Clock Generator 82C84A

The Intel Clock Generator chip 82C84A includes three functional blocks:

Clock Generation circuitry
Ready Synchronization Circuitry
Reset Synchronization Circuitry

1. Clock requirements of the 8088/8086 processor:

- \( T_{CLCH} = 118.33 \text{ns} \text{ min.} \)
- \( T_{CHCL} = 68.66 \text{ns} \text{ min.} \)
- \( T_{CLCL} = 200 \text{ ns min.; 500 ns max} \)
- Frequency = 5MHz max; 2MHz min.

Note: 118.33 : 68.66 is about 2:1

2. 8086 processor vs. 80C86 processor: What does that “C” mean?

Why there is a minimum frequency (2 MHz) requirement for the 8086 processor?

8086 processor designer may be performing sub-clock operations (example: ALE, /DEN, etc.) So, duty cycle of 33% for the clock may be the best for the processor. 33% Duty Cycle is not a requirement. We can satisfy all three requirements, \( T_{CLCH}\text{min} \), \( T_{CHCL}\text{min} \), and \( T_{CLCL}\text{min} \) by choosing 50% duty cycle clock of clock period 400ns!

2. 8086 processor vs. 80C86 processor: What does that “C” mean?

Why there is a minimum frequency (2 MHz) requirement for the 8086 processor?

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>8086</th>
<th>8086-1</th>
<th>8086-2</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min.</td>
<td>Max.</td>
<td>Min.</td>
<td>Max.</td>
</tr>
<tr>
<td>TCLCL</td>
<td>CLK Cycle Period</td>
<td>200</td>
<td>500</td>
<td>100</td>
<td>500</td>
</tr>
<tr>
<td>TCLCH</td>
<td>CLK Low Time</td>
<td>118</td>
<td>53</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>TCHCL</td>
<td>CLK High Time</td>
<td>69</td>
<td>39</td>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>80C86</th>
<th>80C86-2</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min.</td>
<td>Max.</td>
<td>Min.</td>
</tr>
<tr>
<td>TCLCL</td>
<td>CLK Cycle Period</td>
<td>200</td>
<td>D.C.</td>
<td>125</td>
</tr>
</tbody>
</table>

80C86 CHMOS uP **High performance CMOS**

© Copyright 2007 Gandhi Puvvada
CMOS is known as static logic to differentiate from the dynamic logic. In the case of dynamic sequential logic, data sometimes stays on the capacitors. So if the frequency is too low, the data will be lost due to leakage on the capacitors. More in EE477A/577A.

3. How to generate a 33% duty clock

The piezoelectric crystal oscillator produces 50% duty clock. So to produce 33% duty cycle 4.77 MHz clock (as in the case of PC XT), they started with 14.31 MHz crystal, produced 14.31 MHz 50% D.C. clock and then passed it through a divide-by-3 counter.

4. Asynchronous Inputs:

Examples of asynchronous inputs:
Inputs from human, inputs from other subsystems working on a different independent clock

If the asynchronous input changes too late into the clock, the system can go into wrong states!
5. Synchronization of asynchronous signals by using a synchronizing FF:

Simple intuitive answer: Well if it is the same edge we create a RACE condition. So it should be opposite edge. That is in fact a naive answer!
A more thoughtful answer is that we use the same edge. Because of the finite propagation delay of the synchronizing FF (which is made sure to be greater than the hold time requirements of the receiving system/FFs), the RACE condition is carefully overcome in EVERY digital design.

A 16-bit shift register needs only one clock to shift all 16 bits by 1-bit right. It does not need 16 clocks or 16 half-clocks! It works on one single edge of a clock because of the finite propagation delay of the left side FF exceeding the hold time requirement of the right-side FF.

More on timing design in EE560.

6. The clock for synchronization shall be the __________________________ (sending/receiving) system's clock. Example from EE201L homework:

What if the above relation is not satisfied? How do you fix it? __________________________
Design #1 experiences synchronization problems as the DO-IT signal is asynchronous to the \( \text{System-33/ System-44} \) and also the DONE signal is asynchronous to the \( \text{System-33/ System-44} \). Between Design #2 and #3, \( \text{_________} \) is right and \( \text{_________} \) is wrong.

7. Metastability

Even the sampling and hold flip-flop may not be able to help, if the asynchronous input actually changes right at the time of sampling. The output of the sampling flip-flop may go into metastability (a state which is neither a “1” nor a “0”).

To understand this, analyze what happens when you try to set a simple cross connected NAND-pair bistable, but do not hold the set input active long enough.

In the case of a master-slave D-FF, the master can go into a metastable state if the input changes during the setup time; and consequently, the slave goes into a metastable state.

8. What is meant by \textit{flip-flops hardened against metastability}?

It should be noted that even if a flip-flop goes into a metastable state, it is generally difficult for it to remain in such a state. It is like ..., even if you manage to make a knife stand on its edge for a second, it is likely to fall one way or other very soon. By \textit{making the edge of the knife sharper}, you reduce the probability of its standing on its edge for notable length of time. Similarly, by \textit{increasing the loop gain of the cross connected pair} of NAND gates/NOR gates making a bistable, any small disturbance (\textit{disturbance to the precarious metastable state}) will get amplified and push the system to go into one of the two stable states of the bistable. Such flip-flops, which are made to very quickly come out of the metastable state even if they very rarely went in are called \textit{flip-flops hardened against metastability}.

9. Double-synchronization helps to reduce the probability of failure due to metastability:

Increases MTBF (Mean Time Between failures)

Here we employ two Synchronizing flip-flops. Even if the first one goes into a metastable state, it is expected that it would come out of the metastable state by the time the second flip-flop tries to sample the output of the first flip-flop.

More in EE552/EE560: What do we do if we received a multi-bit asynchronous data? No, we do not use multiple synch. FFS.
10. Why does \texttt{~RESET} (asynchronous \texttt{RESET}) need to be synchronized to produce synchronous \texttt{RESET}?

Isn’t it true that if we are resetting the system anyway, we are aborting whatever we are doing and going into the INITIAL state. If so, does it matter if we abort synchronously or asynchronously?

It is not about \textit{when you go into reset}. It is about \textit{when you come out of reset}.

The \texttt{RESET} signal is usually generated using an R-C network. Hence it is an asynchronous signal. After the R-C time constant, if the reset signal becomes \textit{inactive} just before (or at) the significant edge of the clock, then some flip-slops in the state memory may be able to come out of the reset state and start honoring the next-state bits standing at their D-inputs, \textbf{while} some other flip-flops may still be in the reset state. This causes the system to go into wrong/illegal states.

Let us first explain the problem using the 5-state dish-washer state-machine of EE201L. Let us assume that the \texttt{~RESET} is asynchronous and becomes inactive just before (or at) the significant edge of the clock (positive edge in this problem). Also assume that the \texttt{START} happens to be true at that time. It is possible for the one-hot flip-flop \texttt{QDONE} to continue to remain in the reset state (\texttt{QDONE} = 1), while at the same time the \texttt{QADD\_WATER} flip-flop may come out of reset and go to 1 (\texttt{QADD\_WATER} = 1). So we have two FFs hot in the one-hot system!

Solution: \textit{synchronize} the asynchronous \texttt{~RESET} to produce a synchronous reset signal \texttt{~SYNC\_RESET}.

\begin{center}
\includegraphics[width=\textwidth]{diagram.png}
\end{center}
11. Why does RDY need to be synchronized?

It depends on whether RDY is synchronous or asynchronous.

Why do you expect that it would be asynchronous? Isn’t it true that the WSG (wait state generator) is clocked by the same clock as the processor? Yes and No!

If the WSG is clocked by the same clock as the processor, then the RDY is synchronous. Then we can choose to avoid going through the synchronizing FFs in the 82C84A and go directly to the READY input of the processor.

Or, alternatively, the RDY output of the WSG can be taken through one or two flip-flops in the 82C84A before going to the READY input of the processor. In that case, we simply treat the synchronizing FFs in the 82C84 as simple “delay” flip-flops and accordingly design our WSG.

If the WSG is clocked by a different clock (different from the processor clock), then the RDY is asynchronous. If it is asynchronous, then it needs to be synchronized (perhaps by going through one of the two FFs in the 82C84A).

It is very common to have multiple clock domains in a major system such as a PC. The board runs at one clock and the PCI bus runs at a different clock for example. So it is possible that the WSG is running at a clock different from the clock of the processor.

12. If the WSG is working on a different clock, then it shall be designed as a NORMALLY NOT READY SYSTEM (NNR). Why?

How about in the case of the “same clock” situation? _______ (NR / NNR)
13. When should an input such as READY, INTR, HOLD, etc. be valid and stable?

Based on the bus activity sensed on the pins of the processor (as seen on logic analyzer in lab #4), we can assume that the BIU of 8088/8086 follows the simplified State Diagram given on the side.

Then, in the case of READY, you (the board-level designer) would expect that you need to provide a VALID and STABLE READY input to the processor well-in advance (or reasonably in advance) for the NSL (Next State Logic) of the BIU to produce proper desired next state after T3 (as T4 or TW) and after TW (as T4 or TW).

Based on our synchronous system design experience in EE201L, we would expect that the READY should be valid and stable starting from Tsu (T setup time) before the state transition edge of the clock (negative-edge of the clock here) to Th (T hold time) after the edge.

So the following is our expectation for the timing requirements imposed on the READY input:

While the expectation is reasonable and justified, the actual timing requirements published in the data sheets are quite different. It may be because, the actual design may not be following pure synchronous design principles, may be using sub-clock operations, may be affecting circuitry other than the simplified state machine for the BIU that we assumed.
14. A Normally Ready (NR) System and a Normally Not Ready (NNR) System

Reproduced above are diagrams from the 8086/8088 user manual. Several things to notice are:

1. There are two systems, NR and NNR, with different timing requirements.

2. The setup time and hold time are specified sometimes with respect to the middle-edge (positive-edge) of a T-state rather than the significant (state-transition) edge namely the negative-edge.

3. In the case of NNR, the setup time requirement is strangely negative (-8 ns)!
14.1.A What is NR and what is NNR? Which is safe?

Let us take an example of a repair work in a chemical plant. There are 2 operators A and B and one assistant C. After detecting that the water pump in the field is not coming on (even after a low-water alarm from the water tank), operators A switches off power to the pump and sends B and C to the field to open the pump and fix it. They estimated that it would take 30 minutes to fix the pump. Assume that there were no cell phones (or walkie-talkies) in those days! After B and C fix the pump, A needs to switch-on power so that the operator B in the field can see if the pump is now starting.

Consider the following two understandings (protocols) among the operators.

**Normally Ready system**: If the fixing job does not seem to be finishing in 30 minutes, B should send the assistant C to the control room to tell A not to switch-on power. In the absence of any such communication, A can assume that the job is done in 30 minutes (Normally Ready in the expected time) and A may switch-on power at the end of 30 minutes.

**Normally Not Ready system**: Here A assumes that B and C are Normally Not Ready and will not switch-on power until there is a positive confirming message from B (through the messenger C) that he is done.

While both systems can work, which is safe? ________________________________

And why? __________________________________________________________________

14.1.B

Well, we said that the Normally Not Ready system is safe because we do not want B and C to get hurt if they get so involved in fixing the pump that they forget to tell A that they are not done. So we are taking into account the human forgetfulness and consequent safety aspects. But if we are considering designing a wait-state generator, which is the safe system? First consider the following definitions of a NR and a NNR system in the context of a WSG (Wait State Generator).

**A Normally Ready System**: The wait-state generator keeps the READY line by default active (i.e. normally \( \text{READY} = 1 \) during T1 and T2 for example) and will inactivate it (make \( \text{READY} = 0 \)) if needed (i.e. if the WSG determines that the current bus cycle is addressing a slow memory or I/O device needing one or more wait states) and will eventually activate the READY (\( \text{READY} = 1 \)). See fig. 6-83.

**A Normally Not Ready System**: The wait-state generator keeps the READY line by default inactive (i.e. normally \( \text{READY} = 0 \) during T1 and T2 for example) and will activate it (make \( \text{READY} = 1 \)) as needed (i.e. if the WSG determines that the current bus cycle is addressing a fast or slow memory or I/O device decides when to make the READY active). See fig. 6-84.

Safety? Preference? Well, unlike a human, who tends to be forgetful, an electronic system is not expected to be forgetful. It can malfunction and if it does malfunction, anyway we need to fix it.
But how about multiple clock-domains and synchronization across clock domains? What do you mean?

Well, what if the processor is working on a clock different from the wait state generator? A simple example of such a situation is in the context of a multi-processor system where each processor is working on its own clock and the global memory has its own clock.

Even in a single-processor system, in current day systems, the processor often works at a very high frequency (say, 3 GHz) compared to memory and I/O. Subsystems such as DRAM subsystem, EPROM subsystem, and PCI subsystem have their own clocks. Even if all these clocks are related to each other (perhaps derived from one primary clock source), we can not guarantee good phase relation between them because of varying delays on clock traces on PCB. So these clocks have to be treated as asynchronous to each other. So a signal originating from one subsystem is an asynchronous signal input to another subsystem.

OK, how would we judge if NR is better or NNR is better (or neither is good enough!) if the WSG is using a clock different from the processor’s clock.

Whenever, we convey information (such as READY active or inactive) from one subsystem to another subsystem working on a different clock, we need to go through the synchronizing flip-flops (sample and hold flip-flops). The synchronizing flip-flop may miss to capture the latest value if the latest value has arrived at or around receiving systems sampling.

You have heard the phrase, “err on the safe side”. It means that even if we are making an error, it should only be increasing safety rather than decreasing it. So a loss of a clock or two in synchronizing (in producing the synchronous READY output from the asynchronous RDY input) should not result in producing wrong results. It is fine if we end up adding a couple more wait states where we DO NOT need, because it only wastes a couple of clocks but still produces correct result. However, you should never end up missing to add the necessary wait states because then you will be reading/writing wrong data from/into memory.

Please note that the NNR system is safe as you started with saying that you are NOT READY. After sometime you say that you are ready. This new information may reach sometimes quickly and sometimes with an additional clock delay of the receiver’s clock. That means you will have either the exact number of wait states or sometimes one extra wait state (which is safe). You will never have one less wait state.

14.2 We notice that the setup is measured with respect to the middle edge (positive-edge) of TW (and T3 in the case of NNR system) where as we were expecting that it should be measured with respect to the ending-edge (negative-edge) of these T states.

True, but then we do not know the actual internal design of the 8086/8088 CPU. They may have some of the internal circuitry sensitive to positive edges, they may be doing gated-clock operations (covered in EE560), and so on. So we need to just go by the data sheet and meet all timing requirements stated there (collectively and individually).
14.3 In the case of NR system, setup time requirement was stated as -8 (negative 8, minus 8) with respect to the beginning of T3 (end of T2). How is it negative??!!
Well, let us go through the EE201L homework problem below.

So now you agree that even though the internal FF has +20 ns setup time, due to a 28 ns delay in the clock buffer, externally it looks like a FF with -8 ns (negative 8 ns) setup time!
i.e. The external D input can start 8 ns after the external clock and yet be 20 ns before the internal clock because the 28 ns delay incurred by the clock signal.
15. F/C and EFI inputs to 82C84A:

You see a 2-to-1 mux controlled by the select line F/C (Frequency/Crystal) to select between the External Frequency Input (EFI) and clock produced by the crystal oscillator.

In a multiprocessor system of (say) 2 processors, to keep the two processors to work of the same frequency, you employ two 82C84A chips (one in each processor subsystem), but you would use a crystal with one of the two and use EFI for the other as shown below. The common CSYNC (which is active initially for a short time) will start the two divide-by-3 counters in the same phase.

16. Double synchronization and the ASYNC input:

The system designer (board-level designer) has a choice of using two Flip-Flops or one Flip-Flop for synchronization by tying the ASYNC input to GND or VDD. Also note that they used a positive-edge triggered flip-flop for FF1 (flip-flop1). It means that they are allowing only one-half of a clock (to be precise one-third of a clock if we are using a 33% D.C. clock) for the FF1 to come out of metastable condition (if it ever goes into metastable condition). That is reasonable because the clocks at that time were very slow. A clock with 210 ns clock period will provide 70ns for the FF1 to settle. In some of the current high-speed designs, sometimes they do one of the following two:
(i) allow multiple clocks for the synchronizing flip-flops to settle by clocking them by a derived clock of say one-fourth frequency.
(ii) use three or four flip-flops to make sure that even if the first could not come out of the metastable condition at the end of a clock, hopefully the second, which consequently goes into a metastable condition, would come out of that condition in one clock. The expectation is that we shouldn’t be that unlucky to have a series of flip-flops to go into metastability and not come out of this condition in one clock!

17. To satisfy the timing requirements of the READY input on the CPU, we can choose the **severe most requirement** of the multiple requirements and satisfy the same.

Out of the three cases shown here, the severe most requirement comes from the normally ready system which requires that the READY input be stable latest by 8ns into T3. So if we make sure that we follow this requirement by using the synchronizing Flip-Flop FF2 in 82C84A (triggered by the neg. edge of CLK), then the READY input will be valid from the beginning of T3 (and Tw). And since FF2 updates the sample at every neg-edge, the output of FF2 is definitely stable at the positive edges. This explains how the 82C84A (with FF2 producing READY) satisfies the requirements of the READY input of the CPU.

![Diagram](image1)

![Diagram](image2)

![Diagram](image3)

**Figure 6-86. 82C84A with 8086/8088 Ready Timing**
Does the fig. 6-86 depict the timing requirements for the RDY input assuming that it is lead to the FF#2 directly or it is led to FF#1 and FF#2? How do you tell? How would you draw a timing requirements diagram for the other?

18. **Wait State Generator (WSG) design:**

18.1 Here, in our EE454L, most of the time, we are using the processor clock or some clock derived from the processor clock to control the WSG. So it is possible to bypass the RDY synchronization circuit in the 82C84A. Or if we go through the synchronization Flip-Flop in the 82C84A, we can accordingly produce an appropriate RDY (appropriately timed RDY).

18.2 While we can generally say that if we are planning to go through an intermediate stage/FF, we need to say whatever we would like to say a little early, we should be careful in deciding whether it should be early by one clock or half clock or one and a half clock or zero clocks! Do not make a hasty decision.

18.3 Basic design idea we use here is as follows: The WSG is a state machine. It gets initialized at the beginning of a bus cycle (i.e. during T1 state). The **special signal which identifies the T1 state** is the ALE (Address Latch Enable). When the BIU in the CPU) is going through the T1 state the ALE puts the WSG in a state called (let us call) **WT1** (Wait State Generator’s T1 state). We want the WSG to track the CPU. So when the CPU moves to T2, the WSG shall move to **WT2**. Whether WT2 matches T2 exactly in time or with a half clock delay or a full clock delay depends on your design. It depends upon two aspects:

1. **when** is WSG coming out of asynchronous/synchronous resetting and
2. what **significant clock edge** is used for WSG state transitions.

18.4 A simple design would use a shift register asynchronously initialized by the ALE when it is active. We can first clear the shift register using ALE, then we can start shifting a “1” from one end of the shift register. So the shift register’s outputs provide a **step-like waveform** (Q0 goes high, then Q1 goes high, so on and so forth).

18.5 Note that, once the CPU (the BIU, the Bus Interface Unit) reaches T4 state, it does not care what your WSG is telling and what value it sees on READY until the T3 of the next bus cycle. So, it is fine to have READY = 1 or READY = 0 constantly during T4 and subsequent Ti (T_idle state of the BIU). It is fine even to make the READY signal go up and down!

18.6 While ALE initialization of WSG is common to bring the WSG to WT1 state (Wait-State Generator’s T1 state), some designers may use the inactive-state of the RD and WR signals (both inactive together, i.e. $\text{RD} = 1$ and $\text{WR} = 1$) to initialize the WSG.

18.7 Consider the interrupt acknowledge (INTA) bus cycles also in designing the WSG and note that the address (A19-A0) is unknown during the INTA bus cycle. You can use ($\text{RD} = 1$ and $\text{WR} = 1$ and $\text{INTA} = 1$) as a better initialization mechanism. You can consider producing a different
number of wait-states for the read bus cycles and another different number of wait-states for the write bus cycles.

For example, a design problem statement could be as follows:

The address range of 80000-8FFFF is occupied by a memory which needs one wait state for the read bus cycles and two wait states for the write bus cycle. In the same system, the address range of 90000-9FFFF is occupied by a memory which needs two wait states for the read bus cycles and one wait state for the write bus cycle. During the interrupt acknowledge (INTA) bus cycles we need 2 wait states. Note that the address is unknown during the interrupt acknowledge (INTA) bus cycles.

In most of the problems in the homework on the WSG design, we do not bring-in the interrupt acknowledge (INTA) bus cycles, as interrupts are covered much later.

18.8 One could think of using a regular state machine instead of a shift register to have greater flexibility in what to do in what bus cycle.

18.9 ALE, (which goes active consequent to the beginning negative edge of CLK in the T1 state), goes inactive consequent to the middle positive edge of CLK. So it is important to understand that at the middle positive edge of the T1 state, the FFs in the shift register of the WSG can not start shifting data from the D inputs as they are still forced to remain in the initialization state. Similarly, if you are using “RD = 1 and WR = 1” as an initialization mechanism, since one of them goes inactive only consequent to the beginning edge of T2 state, the initialization is in force at the beginning negative edge of T2.
18.10 An example design of a WSG using a shift register:

**18.10.1** Draw the waveforms of Q1, Q2, Q3, Q4 and Q5. If Q4 is used as RDY1 and is passed through FF2 (only FF2) in the 82C84A (i.e. with Async tied to VDD) how many wait states are inserted in each bus cycle.

![Waveforms of Q1, Q2, Q3, Q4, and Q5](image)

**18.10.2** Now consider a slightly modified version of the above problem. RDY1 is passed through FF1 and FF2 by tying the Async input to GND. Are we violating any setup time or hold time?
18.10.3 In the 18.10.1 problem, if we used negative-edge triggered flip-flops instead of positive-edge triggered flip-flops and passed RDY1 through FF1 and FF2 by tying the ASYNC input to GND, what happens?