Massachusetts Institute of Technology
Department of Aeronautics and Astronautics

16.682 – Special Subjects in Aeronautics and Astronautics
"Prototyping Avionics"

Homework #3
Out: Wed Mar 8, 2006
Due: Wed Mar 15, 2006

Topics:
- Digital Logic elements
  - Basic Logic
  - Multiplexers
  - Latches, Flip-Flops, & Registers
- "Thinking Digital"
  - Notation and bases
  - Bitwise vs. Wordwise
  - Shifting
  - Data types, 2's complement

Problem 1 – Basic Logic

First, lets make sure that you have a quick feeling for how the basic two-input logic gates work.

Using 2-input AND, NAND, OR, and NOR gates make:

1) a 2-input XOR gate

Using **only 2-input** logic gates (AND, NAND, OR, NOR, or XOR) make:

1) a 5-input AND gate
2) a 5-input OR gate

Show the truth-tables for both gates with as few rows as possible.
(tip: try to use a lot of "don't care" x's)

The idea to take with you is to see how:
a) Complex circuits are a lot of simple ones put together
b) To make complex design you start to use a lot of logic gates (many transistors)

Problem 2 – Multiplexers

In class we saw a 2-input and a 4-input multiplexer. Now it is your turn to show how a 6-input multiplexer would look like:

1) How may "select" control lines are needed?

2) Draw the symbol for a six input multiplexer – a single symbol, do not use many multiplexers to create this one.

3) Show the truth table as simple as possible… do you have extra rows?

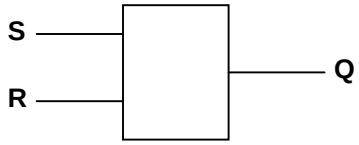
4) What is the number of inputs needed to use all select lines?

5) What if you used a 2-input and 4-input multiplexer, and then combined the signal, would you use all the select lines?

Problem 3 – Latches, Flip-Flops, and Registers

1) Latches

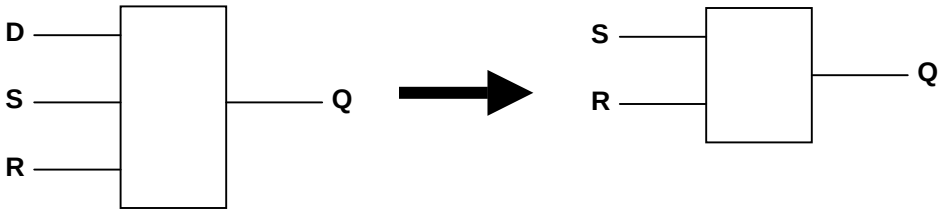
The most simple latch out there is one that only has only three connections: set, reset, output:



The signal Q goes high when Set is asserted, and it goes low when R is asserted.

a) Draw the truth table for this latch --- point out any states you're not sure about

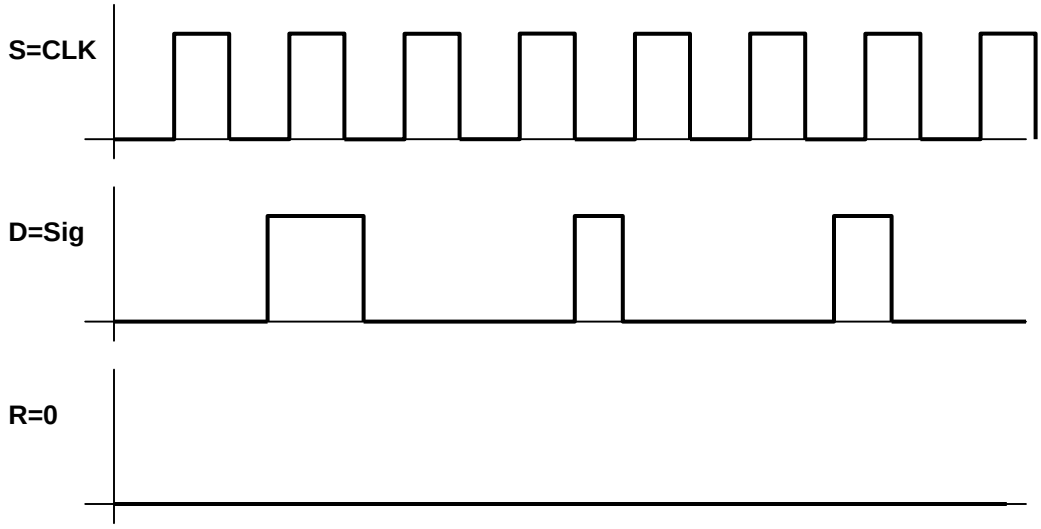
b) Create this latch by using the latch we saw in class (inputs: D, S, R; output: Q) and connecting the input signals as needed. The diagram below illustrates this question.
(Tip: don't over think it, this part should take 10 seconds, the answer is extremely simple!)



Lets re-visit the timing problem with latches.

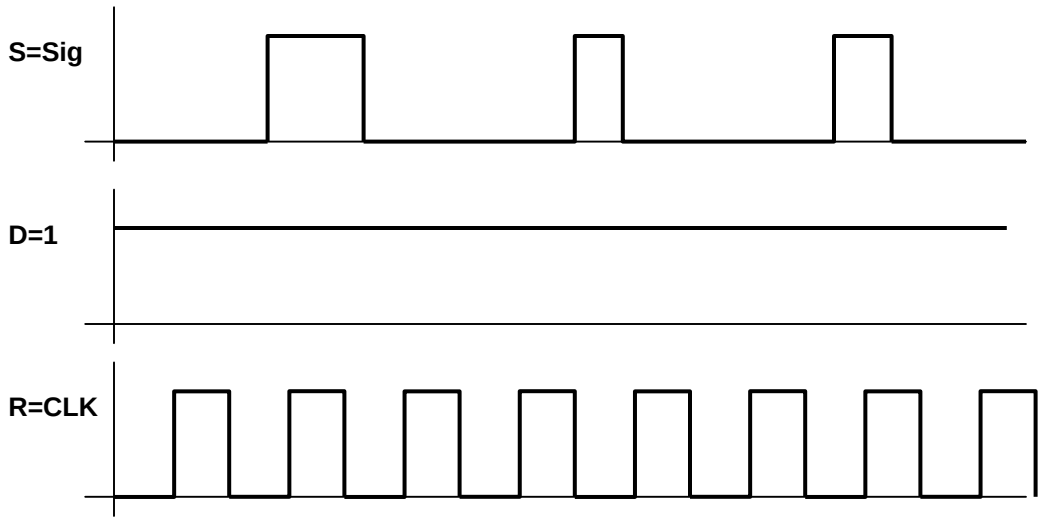
First, assume that you have a microcontroller which must have a stable input when its input clock is **low**. We want to read a signal during this period.

a) Lets setup the latch using the signals as follows (try for output Q to not change when the clock is low):



Draw Q, the output, and show when the data was captured correctly and when the data is missed.

b) Lets try this setup now (never miss data, and reset to 0 when clock is high):



Once again, draw the output Q and show when the data is good and bad.

c) Try at least one other setup which you think may work…

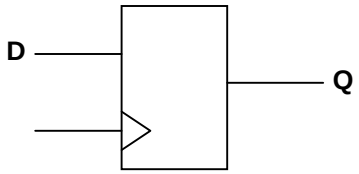
Latches are still useful for many things, especially those which really don't require any special timing (no synchronization) or where the timing is very slow (example, a person pushing a button to turn a light on (set) and then another button to turn the light off (reset); in this case the person reacts very slowly compared to the speed of the latch, so nothing is lost). But the latch is not a great interface to things that work in a synchronized way with fast clocks.

In the second case, an option would be to have reset come back from the processor, instead of the clock, but that would be cheating. We don't want to use processor time to capture a signal, we should be able to do a good job without it.

Like it was mentioned in lecture, a big problem here is that the latch is "level" triggered, so the data must hold valid for a long time and any change in data/signal creates a response we may not want. Therefore, we have…

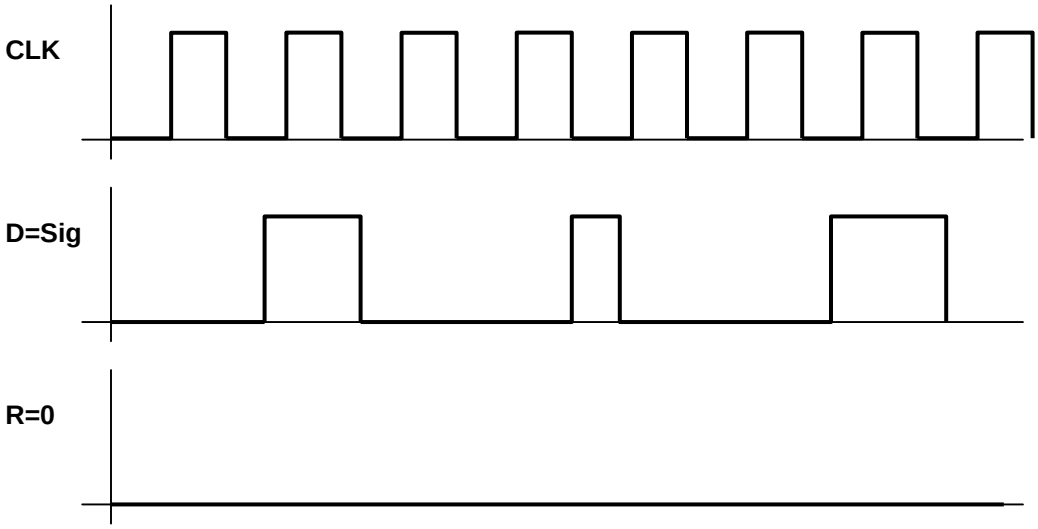
2) Flip-Flops

Remember how a flip-flop works: when the clock input (the triangle) has a "rising edge" (goes from low to high) the value of data at that time is stored in Q until the next rising edge. Here is how the flip-flop looks:



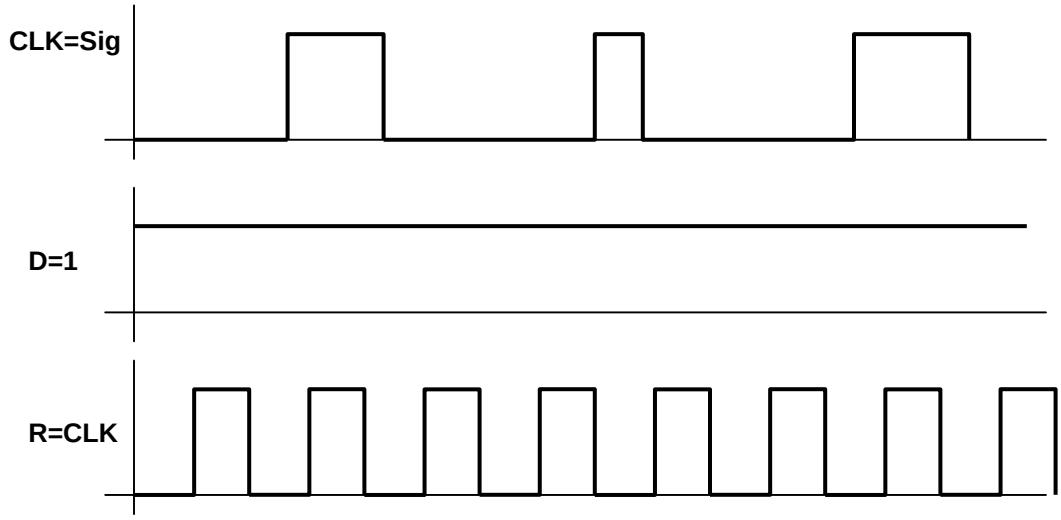
Lets use the same timing diagrams as for the latch:

a) Draw Q, the output, and show when the data is valid and invalid.



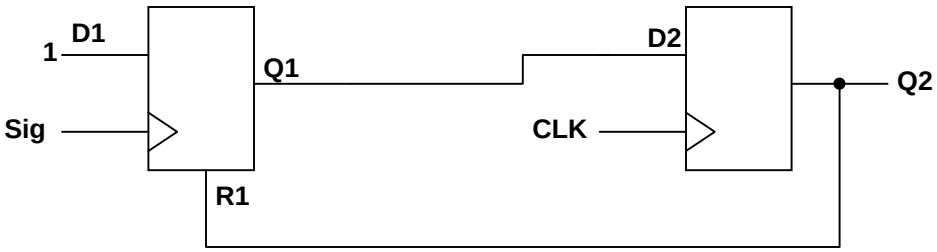
Is this any better than for the latch?

b) Draw the output Q and show when the data is valid and invalid for the second setup:



Is this one better or worse?

c) Lets solve this problem completely by using two flip-flops, instead of just one:



Use the same signals as in the previous part.

To figure this one out, follow the next steps:

i) Look for the first rising edge of the signal “sig”, that should make Q1 go high.

ii) Since Q1 is D2, the next time the CLK signal has a rising edge, the output Q2 will go high

iii) When Q2 goes high, it forces the first flip-flop to reset, making Q1 low

iV) On the next rising edge of CLK Q2 will go low, and the first flip-flop can operate again.

Note: it takes some finite amount of time for signals to get through a flip-flop, therefore when figure out the timing account for some small amount of time between a signal going into the flip-flop and the change being visible in Q.

d) The resulting signal has something very special related to the CLK signal, what?

e) Why would you not be able to do this with a latch?

3) Registers

If you understand latches/flip-flops, you basically understand registers, so this question goes into an a common application.

Many systems use counters to check either time or number of events that occur, and want to store that value at specific known times. Assume that you have an 8-bit counter which you want to store whenever a sensor is triggered (signal goes from low to high).

Find an actual 8-bit register with a rising edge clock and using its connection diagram describe how you would store the counter value until the next time the sensor triggers. *Ask for help if you're overwhelmed by the number of registers out there!*

Problem 4 – Thinking Digital

1) Binary and Hexadecimal: Masking

Lets apply the bitwise operations we learned in class in one of the most useful things to do with bytes/words: masking. A "mask" is a value you use to keep some things the same and other things different. For example:

```
0xFF AND 0xXX
```

where `0xXX` is a random hexadecimal value, **always** returns `0xXX` – `0xFF` is a mask.

Lets figure out how to use other masks and the logical operations AND, OR, and XOR to modify the hexadecimal random value `0xXY`. All your answers will be of the form:

```
0xXY [AND, OR, XOR] <mask>
```

a) Make X all 0's and keep Y the same.

b) Keep X the same and make Y all 1's.

c) Make the second least-significant bit (LSB) 0, keeps everything else the same.

d) Invert X and keep Y the same.

e) Returns all 0's (a logical false) if the fourth LSB is 0

2) Shifting data

A very important use of shifting is to create “serial” data. An 8-bit byte is “parallel” data, all of the data exists at the same instant in time. In serial data, only one bit exists at a time, followed by the next bit, and the next bit.

Say you have the hexadecimal value

```
0x36
```

stored in a register, and you want to “shift” the byte into a serial line. To do this, we will use the following procedure:

a) Write the number in binary

b) Start a timing diagram which has two traces, a clock and a signal; draw a square wave for the clock with at least 10 full cycles

c) On the **rising edge** of the clock do the following in order:

i) In the signal trace draw the LSB of the byte in the register
then
ii) Shift the signal right by 1, assuming a circular-buffer shift

Repeat this until you have shifted the whole byte

d) Examine the timing diagram: for each full-clock cycle write whether the signal is a 1 or a 0. How does this relate to the original hexadecimal value?

Microcontrollers are much better at figuring out how the first bit works than trying to access a random bit in a byte, therefore this is many times the way to serialize data or otherwise analyze individual bits inside a byte.

3) Shifting: math

It is also important that you have the basic concept of how shifting performs mathematical operations, specifically multiplication and division by two. Therefore, using as many shifts as you need (you can store the value between shifts as many times as you need separately from the original value) but at most one addition or subtraction, explain how you would:

a) Multiply an integer number by 7

b) Multiply an integer number by 12

c) Divide an integer number by 16

4) Data types (& thinking digital as a whole!)

Data types have a maximum range (8bit, 16bit, etc), but many times these days memory is larger than the data-type.

Assume that our memory is 32-bit wide, and that we'll use the conventions that:

Data Type	**Size**
char/uchar	8 bit
int/uint	16bit

a) We want to use our memory as efficiently as possible: how many char type variables should we store in each memory space?

b) Lets do it using the masking and shifting functions:

i) Assuming that the memory space is

```
mem_val=0xXXXXXXXX
```

create the necessary masks to insert the value

```
char_val=0xYY
```

so that the final result is:

```
mem_val=0xXXXXXXYY
```

Tip: you will need to use two instructions.

ii) Now we want to insert `0xZZ` so that

```
mem_val=0xXXXXXXYY
```

becomes

```
char_val=0xXXXXZZYY
```

iii) Repeat for `0xWW` and `0xVV` so that you end up with

```
mem_val=0xVVWWZZYY
```

c) Now lets deal with signed and unsigned values. Assume that the actual value in digital is:

```
1001 1000 0110 0011 0101 0101 1010 0111
```

i) If we assume an unsigned character value which looks at the least-significant byte (LSB, to the right), what is its decimal or hexadecimal (your choice) value?

ii) What if it is an *normal* character (ie, signed), what is its decimal value?

iii) If we had signed integers, what is the sign of the lower-16-bit integer?

iv) For the higher 16-bit integer, what is its sign? What is its magnitude? Use binary/hexadecimal only to get the magnitude (not decimal).