The 4-bit adder from the previous lecture raised an interesting point: what happens when you have a circuit in which the output of one circuit is the input to another? Clearly there is some “propagation delay” as the 1s and 0s make their way through to the output.

But what if the output of a circuit is its own input?

Consider the following circuit:

```
S
\-----------\n|            |
|  B        |< a
|           |
\-----------\n|            |
|  A        |< b
|           |
\-----------\nR
```

It's hard to create a truth table for this because the input $B$ depends on the output $b$, which depends on the input $A$, which depends on the output $a$, which depends on the input $B$. Hmmm.

Let's simply things a bit and start out by assuming that $S$ and $R$ are both 1. If we also assume that $a$ is 0, then $A$ will be 0. That means that $b$ will be 1. This means that $B$ is 1, and thus $a$ is 0 (which is where we started). So far, so good.

Now what happens when $S$ becomes 0? In the top NAND gate, we have $S=0$ and $B=1$, which produces an output of 1 for $a$. That means that $A$ is 1, and now $b$ becomes 0. Thus, $B$ becomes 0 and $a$ is still 1. The effect of changing $S$ from 1 to 0 is to change $a$ from 0 to 1 and $b$ from 1 to 0. In a sense, we have “flipped” the values of $a$ and $b$, and this circuit is sometimes referred to as a “flip-flop”.

What if $S$ goes back to 1? Nothing changes! Because in the top NAND gate, we’ll have $S=1$ and $B=0$, which still produces an output of 1 for $a$. So at this point, the circuit “remembers” the value of $a$ when both $S$ and $R$ are 1.

Now what if $R$ becomes 0? Since $A$ is 1, the output $b$ will be 1, and so will the input $B$. Since $S$ is 1 now, the output $a$ “flips” from 1 to 0, and now $A$ is 0 and $b$ is still 1. The effect of changing $R$ from 1 to
0 is to change \( a \) from 1 to 0. As with \( S \), if \( R \) now goes back to 0, nothing will change: \( a \) will still be 0 and \( b \) will still be 1.

To summarize:
- when \( S \) goes from 1 to 0, \( a \) becomes 1
- when \( R \) goes from 1 to 0, \( a \) becomes 0
- when either \( S \) or \( R \) become 1, \( a \) holds its value

Note that we started with an assumption of \( a \) being 0. But as you can tell, all of the observations we made would be similar (in terms of the circuit's behavior) if we started with \( a \) being 1.

Because we only care about the output \( a \), we can draw a truth table like this:

<table>
<thead>
<tr>
<th>( S )</th>
<th>( R )</th>
<th>( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>illegal</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>stays same</td>
</tr>
</tbody>
</table>

This circuit is known as an **R-S latch** and is the basic element of 1-bit memory. That is, the output \( a \) is being “remembered” by the circuit when \( S \) and \( R \) are both 1. We can set it (i.e., make it equal to 1) by changing \( S \) from 1 to 0, and we can clear it (i.e., make it equal to 0) by changing \( R \) from 1 to 0.

Now that we have a way of remembering a value, we need some way of putting the value there in the first place.

Look at the figure below (clearly stolen from a French website!). The right side is an R-S latch (with \( a \) and \( b \) replaced by \( Q \) and its inverse)

\( D \) is the data and \( C \) is the control (the book calls this “WE” for “write-enabled”): we want the output of the latch \( Q \) to be equal to \( D \) when \( C \) is 1, and to hold its value when \( C \) is 0.

We can see that this circuit accomplishes this by looking at the truth table. Note that although \( S \) and \( R \) are not explicitly shown in the diagram, they are intermediate values that are inputs to the latch.

<table>
<thead>
<tr>
<th>( D )</th>
<th>( C )</th>
<th>( S )</th>
<th>( R )</th>
<th>( Q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>stays same</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>stays same</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Now we have not just a way of remembering a value, but also a way of controlling when it is set or cleared. This type of circuit is known as a **gated D latch** or a D-type latch.
Often we want to store not just a single bit, but multiple bits that go together to form a number, character, etc. A collection of D-type latches that stores multiple bits as a single unit is referred to as a register. There is an example in the book as Figure 3.20. The single-bit values of Q will stay the same as long as WE is 0; when WE becomes 1, then the data D is “written” into the register, and the bits of Q will take on those values.

**Memory**

As we will see next week (and as you probably know already), memory is an important part of the computer. Memory can store program data, as well as instructions, system state, etc.

In answering the question “how much memory does the computer have?” there are two deciding factors: address space and addressability.

The address space refers to the number of bits used to represent an address. If there are $k$ bits in the address, that means we can represent up to $2^k$ distinct addresses. In that case, we’d usually say that there are thus $2^k$ registers in memory, each of which holds some number of bits that forms a related unit.

How many bits are in a unit? That is determined by the addressability. That refers to the number of bits held at a single address.

Modern computers generally have a 32-bit address space and 8-bit (or one “byte”) addressability. That means there are $2^{32}$ addresses, each of which refers to an 8-bit register. How much total memory is that? Well, $2^{32}$ is approximately 4 billion, so there are around 4 billion addresses, each of which holds one byte. So the total memory is around 4 billion bytes, or 4GB.

How does memory work? Figure 3.21 shows a diagram of memory with a 2-bit address space and 3-bit addressability. Note how a decoder is used to activate one of the registers, and note how a multiplexer is used to choose which register to read from.