Design Example: 4-bit Multiplier

Consider how we normally multiply numbers:

```
      123
x    264
      492
     7380
    24600
    32472
```

Binary multiplication is similar. (Note that the product of two 4-bit numbers is potentially an 8-bit number).

```
Multiplicand      1011
Multiplier    x   0110
                  0000     Note that each of these lines
                 10110     here is either zero or just a
                101100     shifted venison of the
               0000000     multiplicand.
Product        1000010
```

Note that in binary multiplication, the processes involves shifting the multiplicand, and adding the shifted multiplicand or zero. Each bit of the multiplier determines whether a 0 is added or a shifter version of the multiplicand (0 implies zero added, 1 implies shifted multiplicand added). Thus, we can infer a basic shift-and-add algorithm to implement unsigned binary multiplication.

Can we design a circuit to implement this? We must start with the datapath.

Datapath

Well, because of the way the multiplication works, the algorithm involves *shifting* and *adding*.

- This tells us that we'll probably need shift registers and an adder.

We also need somewhere to store the product.

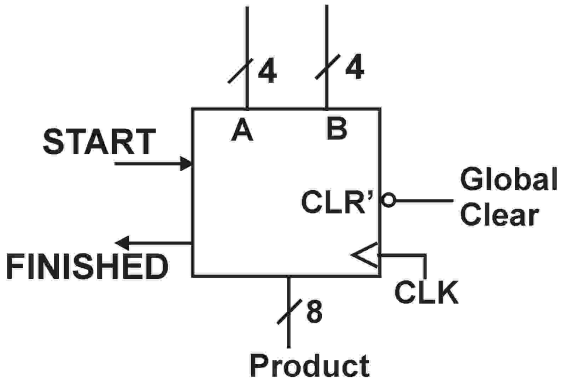
- We can use registers or flip-flops. *Registers* will allow us to hold the value as long as we like.

Sometimes, we add zero, sometimes we add a shifted version of the multiplicand.

- this suggests AND gates will be needed

So, say we consider the expression we want as P = A x B, where P is the 8-bit product, A is the 4-bit multiplicand, and B is the 4-bit multiplier.

We define the system to work as follows. A and B are input into the system. The system waits for a user to assert a START signal. At that point, the system begins multiplication, and when finished, asserts a FINISHED signal. When the FINISHED signal is asserted, the final value of the product is available on the outputs. The START signal must go low before and then high again before a new multiplication can be started.



Our basic approach, from our understanding of how multiplication works, is that it is just the sum of shifted versions of the multiplicand or zeros. If, instead of adding all the terms at the same time, we use one adder over and over again, we can save on a lot of hardware. So, we do one addition each clock cycle.

We initialize the product register to zero. Then, we add the first term. (In the above binary multiplication example, we add zero). On the next clock cycle, we add the second term (10110 in the example above), and so on. Thus, we complete the multiplication in four clock cycles.

Let's think about the multiplicand first. We've repeatedly said that we add shifted versions of the multiplicand to generate the product. This leads us to think that the multiplicand should be loaded into a shift register at the beginning of our multiplication.

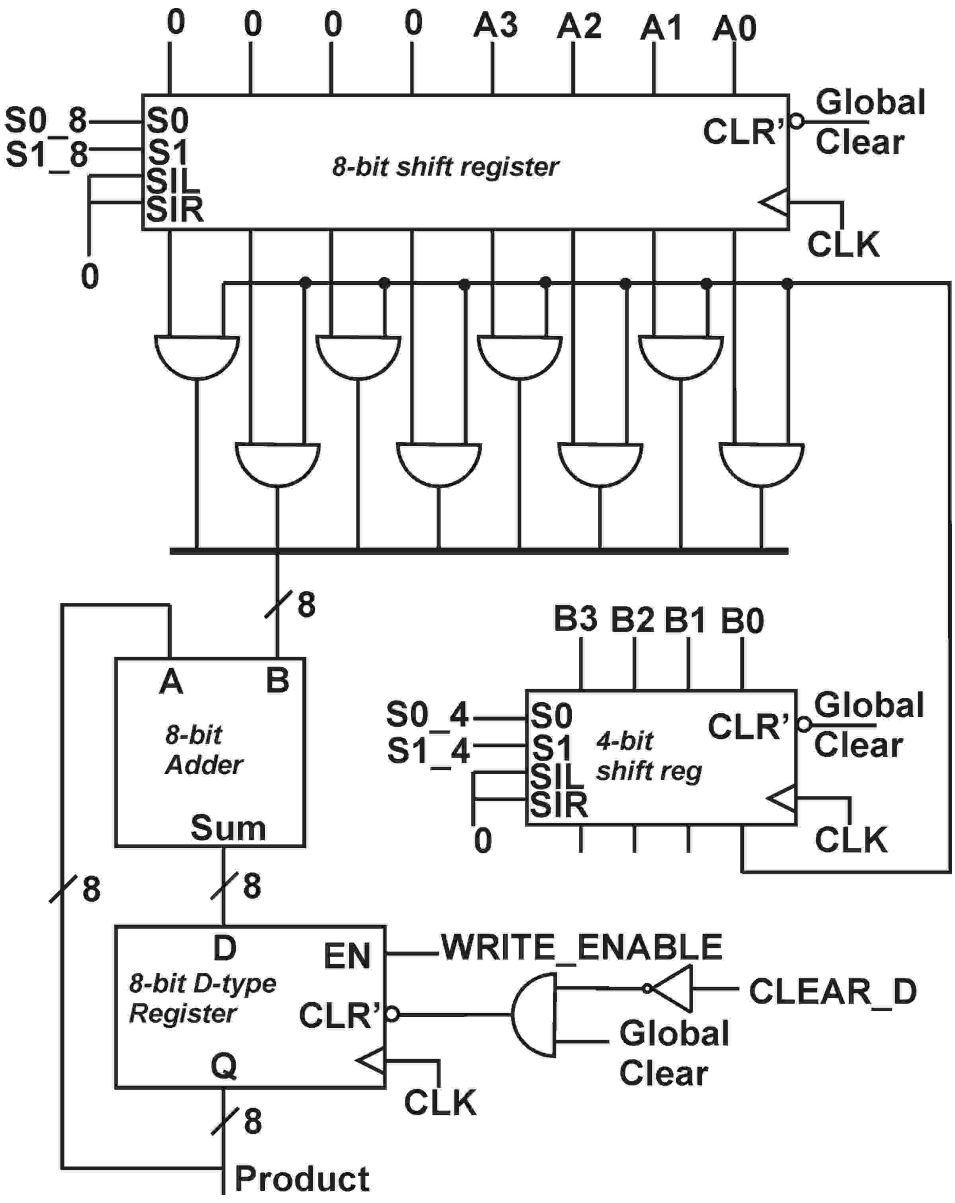
How big should the shift register be? Well, it needs to be at least 7 bits, since if the most significant bit of the multiplier is a 1, a version of the multiplicand shifted left 3 places must be added. However, we might as well use an 8-bit shift register, since 7-bits is not likely a standard part. Each clock cycle will shift the register one place left.

Also, since the product is up to 8-bits long, the adder used to generate the product must be 8-bits in size. Likewise, the register to hold the product must be 8-bits.

What about the multiplier? We use it to determine whether we add zero or a version of the multiplicand into the product. In each clock cycle of the multiplication, we check one bit, and use that to determine whether the output of the multiplicand shift register is passed into the adder, or zeroed. We can load the multiplier into a shift register, too. But this one only need be 4 bits long, and shifts to the right. That way, we only need to look at the least significant bit, and use that to AND the output of the multiplicand shift register.

(Note that we assume a general or universal shift register exists, something like the 74194. This shift register we assume has active-low clear, parallel inputs and outputs, and shift-in-left and shift-in-right inputs, and two control inputs S0 and S1. If S0S1 is 00, the shift register holds; if 01, the register shifts right; 10 shifts left; and 11 causes parallel load.)

So we have our datapath architecture as depicted here.



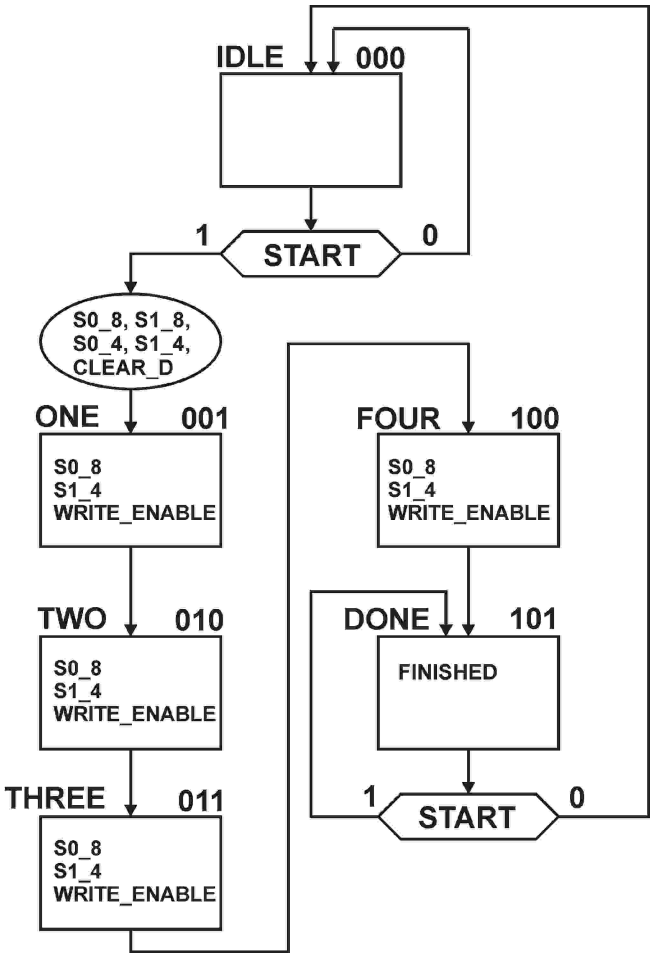
Controller

Now that we have a defined datapath architecture, we can do an ASM chart for the control.

The controller has several responsibilities:

- must iterate four times
- must load A and B into their shift registers
- must wait for START input
- must assert the FINISHED signal well done
- must generate the outputs to control the two shift registers: S0_8, S1_8, S0_4, S1_4
- must generate the WRITE_ENABLE signal to the product register when needed
- must clear the product register when starting a new multiplication (CLEAR_D)
- must still allow global clear from a switch

See the following ASM chart for one implementation.



Now we can derive the equations.

State sequences:

Current state	C_k	B_k	A_k	Next state	C_{k+1}	B_{k+1}	A_{k+1}	Conditions
IDLE	0	0	0	IDLE	0	0	0	START’
				ONE	0	0	1	START
ONE	0	0	1	TWO	0	1	0	
TWO	0	1	0	THREE	0	1	1	
THREE	0	1	1	FOUR	1	0	0	
FOUR	1	0	0	DONE	1	0	1	
DONE	1	0	1	DONE	1	0	1	START
				IDLE	0	0	0	START’

C_{k+1} = THREE + FOUR + DONE·START
= C_k’·B_k·A_k + C_k·B_k’·A_k’ + C_k·B_k’·A_k·START

B_{k+1} = ONE + TWO
= C_k’·B_k’·A_k + C_k’·B_k·A_k’

A_{k+1} = IDLE·START + TWO + FOUR + DONE·START
= C_k’·B_k’·A_k’·START + C_k’·B_k·A_k’ + C_k·B_k’·A_k’ + C_k·B_k’·A_k·START

The outputs are mostly generated from the states and the inputs. Most of these can likely be minimized to some extent.

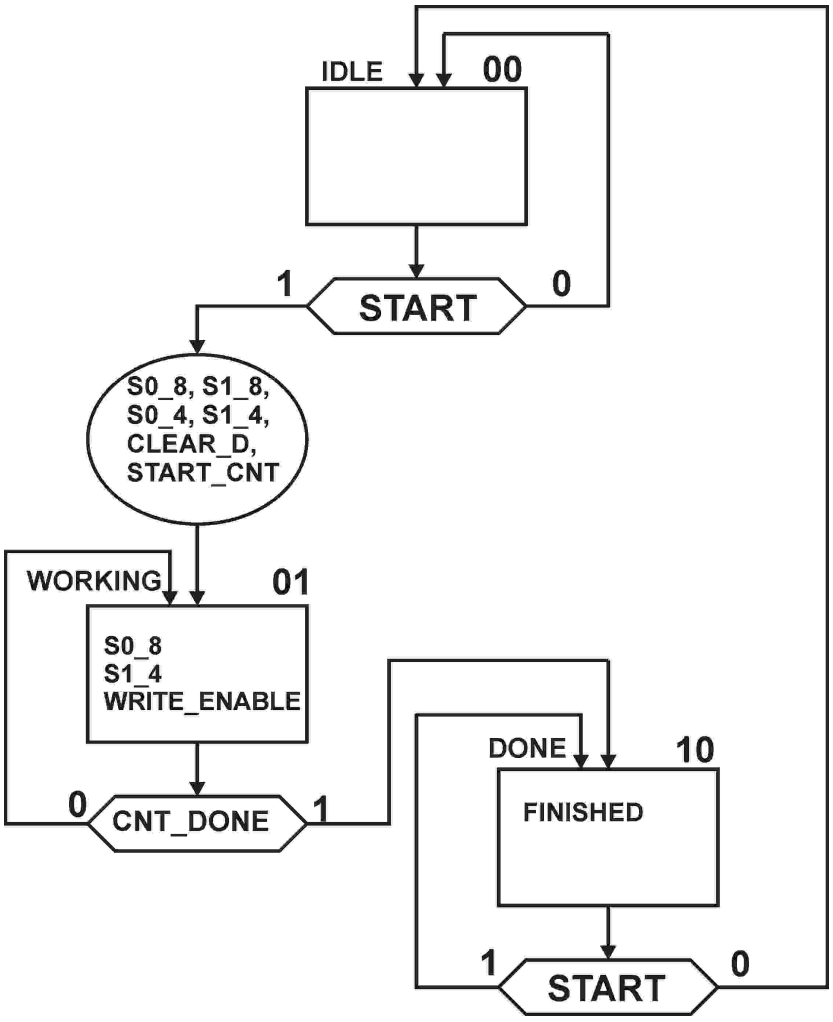
WRITE_ENABLE = ONE + TWO + THREE + FOUR
= C_k’·B_k’·A_k + C_k’·B_k·A_k’ + C_k’·B_k·A_k + C_k·B_k’·A_k’

S1_8 = S0_4 = CLEAR_D = IDLE·START = C_k’·B_k’·A_k’·START

S0_8 = S1_4 = IDLE·START + ONE + TWO + THREE + FOUR
= C_k’·B_k’·A_k’·START + C_k’·B_k’·A_k + C_k’·B_k·A_k’ + C_k’·B_k·A_k + C_k·B_k’·A_k’

FINISHED = DONE = C_k·B_k’·A_k

It is possible to have a smaller implementation by utilizing a 2-bit counter and staying in a single multiplying state until all 4 iterations have been done. Try working out this alternate on your own, then see the following ASM chart and solution.



State sequences for alternate implementation:

Current state	B_k	A_k	Next state	B_{k+1}	A_{k+1}	Conditions
IDLE	0	0	IDLE	0	0	START’
			WORKING	0	1	START
WORKING	0	1	WORKING	0	1	CNT_DONE’
			DONE	1	0	CNT_DONE
DONE	1	0	DONE	1	0	START
			IDLE	0	0	START’

The state equations are easier, since there are fewer states.

B_{k+1} = WORKING·CNT_DONE + DONE·START
= B_k’·A_k·CNT_DONE + B_k·A_k’·START

A_{k+1} = IDLE·START + WORKING·CNT_DONE’
= B_k’·A_k’·START + B_k’·A_k·CNT_DONE’

The outputs in this case are also easier to generate.

WRITE_ENABLE = WORKING = B_k’·A_k

S1_8 = S0_4 = CLEAR_D = IDLE·START = B_k’·A_k’·START

S0_8 = S1_4 = IDLE·START + WORKING
= B_k’·A_k’·START + B_k’·A_k

FINISHED = DONE = B_k·A_k’

