CPS311 Lecture: CPU Building Blocks

Last revised July 12, 2021

Objectives:

1. To introduce the basic building blocks of more complex systems that can be built from gates and flip-flops: adder, decoder. mux, register, shifter
2. To show how these can be combined into multi-bit units as part of the ALU

Materials:

1. Circuit Sandbox and demonstration circuits 4 bit adder, decoder with enables. Multiplexer, register with enables single bit, 4 bit Register, shl, shl2, shr, ashr
2. Projectables

I. More Complex Building Blocks

A. The individual gates and flip-flops we have been talking about can be combined to produce larger building blocks used at the microarchitecture level of design.

B. The ALU part of the CPU, in particular, is largely composed of three basic kinds of building blocks, together with a few individual gates and combinations of gates.

1. Adders

2. Decoders

3. Multiplexers

4. Registers

5. Shifters

II. **Full Adders**

A. All four of the basic arithmetic operations - addition, subtraction, multiplication and division - make use of addition in their implementation.

1. Subtraction is simply adding the negated subtrahend - e.g. A+B is implemented as A + (-B). (We will see later an easy way to negate a binary integer).

2. Multiplication is implemented by adding partial products, similar to the way you learned to do multiplication in elementary school:

```
   123
x  345
------
   615      <-  partial products
  492
 369
------
42435
```

PROJECT

a) Decimal multiplication requires using a 10 x 10 multiplication table for the digits, which you probably memorized in elementary school. Binary multiplication is much easier, because the only possible digits are 0 and 1, and multiplication by these is trivial!

b) Later in the course we will look at a hardware multiplication algorithm and hardware implementation.

3. In similar fashion, division is implemented using trial subtraction, similar to the way you learned to do it in elementary school.

```
          345
123 | 42435
     -369
     ----
       553
      -492
      ----
        615
       -615
       ----
          0
```

PROJECT

a) But again, in decimal trial subtraction involves considering several different products of the divisor (e.g. in this case 1 x 123, 2 x 123, 3 x 123, 4 x 123) and choosing the largest one that "works" In binary, all that is needed is a simple comparison and then you either subtract the divisor or 0 at each step.

b) Again, you will see a binary approach to doing this later in the course.

4. So all we need to be able to do is to perform addition in binary and the other operations can also be realized.

B. Now consider the task of adding two one bit numbers. Recall that the result is both a sum and a carry, since in binary 1 + 1 (arithmetic +, not or!) is 0 with a carry of 1. This yields a two-output truth table:

A	B	∑	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Which corresponds to the boolean equations

$\Sigma = A \oplus B$
$C = A \bullet B$

PROJECT

Which can be easily implemented using an XOR gate and an AND gate.

C. Of course, we need to be able to add numbers with more than one bit! To do this, we need to consider the possibility - for each bit - that it gets a carry from the bit to its right. Thus, the truth table we need involves a carry-in input as well, so we get (written in a non-standard ordering of the inputs to make the equation more obvious)

A	B	C_{in}	Σ	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Which corresponds to the equations

$\Sigma = A \oplus B \oplus C_{in}$
$C_{out} = A \bullet B + (A \text{ or } B) \bullet C_{in}$

PROJECT

1. This could be realized by using two XOR gates, two AND gates, and two OR gates.

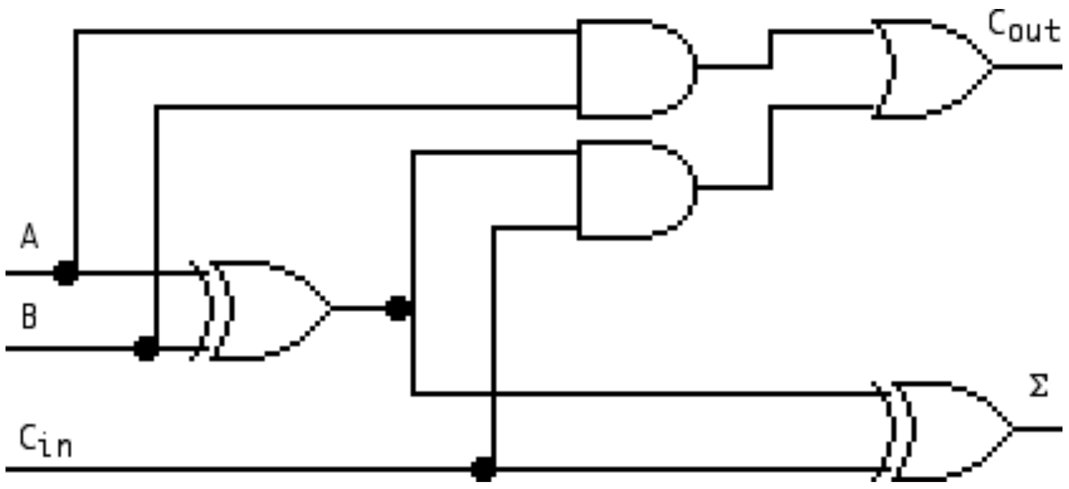
2. However, by sharing one XOR gate between both the sum and carry out equations, this could be simplified to

$\Sigma = A \oplus B \oplus C_{in}$
$C_{out} = A \bullet B + (A \oplus B) \bullet C_{in}$

which only needs one OR gate

PROJECT

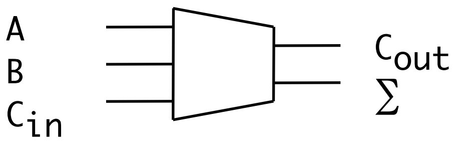
3. This yields the following circuit:



PROJECT

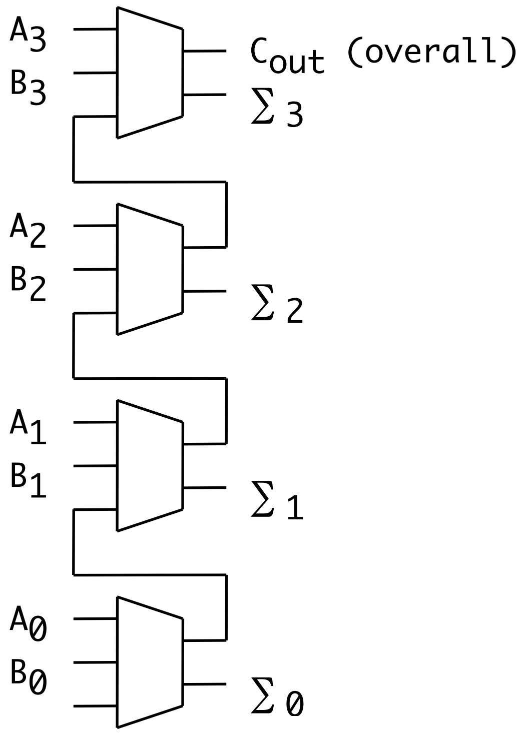
which is known as a full adder (an adder for two bits without carry is known as a half adder).

4. as we noted earlier, it has the following special symbol:



PROJECT

D. Any number of full adders can be connected to produce a multi-bit adder. For example, this is an adder for two 4-bit numbers using 4 full adders connected in ripple-carry fashion.



PROJECT

1. DEMO with Circuit Sandbox (file 4 Bit adder)

2. As we noted earlier, a full adder can take a long time to propagate a carry with certain inputs, so in practice a more efficient way of connecting the carries is used - e.g. a technique known as carry-lookahead). (This is discussed in the remainder of §5.2.1 not assigned)

   DEMO Delay with A = 0, B = 1111 with speed set to 1000 ns - then change A to 1 and watch how carry lines change from 0 to 1 in ripple fashion.

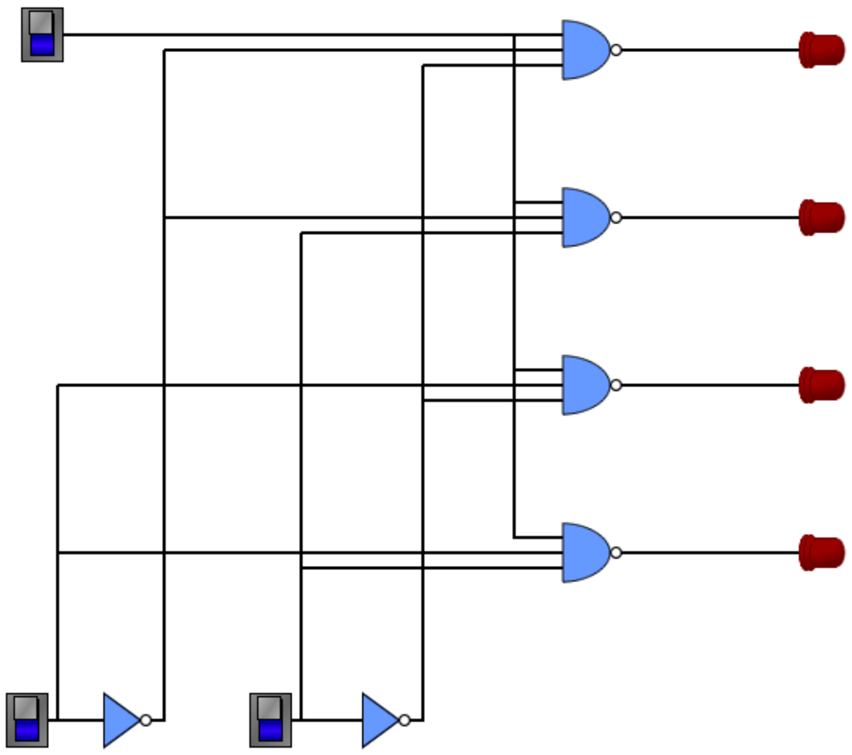
III. **Decoders**

A. A decoder is a circuit that has n inputs and 2^n outputs (for some small integer n - typically <= 4 with discrete chips). At any given time, exactly one of the outputs is active, as selected by the inputs.

Example: a 1 out of 8 decoder has 3 inputs and 8 outputs. Based on the value of the inputs, exactly one of the outputs is active.

B. Often decoders are built using NAND gates - in which case the selected output is 0 and the others are 1. Sometimes a decoder also has an enable input which - if it is not true - means that one of the outputs are selected.

C. DEMONSTRATE: Circuit Sandbox realization - file Decoder - show how individual outputs are selected with enable on, and then none selected with enable off.



a) Note that the selected output is low (0), and the other three outputs are high (1). This is an example of negative logic. Decoders are typically designed this way because a level of inversion is inherent in the typical transistor circuits used to implement gates.

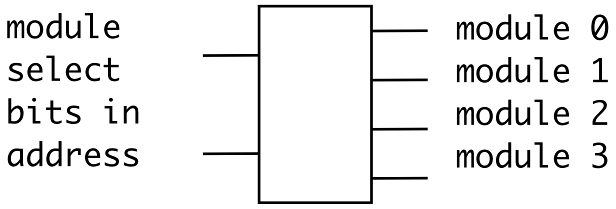
b) Devices that are designed to be used with decoders (eg memory chips) often have active low enable inputs as a result.

D. A typical application is in specifying which one of a group of similar devices is to respond on a particular operation.

Example: A memory system consists of 4 modules. When an operation is done on the memory, 2 bits of the address are used to specify which of the 4 module is to perform the operation.

00	= module 0
01	= module 1
10	= module 2
11	= module 3

A one out of 4 decoder could process these two bits to select one of the 4 modules:



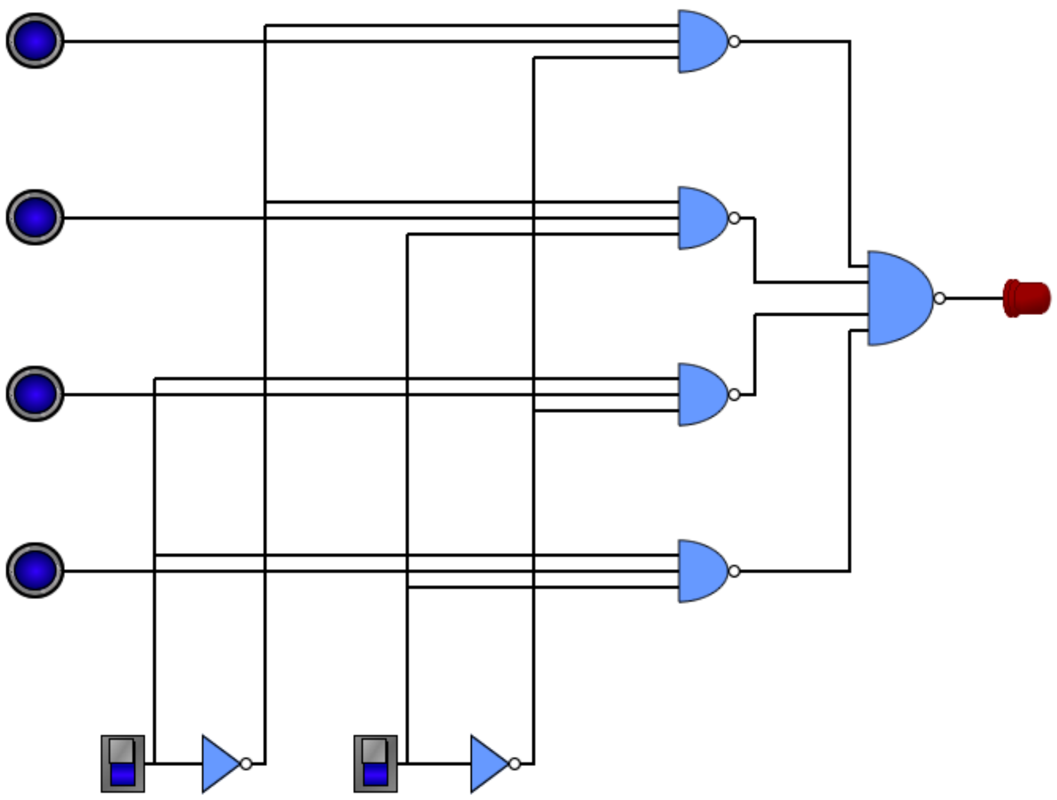
PROJECT

IV.Multiplexers

A. A multiplexer (MUX) is - in some sense - the opposite of a decoder. It has n selection inputs and 2^n data inputs (for some small integer n - typically 2-5), and one output. It selects one of the 2^n data inputs to appear on its output, as determined by its n selection inputs.

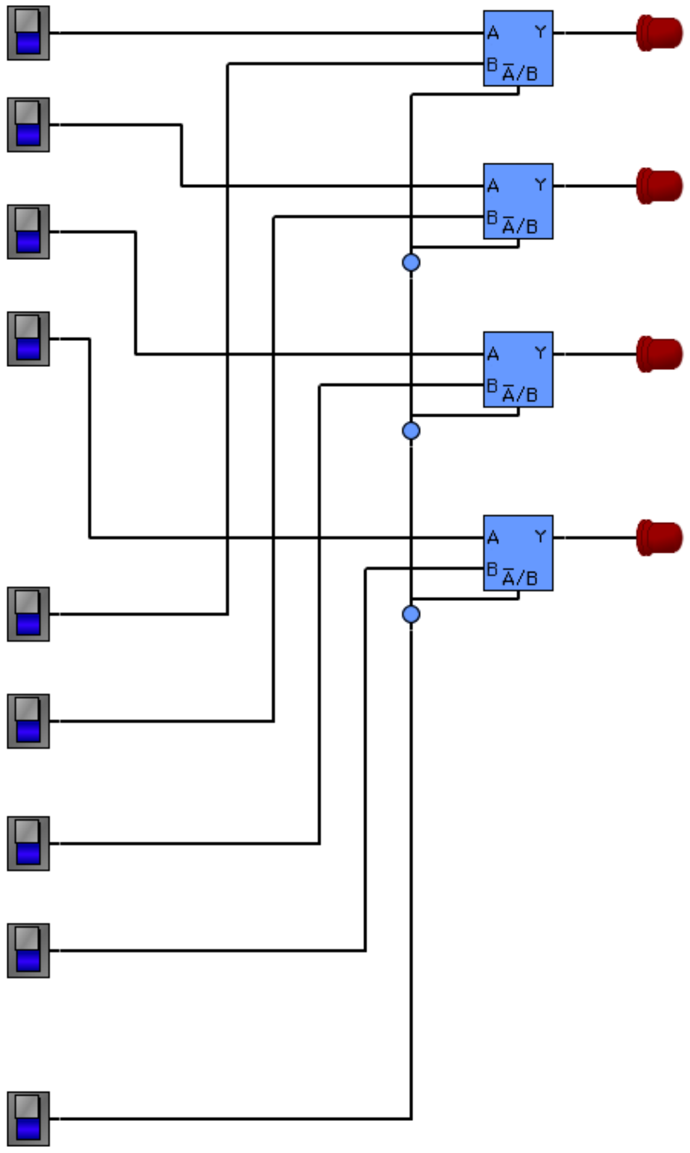
B. A multiplexer resembles a decoder, but with data inputs to each of the final gates plus a single or gate to pass the selected result through..

C. DEMONSTRATE Circuit Sandbox file Multiplexer



D. Multiplexers are often used to build data paths in the ALU.

1. A simple example - a 4 bit wide 1 out of 2 multiplexer can be built from 4 1 out of 2 MUXes



DEMO CSB 4 wide 2 MUX

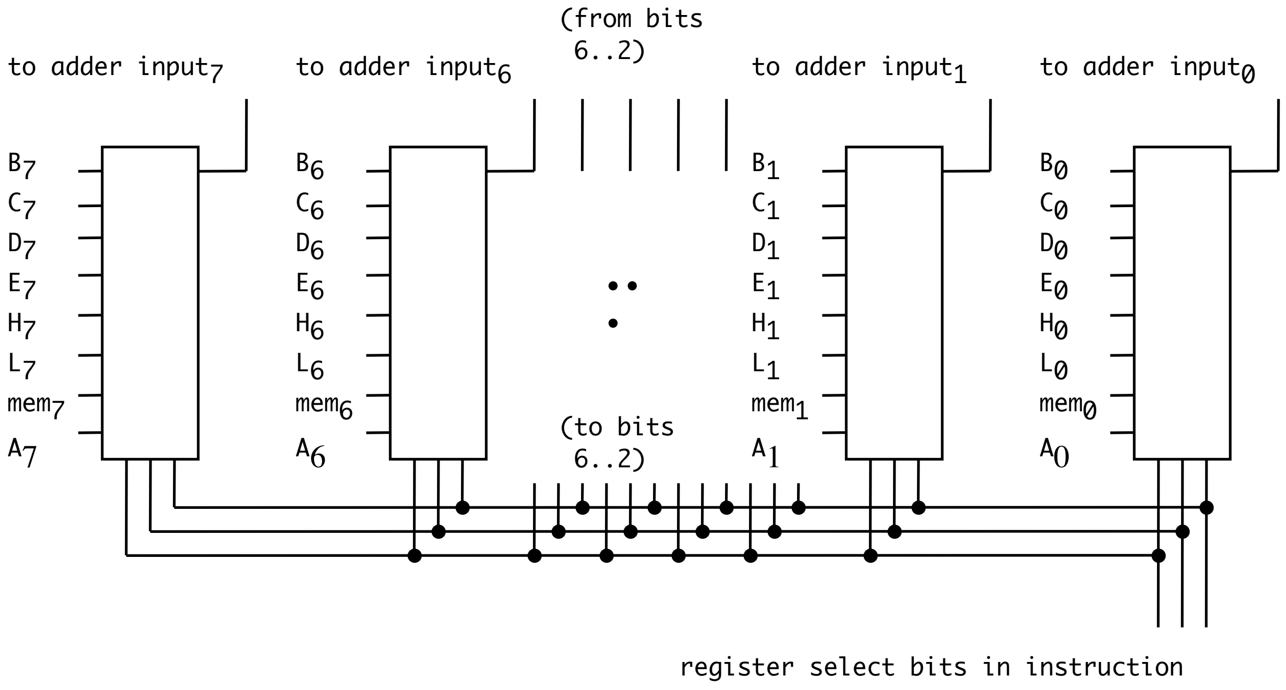
2. One typical application in CPU's arises when a group of bits in an instruction is used to select one of several possible registers to provide data for an operation.

Example: In the Z80, quite a number of instructions use a 3-bit field to select one of 8 registers (B, C, D, ...) to provide data. For example, there is a family of 8-bit add instructions that use the value of 3 bits in the instruction to determine which register gets added to A:

1 0 0 0 0 r r r - where r r r designates which register to add:

0 0 0	add contents of B
0 0 1	add contents of C
0 1 0	add contents of D
0 1 1	add contents of E
1 0 0	add contents of H
1 0 1	add contents of L
1 1 0	add contents of a memory location

This can be implemented by using one MUX per bit, with the selection inputs tied to the appropriate field of the instruction register.


(from bits 6..2)
to adder input$_7$
to adder input$_6$
to adder input$_1$
to adder input$_0$
B_7 C_7 D_7 E_7 H_7 L_7 mem$_7$ A_7
B_6 C_6 D_6 E_6 H_6 L_6 mem$_6$ A_6
B_1 C_1 D_1 E_1 H_1 L_1 mem$_1$ A_1
B_0 C_0 D_0 E_0 H_0 L_0 mem$_0$ A_0
(to bits 6..2)
register select bits in instruction

PROJECT

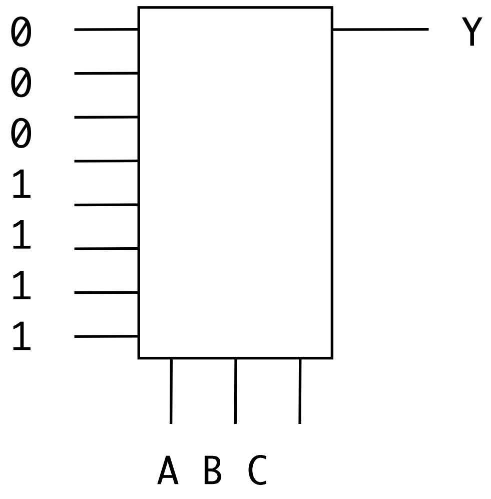
E. Another interesting application for a multiplexer is this: any n-input boolean function can be realized using an n selection-line MUX. The method is this:

1. The inputs (which we will call $A_0..A_{n-1}$) are applied to the selection lines of the MUX.

2. Each possible combination of $A_0..A_{n-1}$ selects one of the 2^n data inputs of the MUX and routes it to the output. We connect the corresponding MUX input to 1 if the truth table shows a 1 output for that row, and to 0 if it shows a 0 output:

EXAMPLE: Realize the following truth table using a MUX. (This is the same example function we have used in a number of places)

A B C	Y
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0



PROJECT

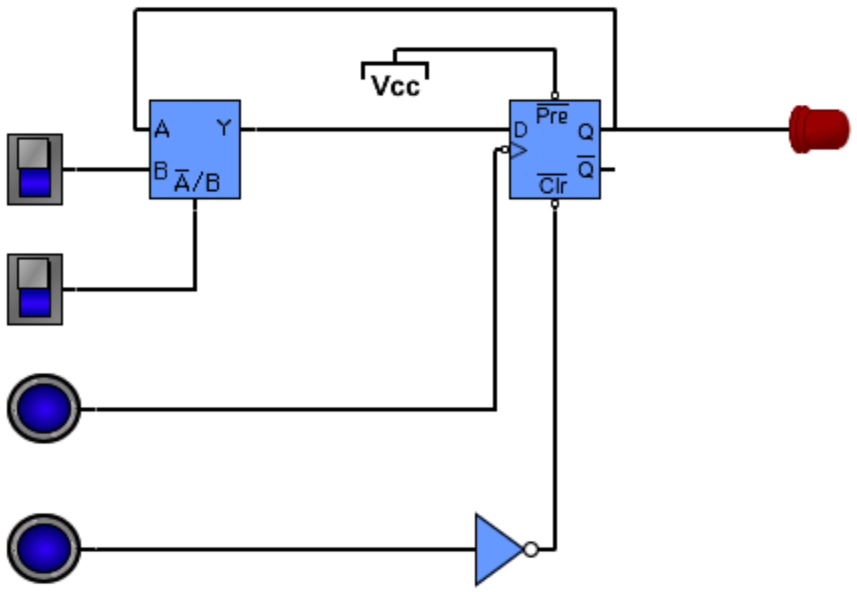
Actually, it turns out one can also realize an arbitrary function of n input variables using an n-1 selection input 2^{n-1} data input) MUX plus an inverter. (This is left as an exercise to the student)

V. Registers

A. The most visible part of an ISA is the set of registers, as you saw in lab working with the Z80.

B. Conceptually, a register is very simple - an n-bit register is just n flip flops (often D's) sharing a common clock. The n D inputs are the input value, and the n Q outputs are the output value. Whenever a clock pulse occurs, the input values are loaded into the register and remain available on the output until a new value is loaded.

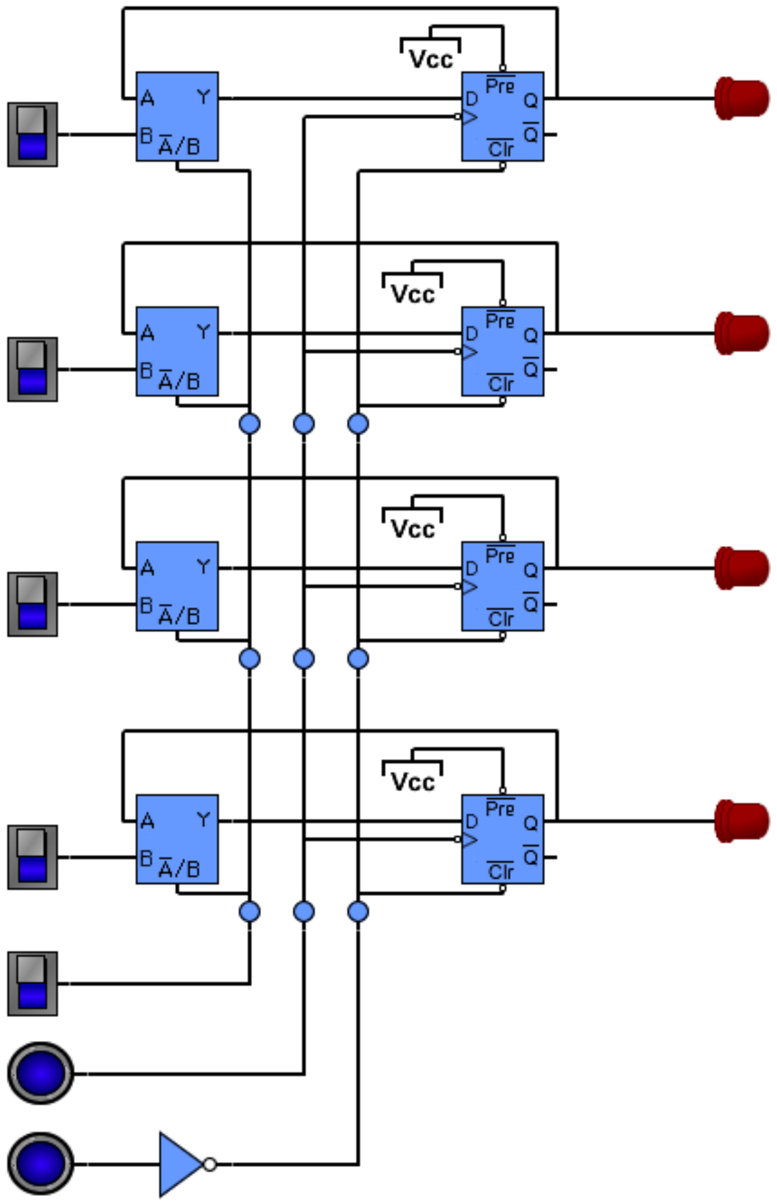
C. However if we use the same clock for multiple registers (as is generally the case), we want to be able to select which register(s) change on a given clock pulse. To do that, we might use a MUX with each flip flop to enable or disable loading, In this case, a typical register bit would look like this.



DEMONSTRATE - CSB Register bit with enable and clear

The upper switch is the value to load into the register; the lower switch is the enable; the first button is the clock and the second is clear.

D. An n-bit version of this register could be constructed by using n copies of this circuit tied to a common clock, enable, and clear enable, with each bit having its own input and output.



If the clock is pulsed when the enable is on, the new data is loaded into the register; if the enable is off, each flip flop copies its current value back into itself so there is no visible change in the stored value.

DEMO CSB 4 Bit Register

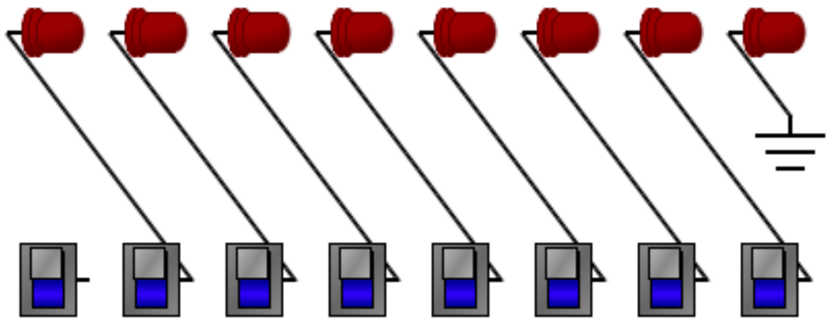
VI.Shifters - Cover only some if time is an issue

A. Circuits that perform simple left or right shifts are easy to build.

In fact, they do not need any gates - just wires.

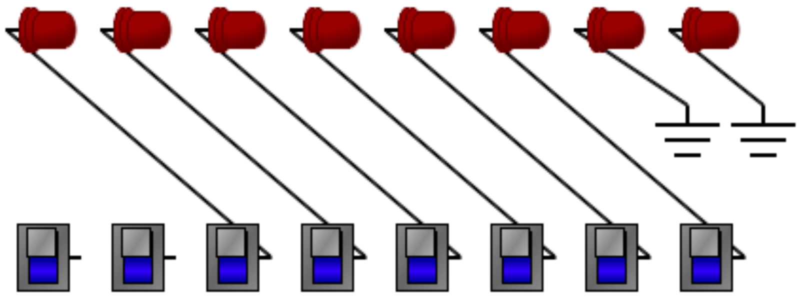
B. For example, here is a 1 bit left shifter:

1. DEMO shl.csm



Shifting left one place is equivalent to multiplying by 2.

2. It is also possible to build a shifter that shifts left 2 places - equivalent to multiplying by 4.



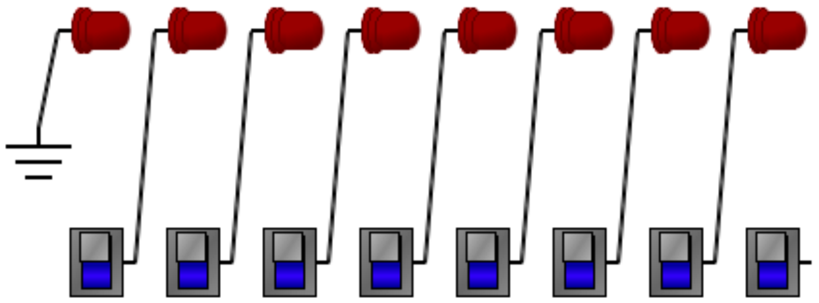
DEMO shl2.csm

3. In fact - as we shall see later - whenever hardware needs to multiply by a power of 2, it is done by an appropriate shift. In general, a shift left of n places is equivalent to multiplication by 2^n.

4. Note that a left shifter always puts a 0 in the rightmost place(s_, and discards the leftmost place(s).

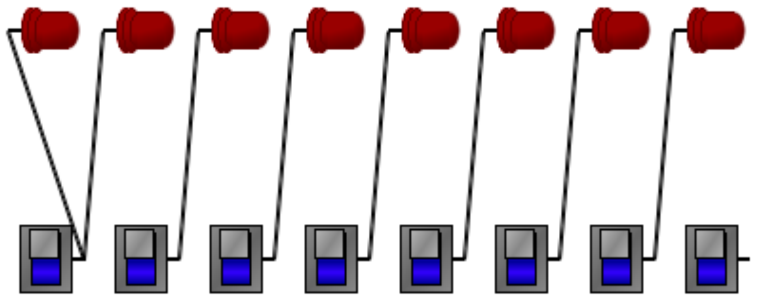
C. It is also possible to build a shifter that shifts right.

1. DEMO shr.csm



A shift right of one place is equivalent to dividing an unsigned number by 2. Note that, in this case, a 0 is shifted into the rightmost place(s), and the leftmost place(s) are dsicarded.

2. When dividing two's-complement signed numbers, it is necessary to propagate the sign into the vacated places instead.



DEMO ashr.csm

3. Right shifts that shift a 0 into the leftmost position are called logical shifts, while those that propagate the sign are called arithmetic shifts.

VII. **Conclusion**

Strange as it may seem, if you understand things the building blocks we have discussed - especially adders, registers, and muxes, as well as implementing boolean equations by gate networks, you'll be in great shape to understand how a CPU is implemented. More on this later!