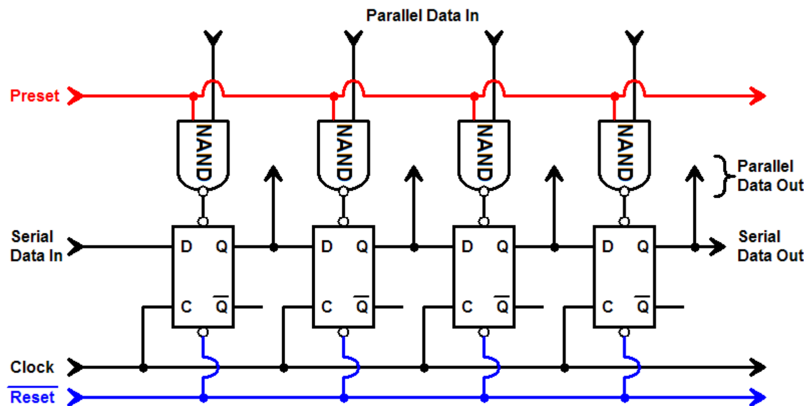
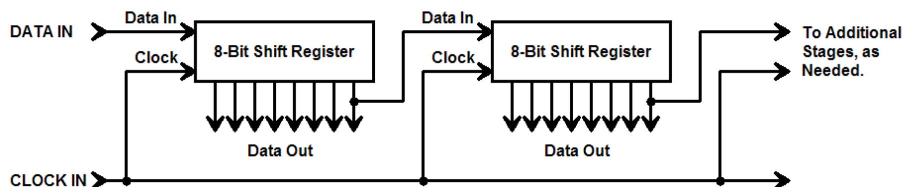
Lecture #25 – April 9, 2004

Shift Registers

We have seen shift registers in earlier lectures as serial-input-only (one bit at a time) devices, but with a little bit of additional circuitry they can be loaded in parallel (all bits at the same time). In the following circuit an extra NAND-gate has been connected to the Set input of each flip-flop in the shift register. One input from each NAND-gate is connected into a single control line, called the Preset line. When the Preset line is low (0) all the NAND-gate outputs are forced high (1), which is the resting state for the Set inputs to the flip-flops (the Set line must go to 0 to set the flip-flop). When the Preset line goes high, the NAND-gates effectively turn into NOT-gates and invert the parallel inputs, passing those inverted inputs through to the Set lines of the flip-flops. Thus, any parallel inputs equal to 1 set their corresponding flip-flops to 1. Once set, the Preset line must go back to 0 before any shifting can take place.



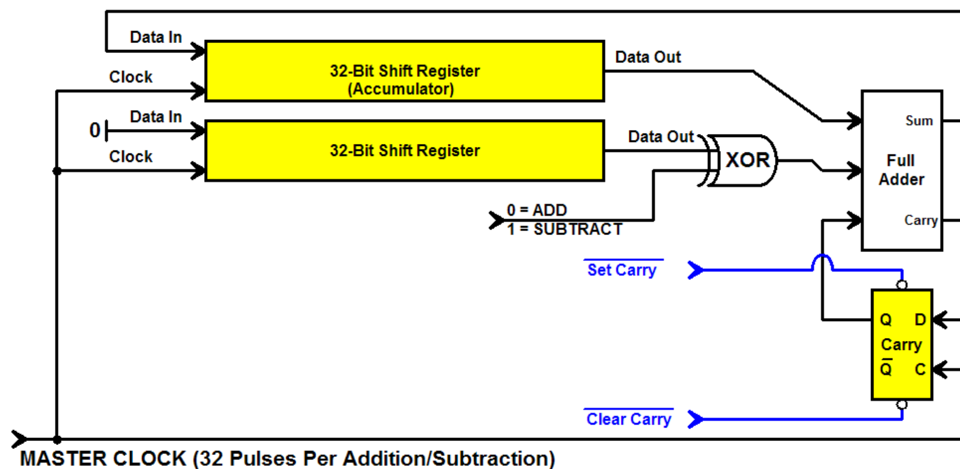
Shift registers with some fixed number of bits are packaged into integrated circuit chips. The number of bits is a function of the number of pins (wires) leading out of the chip, whether or not the output of every bit leads to a wire, whether or not parallel inputs are possible, and how many control lines are necessary (including the serial data in and clock lines). In the following scenario each chip contains eight shift register bits, parallel outputs, the serial data input line, and the clock line. Should we need a shift register containing more than eight bits, we can create such a device by connecting the rightmost parallel output bit of one shift register to the serial data input line of the next. This approach is continued to create a shift register length of any multiple of the base size, as long as all clock lines are connected together to drive all register bits simultaneously. Tapping some bit other than the rightmost allows registers of arbitrary lengths.



Serial Adder

So, let us assume that we have two 32-bit shift register modules. (See how quickly we go up the levels of abstraction from gates to flip-flops to shift registers to modules?) If the outputs of the two shift registers go into a full adder where the sum output from the adder recirculates back to the serial data input of one shift register and the carry output drives the third full adder input through an extra flip-flop (essentially a 1-bit shift register), then clocking the system with 32 pulses will add one shift register's value to the other. The extra flip-flop creates a one-bit delay so that a carry out of one bit gets added to the sum of the next on the next clock pulse.

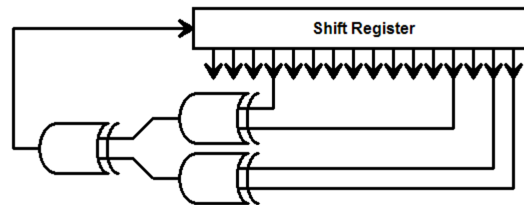
The shift register that receives the sum is effectively the accumulator register. If the other register is fed through an XOR-gate, its output can be selectively passed through to the adder unchanged or complemented. To add, you initialize the carry bit to 0 and set the XOR-gate control line to 0 (pass through). To subtract you initialize the carry bit to 1 and set the XOR-gate control line to 1 (complement), thus generating the two's complement of the second operand one bit at a time. Once an add or subtract is set up, the first clock pulse shifts both registers right by one bit, and shifts one bit of the result into the left bit of the accumulator. One bit of the result shifts in to the accumulator every clock pulse.



The advantage of this approach is that you need only a single full adder to add any number of bits, instead of one full adder per bit as in the earlier parallel adder/subtractor circuit. The major disadvantage of this approach should be fairly obvious: you are trading simplicity of hardware for longer execution times. In the earlier parallel adder the result is complete as soon as the carry values *passively* ripple from the rightmost bits through the leftmost bits, but in this version the sum is not complete until all bits have *actively* shifted through the registers. Lengthening the registers to accommodate larger numbers only exacerbates the problem. Serial adders were used in some early computers to reduce hardware costs, but those computers were too slow to be very successful commercially. (To be completely honest, parallel ripple-carry adders aren't used very often today either; faster adders that use a ***carry-look-ahead*** technique are preferred.)

Pseudo-Random Number Generators

Shift registers are also used to generate bit sequences that are statistically “nearly” random. By tapping output bits at certain places in the shift register and feeding those bits through a tree of XOR-gates back into the serial data input line, a shift register of N bits in length can generate a sequence of 2^N-1 pseudo-random values. The sequence repeats after 2^N-1 clock pulses, so the longer the shift register is the longer the sequence will be before it repeats. Pains must be taken to prevent the shift register from ever containing zero, as its value will never change from then on as a result of the shifting-XOR process. Here is a typical setup for a 16-bit shift register, which generates a pseudo-random sequence of length 65535.



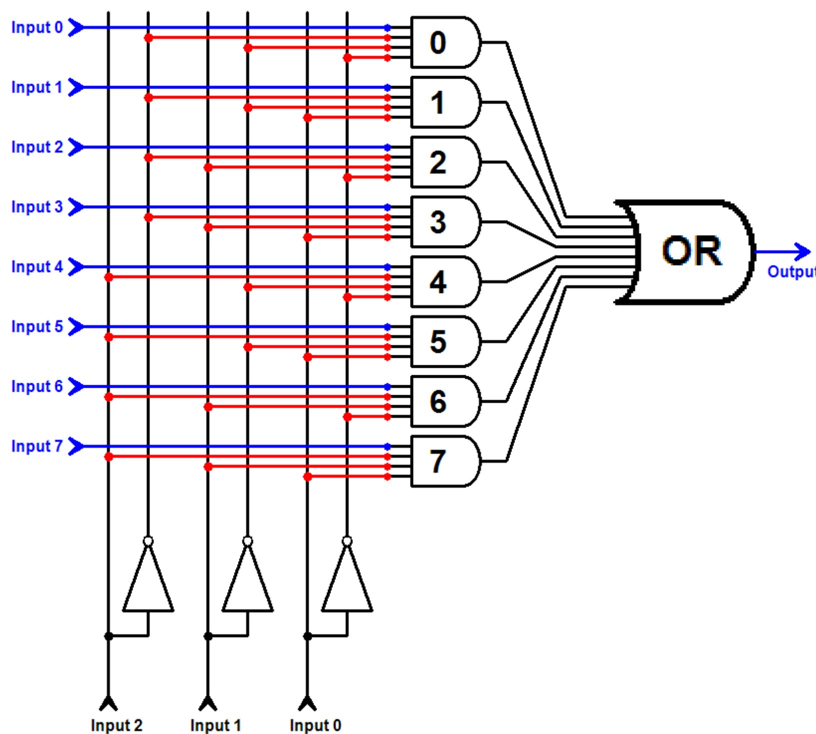
Those values are then further processed to be truly useful. Treated as an N-bit pure fraction, the shift register contents are always greater than zero and less than one. To generate a random integer, this fraction is then *multiplied* by a scale factor and the result truncated to an integer. Treated instead as an integer, the shift register contents are *divided* by a scale factor and the remainder from the division becomes the desired random result. All of these processes are easily performed in assembly language.

Pseudo-random numbers can also be used for encryption purposes. A plain-text message can be viewed as a long string of bits. Before we transmit each bit we first XOR it with the output bit of a pseudo-random number generator; upon receipt the bit is decrypted by XOR-ing it again with the output of an *identical* pseudo-random number generator. While in transit the bit stream may be intercepted by an “enemy” without fear that the message will be broken.

Both pseudo-random generators must start with the same value at the same time, and must be clocked at the same rate, or the process fails. The starting value is the ***encryption key*** for this system, which must be known by both sender and receiver; getting the key from sender to receiver securely can be complicated, and is a topic for another day. Also, the shorter the pseudo-random sequence the more likely that an enemy will be able to crack the code through brute-force techniques. That, too, is a topic for another day.

Multiplexers and Memory

In an earlier lecture we examined a ***demultiplexer*** circuit; an N-bit demultiplexer *asserts* one of 2^N outputs. Among other uses demultiplexers are used to select the word lines in a grid of memory cells. The opposite of a demultiplexer is a ***multiplexer***; a device that *selects* one of 2^N inputs. A multiplexer is constructed from a demultiplexer by the inclusion of an extra input on each AND-gate (forming the 2^N inputs), and combining all the AND-gate outputs together through a large OR-gate. The address lines select one of the AND-gates as in a demultiplexer, so all but one of the AND-gates' outputs are forced to 0, and the output of the remaining AND-gate is controlled by its extra input line. That input line's value is then copied to the output of the OR-gate. Here is the circuit for an 8-input, 1-output multiplexer.



Multiplexers and demultiplexers are used together in memory systems to reduce the amount of required hardware. At one extreme, a complete demultiplexer and no multiplexer is used to pick one of 2^N memory word lines. Removing 1 address bit from the demultiplexer cuts the demultiplexer hardware roughly in half, but the memory must fetch twice as many bits as before and a set of 2:1 multiplexers must select the desired group of bits from those fetched. Removing 2 address bits cuts the demultiplexer hardware in quarter, but now a set of 4:1 multiplexers is required. At the other extreme, no demultiplexer is required at all, all memory bits are fetched at once, and the hardware load is entirely on the multiplexers. Somewhere in the middle is that magic "sweet spot" which minimizes the total number of required gates. We will look at how to determine the sweet spot in a future lecture.