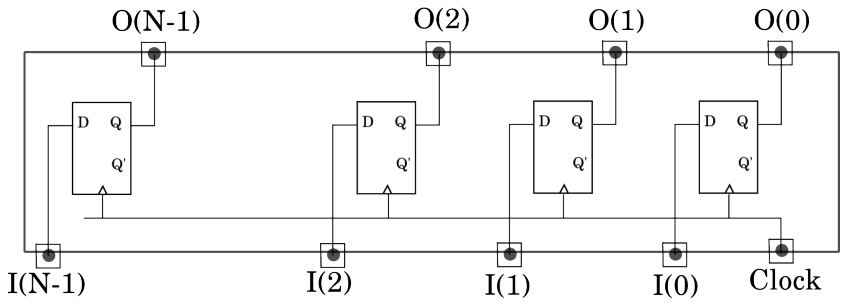
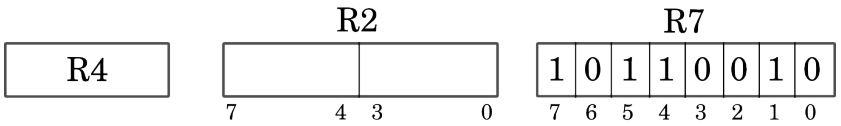
Lecture 12: More on Registers, Multiplexers, Decoders, Comparators and Wot-Nots

Registers

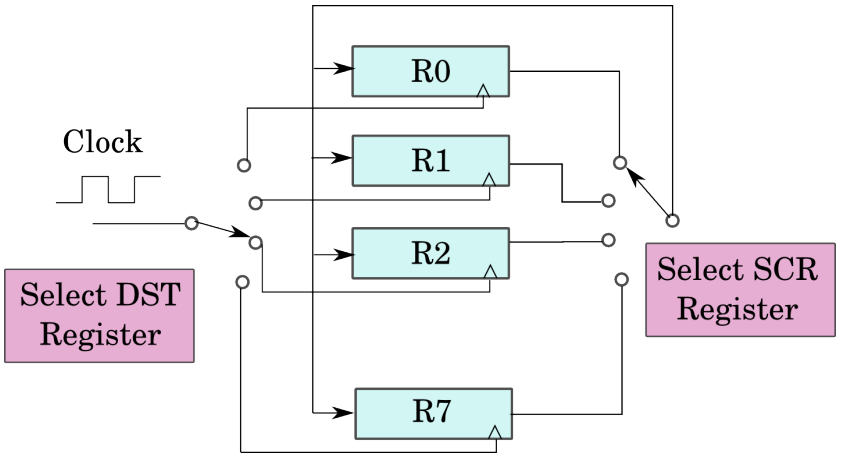
As you probably know (if you don't then you should consider changing your course), data processing is usually done on fixed size binary "words". Data are stored in computers in registers which can be thought of simply as collections of D-type flip-flops. The schematic diagram of such a register is shown as a functional block below:



By convention we number the bits from 0 to $N-1$ for an N-bit register and interpret its contents as the positive numerical value $\sum_{n=1}^{N}(2^n)O(n)$. (However, it is important to remember that bits are bits are bits and the interpretation of what they represent is not fixed). There are many short hand notations for registers. Three of the most common ones are shown below:



Registers can contain either data or control bits and are the fundamental building blocks of digital computers. One of the most basic operation between registers is register transfer which means the copying of the contents of one register into another without the loss of data in the first one. As a first exercise, we will build the hardware for a general register transfer engine which contains eight registers and may be represented by the following schematic diagram:



On each clock pulse (falling edge for example) the selected destination register is loaded from the selected source register. Only the selected register receives a pulse, all others therefore remain unchanged. By setting the switches before a pulse the contents of any one register can be transferred to any other register. Hence, the order of events is:

1. Select the input register
2. Select the output register
3. Generate a clock pulse to transfer

The Register Transfer Circuit using a Multiplexer and a Demultiplexer

Clearly we will not use a mechanical switch in practice. Instead, to select the source register we use the multiplexer circuit which we hae seen earlier in the course. To select the destination register we will use a demultiplexer circuit (sometimes called a decoder or a binary to unary converter). The circuit of a 4-1 multiplexer is shown as a functional block in figure 1. We have added an enable line which must be set to 1 for the connection to be made. We shall see that this is a very useful feature in due course.

Most CAD systems supporting digital circuit design provide a comprehensive range of functional units, but it will never be complete. Suppose we are using a system which has only 4-to-1 multiplexers and we want an 8-to-1 device, we use functional reasoning to do this and arrive at circuit shown in figure 2. Following the same principle we can simply and quickly design a multiplexer of any size.

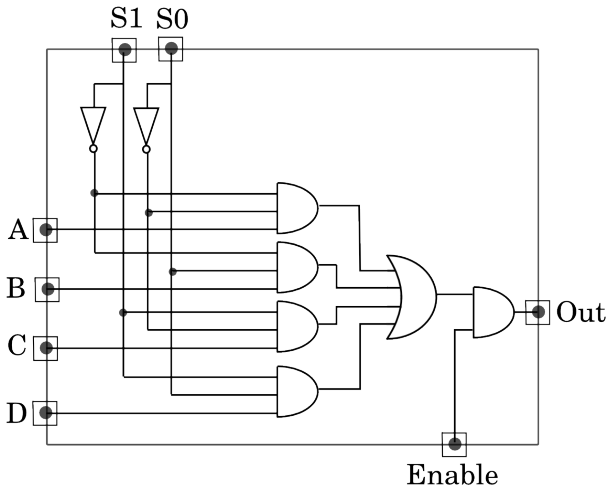


Figure 1: 4 to 1 Multiplexer

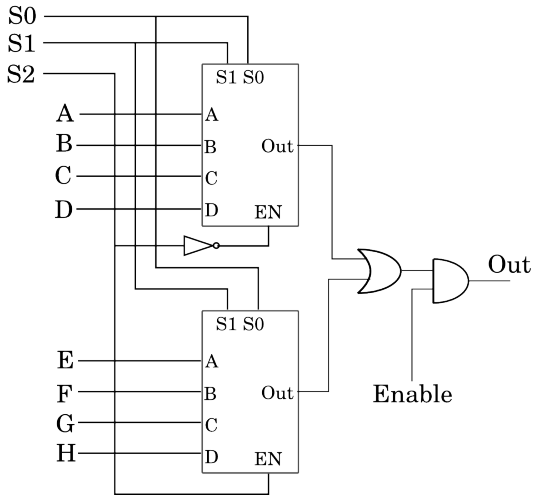
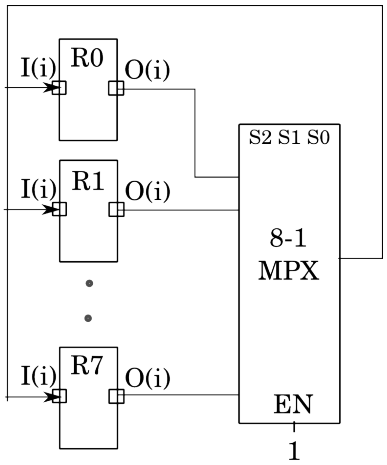


Figure 2: 8 to 1 Multiplexer

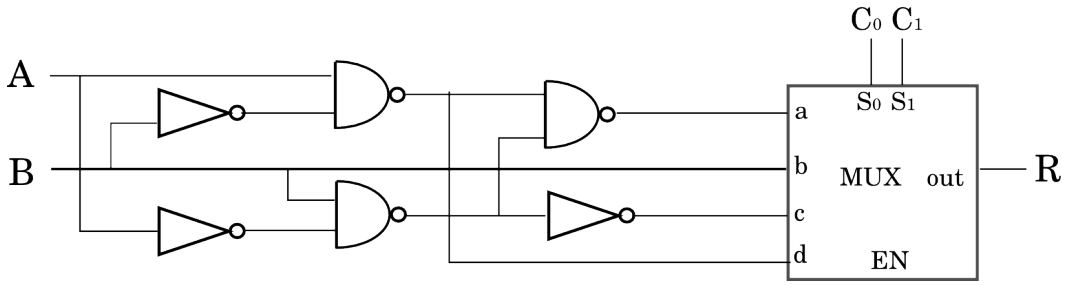
Having implemented an 8-1 multiplexer we can now wire up the "select SRC register" switch. We need one multiplexer for each bit of the registers, and they will all share the same control inputs $(S_2S_1S_0)$. The wiring for bit i is:



Some interesting properties of multiplexers

Functionally, multiplexers have a clear dedicated function. This does not mean that they cannot be used for other purposes. For example, you could have used it for your hardware assessed course work which you have just completed and you could have designed your circuit in ten minutes! Let's say have the functions A(XOR)B for (C1,C2)=00, B for 01, A'B for 10 and function (A'+B) for 11.

Noticing that a multiplexer can provide four different functions, and the simplifying the equations so that $A(XOR)B = (A'B + AB') = ((A'B)'(AB')')'$ and that $A' + B = (AB')'$, you could have built the following circuit:



The control functions select between the four inputs which are generated by simple circuits.

Demultiplexers

Let us get back to our original problem of the general register transfer circuit. We have solved the problem of selecting any one of the registers as a source register by desigining an 8-1 multiplexer. We still have to select a proper destination register and we do this by allowing only one register to receive a clock pulse. All other registers will not change.

The circuit for this function is called either a demultiplexer (aka decoder or binary to unary converter). The circuit diagram of a 2-4 demultiplexer with enable is shown in figure 3.

If the enable is zero then all outputs are zero. However if the enable is set to 1 then exactly one of the four outputs (D00, D01, D10 and D11) is one, and the others are zero. The output that is 1 is selected by the slelect inputs (S1, S0). Does this remind you of anything? Well, it may remind you of minterms. Because a decoder is a minterm decoder! It can also be considered a binary to unary converter. A three-input decoder has eight outputs, a four-input one has sixteen outputs. Using the functional approach we can build 3-8 demultiplexer in the same way that we constructed the 1-8 multiplexer. The circuit is shown in figure 4.

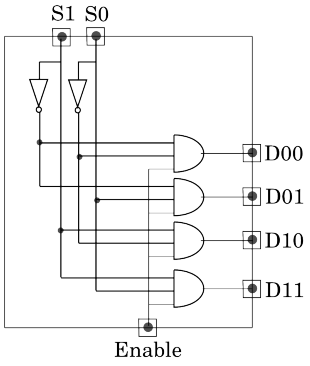


Figure 3: 2 to 4 Decoder

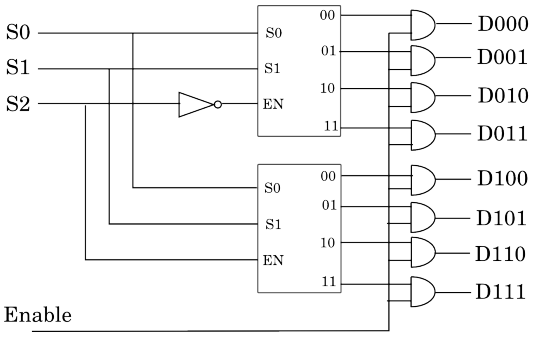


Figure 4: 3 to 8 Decoder

The two 2-4 decoders are neatly combined by making use of their enable inputs, but we need an extra eight AND gates to provide the enable function for the 3-8 decoder. We see that the functional approach to hardware design does not necessarily provide the smallest circuit.

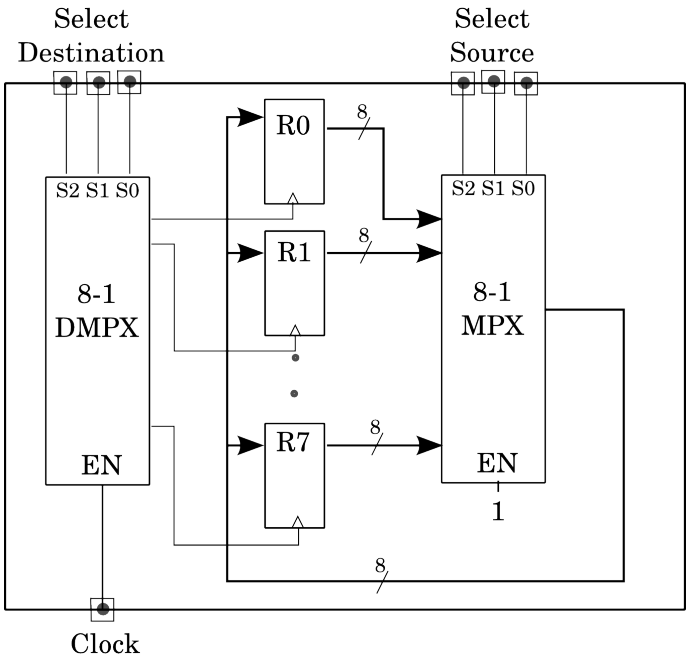
Remembering that the decoder is a minterm generator we note that we could use one to solve the coursework problem in under 1 minute! We use a 4-to-16-decoder,and a sixteen-input OR gate. All we need do is connect the minterms that we need to the OR gate. It may cost more in terms of silicon area, but think of the saving in design time. As we saw in lecture 4, there are special purpose chips called field programmable gate arrays that allow very fast prototype design using this functional design concept.

The Final Circuit

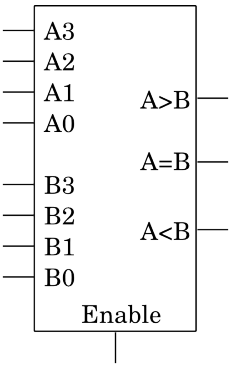
We have shown how the multiplexer and the demultiplexer are used to build our ”register transfer” engine. Remember that with this circuit, when a clock pulse is applied, the contents of any selected register can be transfered into any other register. We write this operation symbolically as:

$$R_{dest} \leftarrow R_{source}$$

The final circuit is:



Comparator Circuits



We will conclude this general discussion on looking at digital circuits from a functional point of view by showing how to build a comparator circuit. The input to a simple binary comparator circuit usually comes from two registers; thus the inputs are considered to be two simple positive binary numbers of "n" bits, say number A and B. There are three outputs and, just like a decoder, exactly one of the outputs is equal to 1. The three outputs indicate the conditions $A > B$, $A = B$, or $A < B$. We can realise (again, functional thinking!) that we need to provide only two outputs, since the third one is the NOR function of the other two.

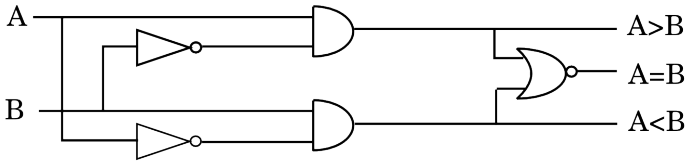
We could design the circuit buy setting up an 8-input (256 line) truth table and attempt to minimise the canonical equation. However, we would not get very far with our K-maps. The other possibility is to write down Boolean equations making use of functional thinking of the sort:

- A is larger than B if A3 is equal to 1 and B3 is equal to 0 (remember, we have only positive binary numbers in the range of 0 to 15)
- A is larger than B if A3 and B3 are equal to each other and A2 is equal to 1 and B2 is equal to 0
- etc

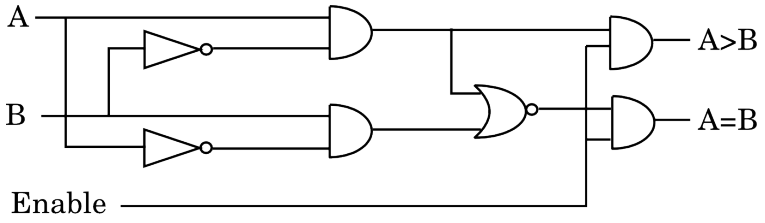
Again, functionally thinking, the fact that $A3 = 1$ and $B3 = 0$ can be expressed by the Boolean equation $A3 \cdot B3' = 1$ and the fact that $A3$ is equal to $B3$ is the Boolean equation $(A3 \oplus B3)' = 1$ which expands to $(A3 \cdot B3 + A3' \cdot B3' = 1)$. Thus, we can write the Boolean equation as:

$$A > B = A3{\cdot}B3' + (A3B3 + A3'{\cdot}B3'){\cdot}((B2A2') + (A2{\cdot}B2 + A2'{\cdot}B2'){\cdot}((A1{\cdot}B1') + (A1{\cdot}B1 + A1'{\cdot}B1'){\cdot}A0{\cdot}B0')))$$

and go from there. But, there is an easier (i.e. functional) way of solving this problem. We can first build a one-bit comparator:



add an enable, and take away the $A < B$ which we don't really need



and then "functionally" build a four-bit one: