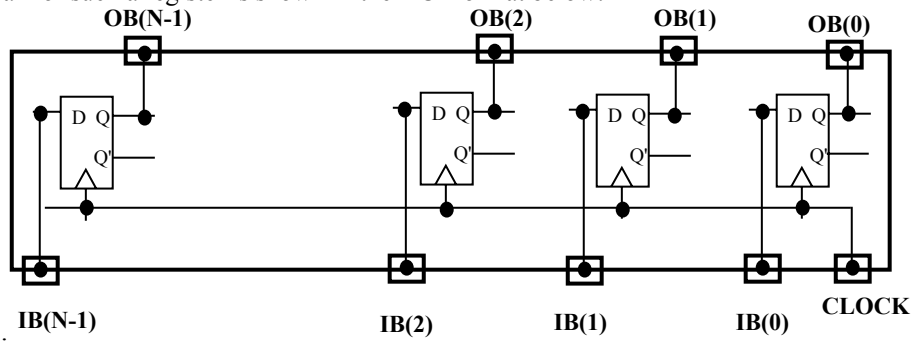
Department of Computing Course
Course DOC 112, Hardware
Lecture 12

Registers, Multiplexers, Decoders, Comparators and What Not

Registers

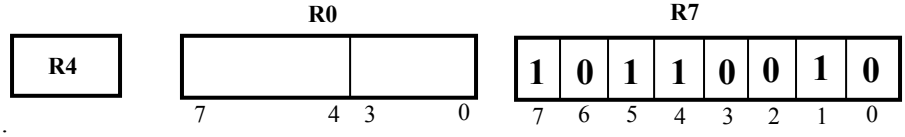
As you know by now from your Computer Architecture course, data processing is usually done on fixed size binary "words". Data are stored in computers in registers which can be thought of simply as collections of **D-type flip-flop**s. The schematic diagram of such a register is shown in the "IC" format below:



By convention we number the bits from 0 to N-1 for a N-bit register and may assign its contents the positive numerical value $\sum((2^n)*OB(n))$. (However, it is important to remember that bits are bits are bits and the interpretation of what they represent can be made up by anybody).

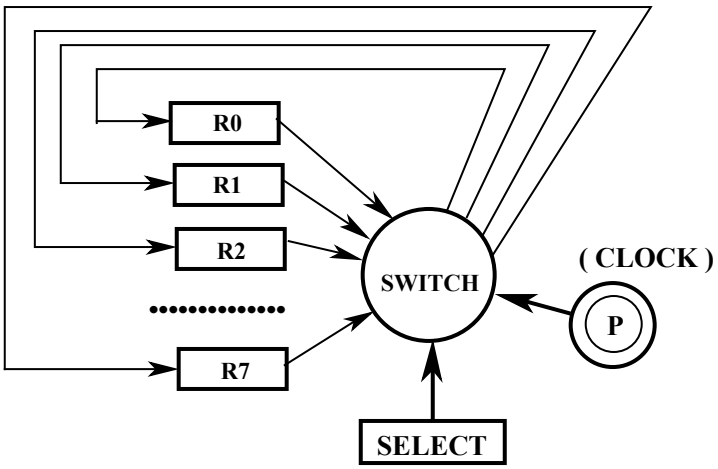
For example, the IC type **SN74273** is such an 8-bit register. It has 20 pins and in addition to the 8 input, 8 output, **CLOCK**, **VCC**, and **GND** pins, it has a pin designated as **CLR** which clears (sets to **0**) all eight bits, a very useful facility some times. (Incidentally, the price of this IC is £1.42)

We also have short hand notations for registers. Three of the most common ones are shown below:

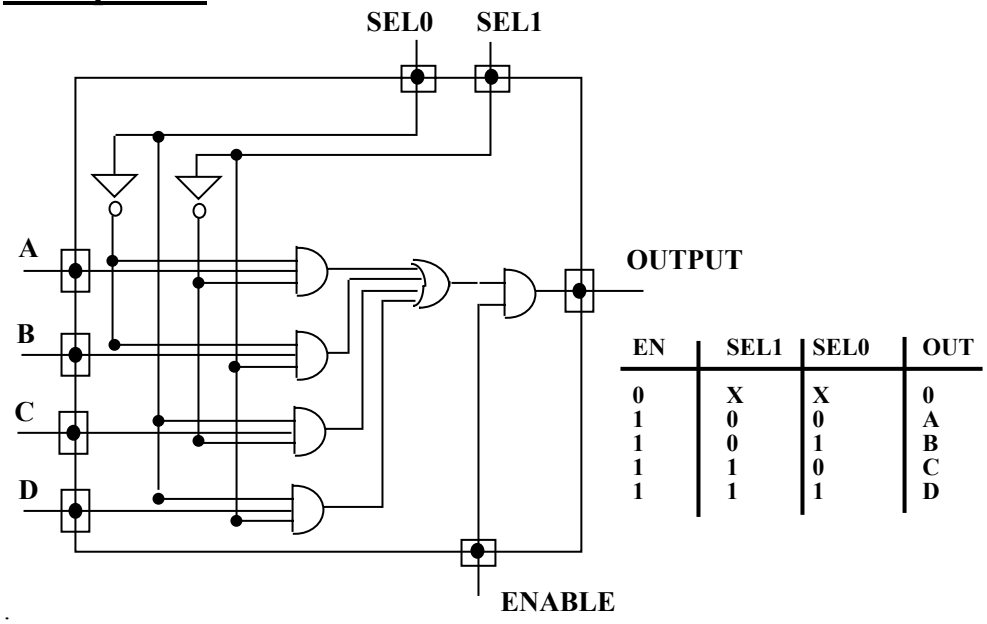


Registers can contain either data or control bits and are the fundamental building blocks of digital computers. One of the most basic operation between registers is **register transfer** which means the copying of the contents of one register into another without the loss of data in the first one.

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:



Multiplexers



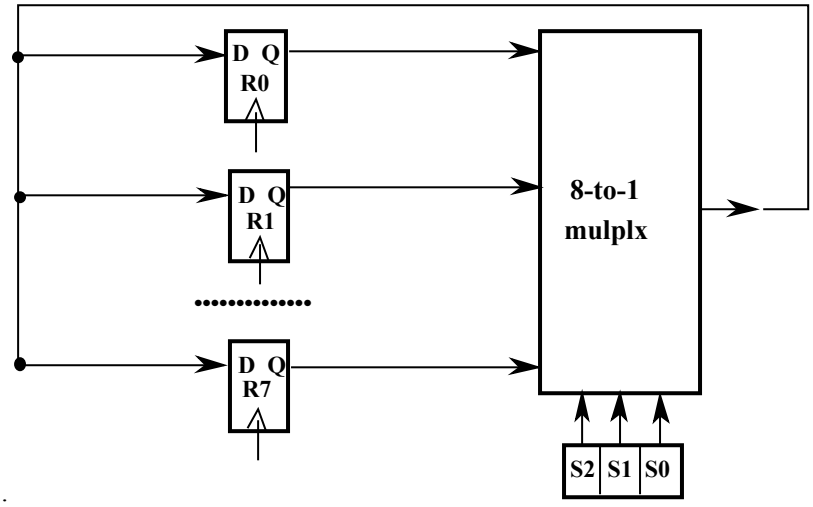
EN	SEL1	SEL0	OUT
0	X	X	0
1	0	0	A
1	0	1	B
1	1	0	C
1	1	1	D

For each push of the button P a clock pulse is generated. The switch is set before the push and it can be set such that the contents of any one register can be transferred to any other register during one clock pulse. Hence, the order of events:

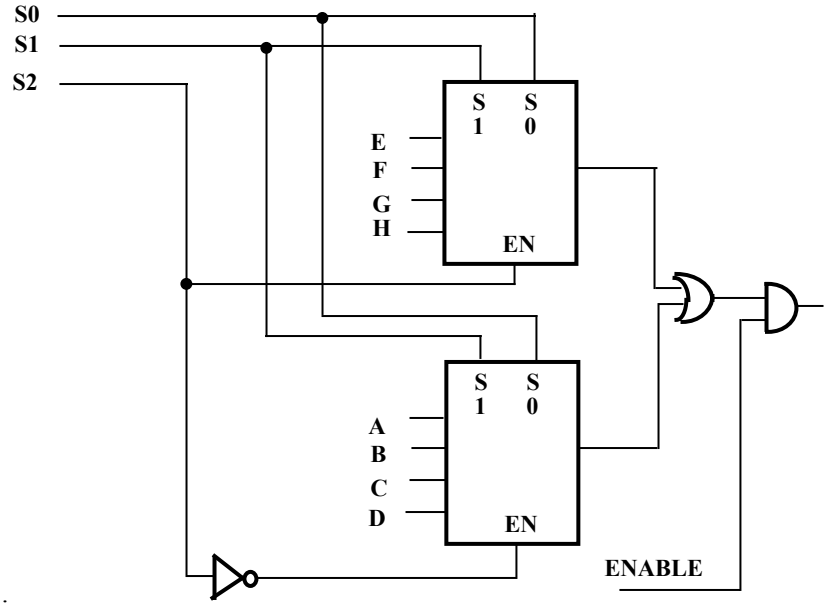
1. **Select the input register.**
2. **Select the output register**
3. **Generate a clock pulse to transfer**

For simplicity, we can assume for now that each register is a **one-bit register**, i.e. a simple **D-type flip-flop**. In order to build this circuit we have to understand how a **multiplexer** and a **demultiplexer** (decoder) works. The whole idea about multiplexers is that the output is equal to one of the inputs when it is enabled. Thus the output side of the switch can be realised with an eight-to-one multiplexer

The Register Transfer Circuit using a 8-1 Multiplexer



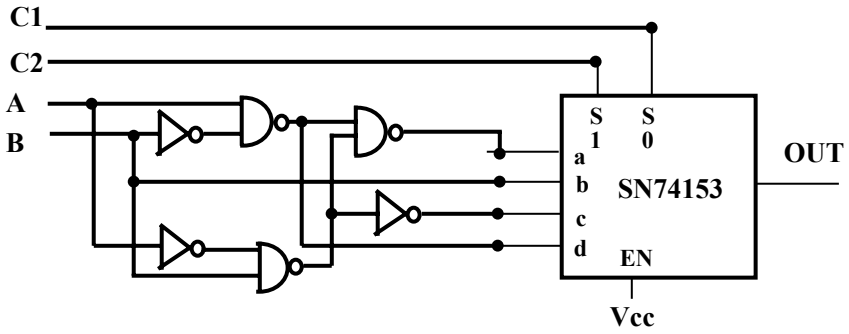
Notice that only one line appears as the output of the multiplexer and this line is connected to the inputs of **all** registers. However, by setting the bits in the "select" register (with bits S0, S1, and S2) any one of the register output may appear as input to all the registers. But, nothing happens yet, because registers are made up of flip-flops and flip-flops require a clock signal before they change their storage outputs. Thus we have solved the problem of selecting any one of the registers as a **source register**, we still have to select a proper destination register (just one specific destination register regardless of the fact that all the inputs are connected).



We also have to see whether one can buy an 8-to-1 multiplexer. The answer is yes, type **SN7400** is just such a device. But, if we had only , say 4-to-1 multiplexers (or we want larger than 8-to-1) with enable inputs, we could use two of these to build an 8-to-1 device. We use functional reasoning to do this and arrive at the following circuit:

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 controls **01**, **A'•B** for **10** and function **(A'+B)** for **11**. According to my fast (and possibly slightly inaccurate) estimation, you would have needed 6 **NAND**, 3 **AND** and 4 **inverter** gates, i.e. 4 **IC**s and estimated cost of £3.24.

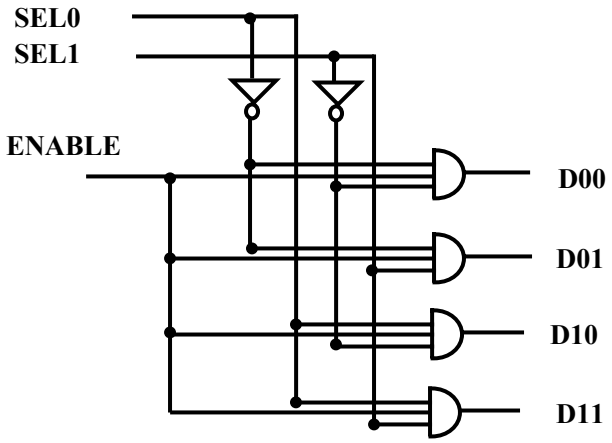
However, noticing that a multiplexer can provide four different functions, and the simplifying facts that **A(XOR)B = (A'•B+A•B')**; and, also that **(A•B')' = A'+B**, you could have built the following circuit:



The control functions select between the four inputs which are generated by a simple circuit. Since the available IC is a type **SN74153** which is £0.90 and has two 4-to-1 multiplexers; therefore, it is 1/2 utilised. The rest of the circuit uses two **IC**s with utilisation factors of 3/4 and 1/2 respectively, so we get a total cost of 3*(0.50) + (3/4+1/2)*0.48 + (1/2)*0.90 and we get £ 2.55. Not only a savings in money but **time!**.

Demultiplexers

Let us get back to our original problem of the general register transfer engine. We have solved the problem (with a multiplexer) of providing all the registers with the output of one selected register (the source register), but now we must apply the clock to only one register (the destination register, that is) in order that the data transfer can take place. There is an **IC** device for this function, it is called either a **demultiplexer** or **decoder**. The circuit diagram of a 2-4 demultiplexer with enable is shown below:

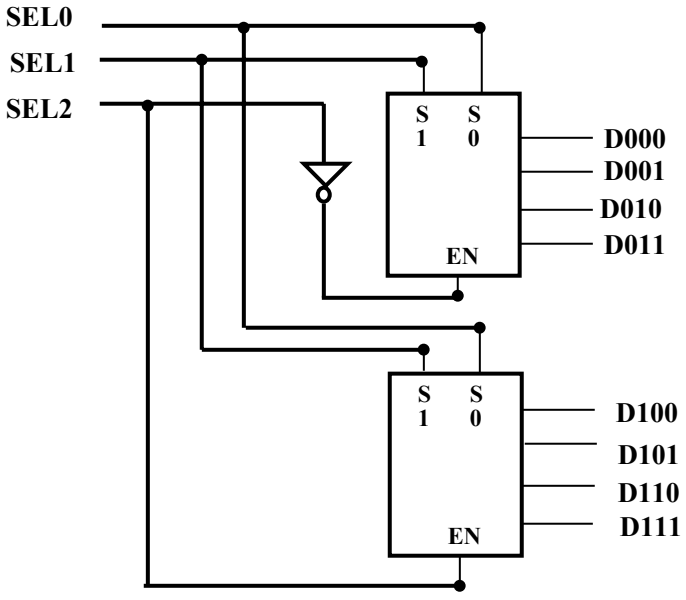


A good way to describe the operation of a decoder/demultiplexer is to look at its functional truth table:

.

EN	SEL1		D00	D01	D10	D11
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

When the enable input is **0** (exactly the same as for all other functional circuits), then all outputs are **0**. However, when the decoder operates properly then exactly one of its outputs is equal to **1**. The selection of the output is determined by the values on the selection lines. Does this remind you of anything? Well, it may remind you of **minterms**. Because a decoder is a minterm generator! Obviously, a three-input (meaning selection input) decoder has eight outputs, a four-input one has sixteen. How do we build a three-input decoder from two-input decoders? Well, all we need is one inverter gate. Look how it is done:



.

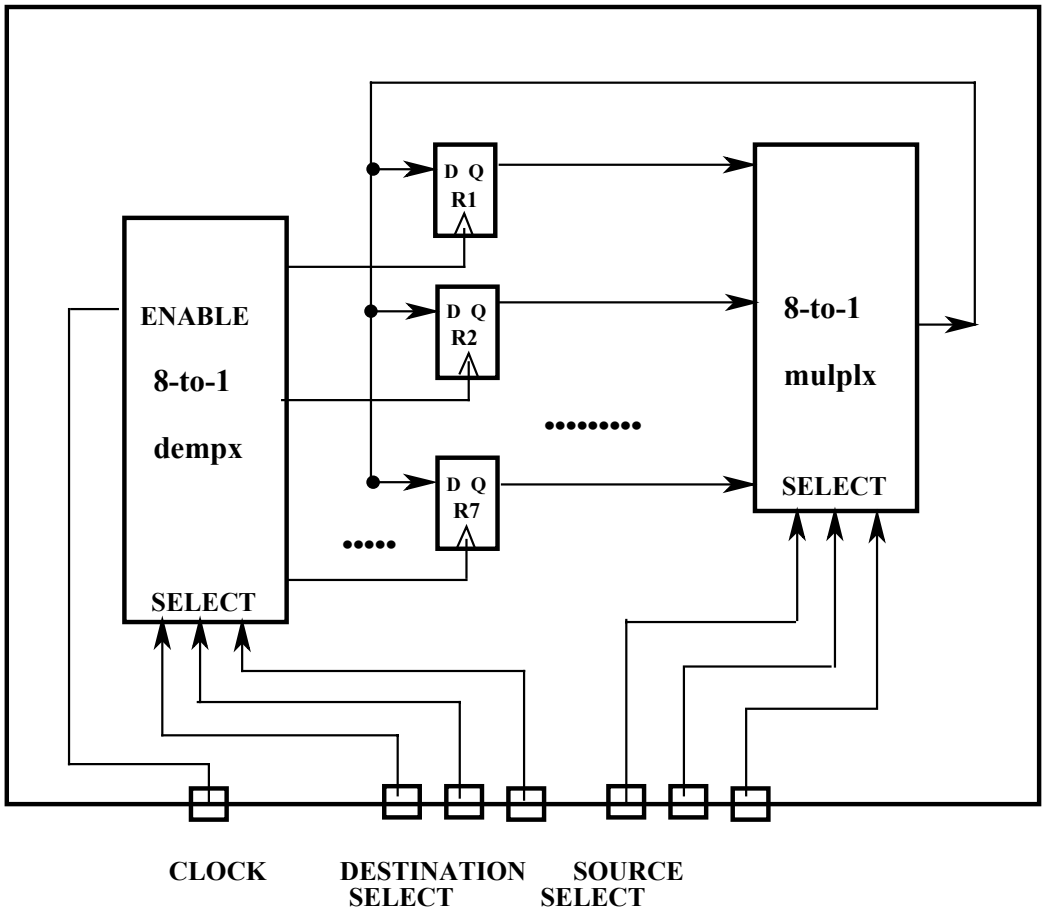
While the three selection inputs do provide eight decoder outputs, we lost our **ENABLE** input. If the **ENABLE** input is required (which it <u>is</u> in our example, as you will see below), then we need eight additional two-input **AND** gates.

There are a wide variety of decoder/demultiplexers. There is a dual 2-to-4 decoder, type **SN74139**, or a 3-to-8 decoder, **SN74138** and even a 4-to-16 decoder (with enable - a device with 24 pins), type **SN74154**, with respective prices of £ 0.90, £ 0.85, and £ 2.30 respectively. Remembering that the decoder is a minterm generator, in order to redo our assigned course work (four inputs, one output), we can do no work at all by buying a 4-to-16-decoder, and a sixteen-input **OR** gate. It may cost us more but no circuits to design. And it will solve any one of the 110 circuits you have been working on!

The Final Circuit

We can show now how the multiplexer and the demultiplexer are used to build our "**register transfer**" engine. Remember that with this circuit, when a clock pulse applied, the contents of any selected register can be transfered into any other register. A computer operation which we show symbolically as:

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

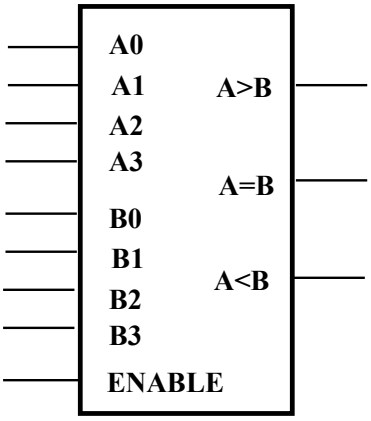


. .

Comparator Circuits (or: creating computer intelligence).

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, only one of the outputs is equal to **1**. The three outputs indicate the fact whether **A>B**, **A=B**, or **A<B**. This starts to look more and more like intelligence in hardware.

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. Let us first look at the block diagram of a 4-bit comparator:



.

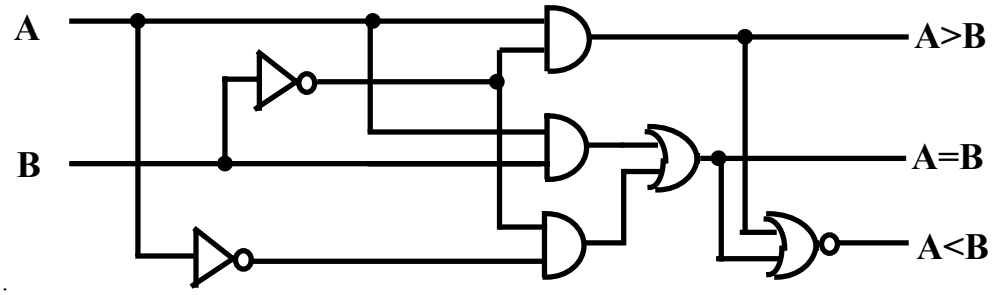
We could build now the circuit buy setting up the 8-input, twice 256-output truth tables and start minimising. Would not get very far with our K-maps. The other possibility is to write down the Boolean equation by functional thinking of the sort: **A** is larger than **B** if **A3 is equal to 1** and **B3 is equal to 0** (remember, we have positive binary numbers in the range of 0 to 15); or if **A3 and B3 are equal** to each other and **A2 is equal to 1** and **B2 is equal to 0**; and so forth.

Again, functionally thinking, the fact that **A3=1** and **B3=0** can be expressed by the Boolean term **A•B'** and the fact that **A3** equal to **B3** with **[A3<+>B3]'** or the expression **(A3•B3+A3'•B3')**. Thus, we can write the Boolean equation as:

A>B = A3•B3' + (A3•B3+A3'•B3')•((B2•A2') + (A2•B2+A2'+B2')•((A1•B1') +

(A1•B1+A1'•B1')•A0•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:



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