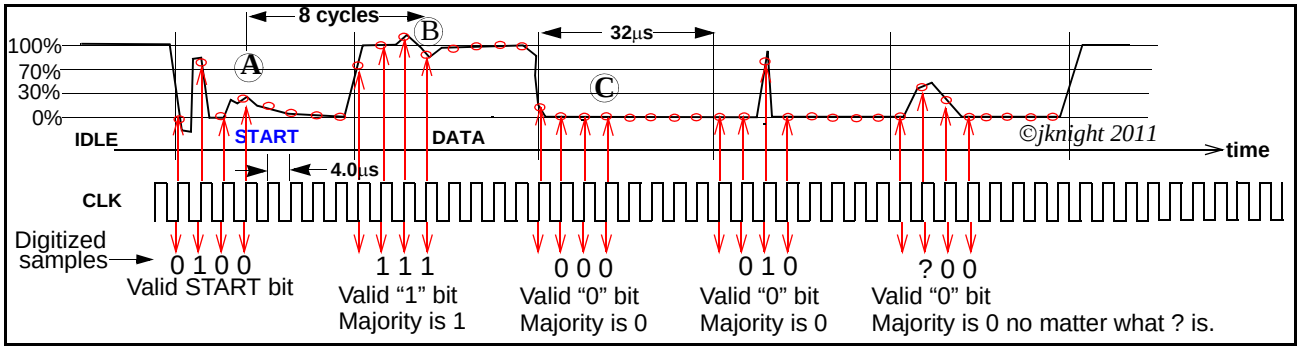
For the START bit, we will be even more careful. We will use 4 samples and demand that the 1st sample, and at least 2 more samples, be low. See the prelab, question. 7.2.

The Clock Rate and Majority Value of the Bit Samples

To have reasonable averaging benefits inside a noisy data bit, this design will have eight samples (i.e 8 clock cycles) inside each MIDI data bit. Since each incoming MIDI bit is nominally[2] 32µs long, Taking a sample on each rising clock edge would make the clock period 4.0µs long (0.25MHz).

FIGURE 4.5 The **SERIN** sampled by 8 clock edges per bit (small circles) with some digitized values. Recall in CMOS, a signal voltage of over 70% of a "1" is taken as a "1", and below 30% is taken as a "0". Between 30% and 70% it may be taken as either.

Here the START bit is recognized after four samples of values 0100. This starts the rest of the circuitry going (A). Then after 8 clock cycles (B) the majority of the last 3 bits is checked and its value is stored as the value for the 1st data bit. One waits another 8 cycles (C) and again stores the majority value as the 2nd data bit, etc. until the eight data bits have been read.



Generating the GRAB Pulses That Capture the Data

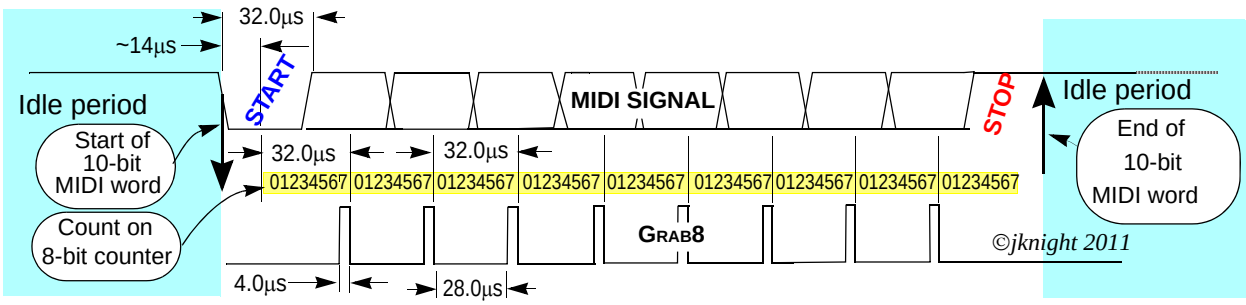
The START bit will be recognized after about four clock cycles, point(A), in Fig. 4.5). This will start a counter going which will count 8 more clock cycles. This should place one timewise near the middle of the 1st data bit (point (B)). At that point, the output of the majority calcuating circuit for the past three samples will be stored as the 1st data bit. Then the counter will count another 8 clock cycles, to point (C), and store the majori-

2. "nominally" means it is supposed to be 32µs but may be higher or lower, perhaps by several percent.

ty circuit output as the 2nd data bit. This will continue until all 8 data bits are stored. The last bit is a STOP bit, and we do not need to store its value. Instead, we count the 8 clock cycles and then start searching for the next start bit.

GRAB8 is the name of the signal that tells when to store the majority circuit's output will GRAB8. GRAB8 will have 8 pulses per MIDI word, each pulse one clock cycle (4.0µs) long, see Fig. 4.6.

FIGURE 4.6 The detailed timing relations between the MIDI Signal and the GRAB8 signal.
We ignore noise here. The falling edge of the START bit is at the big black down-arrow. 4 clock-edges after this arrow, an average[3] of 14µs, the counter will start and count from 0 to 7 (8 clock cycles). On the count of 7, a GRAB8 pulse will be generated. The counter will wrap around and count to 7 again, at which point another GRAB8 pulse is produced. This continues until these 8 pulses have grabbed the probable value of the 8 data bits from the majority circuit. After that GRAB8 goes low and stays low until the first data bit, after the next START bit is found.



When you get to FIGURE 4.14, you will see that when enabled flip flops are used, their output changes just after the rising clock edge, which happens just before the falling edge of the GRAB8 (enable) pulse.

The Block Diagram of the Circuit, See Fig. 4.7

LAST4SAMP captures a sample on every active clock edge and holds the last 4 samples.

FINDSTART checks the samples for 3 zeros out of 4 samples and sends out GOTSTART when this is found.

MAJORITYCIR sends out the majority value of the last 3 samples. These value will be collected by SER2PAR to be delivered as parallel output.

COUNTCLEAR checks that GOTSTART=1 indicating it has found a valid START bit. CLRCOUNT also checks that this was not just a low data bit by checking that the bit counter, **BitCount**, is sending out BIT9 indicating the last known input bit was a STOP bit and we are now in the idle period. If these conditions are met, it lowers CLRSAMPCOUNT so **SampCount** can start counting.

SAMPCOUNT: SampCount rests at a count of zero. It starts counting about halfway through the START bit when CLRSAMPCOUNT goes low. It counts 0,1,2,3,4,5,6,**7**,0,1,2, . . . sending out a GRAB9 pulse during each 7 count. This pulse should happen roughly half way through the data bit where SIGIN is most stable. GRAB8 is the same as GRAB9 except the last pulse is removed.

BITCOUNT keeps track of which bit has been reached in the midi word. It counts the number of GRAB9 pulses generated by **SampCount**, thus it counts 8 bits plus the STOP bit. Its BIT8 signal is used to remove the final GRAB9 pulse, to change GRAB9 into GRAB8. This lets the SER2PAR circuit saves only the data and not the STOP bit. BIT9 is high after all 8 data bits have been captured, and is used as a DATAVALID signal to tell the outside world the data is stable and can be read.

3. Four clock edges take 16µs, but the MIDI signal edges are not aligned with the clock edges, so the black down arrow in Fig. 4.6, might come partway through the first clock cycle in making the delay; 12µs < delay < 16µs.

SER2PAR responds to each GRAB8 pulse and transfers the data bit values from MAJ to the seven parallel data outputs, D0, D1,. . . D7.

FIGURE 4.7 The block diagram.

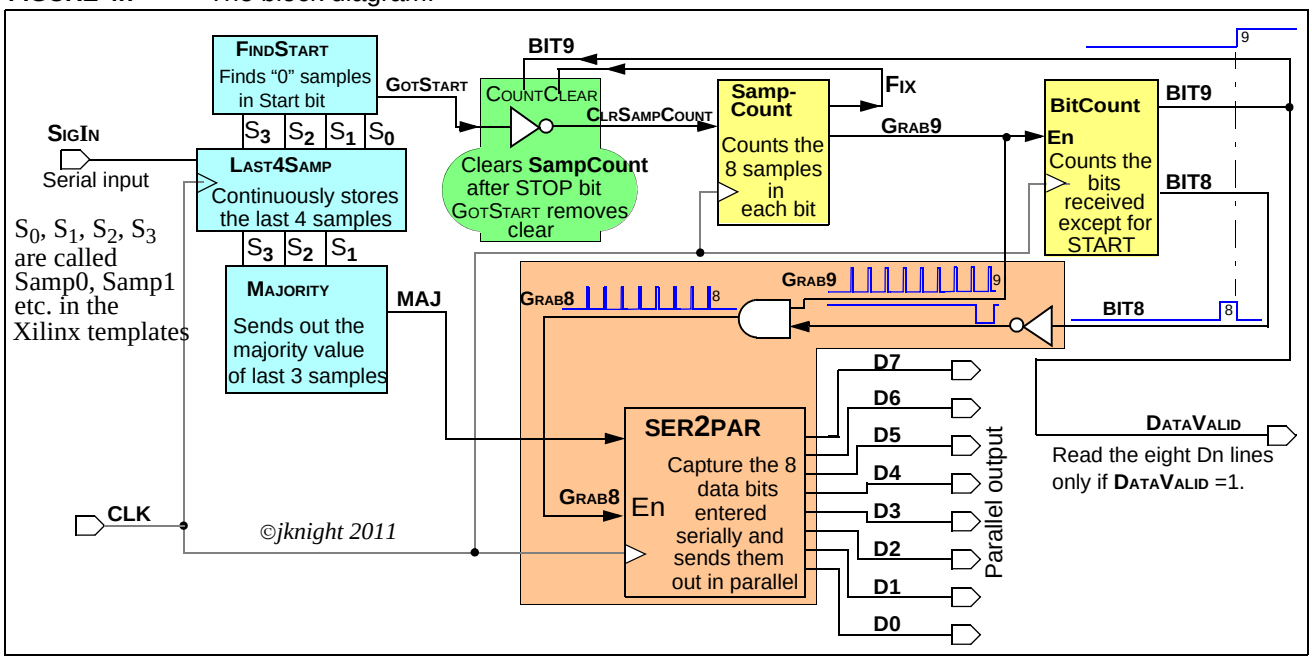
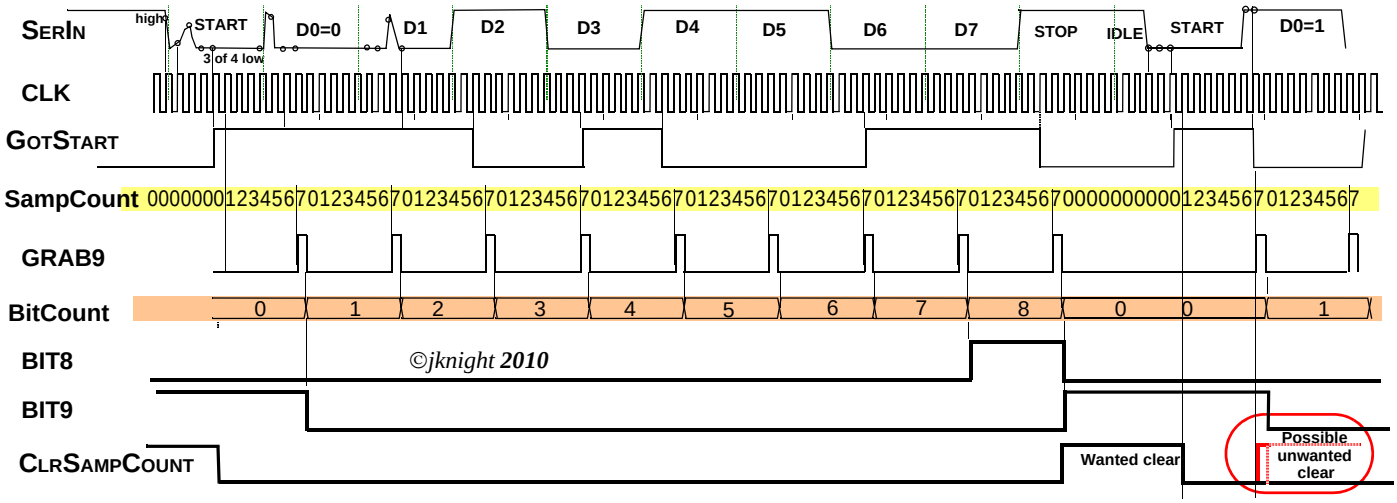


FIGURE 4.8 Signals needed to derive the circuitry to control the counters



Counters Using D Flip Flops

(A) The D Flip Flop

The value on input D is transferred to output Q on every active clock edge. Let:

Q^+ = the value of Q after the clock edge

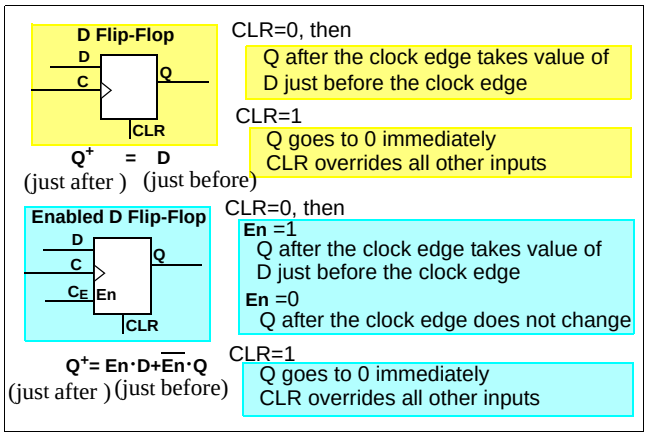
Q = the old value before the clock edge.

D = the value on D just before the clock edge.

The formula for the how the output changes is

$Q^+_{\text{(just after the clock edge)}} = D_{\text{(just before the clock edge)}}$ or

$Q^+ = D$



Counter Design with D Flip Flops

- The design starts with a state table (Fig. 4.9).
- In circuit using D flip flops, the inputs needed to get the Q^+s for the next state are simply D = Q^+.

FIGURE 4.9

State $Q_2\ Q_1\ Q_0$	Next State $Q_2^+Q_1^+Q_0^+$	D inputs $D_2\ D_1\ D_0$
0 1 0	0 1 1	0 1 1
0 1 1	1 0 0	1 0 0

SAMPLECOUNT Design with D Flip Flops

The details of constructing and filling in the **SampleCount** state table are shown in Fig. 4.11. They will be much easier to follow if you have read the lecture notes on how to design state machines with D flip flops.

Counter Design with D Flip Flops and Clear

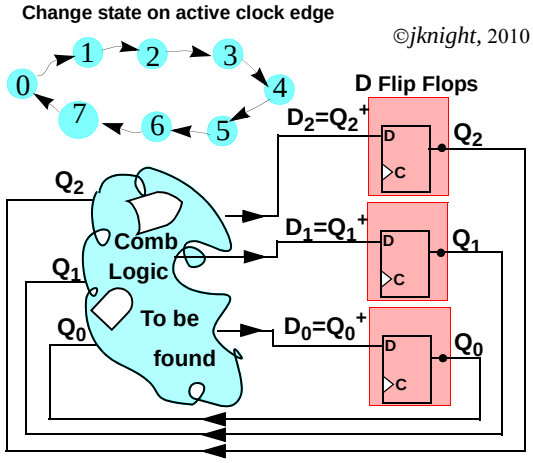
- In the design of **SAMPLECOUNT**, one needs 4 inputs, $Q_2\ Q_1\ Q_0$, and ClrSampCount. (Fig. 4.10). This will make the state table 16 lines long, and three 5 variable Kmaps. (See the complete state table in appendix (Fig. 4.27). This is not necessary. .

FIGURE 4.10

State $Q_2\ Q_1\ Q_0$	Next State C_L=0 $Q_2^+Q_1^+Q_0^+$	inputs $C_L D_2\ D_1\ D_0$	Next State C_L=1 $Q_2^+Q_1^+Q_0^+$	inputs $C_L D_2\ D_1\ D_0$
0 1 0	0 1 1	0 0 1 1	0 0 0	1 0 0 0
0 1 1	1 0 0	0 1 0 0	0 0 0	1 0 0 0
8 states	etc	etc	etc	etc

- When one input has such a simple relation to the output, it can often be added very simply at the end, and save half the work with a little thought. You may may do it either way as you like.

FIGURE 4.11 Design of a 3-bit binary counter using D flip flops. The C_L inputs are temporarily ignored here. They are added later to make a synchronous clear.



Steps in Designing a Finite-State Machine

1. Understand the problem (Usually hard, easier here).
2. Draw a state graph
3. Do state assignment (fill in bits for state names).
   Here that means substituting 010 for 2, 011 for 3 etc.
4. Construct a state table showing the next state.
5. Find the ff inputs to change state –> next state.
6. Put the state table in K-map order
7. Draw the K-maps from the state-table input columns..
8. Loop the K-maps to get the best equations.
   Remember to share gates if it is economical to do so.

Counter state table Using D flip-flops

Count	State $Q_2\ Q_1\ Q_0$	Next State $Q_2^+Q_1^+Q_0^+$	D inputs $D_2\ D_1\ D_0$
0	0 0 0	0 0 1	0 0 1
1	0 0 1	0 1 0	0 1 0
2	0 1 0	0 1 1	0 1 1
3	0 1 1		0
4	1 0 0		1
5	1 0 1		0
6	1 1 0	1 1 1	1 1 1
7	1 1 1	0 0 0	0 0 0

Explanation

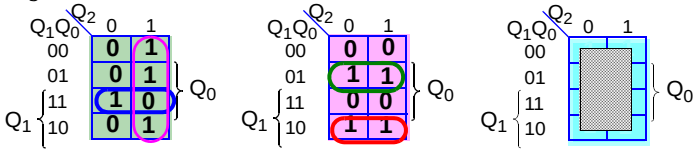
This counter counts:
0->1->2->3->4->5->6->7->0->1 . . .
which is 8 states.

FIGURE 4.12 The counter state table arranged in K-map order.

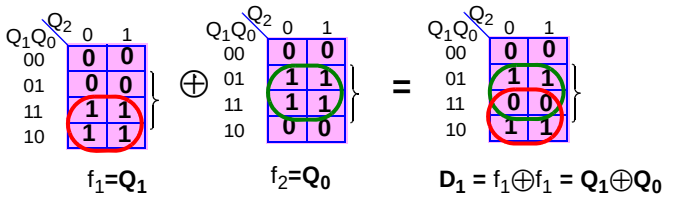
Counter state table In K-map order

©jknight 2011

Count	State $Q_2\ Q_1\ Q_0$	Next State $Q_2^+Q_1^+Q_0^+$	D inputs $D_2\ D_1\ D_0$
0	0 0 0	0 0 1	0 0 1
1	0 0 1	0 1 0	0 1 0
3	0 1 1	1 0 0	1 0 0
2	0 1 0	0 1 1	0 1 1
4	1 0 0	1 0 1	1 0 1
5	1 0 1	1 1 0	1 1 0
7	1 1 1	0 0 0	0 0 0
6	1 1 0	1 1 1	1 1 1



The equations for D1 is generated below with extended K-maps, which use 1⊕1=0



Clearing Counters Synchronously

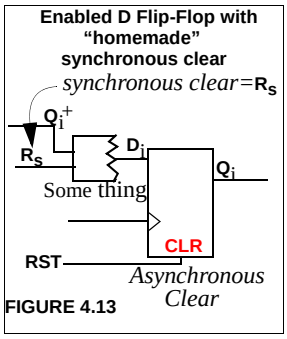


FIGURE 4.13

Don't even think about using the **a**synchronous CLR input on the flip flops to clear the counter. This is a NONO! The asynchronous CLR responds to very fast glitches (hazards). Such glitches are common and you should have seen them in your simulations. The asynchronous CLR will respond to these glitches and may clear the flip flop at times the designer did not expect.

Never use the asynchronous CLR for anything but start-up or recovery reset.

To clear SampCount either:

- use the state table Fig. 4.27 and three 5 variable K-maps,

or

- put a gate on the D_i input of each flip flop (Fig. 4.13) to give the correct Q_i after the active clock edge.
  This gate must give the normal D_i input, that is Q_i^+, when ClrSampCount is "0".
  It must make D_i a value which will make $Q_i^+ = 0$, when ClrSampCount is "1"
  Remember. A signal performs the action given by its name when it is TRUE.
  In this case, it clears when ClrSampCount =1.

Q_i^+ \ R_s	0	1
0	**0**	**0**
1	**1**	**0**

(B) The Enabled D Flip Flop

They act just like a plain D-flip flop if En =1
If En =0, Q just holds its old value.
The Enable pin is labeled:
$\mathbf{C_E}$ by Xilinx software, or **En** by the IEEE standard**.**

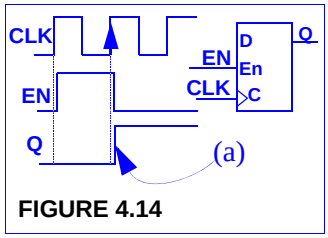


FIGURE 4.14

The formula for how the output changes is

$$Q^+ = En \cdot D + \overline{En} \cdot Q \cdot \quad \{Q^+ \text{ is just after the clock edge, Q is just before}\}$$

En does not clock the flip flop, but it might appear to. (See Fig. 4.14)

Note the exact timing relationship between a one clock-cycle long En (enable) and the change in Q. The output Q changes after the clock edge at the end of the enable pulse (point (a)), slightly after the rising clock edge.

Counter Design with Enabled D Flip Flops

- Use the same state table as for D flip flops, temporarily ignoring the En input.
- In the circuit connect an addional signal to the En inputs.
  Make the En inputs, En =1 if one wants the flipflop to change after the next active clock edge.
  Make En = 0 to ignore the next active clock edge.

In the **BitCount** circuit the flip-flops are restricted to wait eight clock cycles before changing, i.e. they are controlled by GRAB9. One could include GRAB9 as a 5th variable in the Karnaugh maps, but it is much easier to use an En flip-flop, and use GRAB9 as an enable signal. Then the counter just sits until enabled.

BitCount Design with Enabled D Flip Flops

The 0-to-9 counter, **BitCount**, is easier to design using enabled D flip-flop. You will need four 4-variable K-maps, instead of four 5-variable K-maps, and the maps are almost half don't cares.

FIGURE 4.15 Circuit using enabled D flip flops

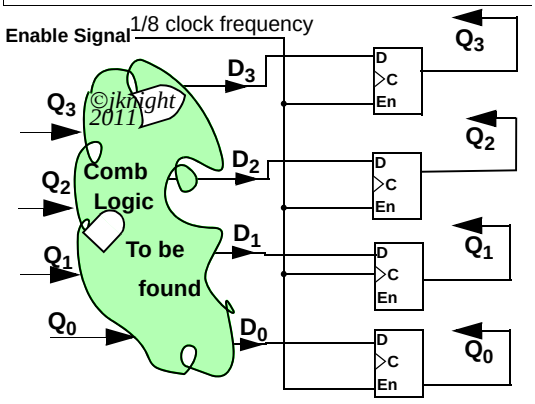
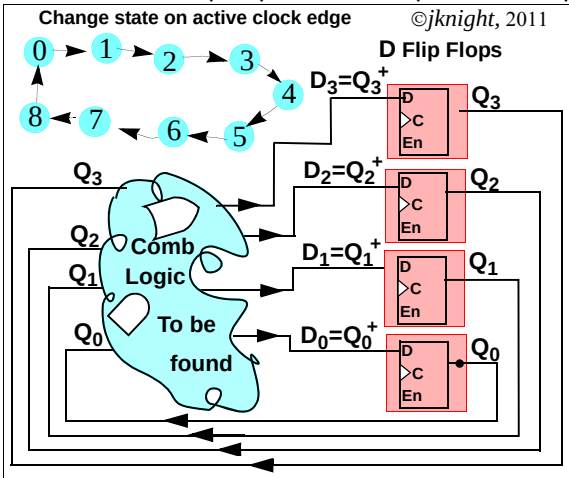


FIGURE 4.16 Design of a 4-bit 0-to-8 binary counter using D flip flops. The En inputs are temporarily ignored.



Steps in Designing a Finite-State Machine

1. Understand the problem (Usually hard, easier here).
2. Draw a state graph
3. Do state assignment (fill in bits for state names).
4. Construct a state table showing the next state.
5. Find the ff inputs to change state –> next state (D inputs)
6. Put the state table in K-map order
7. Draw the K-maps from the state-table input columns..
8. Loop the K-maps to get the best equations.
   Remember to share gates if it is economical to do so.

Explanation

The first counter, counts:
0->1->2->3->4->5->6->7->0->1 . .
which is 8 states.

The 2nd counter has to count 9 states,
i.e. 0 to 8. That requires another flip-flop

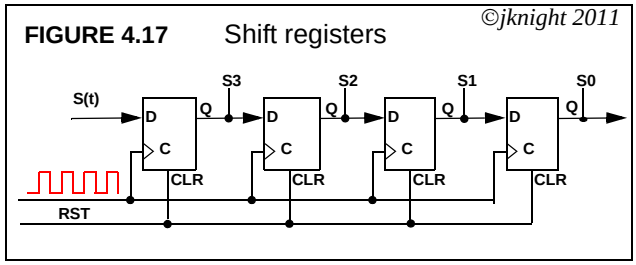
Counter state table Using D flip-flops

Count	State $Q_3Q_2Q_1Q_0$	Next State $Q_3^+Q_2^+Q_1^+Q_0^+$	D inputs $D_2D_2D_1D_0$
0	0 0 0 0	0 0 0 1	0 0 0 1
1	0 0 0 1		
2	0 0 1 0		
3	0 0 1 1		
4	0 1 0 0		
5	0 1 0 1		
	0 1 ? ?		
	0 1 ? ?		
8			
9 etc	16 states		

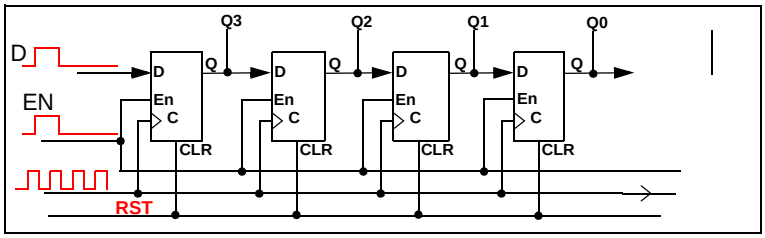
Shift Registers

The **SER2PAR** and **LAST4SAMP** blocks are usually constructed using shift registers.

The upper shift register takes a sample of s(t) every active clock edge and stores it in the leftmost flip flop. On the next edge it shifts the previous sample right and stores a new sample. After four clock cycles the last four samples are stored, the oldest on the right.



The lower shift register uses enabled flip flops so it will only shift when there is an enable pulse. It appears that one could get same result by sending EN into the clock input; however this is a NO NO! This creates a race between the D input and the EN signal pretending to be a clock, and the result will depend on which signal rises first. The details will be taught next year. For those with an unquenchable thirst for knowledge, see the Appendix. For this course one need only remember this rule:



Never send any signal but CLOCK into the C input of flip flops.

The COUNTCLEAR block

The counter, **SampCount**, has a clear input, CLRSAMPCOUNT which clears the count to zero when SIGIN is idle. CLRSAMPCOUNT is removed by the GOTSTART signal after 4 samples indicate a valid **START.**

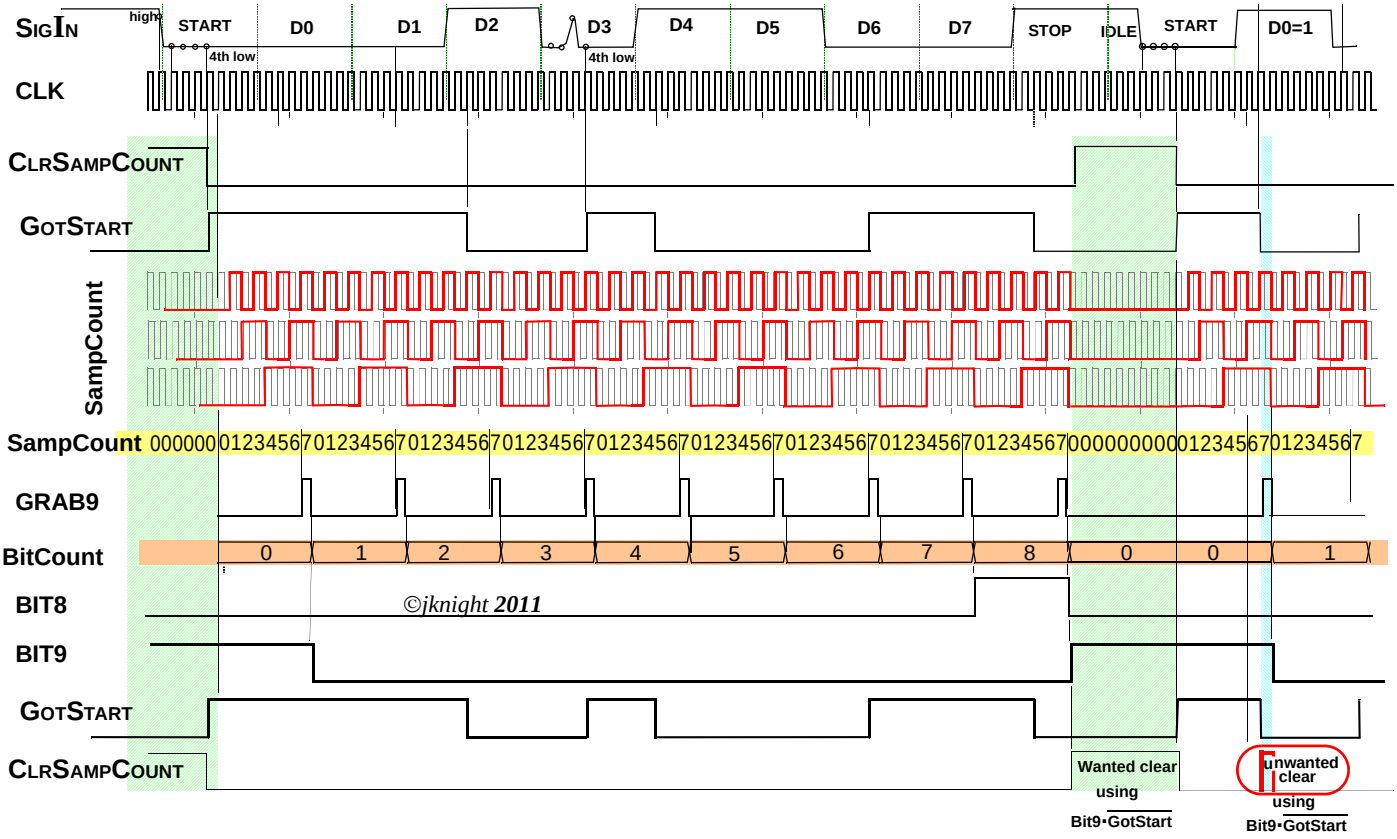
You should design a circuit to generates **CLRSAMPCOUNT** using other signals from Fig. 4.18.
We hope you can see that BIT9 will start the clear signal at the right time, and that GOTSTART can be used to remove the clear signal at the right time.
Follow a vertical slice of the timing diagram where **CLRSAMP**Count is active, and derive the formula.

However **CLRSAMPCOUNT** may get an unwanted pulse if D0 is a "1", as on the top far-right of Fig. 4.18. You need to find another signal, call it FIX, that will lower **CLRSAMPCOUNT** after **SampCount** has started counting. This signal must keep **CLRSAMPCOUNT** low until BIT9 goes low. Then BIT9 can keep **CLRSAMPCOUNT** low**.**
Follow the vertical slice and look for this FIX a signal. This signal must not mess up **CLRSAMPCOUNT** at other times.[4]
Too complicated? Temporarily ground the FIX signal and come back later. Your circuit will work 50% with no FIX.

4. Hint: Draw a graph of the bit patterns of the Q_2, Q_2, and Q_2, bits in **SampCount**. Also note where the FIX signal originates on the schematic

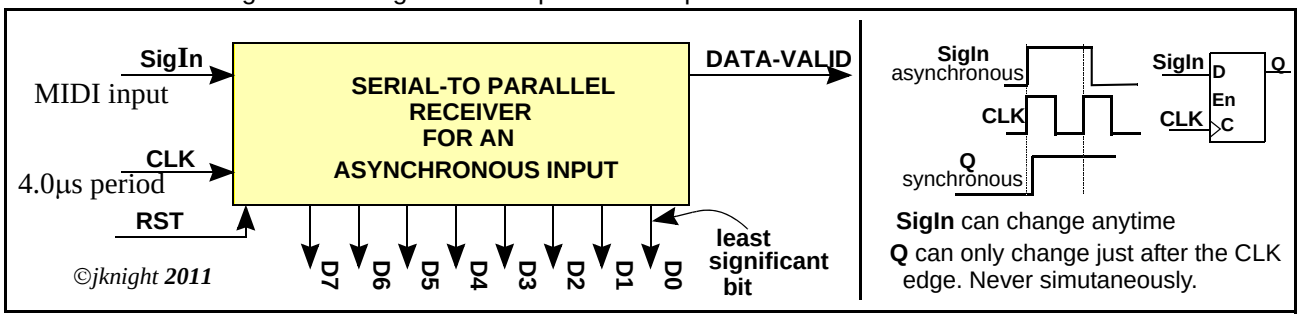
.

FIGURE 4.18 Signals needed to derive the circuitry to control the counters



The High Level Block Diagram and I/O Signals

FIGURE 4.19 Diagram showing receiver inputs and outputs



Signals which are restricted so they cannot change at the same time as the active clock edge are called *synchronous*. If they might be able to change on that clock edge, they are called *asynchronous*.

The receiver input is called *asynchronous* because a bit at the MIDI input (**SigIn**) might change on the clk edge. Thus each new bit may start at a random or *asynchronous* time with respect to the CLK.

The receiver has the following input and output signals:

SigIn **S**ignal **I**nput.....The received serial MIDI signal

CLK This is supplied externally and has a period of 4.0 µs, 8 times the signal bit speed. **CLK** is **not** synchronized with **SigIn**. The rising/falling edges of **CLK** are at arbitrary times with respect to the rising/falling edges of **SigIn**.
The **CLK** is sacred. It must go to the clock pin of all counters and flip-flops. It must not go through gates or be delayed in any way. **All flip-flops must clock at the same time.** If the clock reaches them at different times it is called clock *skew*, which causes many problems

RST Asynchronous reset is used for initializing the circuit on power-up. It is also used in factories for testing. It should go to the CLR connection[5] on every flip-flop. It **must not** be used for things like clearing a counter at the end of its count.

D0...D7 This is the received 8-bit parallel-output word. **D0** is the least significant bit. It follows the START bit in SIGIN. If you draw your shift registers shifting left to right, DO will end up in the rightmost flip flop.

DATAVALID **DATAVALID** = 1 after the STOP BIT has been received, and the parallel data is stable and may be read out. It stays 1 until new serial input data changes the parallel output lines.
DATAVALID = 0 whenever the data byte may be corrupted by new incoming data.

Prelab

Questions **Get as close to 7.14 as you can for the first lab session.**

7.1 Call the four last samples of SIGIN, S_3, S_2, S_1, S_0, with S_0 being the earliest sample and S_3 the latest. This order is the natural order in a shift register with the input on the left, → S_3 → S_2 → S_1 → S_0 but the reverse of the order on timing diagrams like Fig. 4.20.
Write the equation for a circuit that finds the majority of the last 3 samples, S_3, S_2, S_1. See FIGURE 4.4

7.2 Four samples of the start bit must be either 0000, 0010, 0100, or 0001 to turn on GOTSTART[6]. Make a Karnaugh map for the FindStart circuit, and from it write an equation for GOTSTART in terms of S_3, S_2, S_1, S_0.

7.3 Assuming there is no noise, after how many zero samples will GOTSTART go high? Explain.

7.4 At the end of the start bit with no noise, assume D0=1. How many samples of D0 =1 are needed to make GOTSTART turn off? Explain.

7.5 Plot GOTSTART, for all five midi bits, on the timing diagram below.

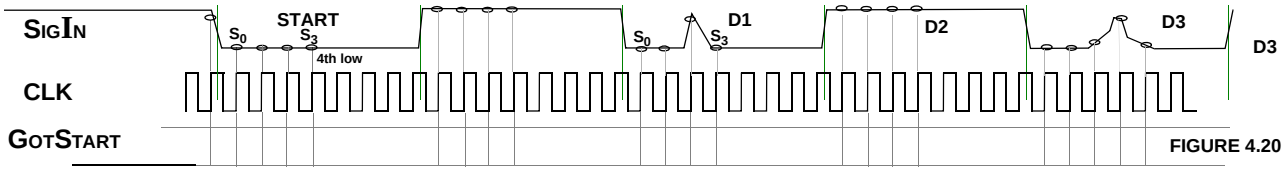


FIGURE 4.20

7.6 Design the LAST4SAMPLES circuit. Be careful to make S_0 the earliest sample and S_3 the latest one

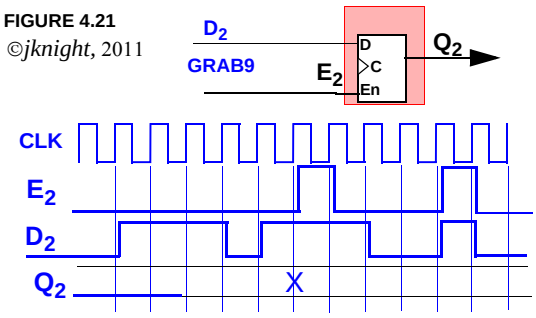
7.7 Omitting the FIX signal, draw the **SampCount** circuit using D flip flops. If you don't know how, reread the lab sheets.

5. Xilinx library flip-flops use CLR for asynchronous reset and R for synchronous reset.
6. This has changed from 2010, and the TAs will be watching.

7.8 Study and design the gates that can make SampCount = 0 without using the asynchronous CLR..

7.9 Complete the timing diagram on the right, showing exactly how Q_2 responds to the inputs. It is very important to show it changing just after, not on top of, the proper clock edge. The X shows one spot where it does not change.

FIGURE 4.21
©*jknight*, 2011



7.10 Look at the figure below. Recall that **BitCount** counts 0,1,2,3,4,5,6,7,8,0,. . . and that the BIT9 signal is high for the 9th count (the STOP bit) when **BitCount** holds 0. On the timing diagram below, plot when **BitCount** changes with respect to the GRAB9. This also determines exactly where BIT9 goes low. Draw hexagons 0 1 2 in the **BitCount** area of Fig. 4.22 below to show exactly where the transitions occur. ***Do not draw sloppy transitions*** 0 1 .

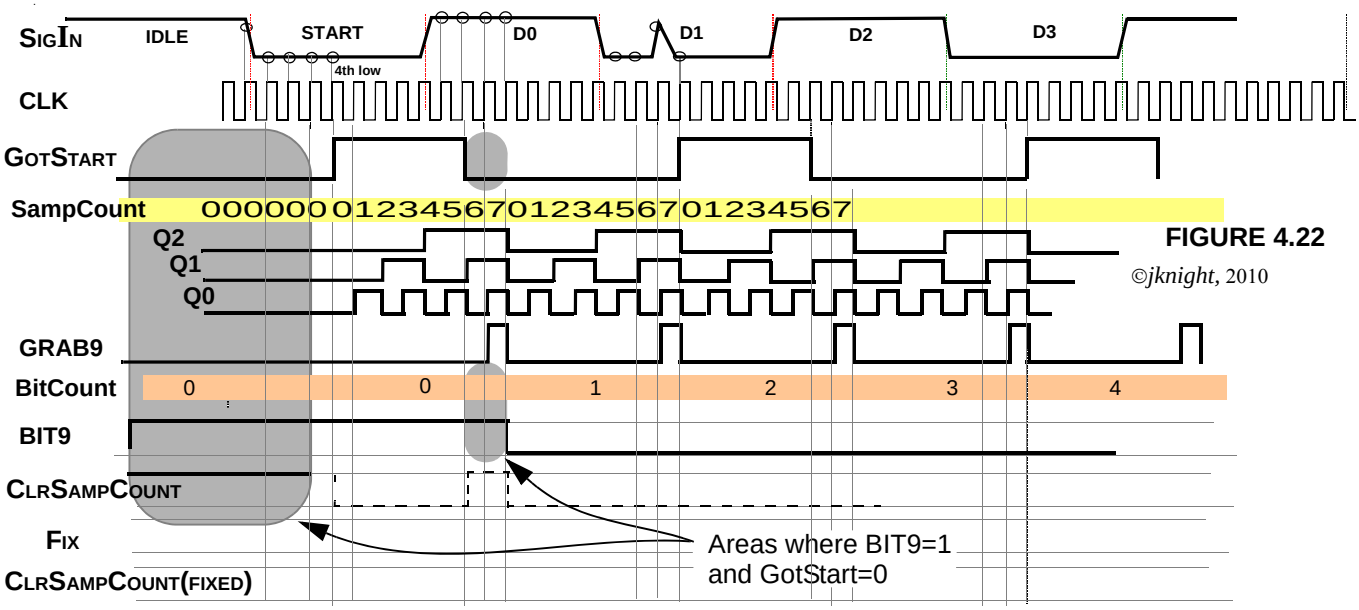


FIGURE 4.22
©*jknight*, 2010

7.11 Add a gate to **SampCount** to give out the GRAB9 pulses when the count is 7.

7.12 Draw the state graph for the **BitCount** circuit.

7.13 Design the circuits Parg. 7.24 to Parg. 7.27

We suggest stopping for the prelab for the first week.

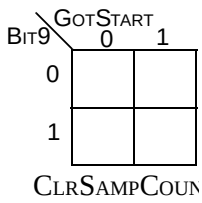
7.14 Make a state table for the **BitCount** circuit. We suggest you order the variables $Q_3Q_2Q_1Q_0$

7.15 Following the example of Fig. 4.11, add a column to the state table for the four D_n inputs needed to give the desired next state.

7.16 Make Karnaugh maps to calculate D_3, D_2, D_1 and D_0. To go with the variable order above you might make the map coordinates $Q_1Q_0\backslash Q_3Q_2$. See Fig. 4.12, only here you will have 4 variables instead of 3.

7.17 Obtain the equations for D_3, D_2, D_1 and D_0 from the maps. Extended maps using XORs are useful here.

7.18 Draw the circuit and add a gate to each enable input so the enables can only get through, and thus the counter can only count, when GRAB9=1.



7.19 Add gates to **BitCount** to give the BIT8 and BIT9 outputs.

Do this in the second week prelab. It is hard!

7.20 Fill in the K-map and design a circuit using BIT9 and GotStart to generate ClrSamp-Count. Add ClrSampCount to the timing diagram, Fig. 4.22 above, making it high at two different times.

7.21 Look at the "possible unwanted clear" pulse in Fig. 4.18. and hopefully in Fig. 4.22. It comes when D1=1, before BIT9 goes low. Carefully highlight the value(s) **SampCount** has during the unwanted clear. List the values here _____.

7.22 Determine a Fix signal that can be generated inside **SampCount** and can be used to hold ClrSampCount low until BIT9 can take over. Check it doesn't clear **SampCount** when it shouldn't.
Draw the circuit to add to **SampCount**.
Plot Fix on Fig. 4.18.
Draw the new ClrSampCount on Fig. 4.18.

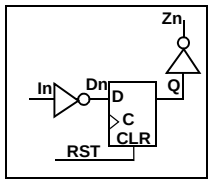
7.23 Design the circuits in Parg. 7.28 to Parg. 7.30

For The Prelab, Implement This Design

Modular Design

The interface can be divided into modules such as shown in Fig. 4.7 and Fig. 4.23. If you use these modules with the same I/O signals, you can use the module test fixtures supplied. We strongly encourage you to enter **and test** each part individually.

7.24 Last4Samp block



Design a circuit to take a sample of **SigIn** on every active clock edge, and store the value of the three previous samples with names S3, S2, S1, S0(oldest). It does this continuously.

On start up the simulator will send out an RST signal which will clear all the flip-flops. Thus FindStart will think it has found a start bit and keep going. This will give a mess at the start of the simulation. If one puts an inverter before and after every flip-flop, the flip-flop will externally appear just the same, except the RST signal will not be inverted, and the output inverter will make it appear that the flip-flop was set to one.

7.25 Majority Block

Design a circuit whose output is the value of the majority of the last 3 samples.

7.26 FindStart Block

Design a circuit whose input is the last 4 samples from the **SigIn** lead. The output **GotStart** is high if the 3 of the 4 last samples of **SigIn** are zero.

7.27 SampCount Block

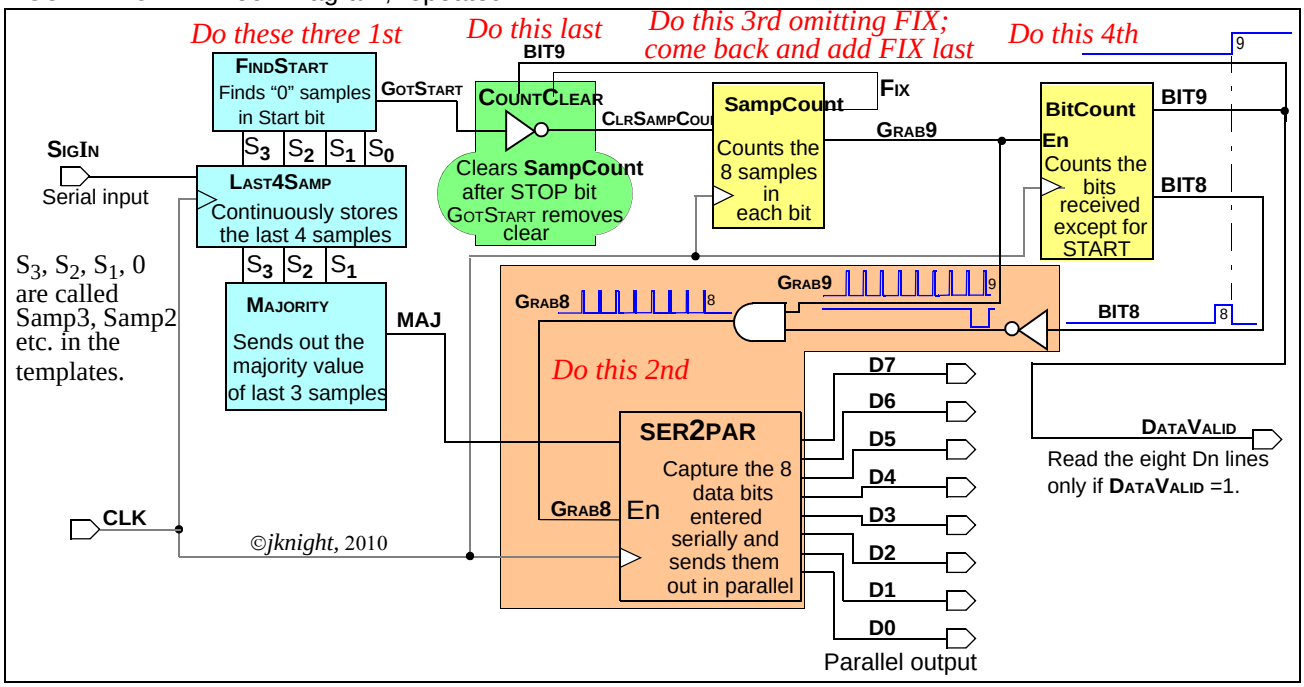
Design a counter which counts from 0-to-7 and then repeats, one count per clock cycle. It should have an input **ClrSampCount** which, when high, sends the count to 0 on the next clock edge. On the eighth count (a count of 7) it should send out a **GRAB** pulse one clock cycle long. Another output may be needed later. Construct the T flip flops, using D (or enabled D) flip flops and an inverter. Alternately most flipflops have a $\overline{Q}$ output which can be fed back without the inverter.

Do the counter by temporarily grounding the **Fix** output. Add that last when you design CountClear.

7.28 Ser2Par Block

Design this as per the description in the box. It only captures a bit when GRAB8 is high.

FIGURE 4.23 Block Diagram, repeated.

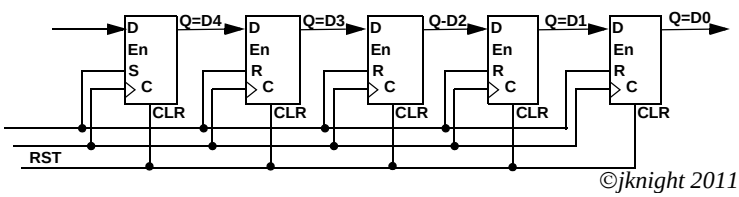


7.29 BitCount Block

This circuit counts the data bits and the stop bit, a total of 9 bits. This means 4 flip flops are needed. The input is the GRAB pulse, and it has two outputs each 8 clock cycles long. One is high during the eighth count and one during then ninth (counts of 7 and 8 respectively).

Alternate BitCount Block (An innovation, the test fixture won't support this. See Prof Knight)

One does not need to design a separate counter. The shift register in the SER2PAR block can keep track of the number of bits coming in. Use an extra D flip-flop and set the first flip flop to "1" and the others to "0" initially and sometime during the start bit. For example when BIT9=1 and SAMPCOUNT= 6. Implementing this will require modifying *midi_top* and the test fixture.



7.30 CountClear Block

Design a block to generate the signal CLRSAMPCOUNT, which clears **SampCount** during the later part of STOP bit through to the first half of the START bit. From Fig. 4.18, generating the "wanted clear" pulse is easy. Avoiding the "unwanted clear" requires studying the figure carefully.

Variations (Innovations)

The criteria for GOTSTART are somewhat arbitrary. One could think of several alternatives. You should be able to argue that your changes are an improvement.

In The Lab.

After you have designed the circuit, you will enter it graphically into the computer as schematic diagrams. After the circuit is entered, you will simulate it to check its operation.

While the blocks are individually simple, connecting the whole circuit at once will make debugging difficult. Implement the above blocks one at a time. *Test fixture* files are available to help you simulate the individual blocks. There is also a test file to simulate the whole MIDI interface.

By the end of the first lab period you should aim to have entered at least-three of the blocks. Simulating and debugging will be slow. Do not leave too much for the second period. Try starting with the MAJORITY block.

```
// Start bit
     SigIn =0;
        startguide=1;
#32;    startguide=0;
// 8 Data bits
     SigIn =0;
#32;
     SigIn =1;
#32;
     SigIn =1;
#32;
     SigIn =0;
#32;
     SigIn =0;
#32;
     SigIn =0;
#32;
     SigIn =1;
#32;
     SigIn =1;
#32;
// Stop Bit
     SigIn =1;
```

FIGURE 4.24
Part of the test file showing **startguide**

Displaying Waveforms

The test fixture file can do a lot to help you check your circuit. There are three waveforms, not in your circuit, which it will displayed.

StudNumb displays your student number for comparison with the midi data. The data in the first MIDI word is the last 6 bit of your student number in binary.

startguide displays a pulse during the start bit to make it easy to find.

serialin displays the serial input data as an integer.

data displays the parallel output data as an integer, with **D0** as the least significant (leftmost) bit.

timetick shows the start and end of each data bit.

samp3 shows the input delayed by the shift register. (it may be hard to display)

The Test for the Complete Circuit

After you have run the simulation file on the full MIDI schematic, you should run it with your own data.

The test files are written in *Verilog*, a language for describing and simulating digital circuits. You will use Verilog next year in ELEC3500.

All Verilog instructions end in semicolons. If there is no "**;**" the instruction is continued on the next line. Comments start with //, or are enclosed in /* . . . */.

In the midi file, the 10 values of **SigIn** which make up the MIDI word are written out serially as in Fig. 4.24 The values of **SigIn** are set. The #32 indicates a 32 μs delay before the next change. Thus both boxes on the right give the pulse shown below them. Writing the #32 between the lines makes it clearer where the delay is, but takes more space.

```
    SigIn = 0;
#32;
    SigIn = 1;
#32;
    SigIn = 0;
```

```
    SigIn = 0;
#32 SigIn = 1;
#32 SigIn =0
```

A typical data word, with the start and stop bits underlined, is 0 0 1 1 0 0 0 1 1 1, see Fig. 4.24 for the Verilog code.

Your student number, or your partners, is used as data in the midi *test_top.tf* test fixture file.

```
    integer StudNumb;
// ENTER YOUR STUDENT NUMBER HERE.
//-------------------------------
    initial StudNumb = 312153;  // skip the initial 100
//-------------------------------
```

SigIn

32 64

The Verilog program will translate it into binary, and use the 8 least significant bits as test data.

In the simulation the incoming midi bits are exactly 32μs long and the receiver clock is exactly 4μs. Normally the transmitters idea of a 32 μs is not the same as the receivers. One might be a little faster. If the receiver clock period were say 3.7 μs, the SAMPLE pulse would not be in the right place to sample the final bit.

The Test Fixture Log

Read the simulation log, particularly at the end. It summarizes the date that went into your design, and what came out. You will need a printout of it for your report.

Noisy Data Tests

The test file *test_top_noisy.tf* which has the student number word (noise free) followed by three samples of noisy data. You can tell how well your circuit rejects the noise.

The signal starts with a good midi word, and changes it by XORing each sample with a random bit string. In 2011 the random bit string was set to a 2% probability of being 1 and thus flipping the sample. With 2% noise your circuit will probably not allow errors to get through. With 15% or more, it probably will.

With high noise the midi bits may be so distorted you cannot tell where they start. There is a variable *timetick* which shows the start and end of each data bit. Also the input, before the noise is added, is called *x*.

If you have noise related errors, you should be able to take the printout, showing one of the midi words that gives an error, and explain in why the circuit failed in your report.

Your Report

The marker will not believe that you intuitively know how the Midi interface works. You must explain, at a high level, the function of the complete interface and the individual blocks. For example, could you not give a better explanation of the ser2par block than is in the lab sheet.

Your design and design methods should be included. Most people's designs will follow the prelab outline. Brag about any changes you made, particularly if you think they are an improvement.

Your report should be coherent, and not jump over design derivations. Put in your schematics and your final simulation. The circuits should agree with those you actually used.

Testing and implementation are important. You should neatly highlight and write comments on the waveforms. It is very hard for someone to make sense of simulation waveforms if they do not know what is being simulated. You must explain what the waveforms mean.

Common reasons for losing marks in MIDI reports

- The author of each section is not identified at the top of each section.
- The report uses two much unexplained jargon. For a reference level, assume your report will be used a crutch for a student doing the lab next year. He/she should be able to understand it. That student might just get a question about serial-to-parallel conversion on the final exam
- The paragraphs or sentences do not make sense. Example from a recent report, *"Since **SigIn** is asynchronous, the **SigIn** data is input when the data is low."*
- Not describing the design of the modules.
- Not describing the final simulation. This is the test that your circuit works. State how you know your circuit works. Be sure to write short neat comments on the waveform printout.
- We repeat! **Annotate your waveforms in detail**.

- Not including the last part of the simulation log, showing input and output data.
- Leaving out sections.
- Copying the sections verbatim from the laboratory write-up.
- Copying pictures from the lab sheet and not acknowledging each of them.
- Not attaching the prelab as an appendix.
- Not placing your names on the schematics.
- Copying a last years report. The changes give this away easily. Plagiarism is a major offense.
- Passing in simulations with a student number that is not yours, your partner's, or 312153.

Appendix

Why one should not use any old signal as a clock.

Suppose one decided they would not use enabled flip flops in SER2PAR because they would apply GRAB directly to the CLK inputs.

Clock signals are carefully designed so that they arrive at every clock input at almost exactly the same time. They are not delayed by going through gates.

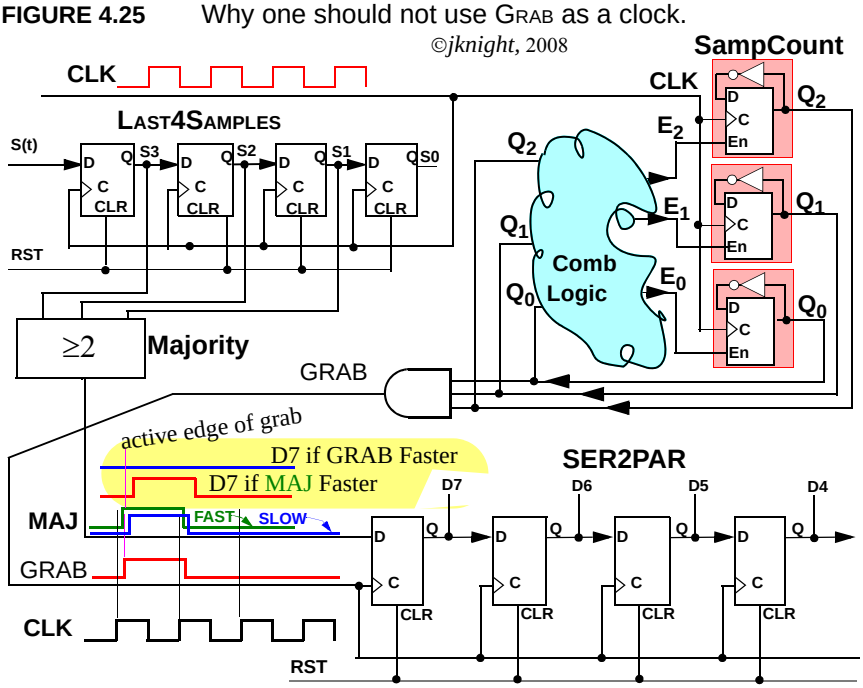
In Fig. 4.25, both the LAST4SAMPLES circuit and the **SampCount** circuit get CLK at the same time. However the GRAB signal and the MAJORITY signal are delayed by going through flip flops and gates. Which signal changes first?

If MAJ is high before the rising edge of GRAB, it will be captured properly. However if MAJ is slower, then GRAB will clock the flip flop while MAJ is still low, and will miss it entirely. This will be taught in more detail in the next course. For now, follow the rule:

Never send any signal but CLOCK into the C input of flip flops.

Another, simpler reason for not sending low-class signals into the C inputs is that any glitch on GRAB will cause the flip flop to capture an unwanted signal from MAJ.

FIGURE 4.25 Why one should not use GRAB as a clock.



SampleCount Using 5-Variable Maps

FIGURE 4.26 State graph for **SampCount**.

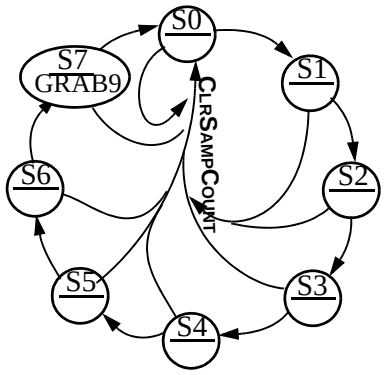


FIGURE 4.27 The SampleCount state table using a C_L (CLRSAMPCOUNT) signal.

Counter state table In K-map order ©*jknight 2011*

Count	**State** $Q_2Q_1Q_0$	**Next State** C_L=0 $Q_2^+Q_1^+Q_0^+$	inputs $C_LD_2D_1D_0$	**Next State** C_L=1	inputs $C_LD_2D_1D_0$
0	0 0 0	0 0 1	0 0 0 1	0 0 0	1 0
1	0 0 1	0 1 0	0 0	0 0 0	1 0
3	0 1 1	1 0 0	0 1	0 0 0	1 0
2	0 1 0	0 1 1	0 0	0 0 0	1 0
4	1 0 0	1 0 1	0 1	0 0 0	1 0
5	1 0 1	1 1 0	0 1	0 0 0	1 0
	1	0 0 0	0 0	0 0 0	1 0
	1	1 1 1	0 1	0 0 0	1 0 0 0

Did you think we would give you the whole table so you could paste it into your lab report without your doing any thinking?

This will require three 5-variable maps. UGH!