Digital System Design

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc350

Simon Fraser University

Slide Set: 11

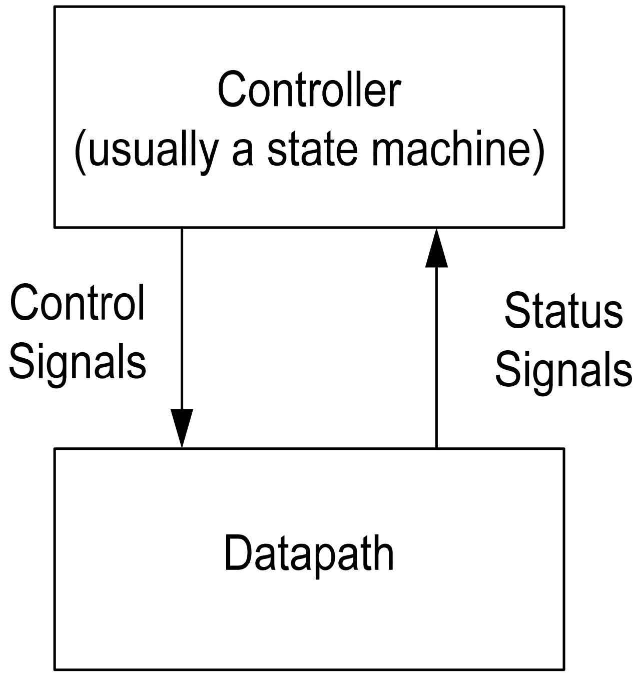
Date: March 9, 2009

Slide Set Overview

- Datapath Circuits
  - Large digital systems are more than state machines and combinational logic. Generally these systems can be divided into two parts:
    - Control
    - Datapath
  - We'll use examples to understand how to do this:
    - There is no real "recipe" for designing these things, but with experience, you get to be good at it.

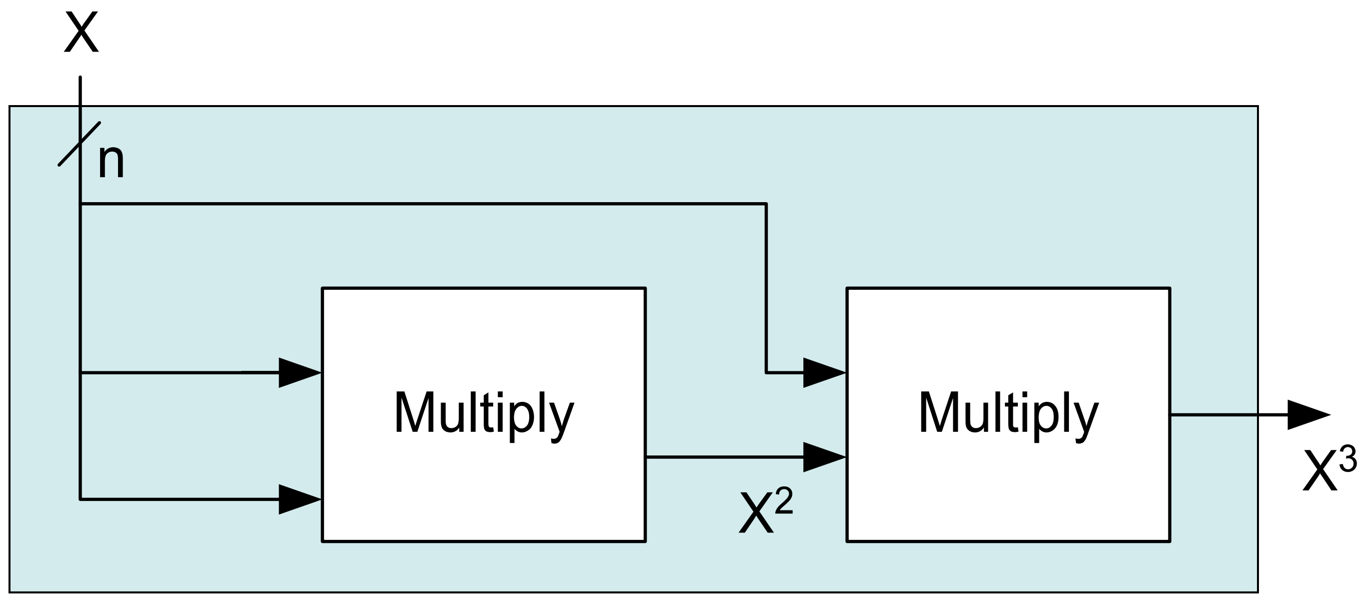
Real Systems

- All but the simplest systems have two parts:



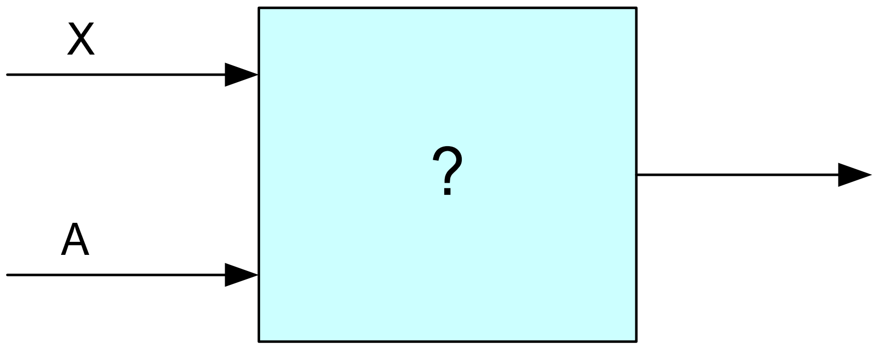
Exponent

- Suppose we want to build a circuit to calculate X^3
  - X is an n-bit input, and assume that the result also fits in n-bits for now
  - This is a relatively simple circuit



Exponent: A bit more complicated

- What if we want to compute X^A where X and A are both inputs?



- If A was fixed, we could figure out how many multipliers we need (as in the previous example)
- But, during the operation of this circuit, suppose A can change. How do we know how many hardware units to put down?

Exponent: A bit more complicated

The algorithm to be implemented in this block:

```
P = 1;  CNT = A;
while (CNT > 0) do
    P = P * X;
    CNT = CNT – 1;
end while;
```

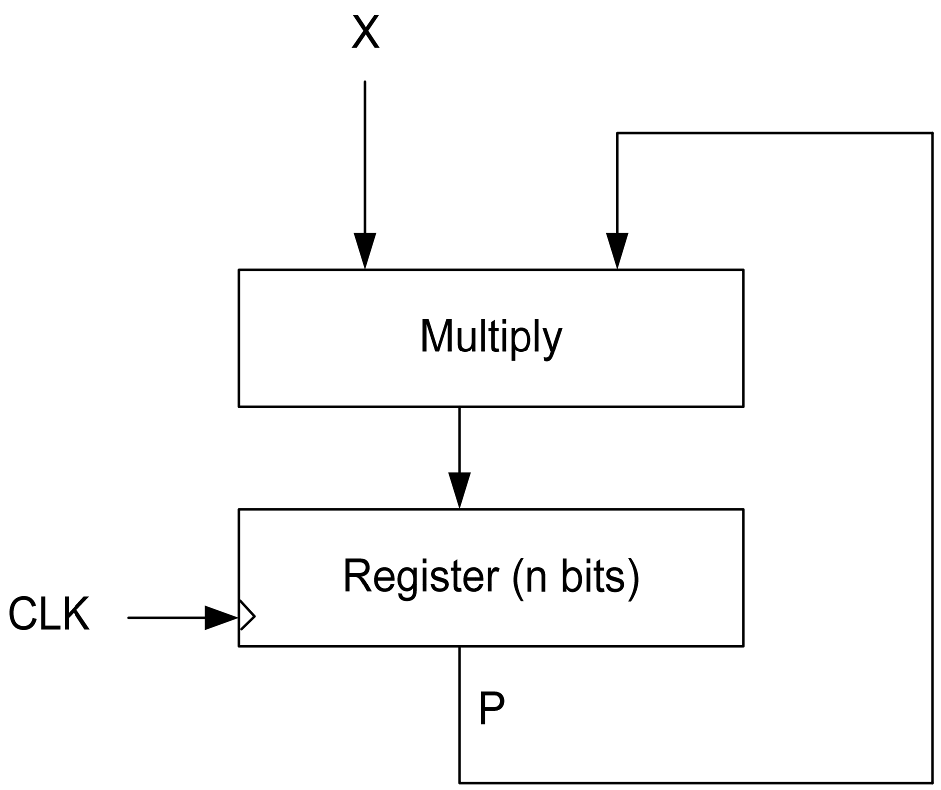
Note 1: this isn't VHDL or C, it just is pseudo-code to illustrate the algorithm.

Note 2: We could write this in VHDL, but it would ***not*** be ***synthesizable***. So, we have to design it using smaller processes (each one synthesizable)

Rule that has never mattered before**: A synthesizable process can only describe what happens in one clock cycle.** This would take more than one clock cycle. So, it would not be synthesizable

Exponent: A bit more complicated

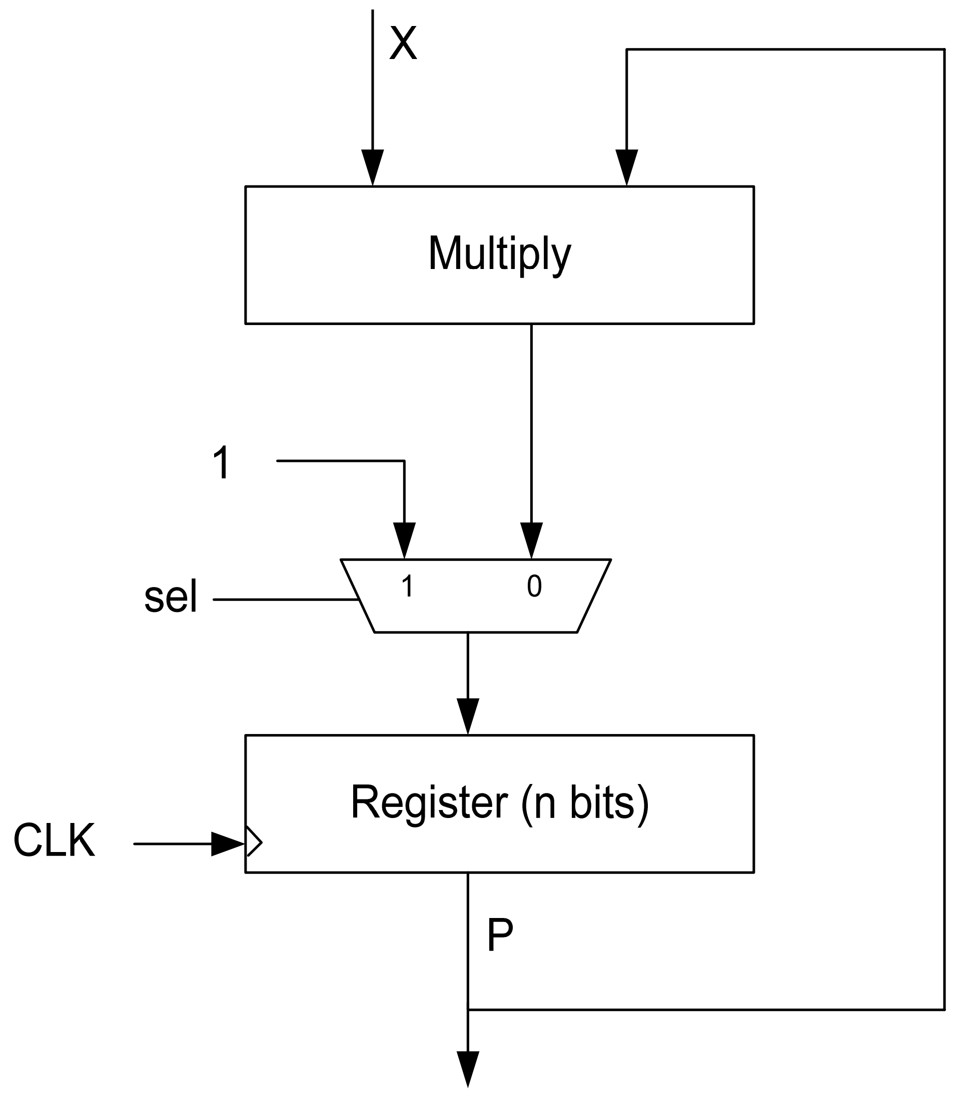
Consider this simple datapath



If we let this run for A+1 clock cycles, we will produce the desired result.

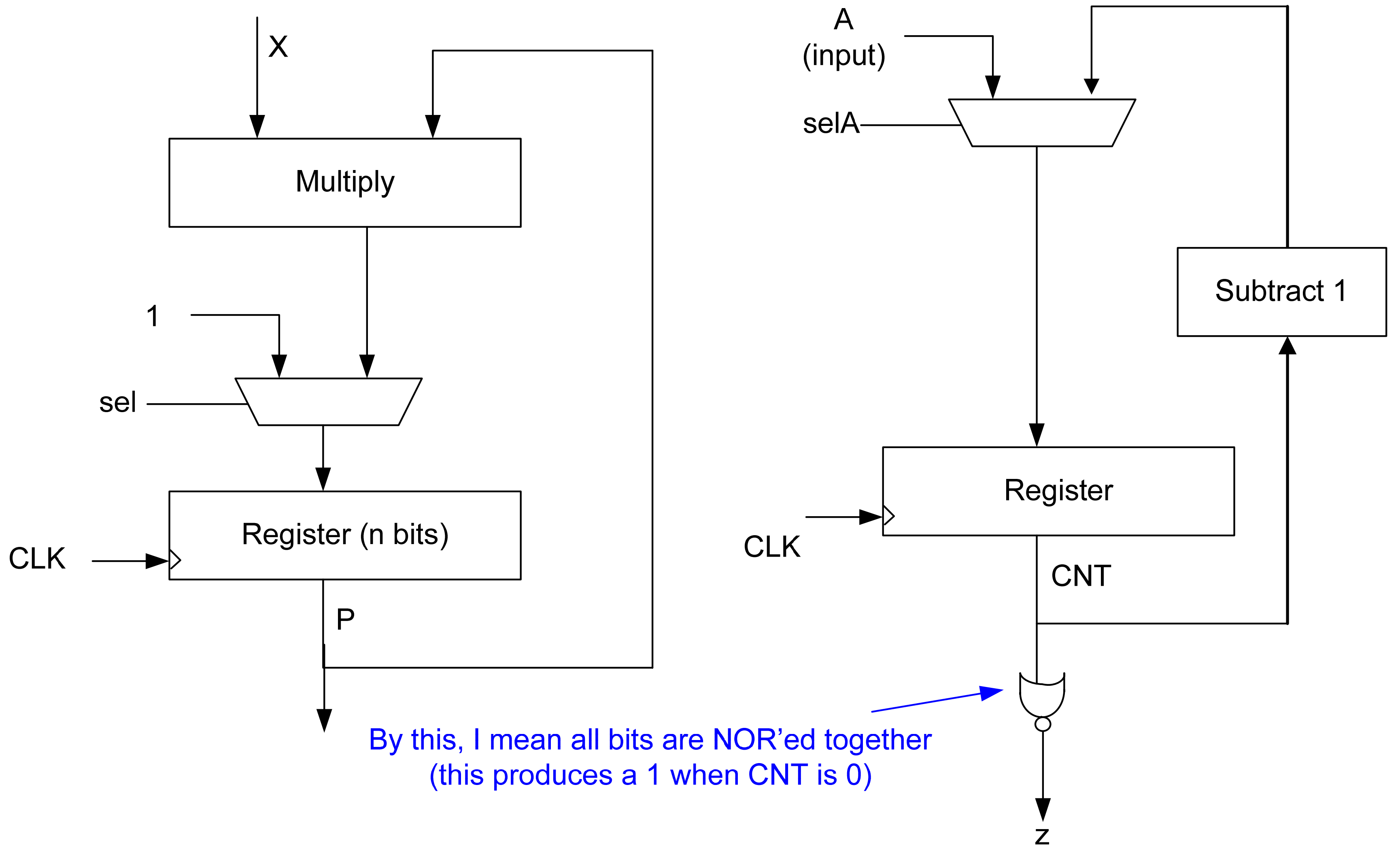
This will work for any value of A

Need a way to initialize P to 1 at the start:



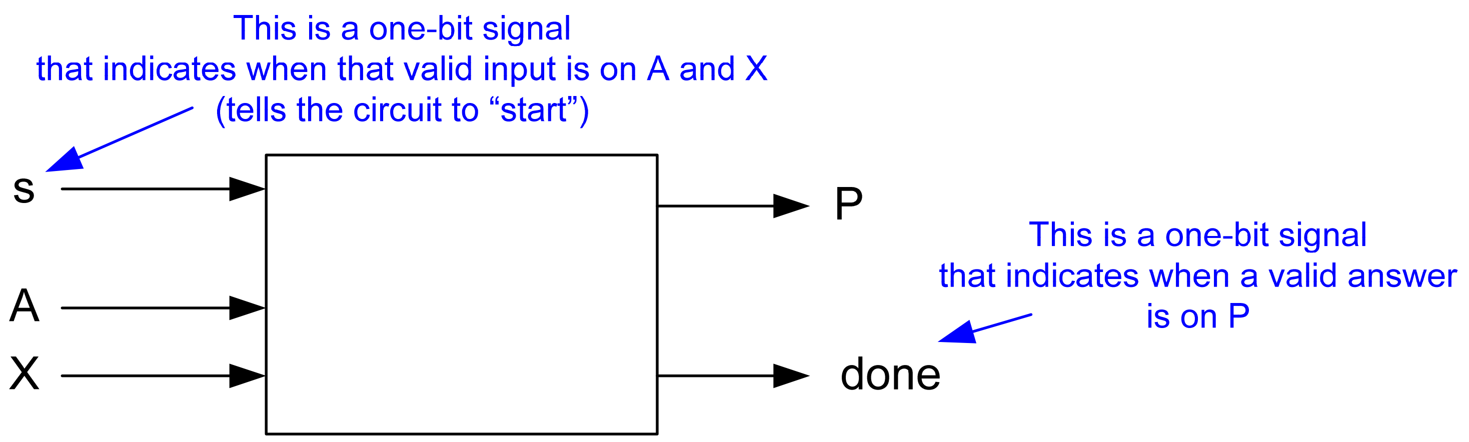
First cycle, set sel to 1, and this will initialize P to 1

We have to let this run for A cycles. We need some sort of counter to keep track of this.



By this, I mean all bits are NOR'ed together (this produces a 1 when CNT is 0)

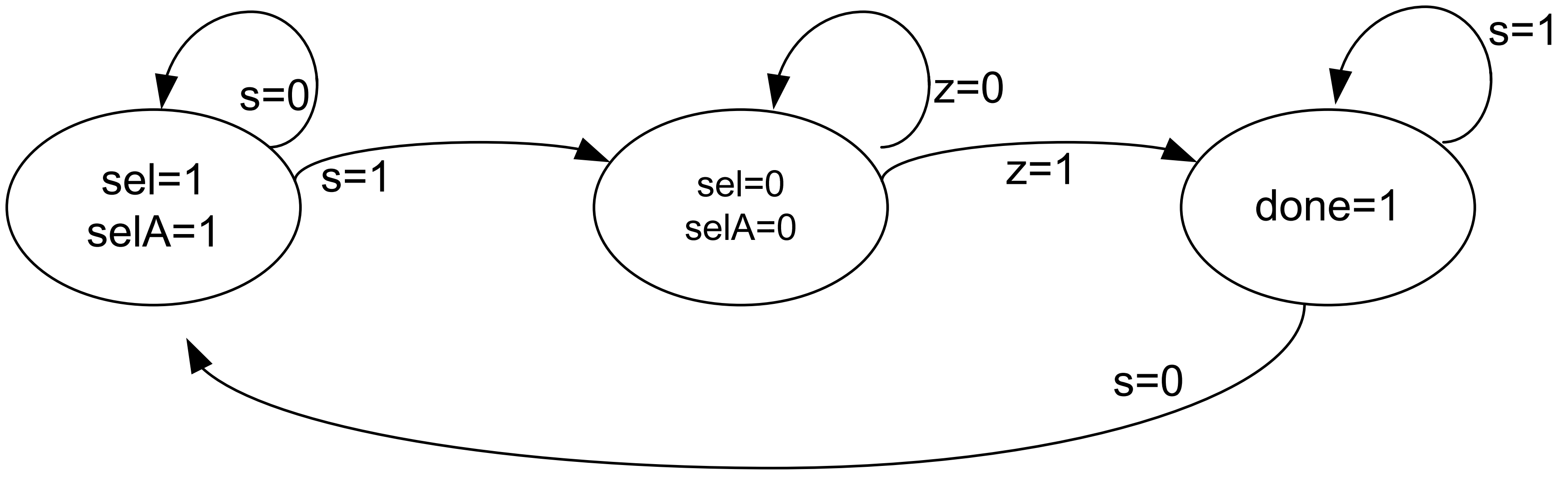
But we are not there yet. What we really want is:


This is a one-bit signal
that indicates when that valid input is on A and X
(tells the circuit to "start")
s
A
X
P
done
This is a one-bit signal
that indicates when a valid answer
is on P

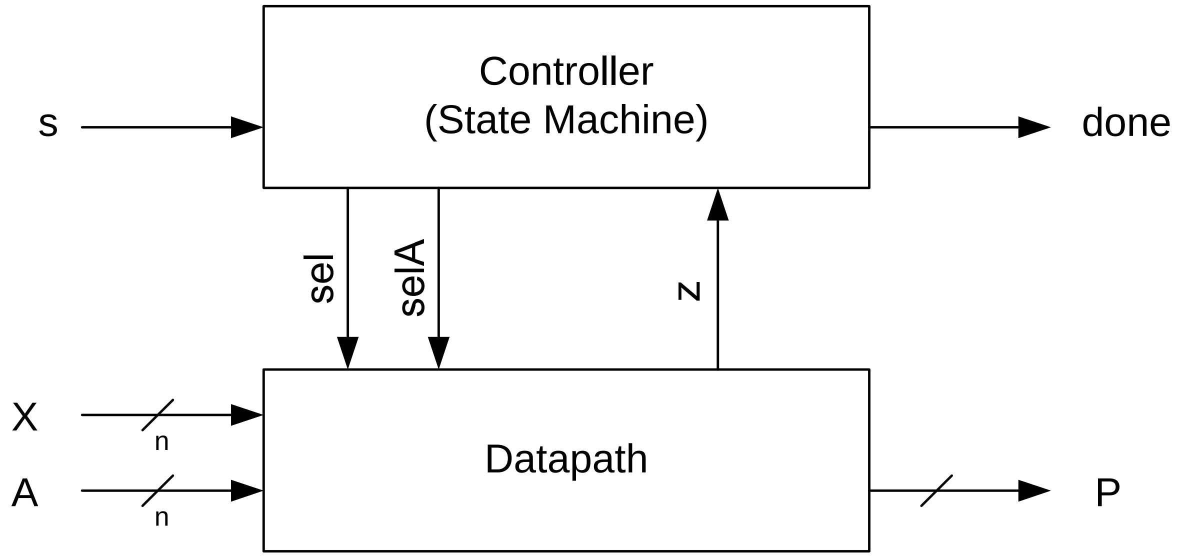
So to implement this, we need a controller that:

- when ***s*** goes high:
  - set ***sel*** and ***selA*** to 1 for one cycle
  - wait until **z** goes high
  - when it does, assert ***done***, and go back to the start

- Here is a simple controller that does that:



- Now combine the state machine and the datapath into one circuit:



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity top is
    port(A, X : in std_logic_vector(7 downto 0);
         s, clk : in std_logic;
         P : out std_logic_vector(7 downto 0);
         done : out std_logic);
end top ;

architecture behavioural of top is

signal curr_state: std_logic_vector(1 downto 0) := "00";
signal z, sel, selA : std_logic;
signal P_int: std_logic_vector(7 downto 0);
signal cnt: std_logic_vector(7 downto 0);
  ….
```

```
begin
   -- Datapath

   process(clk)
   variable tmp_mul_result : std_logic_vector(15 downto 0);
   begin
     if (clk = '1') then
        if (sel = '1') then
          P_int <= "00000001";
        else
          tmp_mul_result := X * P_int;
          P_int <= tmp_mul_result(7 downto 0);
        end if;
     end if;
   end process;
   P <= P_int;
```

--NOTE THIS STILL ISN'T SYNTHESIZABLE BECAUSE MULTIPLICATION IS NOT SYNTHESIZABLE (BUT YOU CAN SIMULATE IT)

…..

```
process(clk)
  begin
    if (clk = '1') then
       if (selA = '1') then
         cnt <= A;
       else
         cnt <= cnt - 1;
       end if;
    end if;
  end process;

  process(cnt)
  begin
    if (cnt = "00000000") then
      z <= '1';
    else
      z <= '0';
    end if;
  end process;
```

-- Controller

```
process(clk)
begin
   if (clk = '1') then
     case curr_state is
         when "00" =>
              if (s = '0') then
                 curr_state <= "00";
              else
                 curr_state <= "01";
              end if;
         when "01" =>
              if (z = '0') then
                  curr_state <= "01";
              else
                  curr_state <= "10";
              end if;
         when others =>
              if (s = '0') then
                  curr_state <= "00";
               else
                  curr_state <= "10";
               end if;
      end case;
    end if;
  end process;
```

```
process(curr_state)
     begin
         case curr_state is
            when "00" =>
                 sel <= '1'; selA <= '1'; done <= '0';
            when "01" =>
                 sel <= '0'; selA <= '0'; done <= '0';
            when others =>
                 sel <= '0'; selA <= '0'; done <= '1';
         end case;
   end process;
end behavioural;
```

Consider simulating this description to see the circuit’s behaviour in Altera’s waveform viewer.

Serial Multiplier

How do you implement a multiplier (multiply two numbers)

Decimal

$$\begin{array}{r}
13 \\
\times\ 11 \\
\hline
13 \\
13 \\
\hline
143
\end{array}$$

Binary

$$\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & & \\
 & 1 & 1 & 0 & 1 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}$$

Serial Multiplier Algorithm

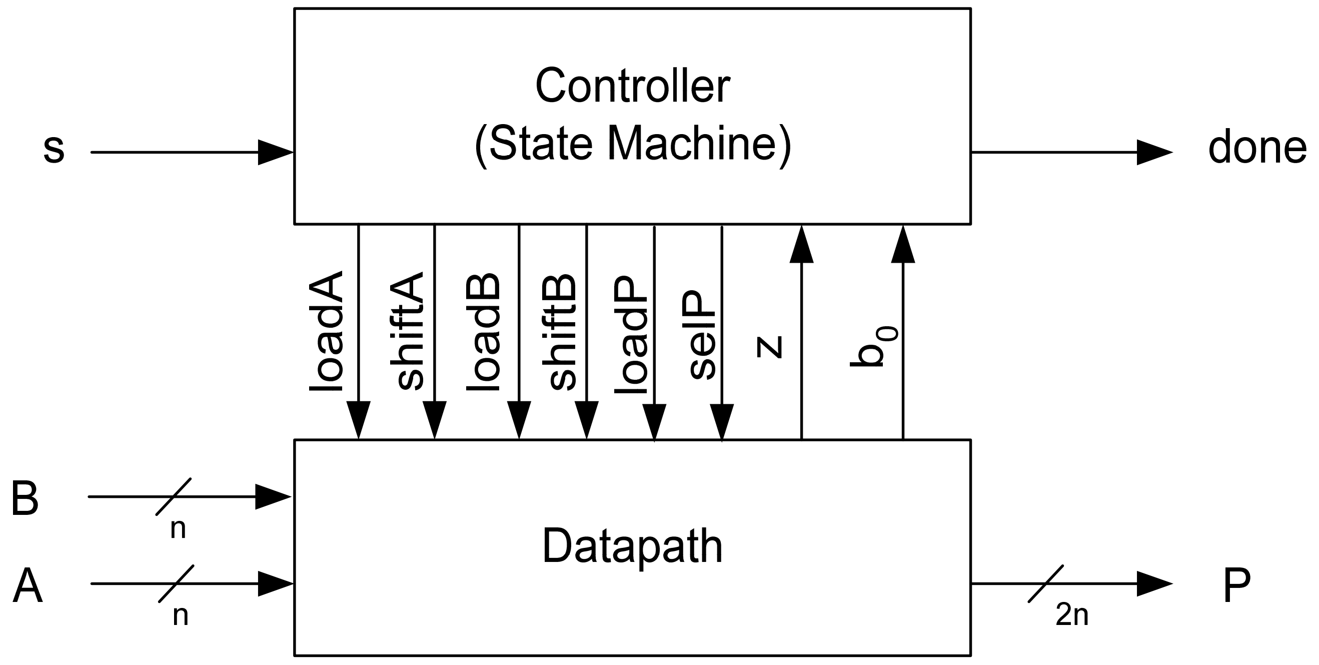
```
          1 1 0 1
        x 1 0 1 1
        ---------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
  ---------------
  1 0 0 0 1 1 1 1
```

Inputs A and B, Output P:

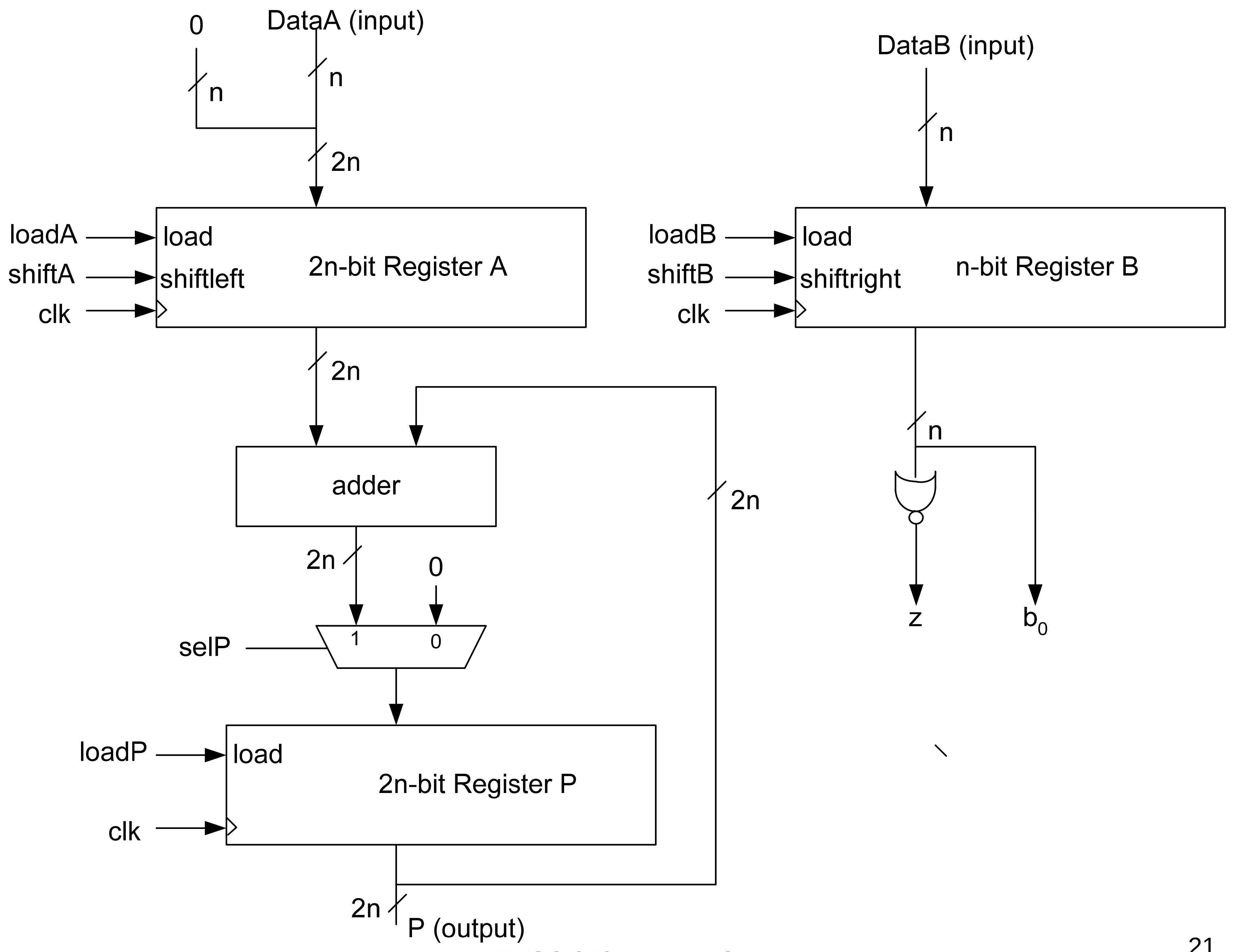
```
P=0
for (i=0 to n-1)
   if (B(i) = 1) then
      P = P + A
   end if;
   left shift A
end for;
```

As before, we could implement this psuedo-code using VHDL. But, it would not be synthesizable. So, we have to break it into smaller processes (a.k.a. design the hardware)

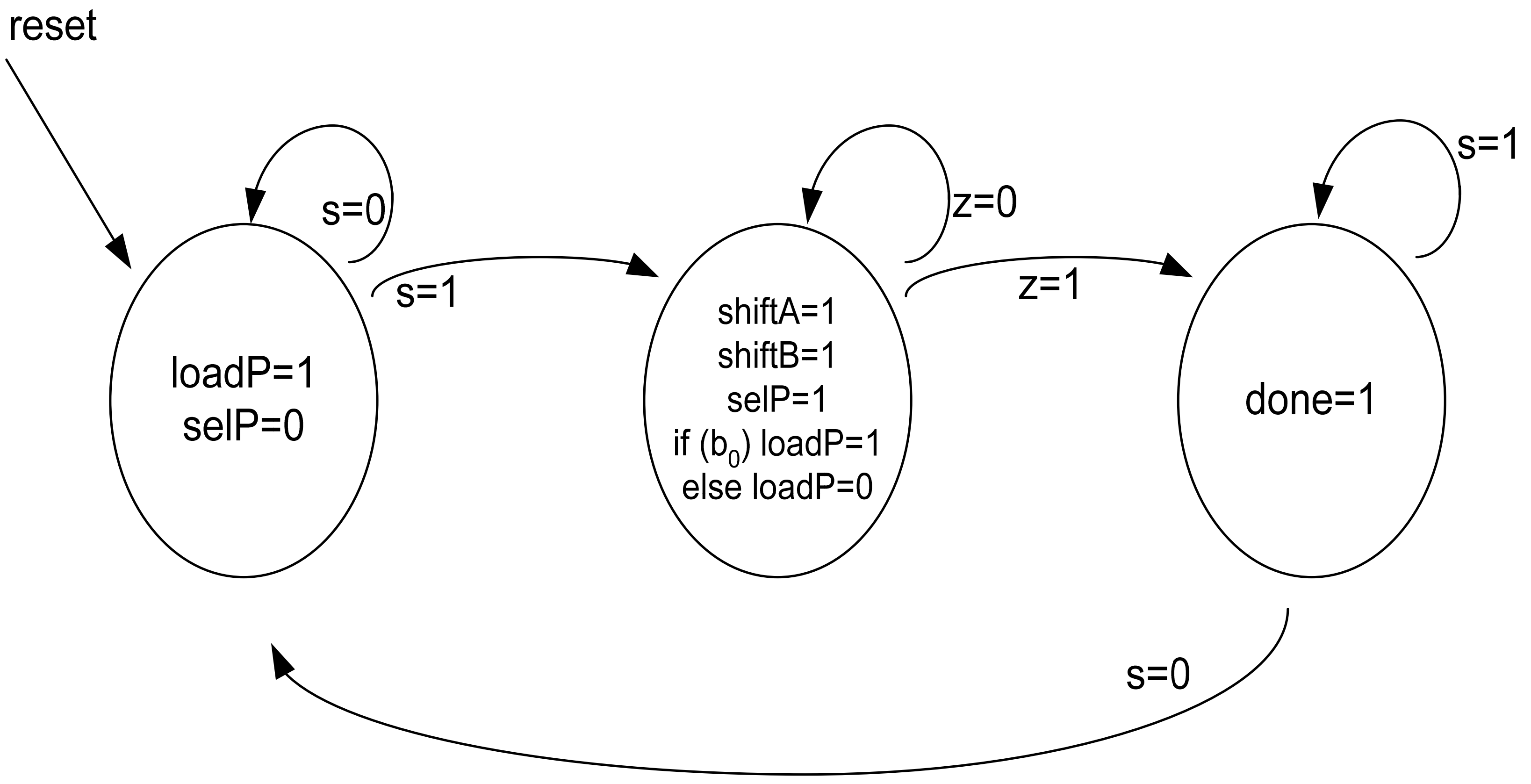
Top level diagram of what we will build:



When ***s*** goes high, a new n-bit values available on ***A*** and ***B.*** The machine then multiplies, and when it is finished, asserts ***done*** and puts the result on ***P***.


0
DataA (input)
n
n
2n
DataB (input)
n
loadA
shiftA
clk
load
shiftleft
2n-bit Register A
loadB
shiftB
clk
load
shiftright
n-bit Register B
2n
n
adder
2n
2n
0
selP
1
0
z
b_0
loadP
clk
load
2n-bit Register P
2n
P (output)

- State Machine:



Together, the state machine and datapath implement the serial multiply. The state machine is a Mealy Machine, so you would need two processes to describe it. The datapath can be described using simple components.

Bit Counting Circuit

Suppose we want to count the number of ‘1’s in a word.

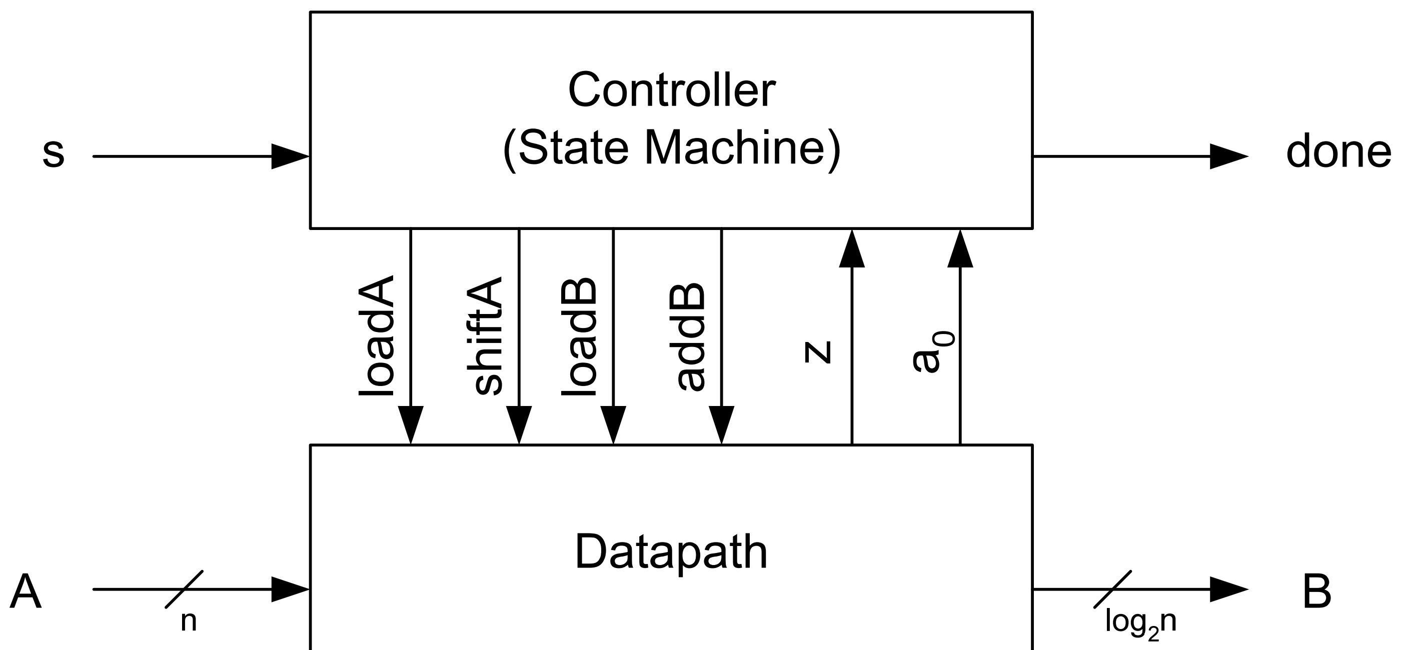
Algorithm to do this:

```
B=0
while (A ≠ 0) do
        if ($a_0$ = 1) then
                B = B + 1
        end if
        Right shift A
end while
```

Note 1: this isn’t VHDL or C, it just is pseudo-code to illustrate the algorithm.

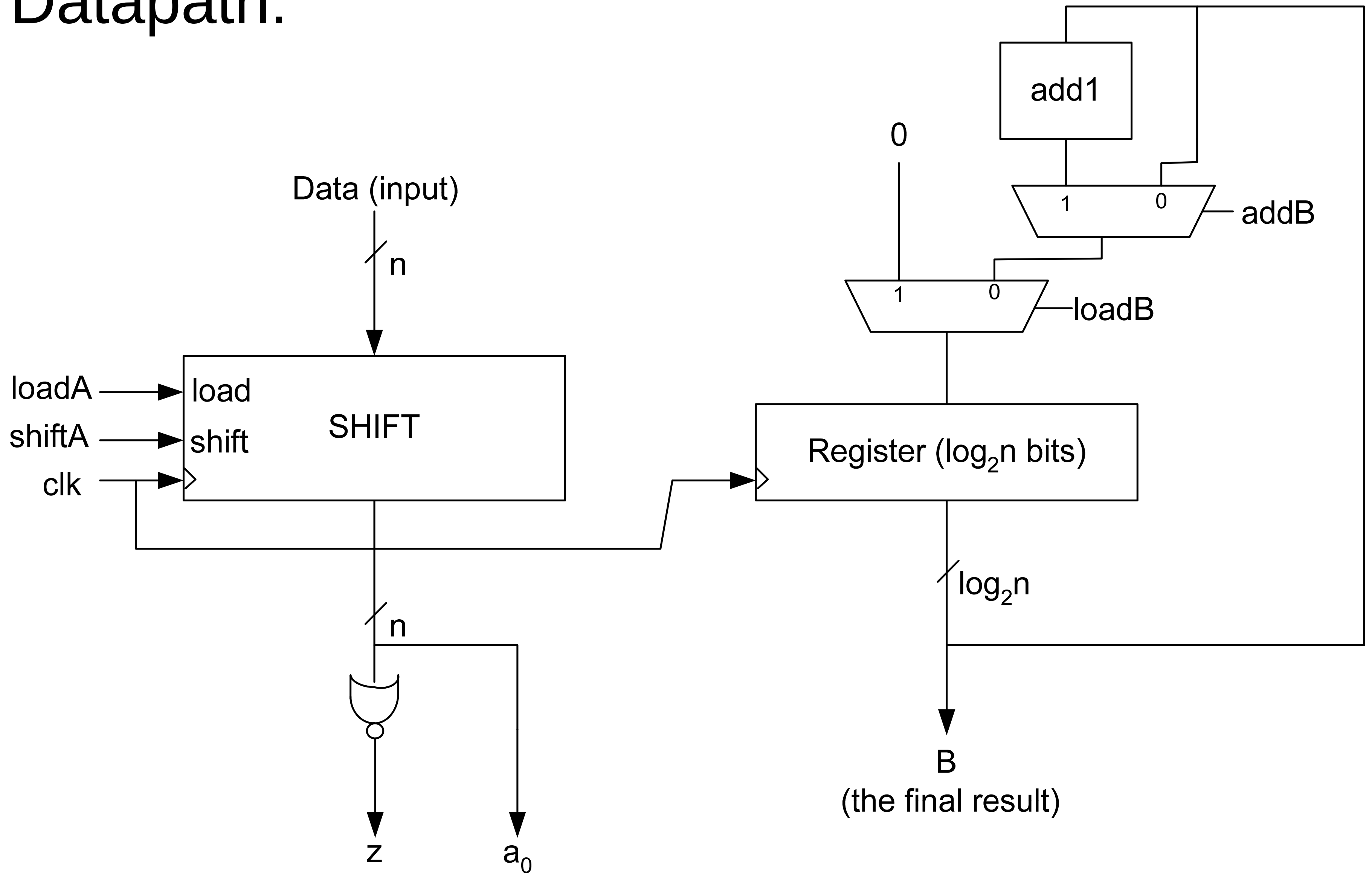
Note 2: We could write this in VHDL, but it would not be synthesizable. So, we have to design it using smaller processes (each one synthesizable)

Top level diagram of what we will build:



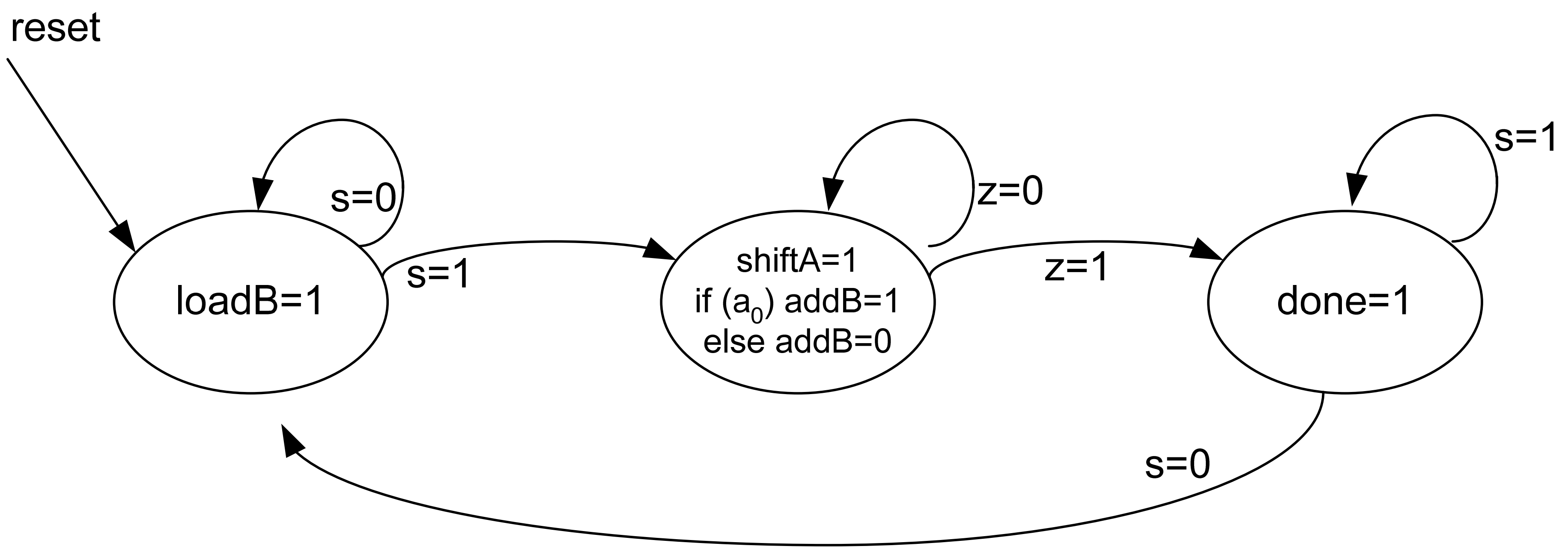
When ***s*** goes high, a new n-bit value is available on ***A***. The machine then counts the bits, and when it is finished, asserts ***done*** and puts the result on *B*.

Datapath:



add1
0
Data (input)
1
0
addB
n
1
0
loadB
loadA
load
SHIFT
shiftA
shift
clk
Register ($\log_2 n$ bits)
$\log_2 n$
n
B
(the final result)
z
a_0

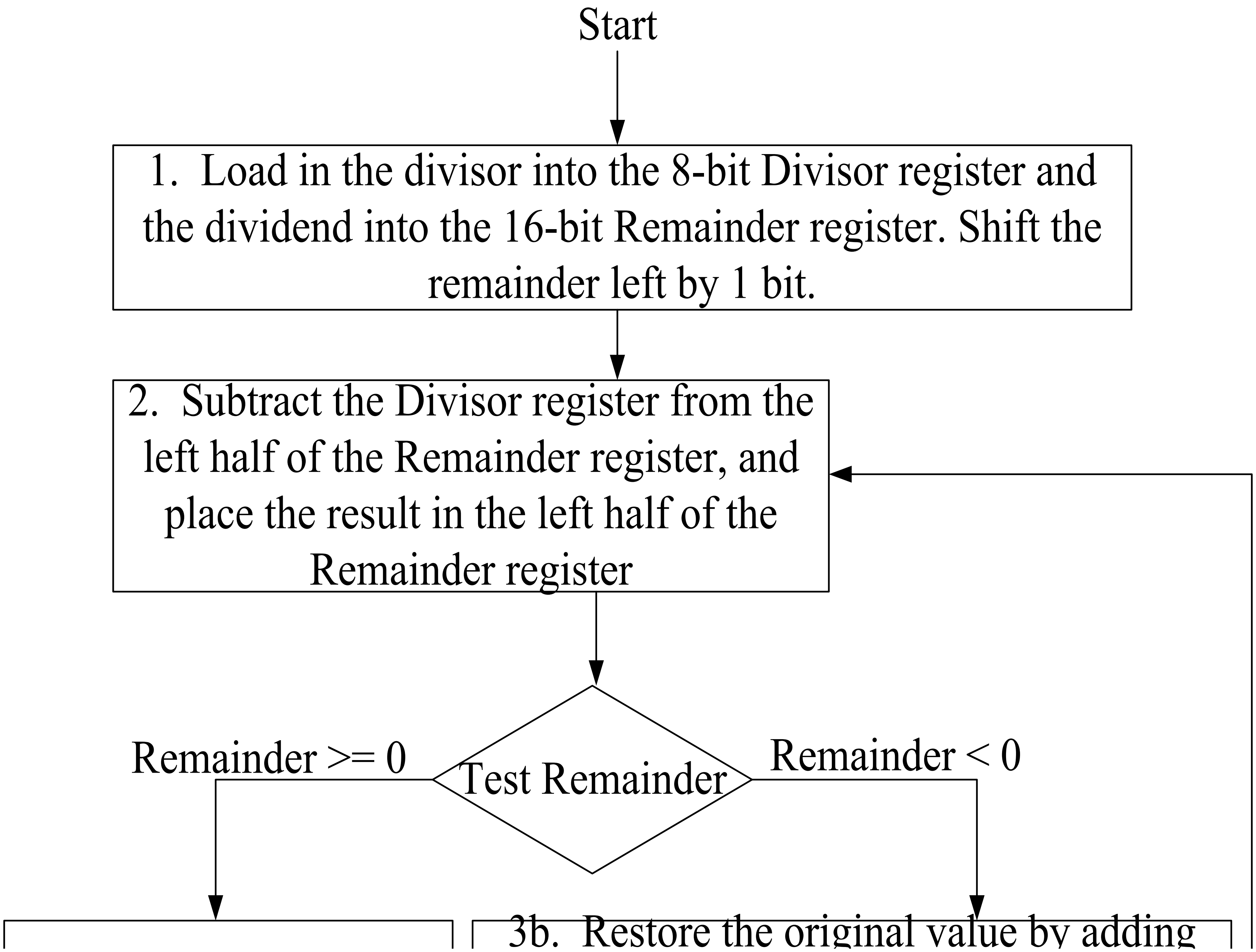
- State Machine:



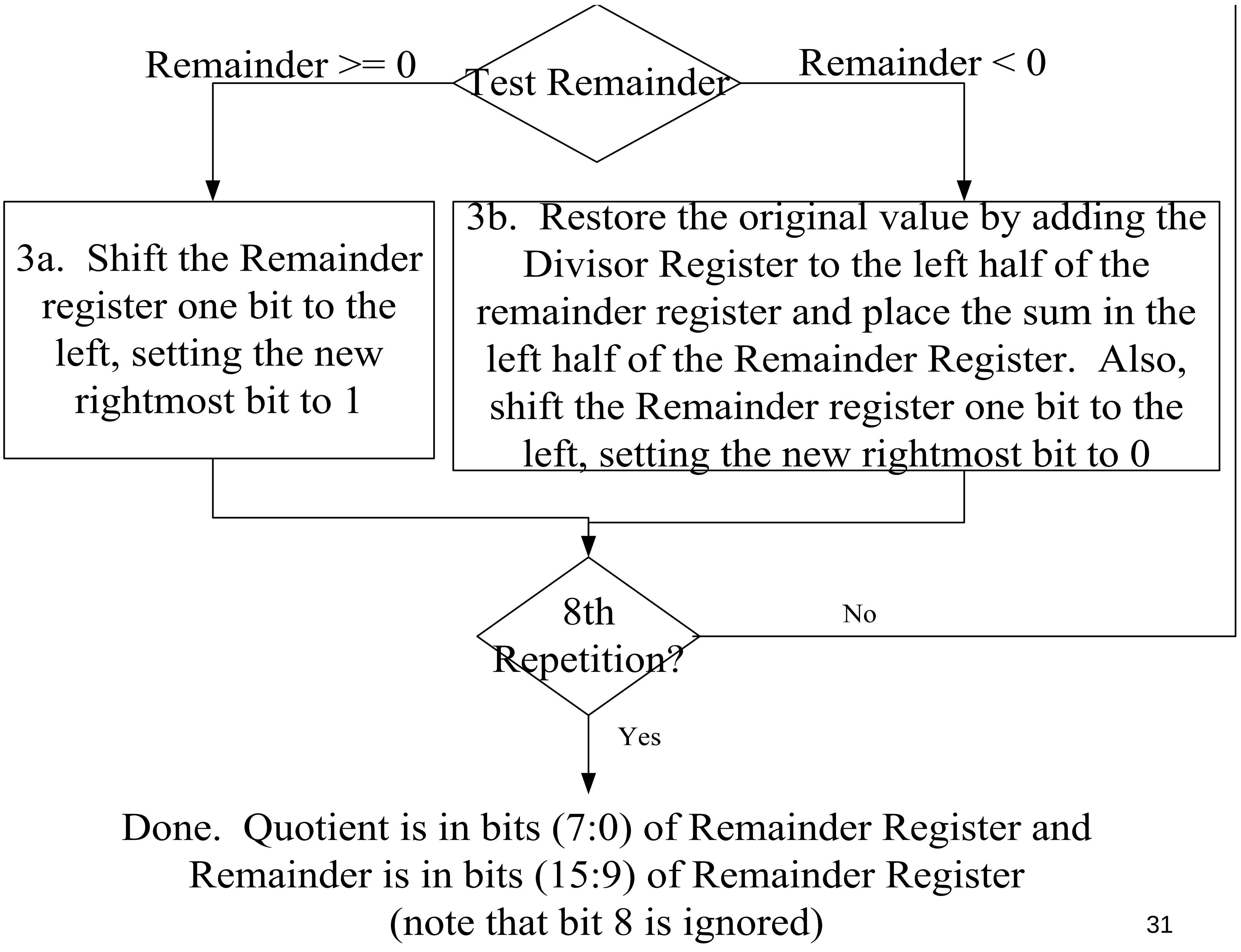
Together, the state machine and datapath implement the bit-counting operation. The state machine is a Mealy Machine, so you would need two processes to describe it. The datapath can be described using simple components.

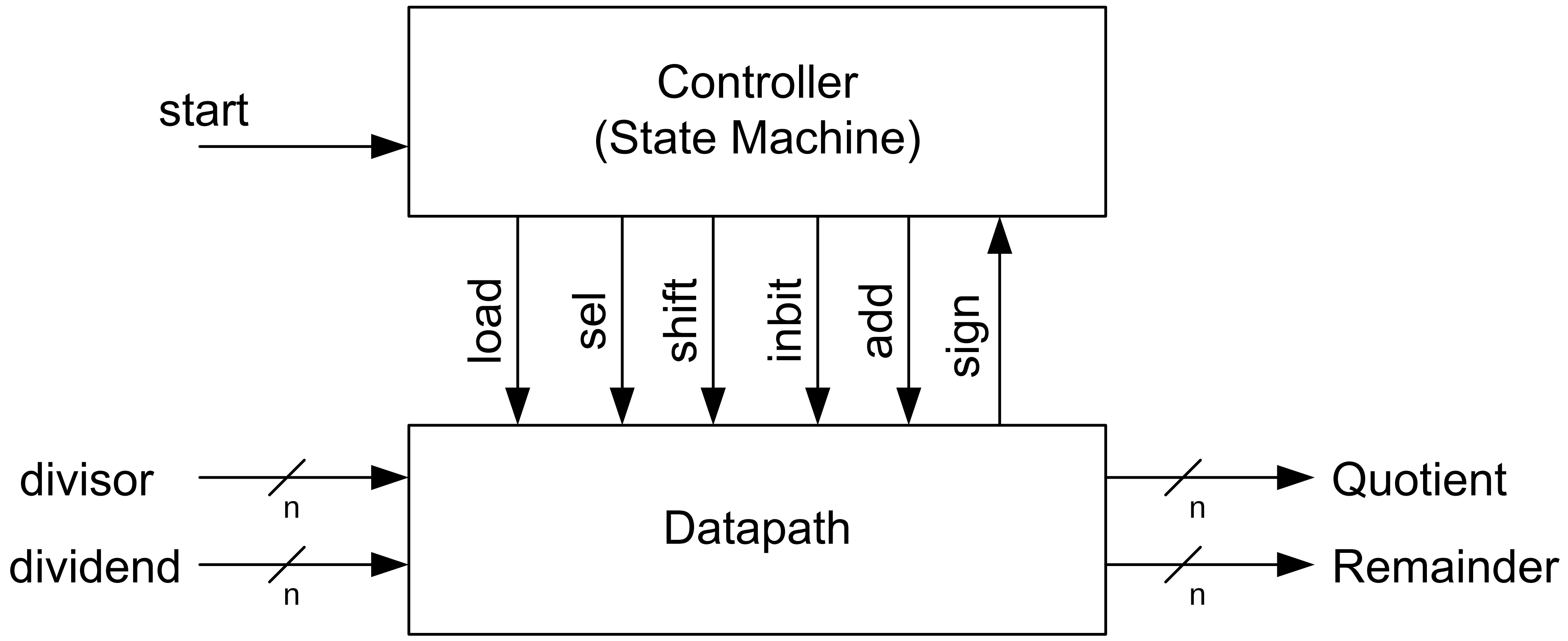
For practice you can try writing the HDL from this description

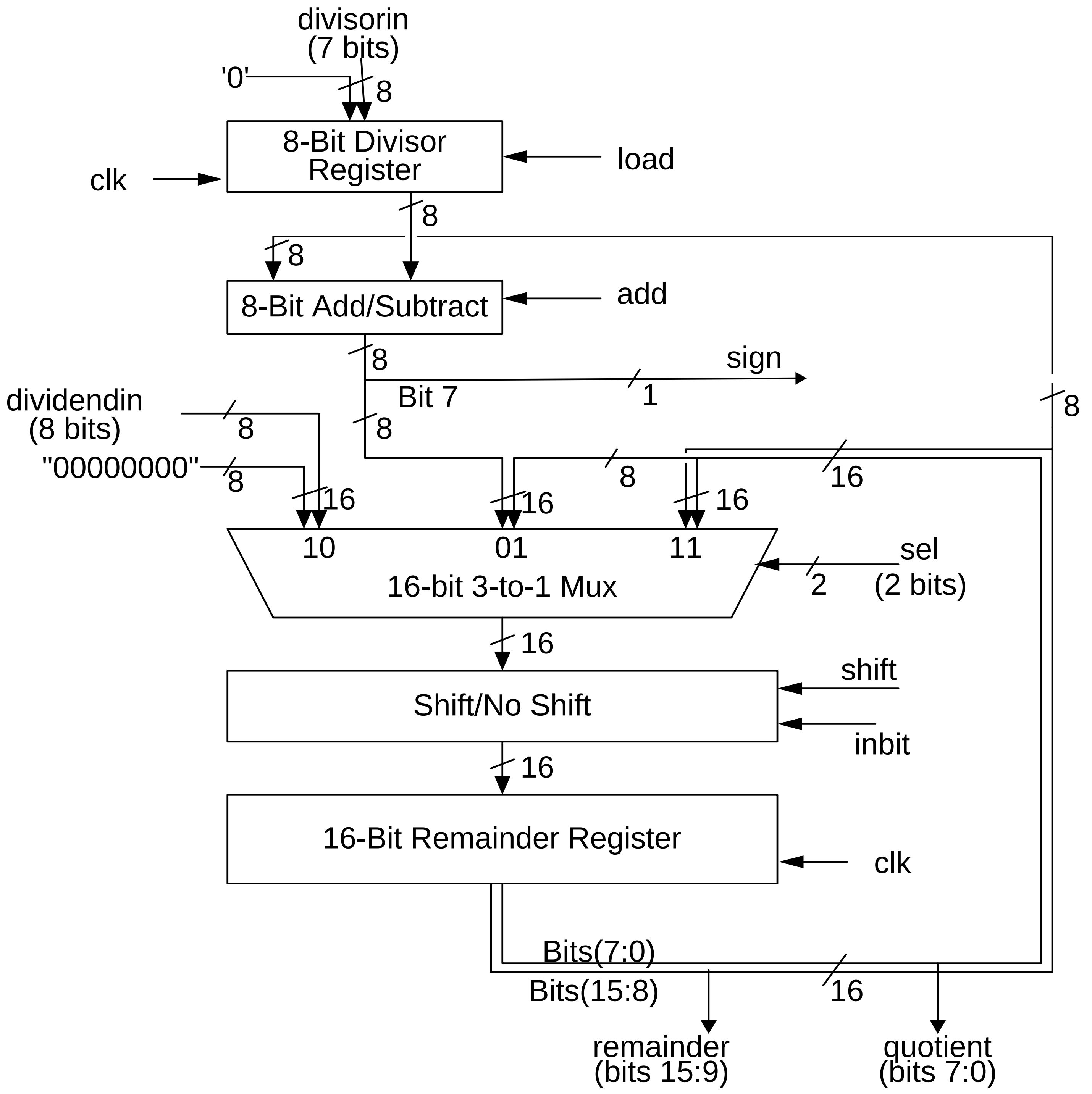
How would you design a divider?

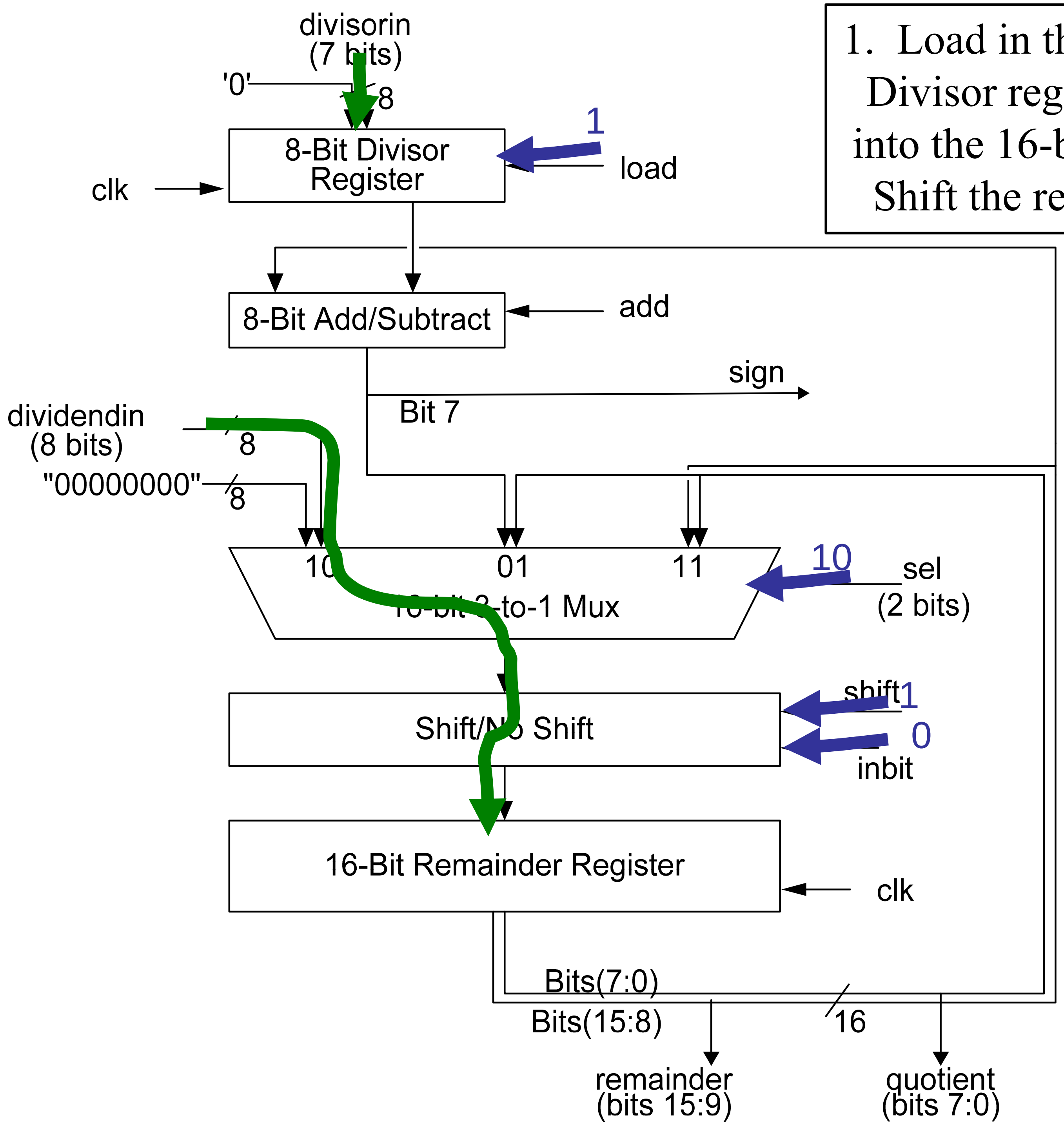


Start
1. Load in the divisor into the 8-bit Divisor register and the dividend into the 16-bit Remainder register. Shift the remainder left by 1 bit.
2. Subtract the Divisor register from the left half of the Remainder register, and place the result in the left half of the Remainder register
Test Remainder
Remainder >= 0
Remainder < 0
3b. Restore the original value by adding

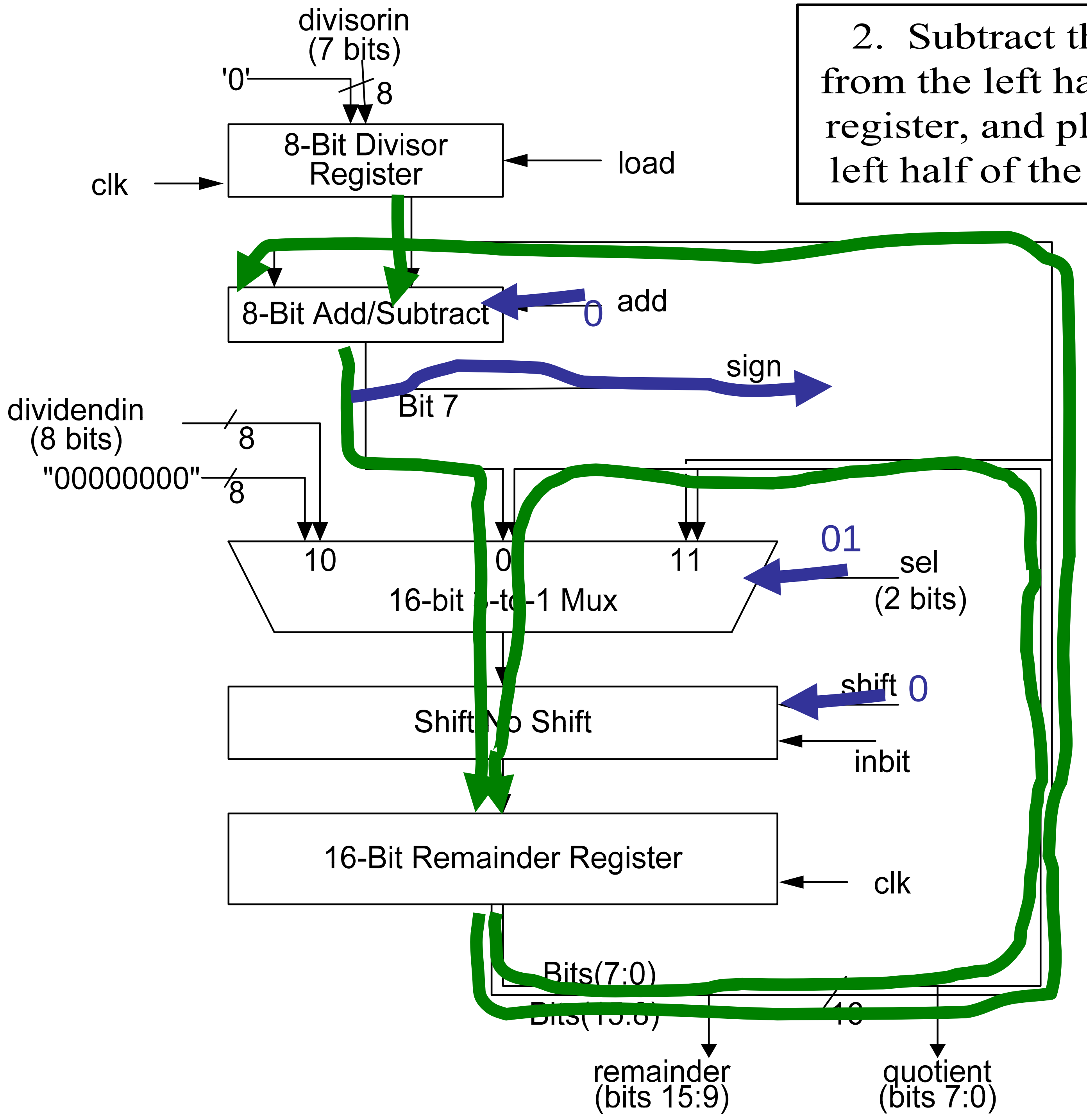

Test Remainder
Remainder >= 0
Remainder < 0
3a. Shift the Remainder register one bit to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor Register to the left half of the remainder register and place the sum in the left half of the Remainder Register. Also, shift the Remainder register one bit to the left, setting the new rightmost bit to 0
8th Repetition?
No
Yes
Done. Quotient is in bits (7:0) of Remainder Register and Remainder is in bits (15:9) of Remainder Register (note that bit 8 is ignored)


start
Controller
(State Machine)
load
sel
shift
inbit
add
sign
divisor
dividend
n
n
Datapath
Quotient
Remainder
n
n


divisorin
(7 bits)
'0'
8
8-Bit Divisor Register
load
clk
8
8
8-Bit Add/Subtract
add
8
sign
Bit 7
1
8
dividendin
(8 bits)
8
"00000000"
8
16
16
8
16
16
10
01
11
16-bit 3-to-1 Mux
sel
(2 bits)
2
16
Shift/No Shift
shift
inbit
16
16-Bit Remainder Register
clk
Bits(7:0)
Bits(15:8)
16
remainder
(bits 15:9)
quotient
(bits 7:0)

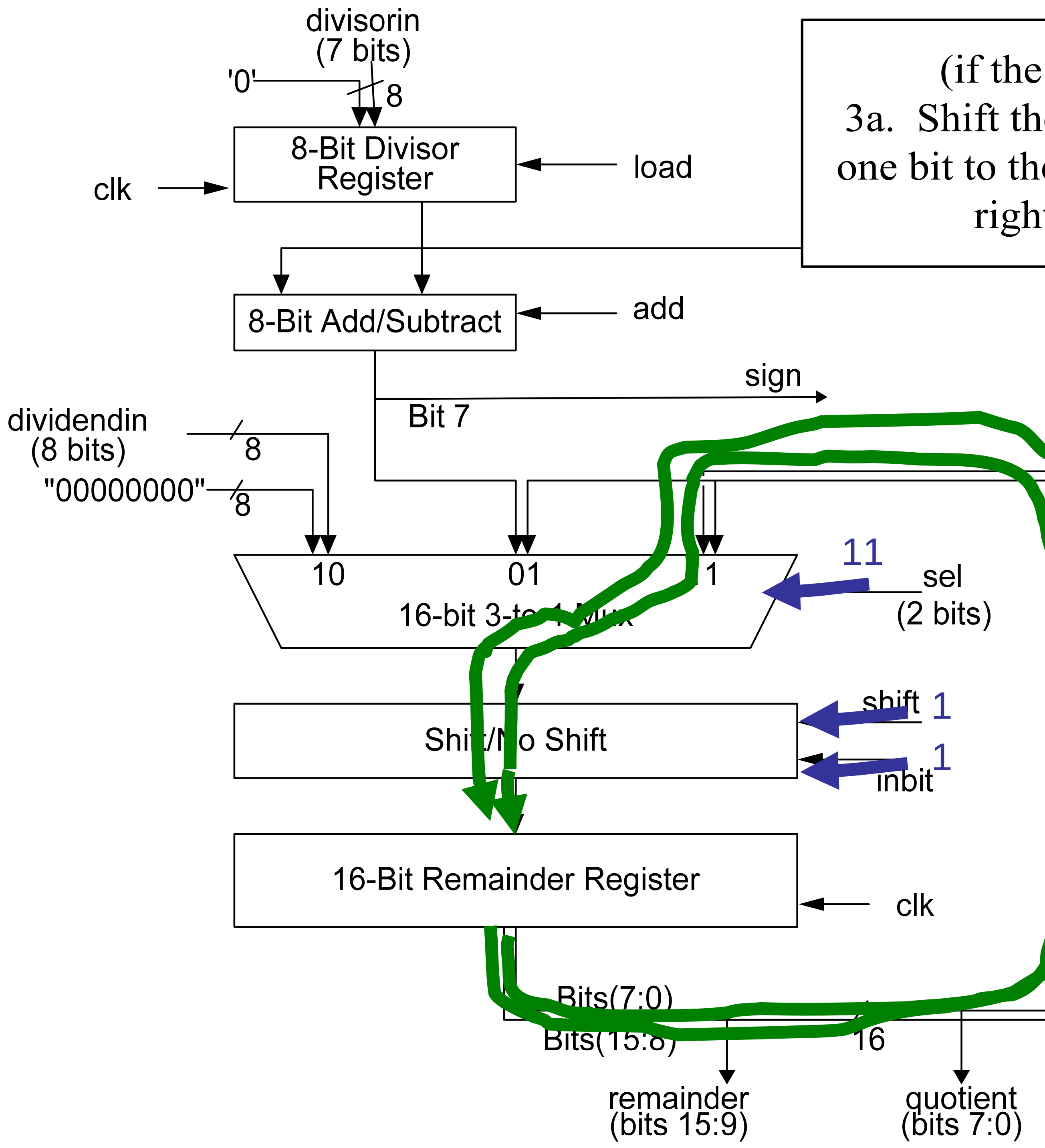


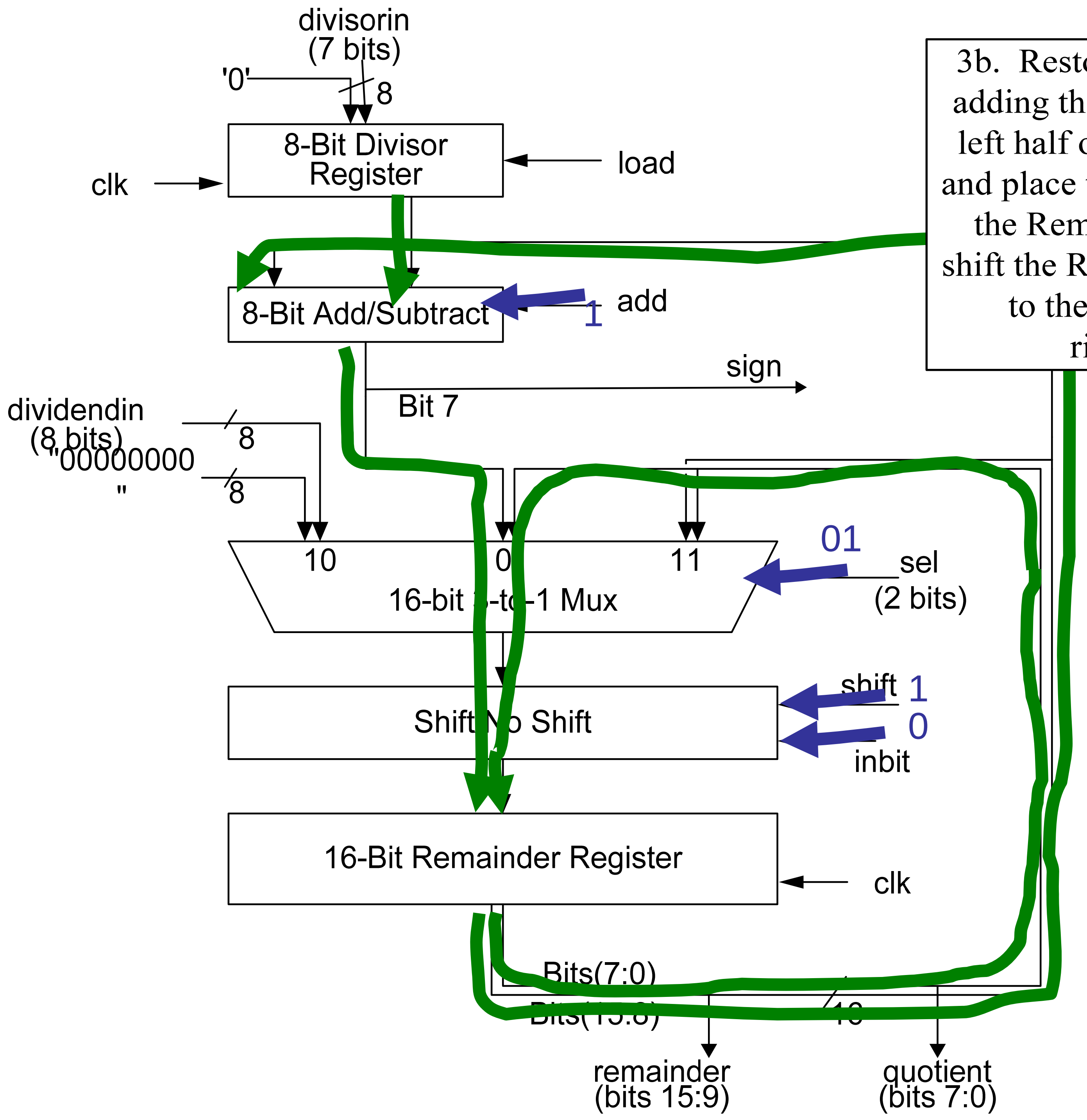
1. Load in the divisor into the 8-bit Divisor register and the dividend into the 16-bit Remainder register. Shift the remainder left by 1 bit.



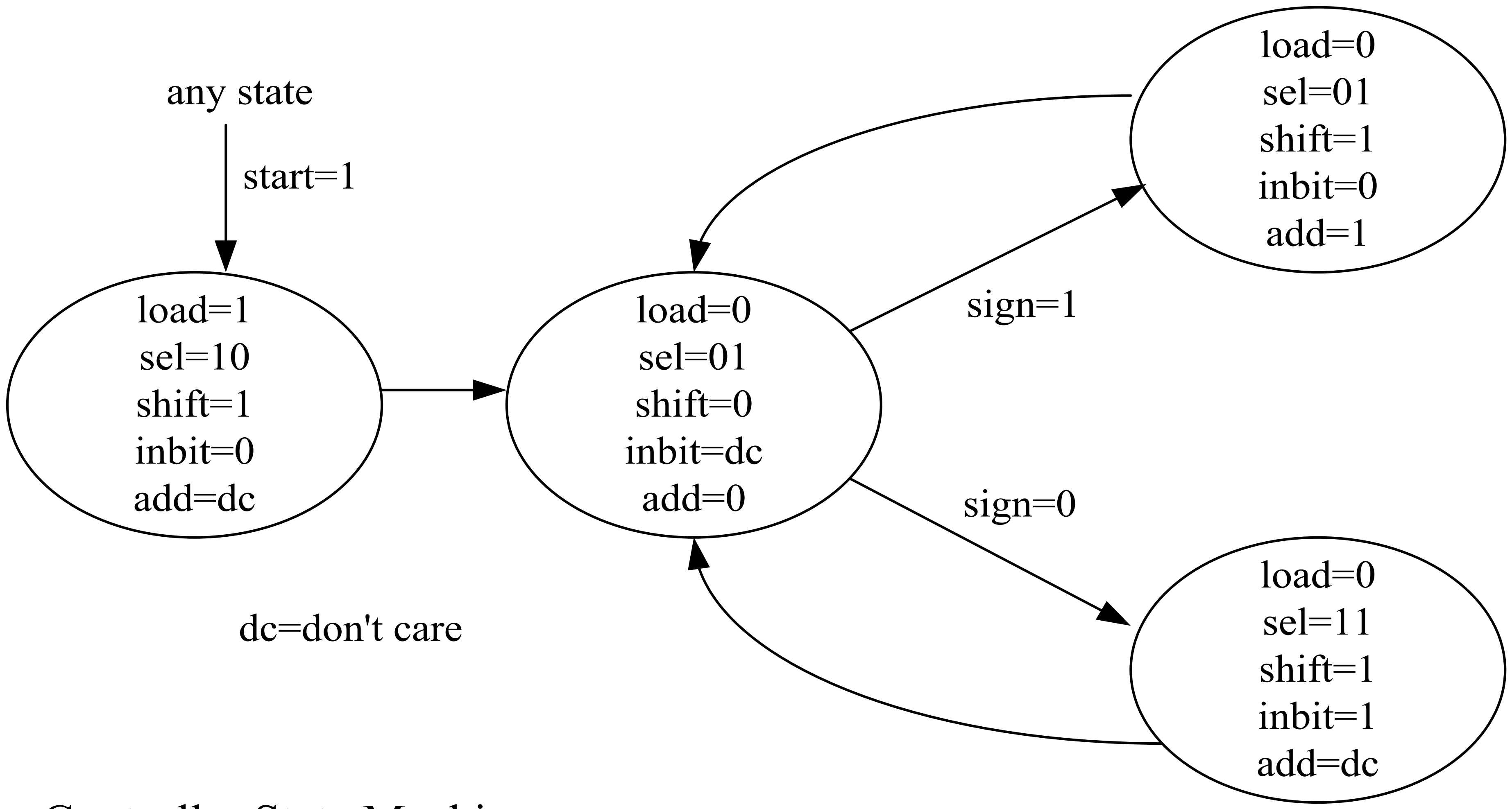
2. Subtract the Divisor register from the left half of the Remainder register, and place the result in the left half of the Remainder register

(if the sign bit was 0)
3a. Shift the Remainder register one bit to the left, setting the new rightmost bit to 1

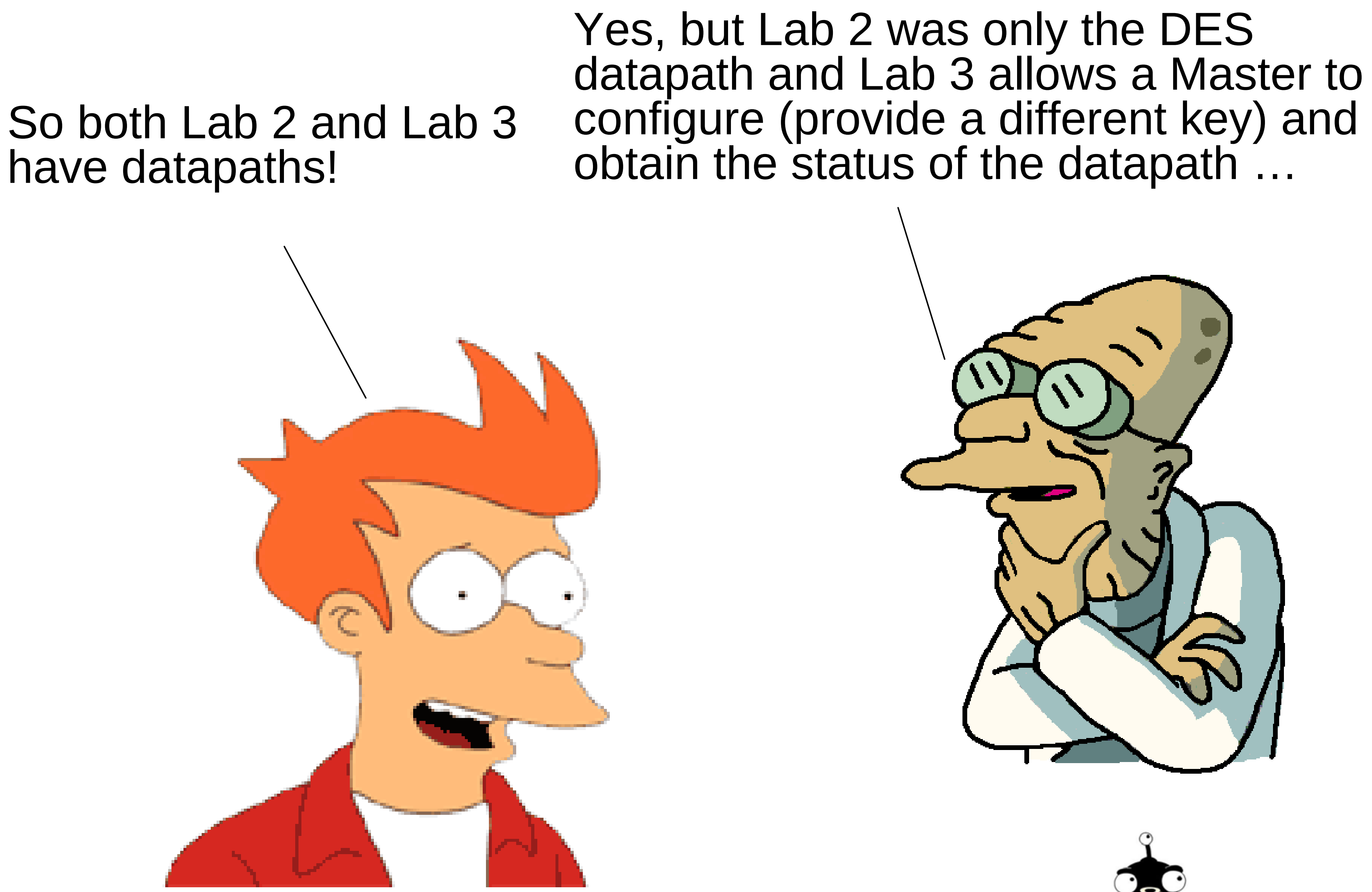




3b. Restore the original value by adding the Divisor Register to the left half of the remainder register and place the sum in the left half of the Remainder Register. Also, shift the Remainder register one bit to the left, setting the new rightmost bit to 0

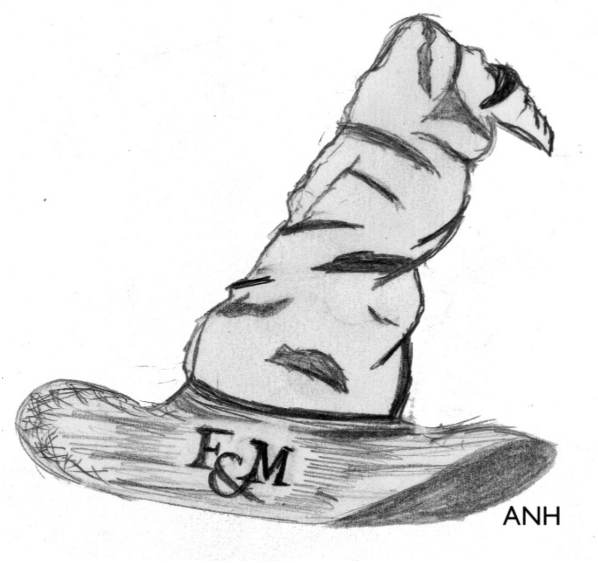


Controller State Machine


So both Lab 2 and Lab 3 have datapaths!
Yes, but Lab 2 was only the DES datapath and Lab 3 allows a Master to configure (provide a different key) and obtain the status of the datapath …

Sorting

Sorting is the type thing that really makes sense to do in software (since it is so sequential). That being said, there may be times that you want to do it in hardware. Let's look at a fairly complex datapath that will perform sorting.



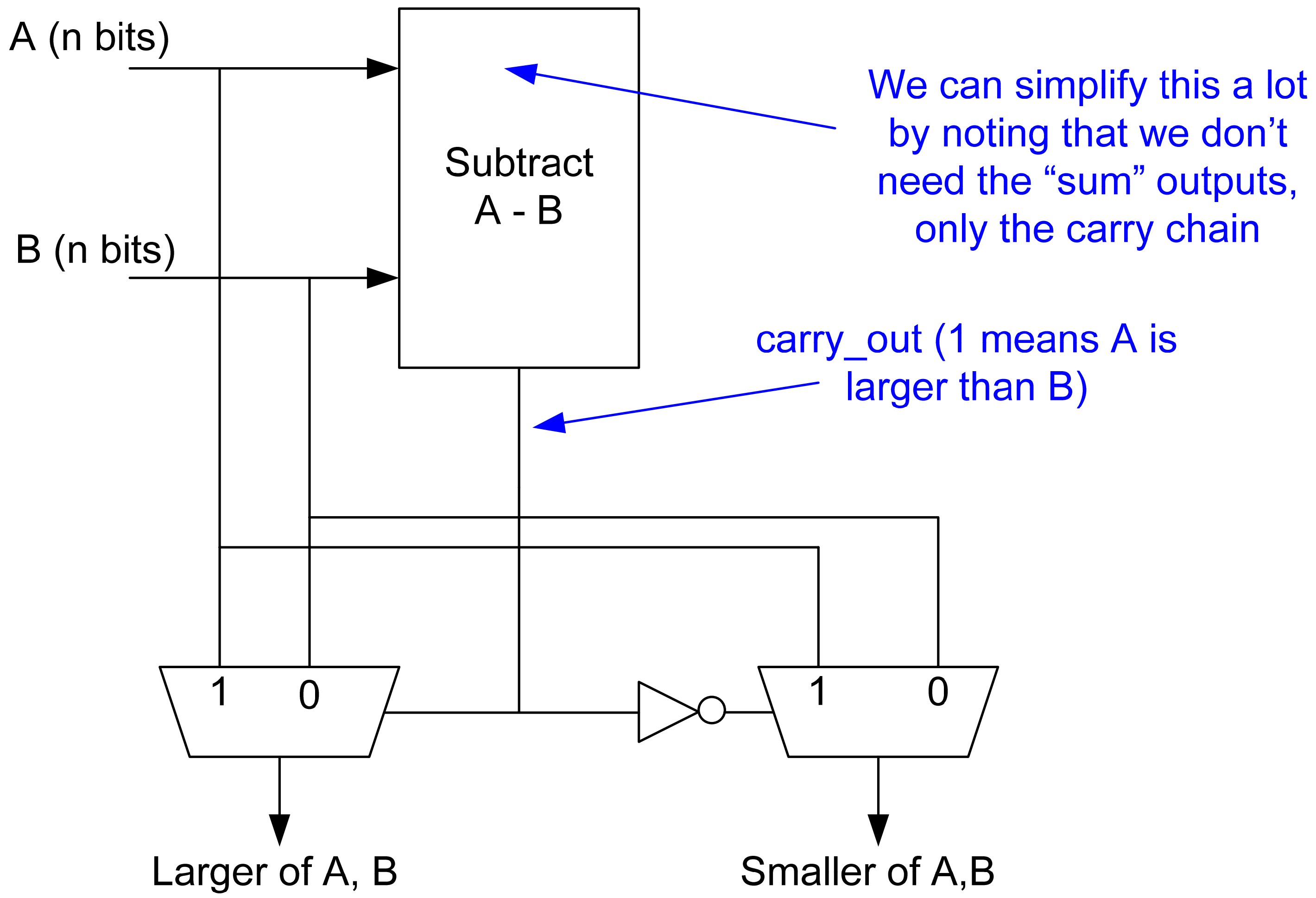
We will consider two approaches:

- Fully parallel (big)
- Serial (slow, but smaller)

The serial version is described in the textbook in great detail.

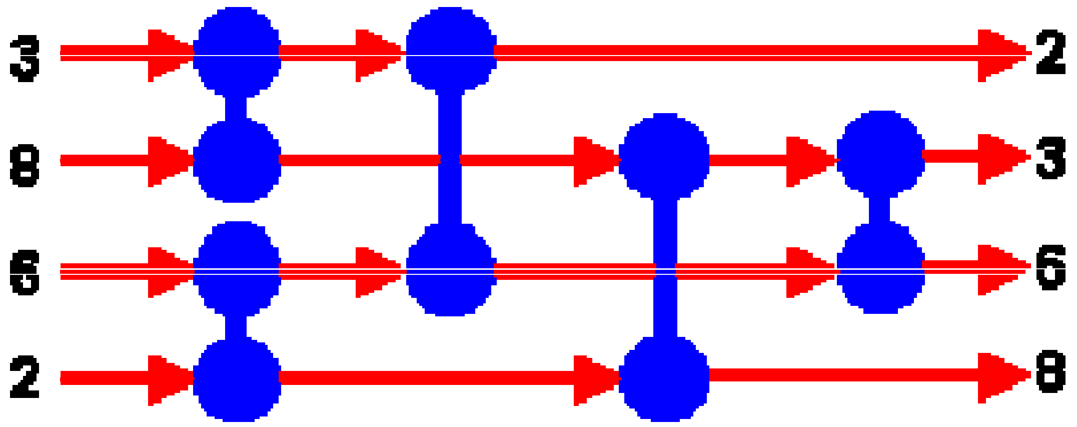
Fully-Parallel Sorting

- First consider designing a block that sorts two input numbers:


A (n bits)
B (n bits)
Subtract
A - B
We can simplify this a lot by noting that we don't need the "sum" outputs, only the carry chain
carry_out (1 means A is larger than B)
1
0
1
0
Larger of A, B
Smaller of A,B

- Note that this is purely combinational (no clock required)

Now build a network of these building blocks:



This will sort four numbers of any bit-width in one cycle

Problems:

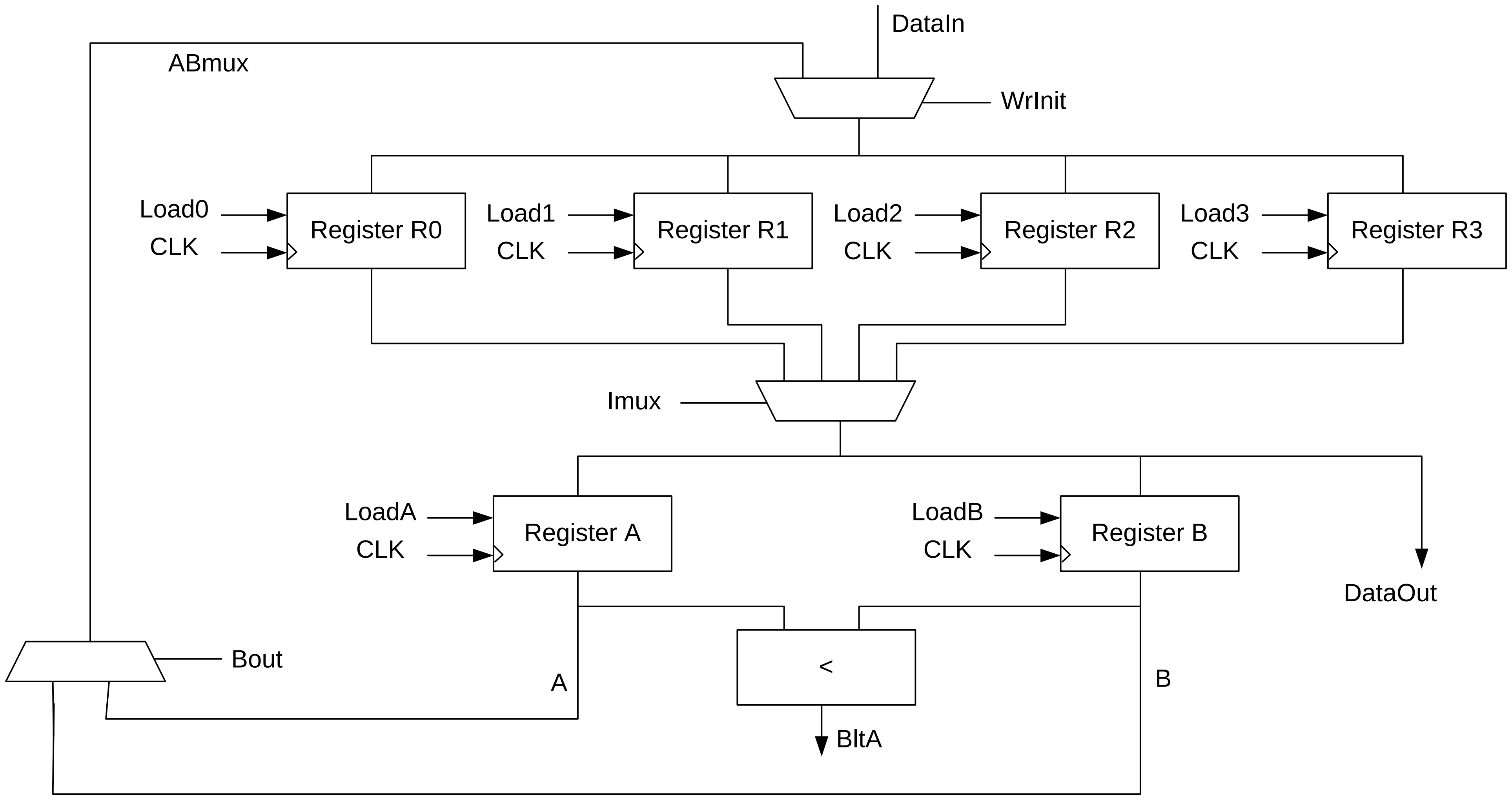
- Gets big of there are more numbers to sort

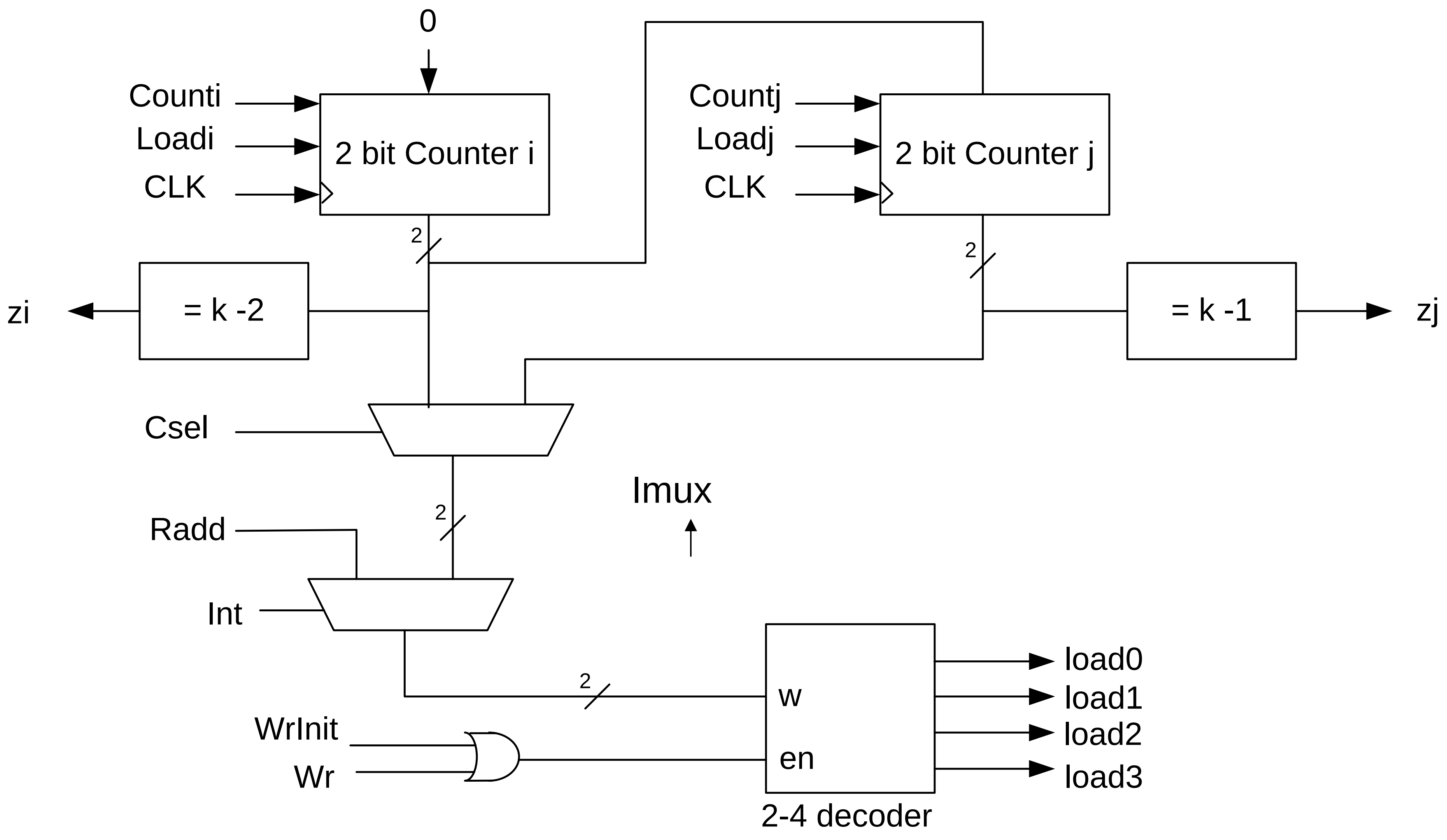
  Best known: O (n log n) blocks for n inputs

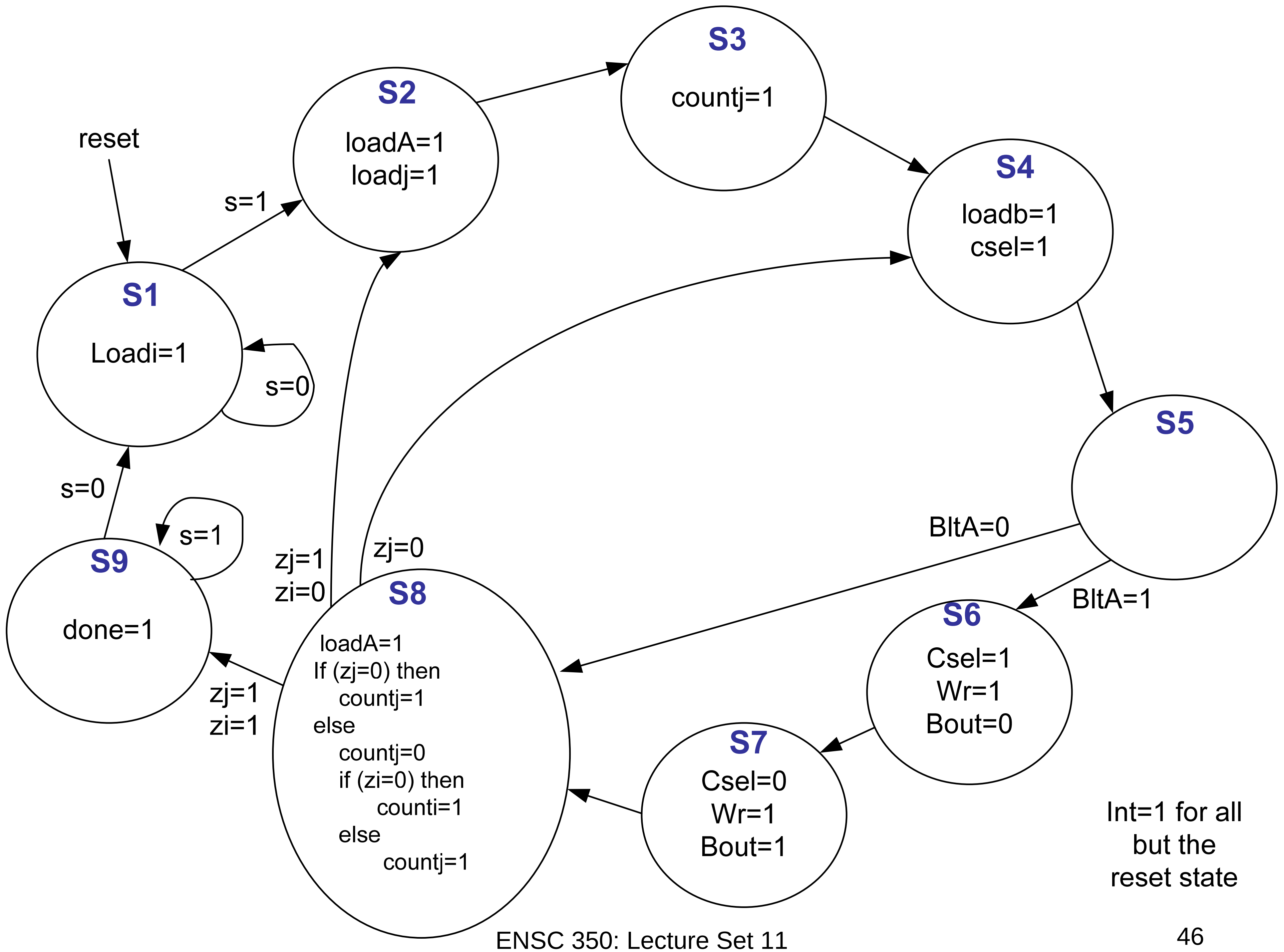
- Can’t use this if n is arbitrary (not known when the chip is designed)

Suppose we want to sort k numbers:

```
for (i=0 to k-2) do
  A = Ri;
  for (j=i+1 to k-1) do
      B = Rj
      if (B < A) then
         Ri = B
         Rj = A
         A = Ri
     end if;
  end for;
end for;
```


DataIn
ABmux
WrInit
Load0
CLK
Register R0
Load1
CLK
Register R1
Load2
CLK
Register R2
Load3
CLK
Register R3
Imux
LoadA
CLK
Register A
LoadB
CLK
Register B
DataOut
Bout
A
<
B
BltA

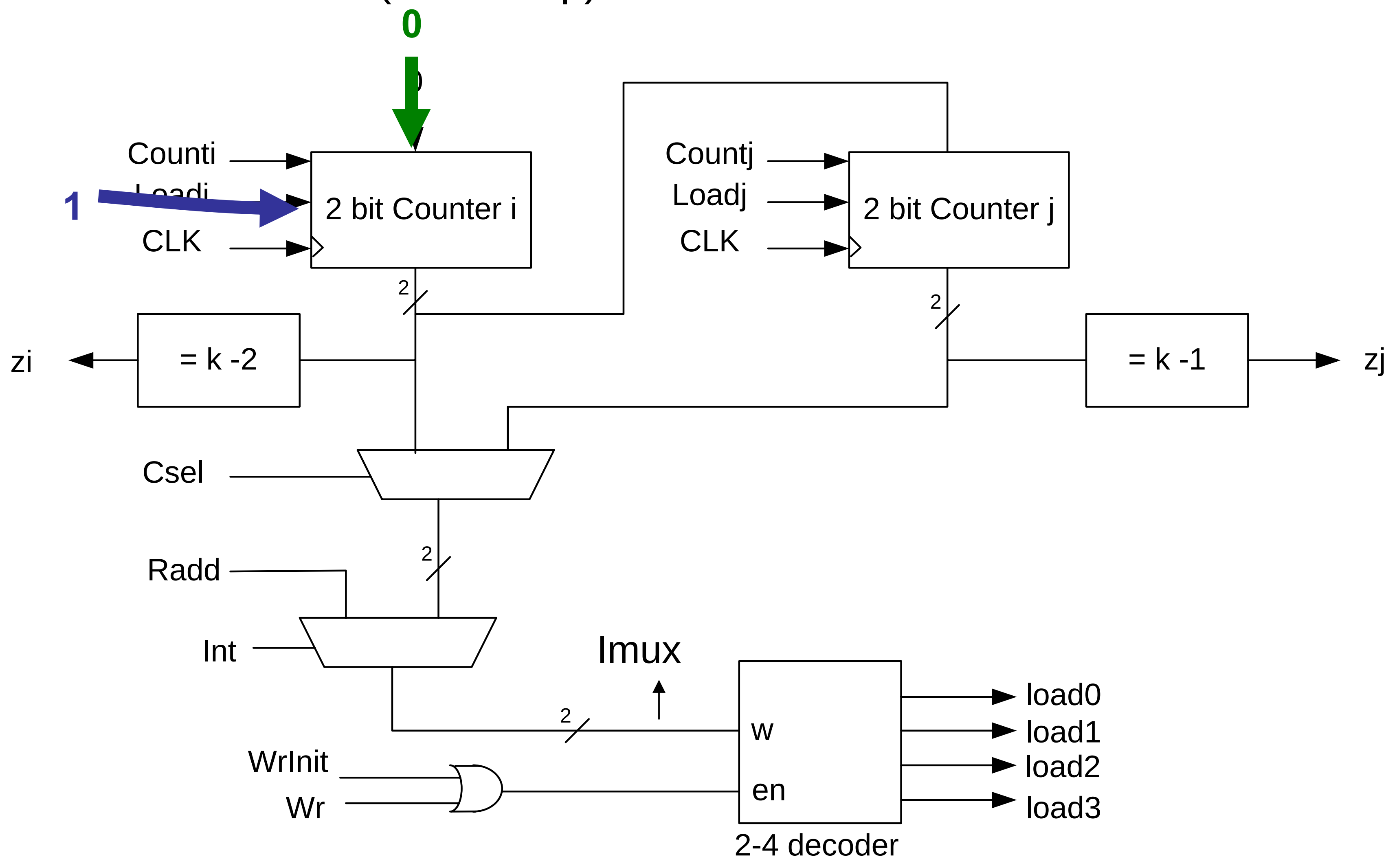

0
Counti
Loadi
CLK
2 bit Counter i
Countj
Loadj
CLK
2 bit Counter j
2
2
zi
= k -2
= k -1
zj
Csel
Imux
2
Radd
Int
2
w
en
WrInit
Wr
load0
load1
load2
load3
2-4 decoder



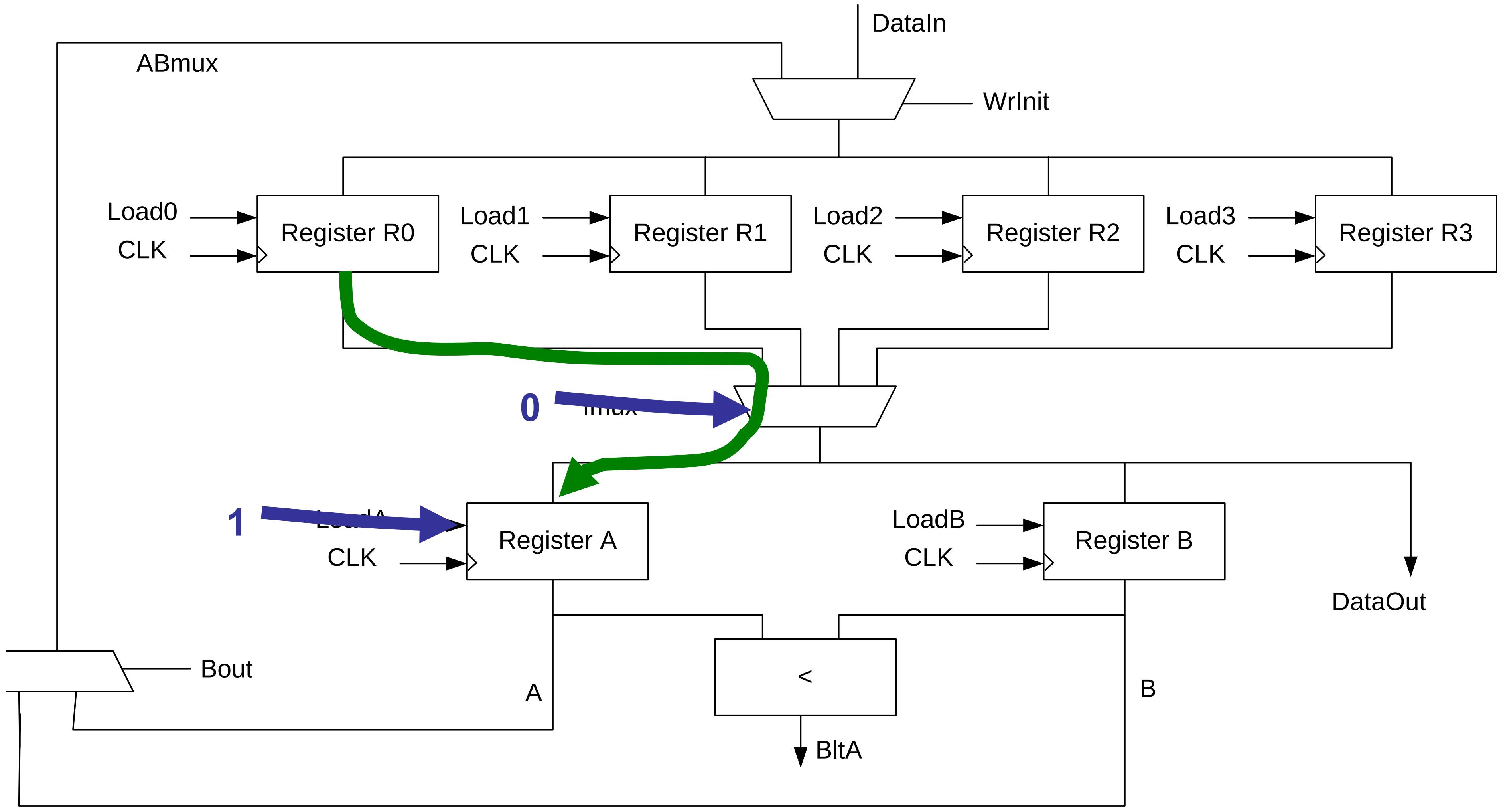
Int=1 for all but the reset state

Assume that the values to be sorted are in Registers R1 to R4 (circuitry is provided to do this, but assume it has already been done)

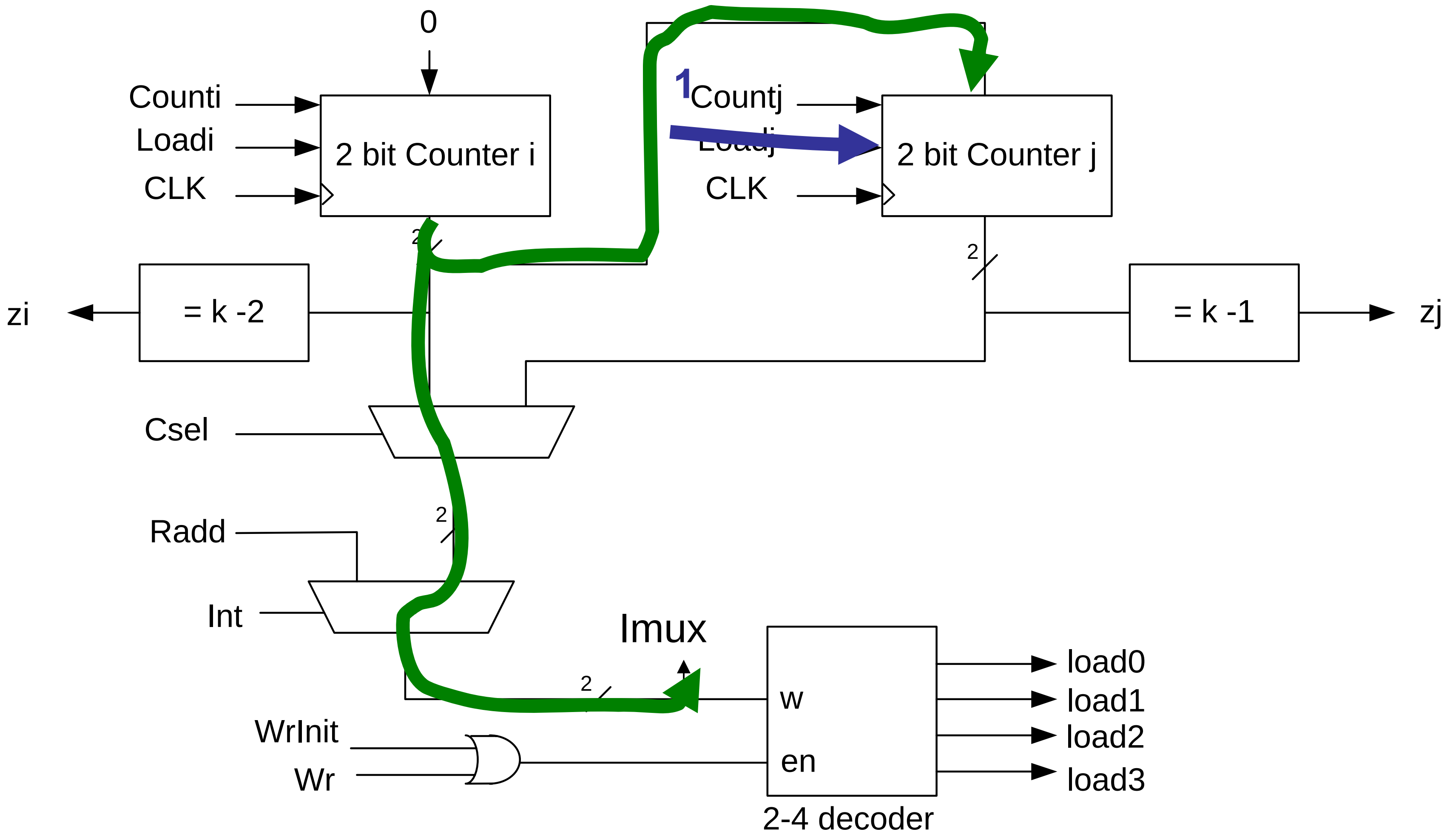
State S1: Initialize i (outer loop) to zero



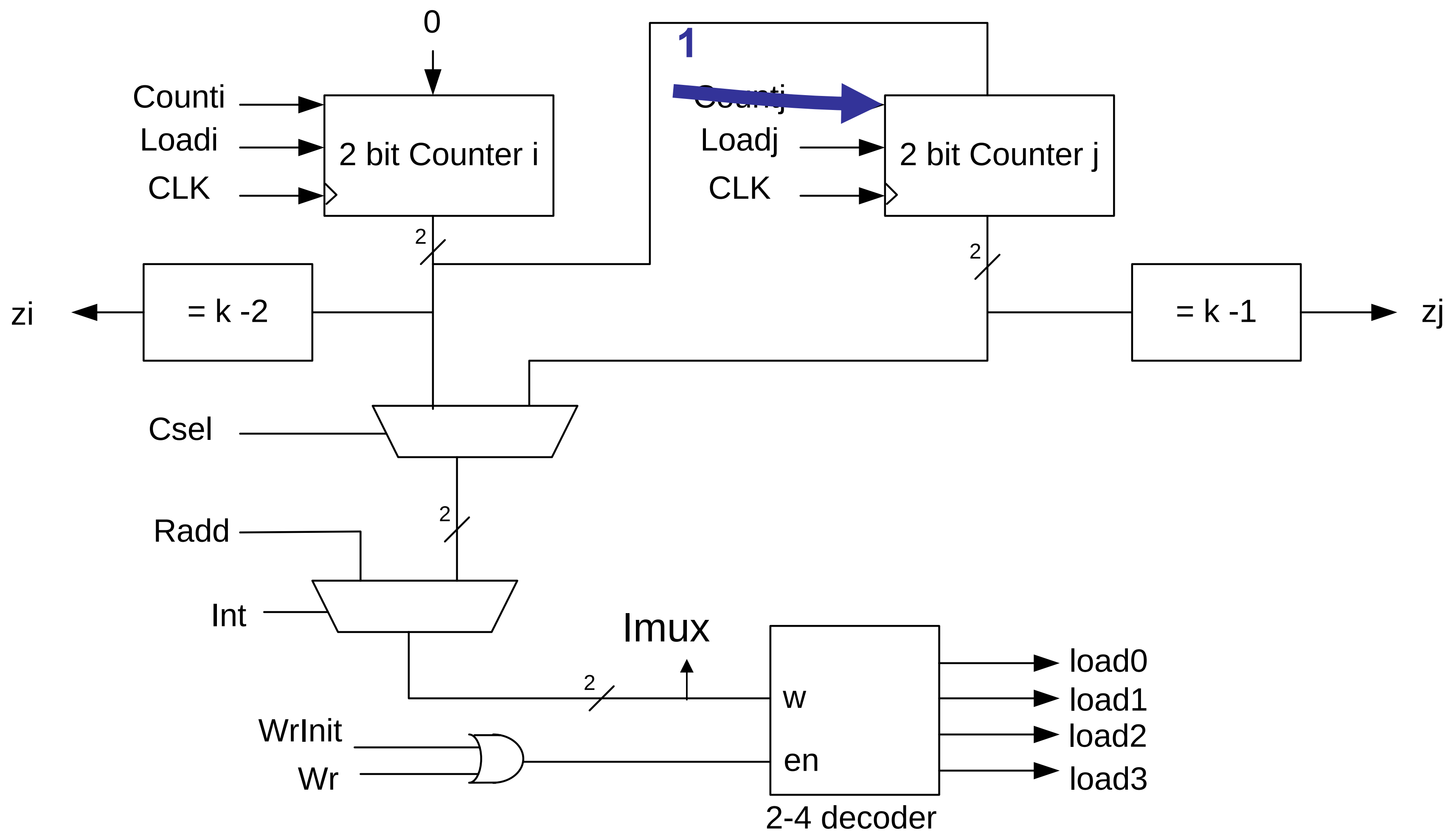
State S2: Load Register A with Ri and initialize j to value of i



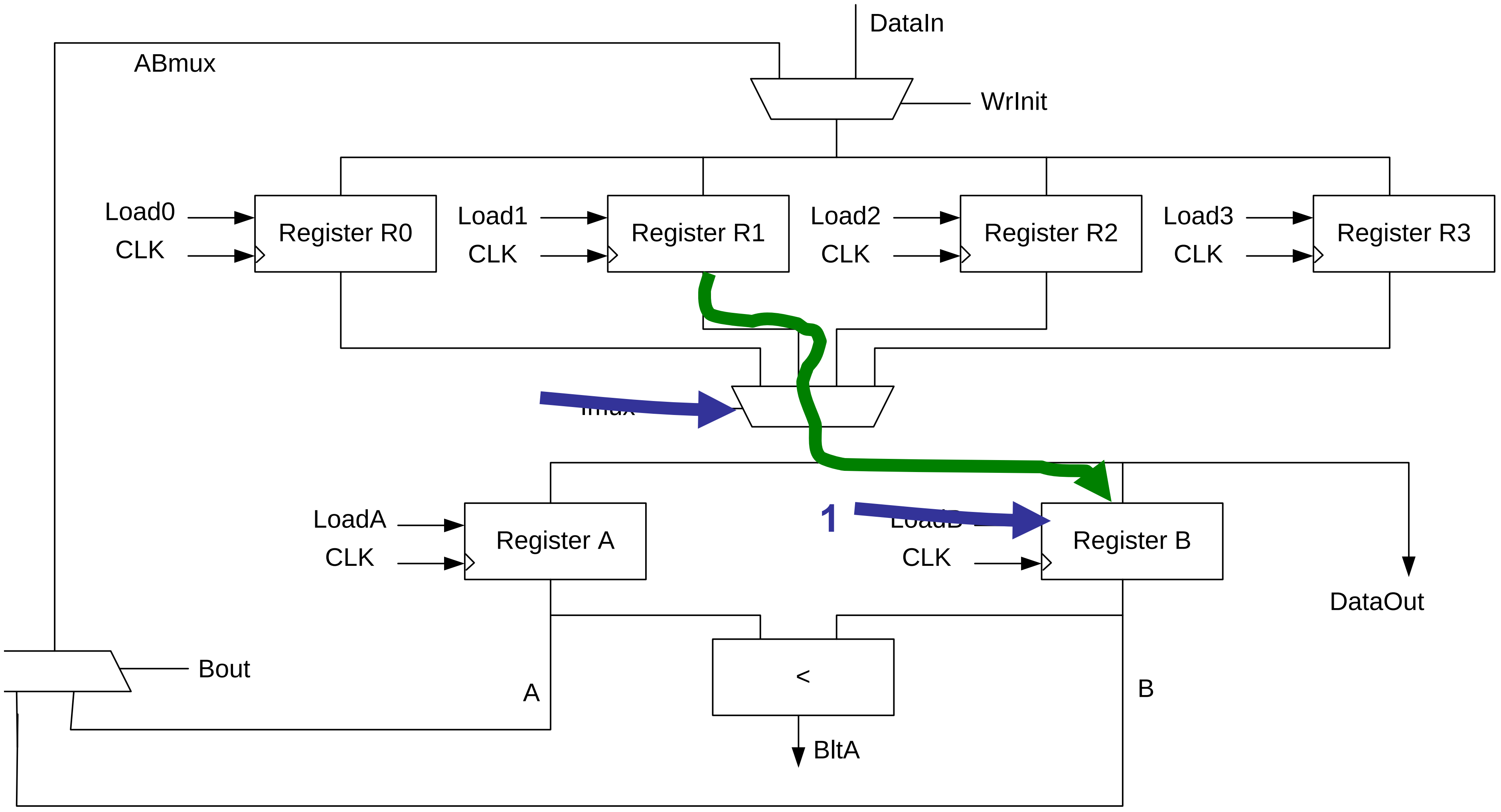
State S2: Load Register A with Ri and initialize j to value of i



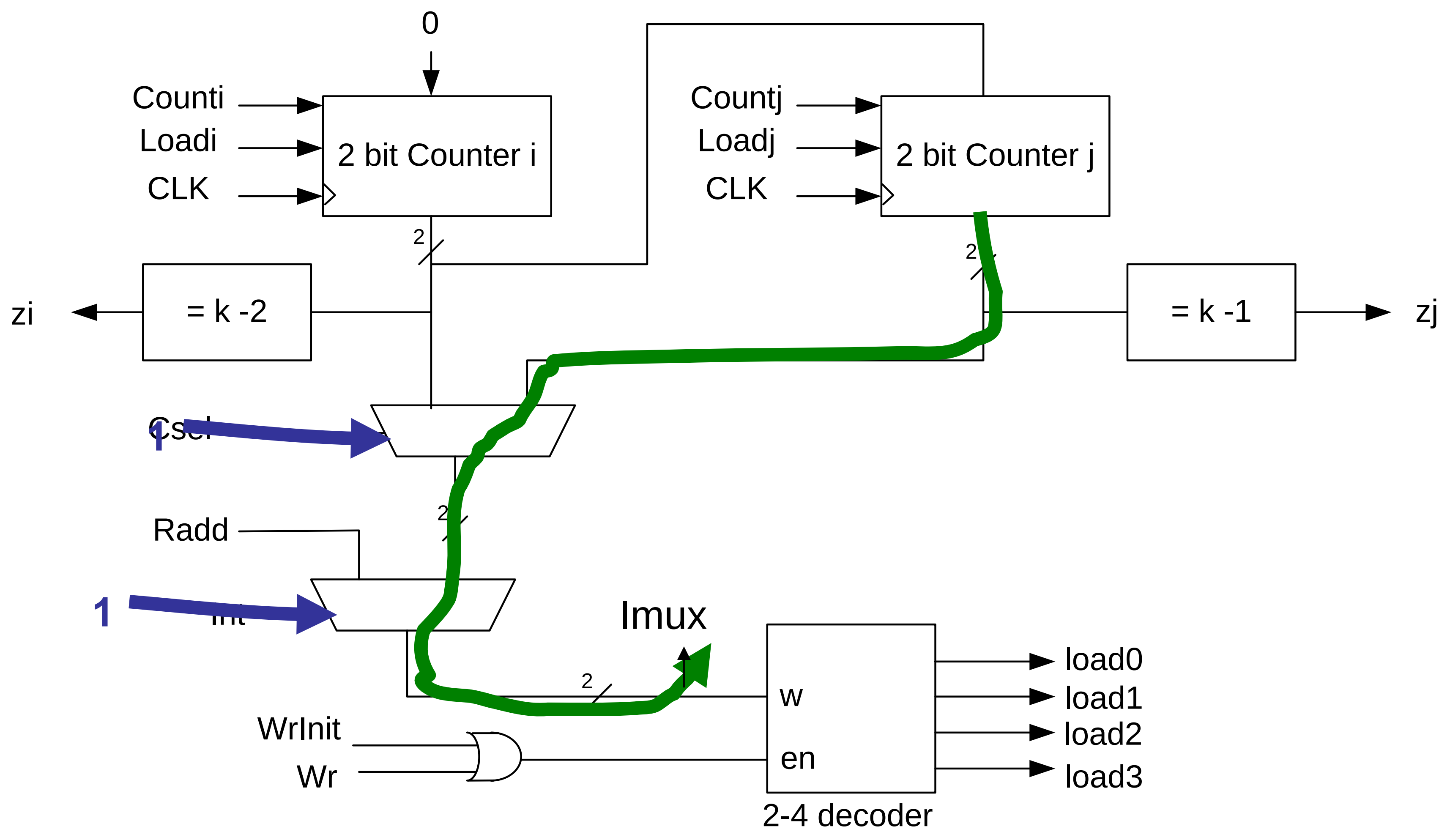
State S3: Increment j so it equals i+1



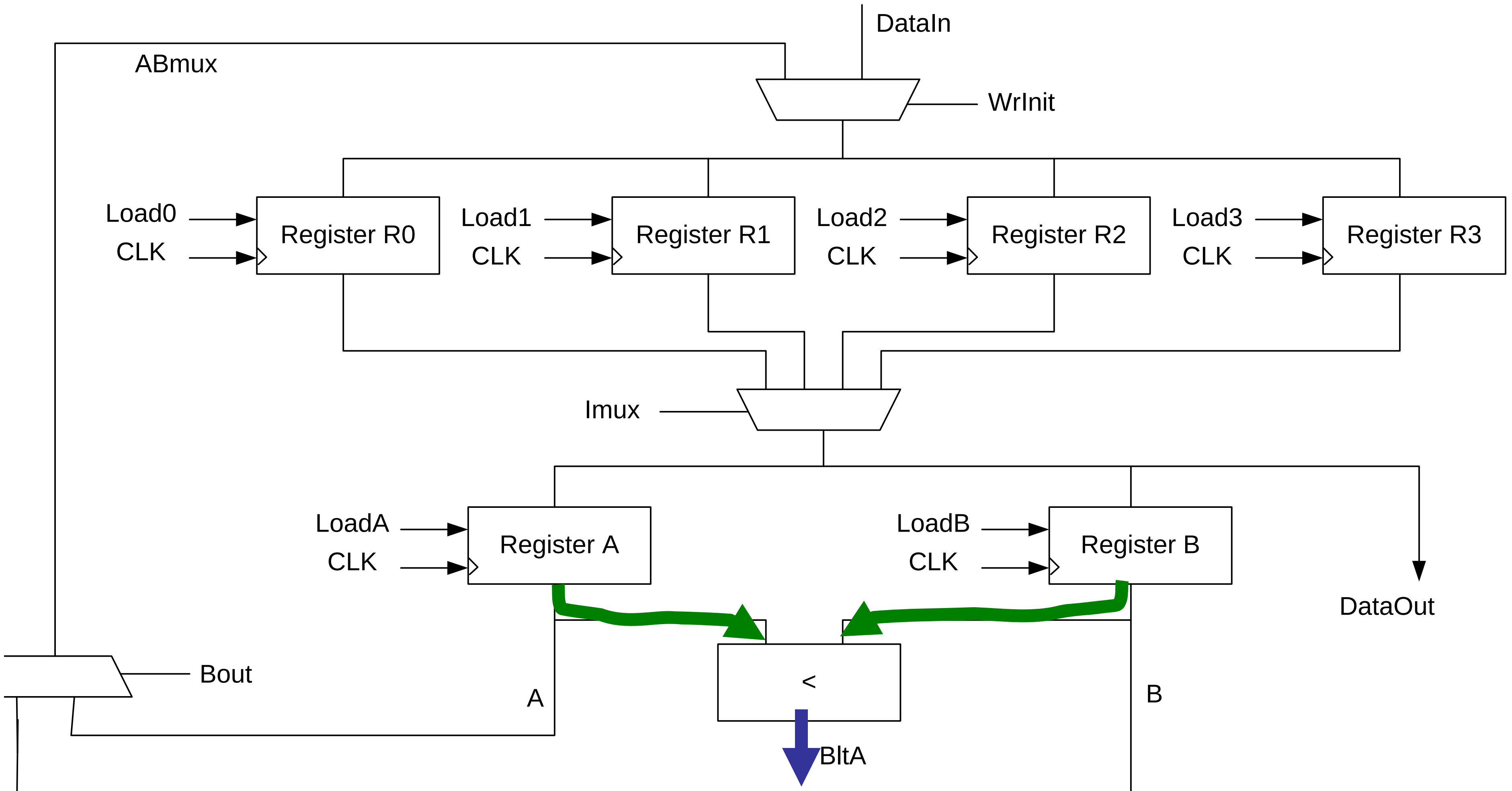
State S4: Load value of Rj into B



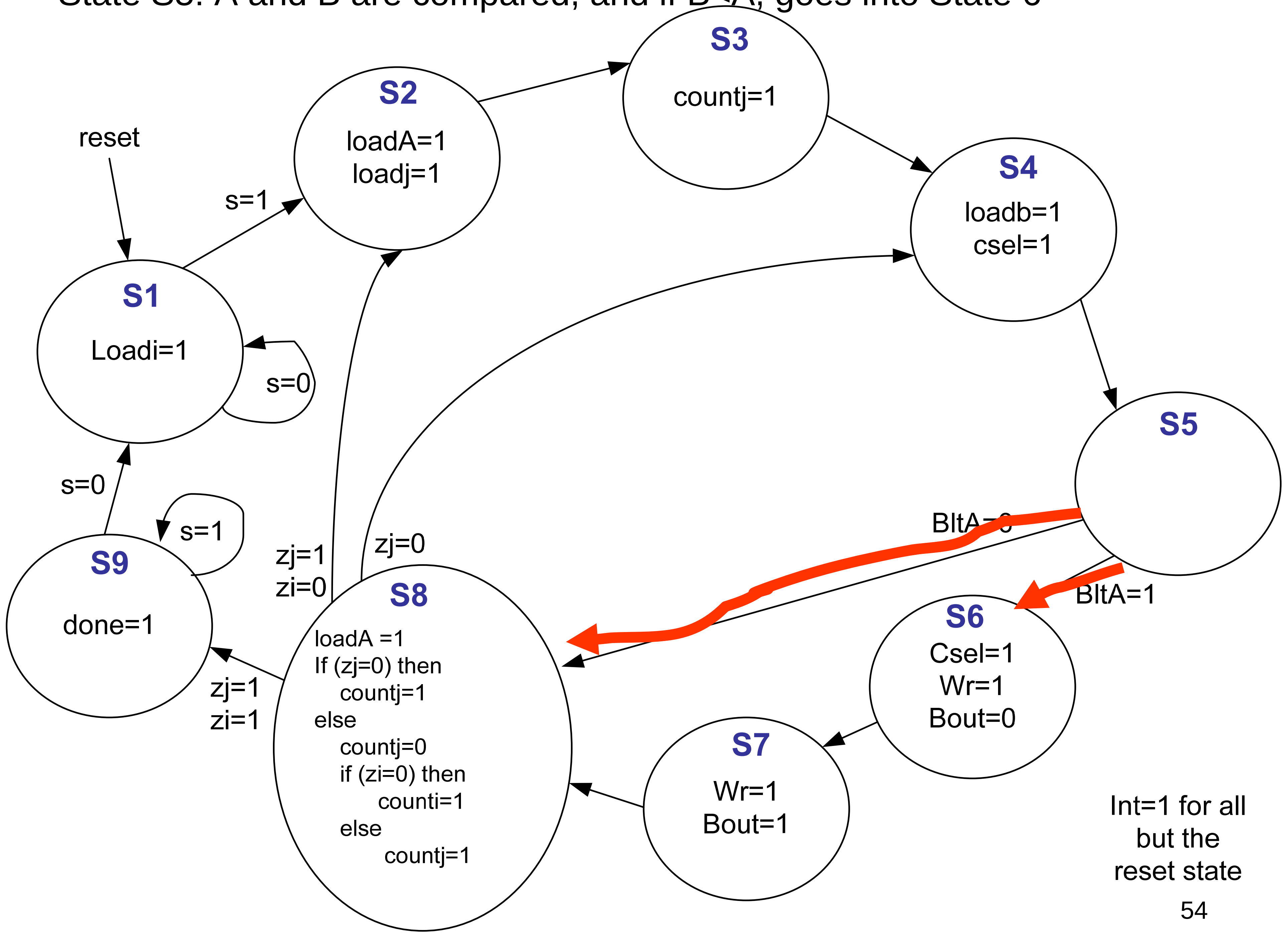
Where does Imux come from in S4?



State S5: A and B are compared, and if B<A, goes into State 6

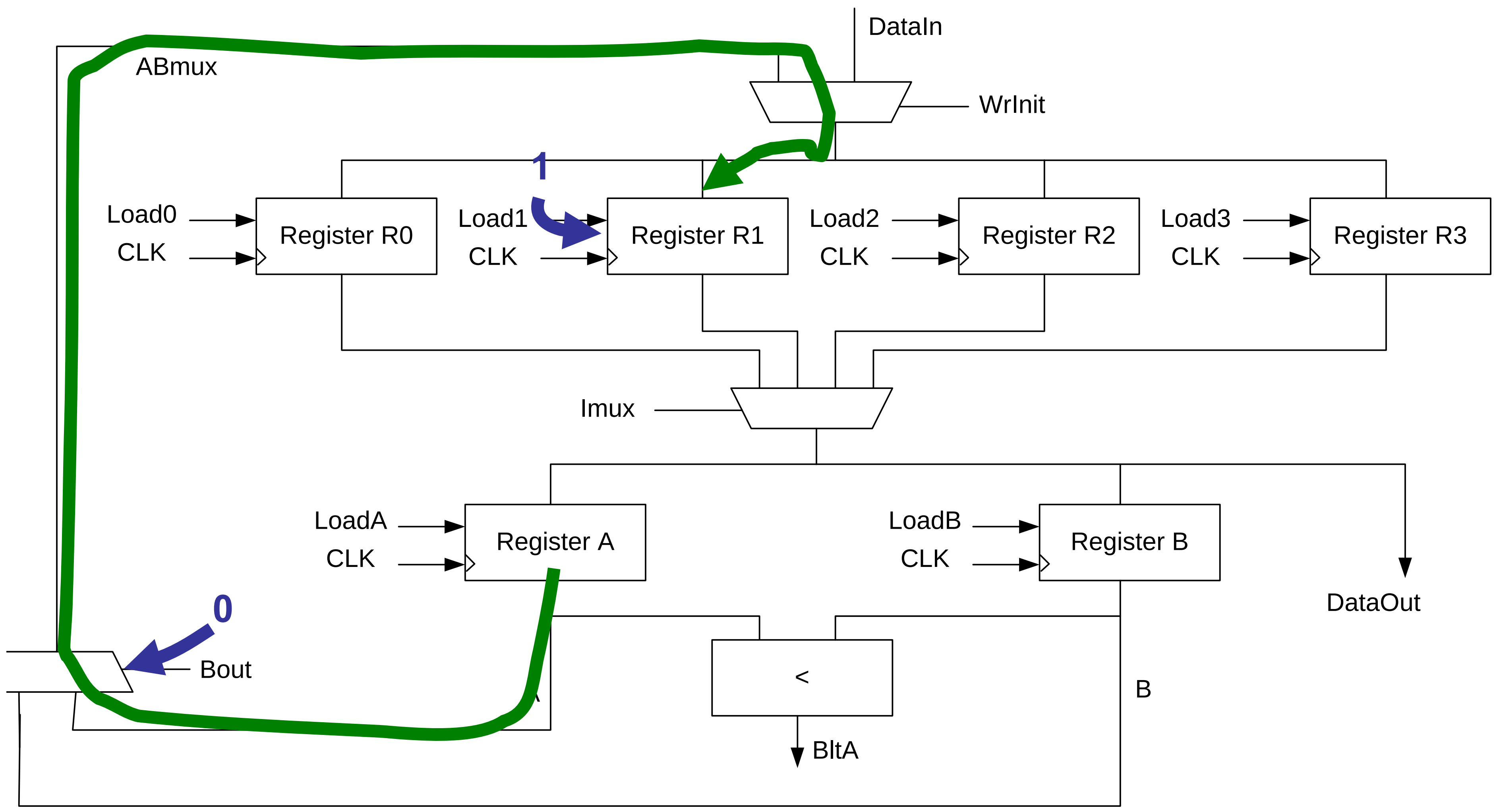


State S5: A and B are compared, and if B<A, goes into State 6

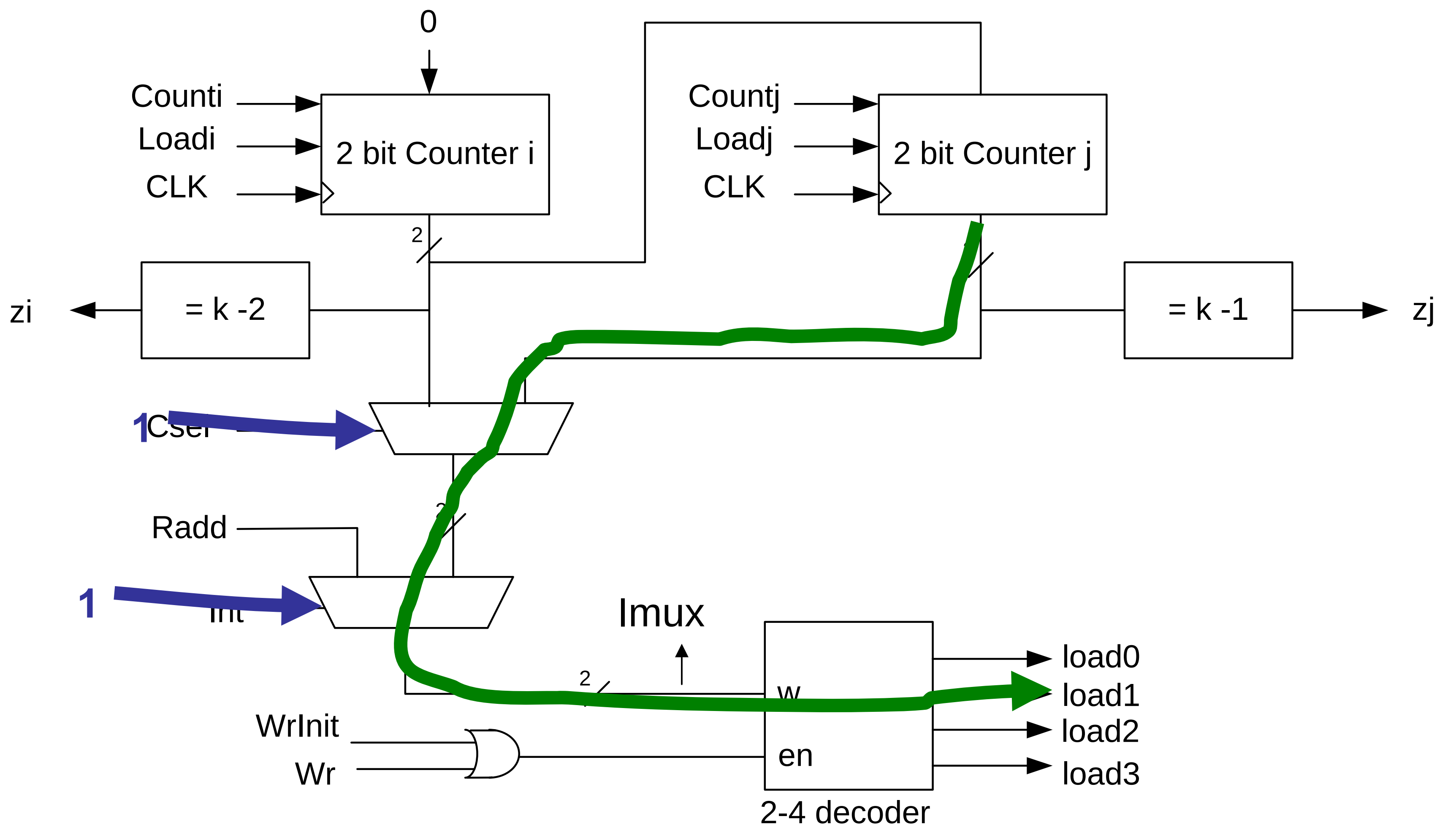


Int=1 for all but the reset state

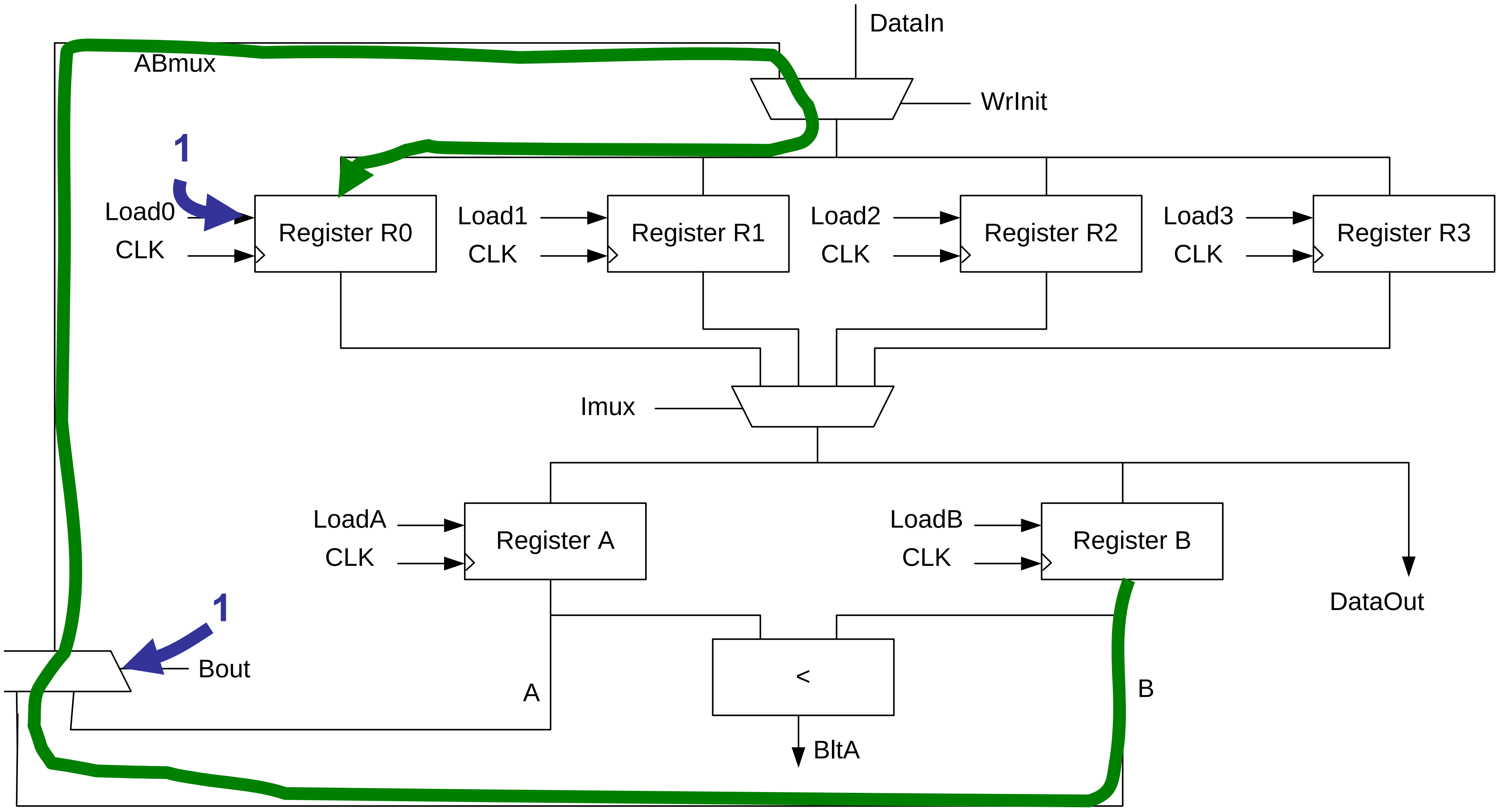
State 6: Swap Ri and Rj (part 1)



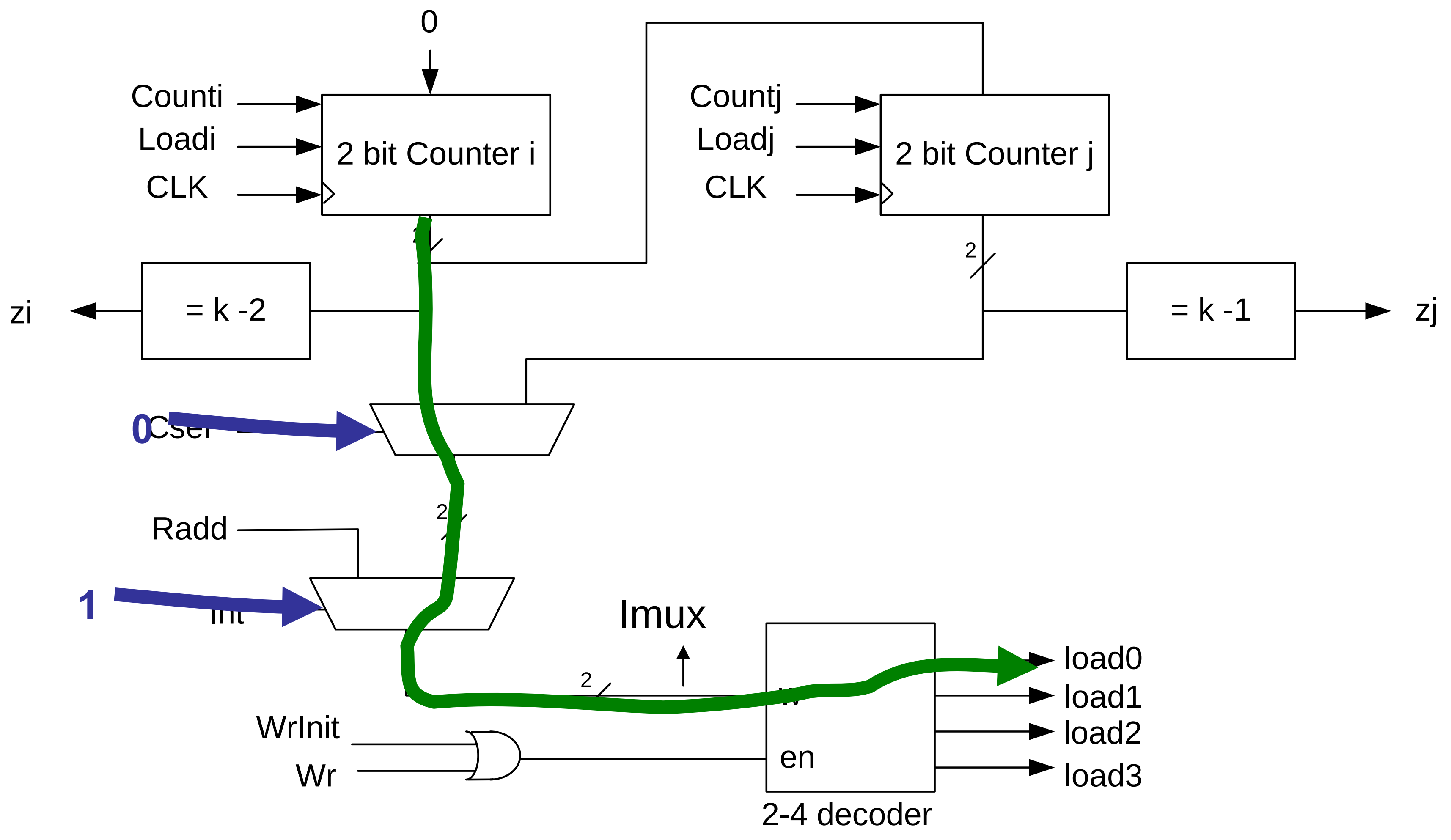
How did it know which loadj to assert?



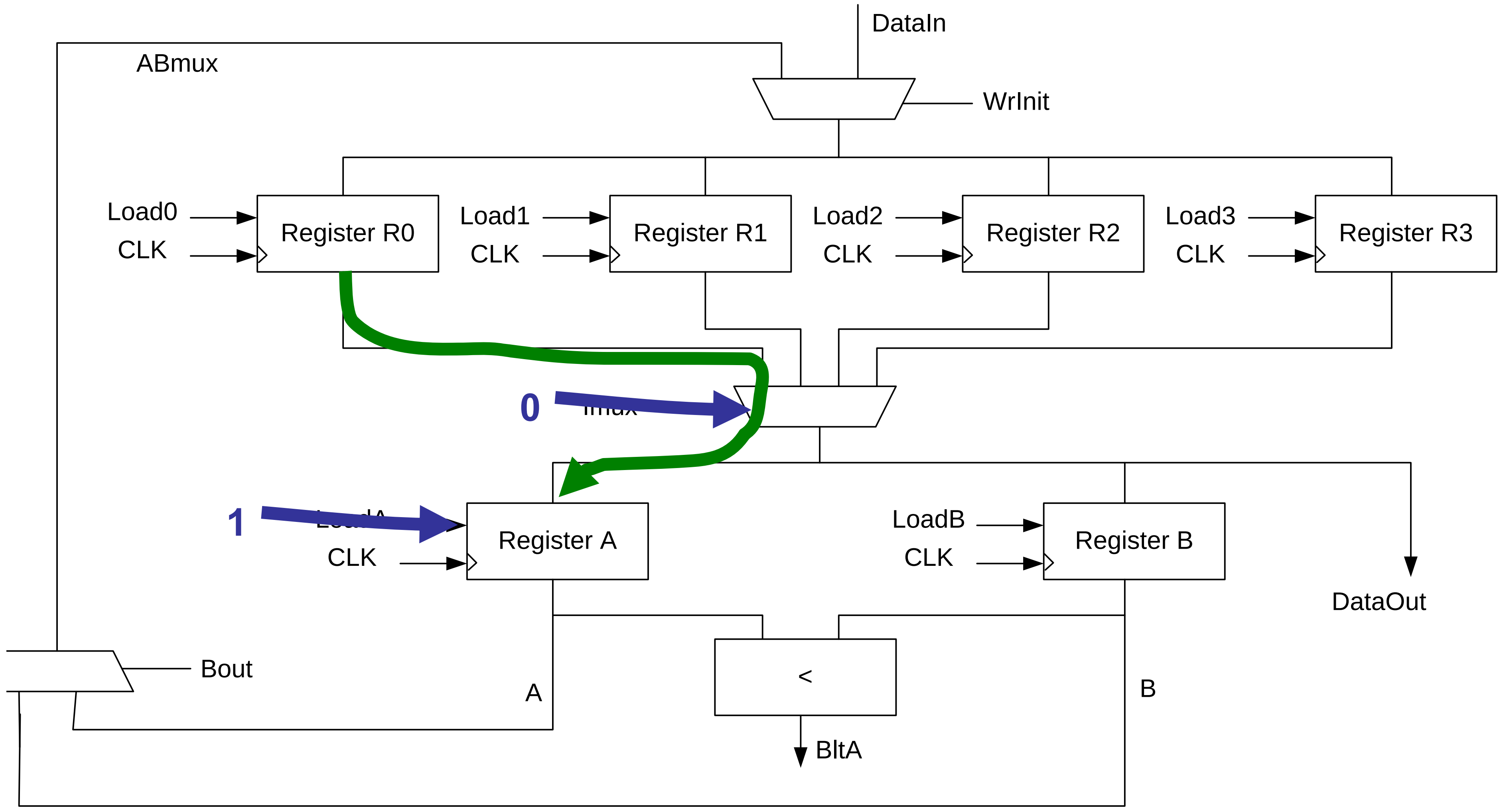
State S7: Swap Ri and Rj (part 2)



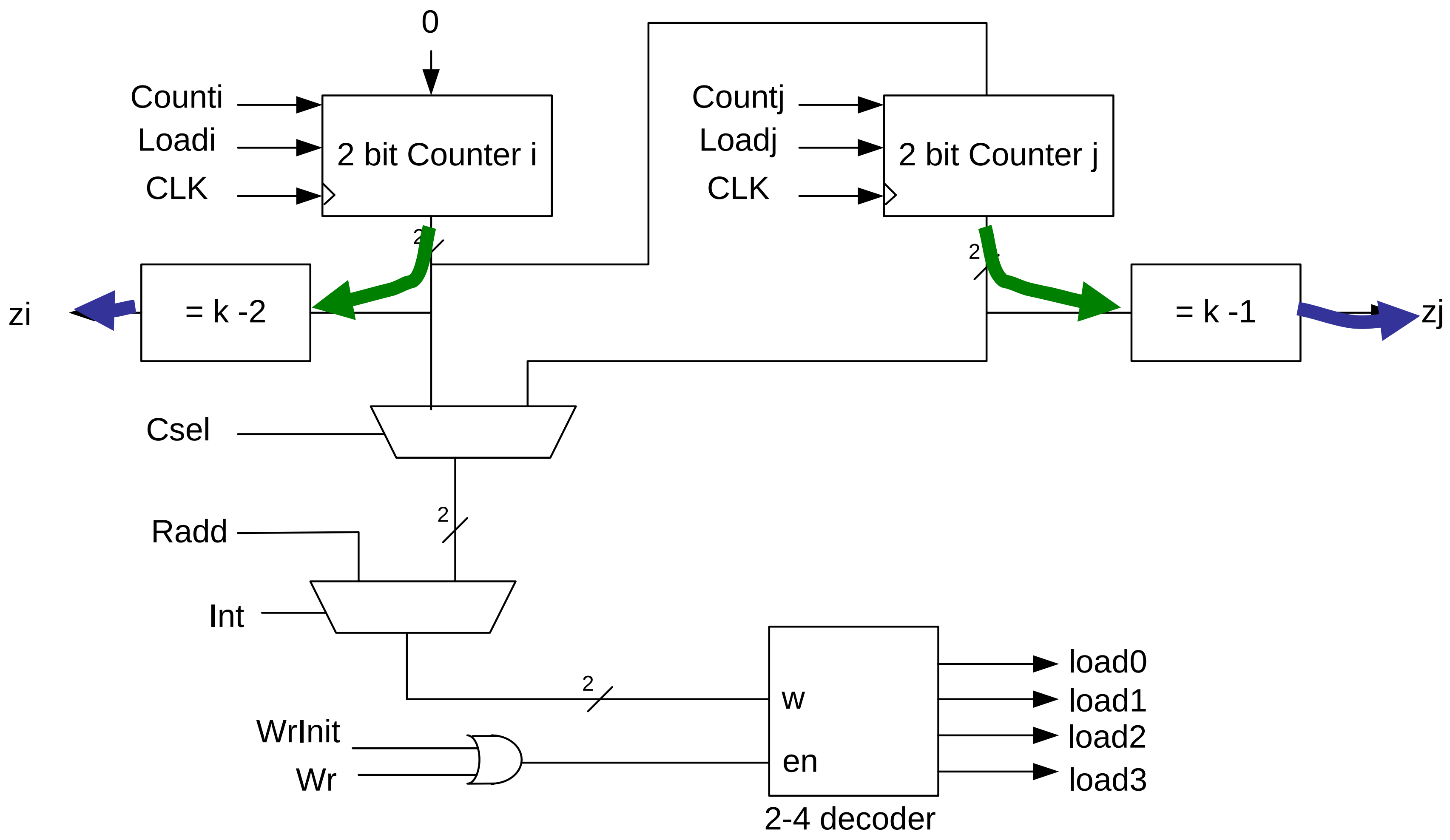
How did it know which loadi to assert?



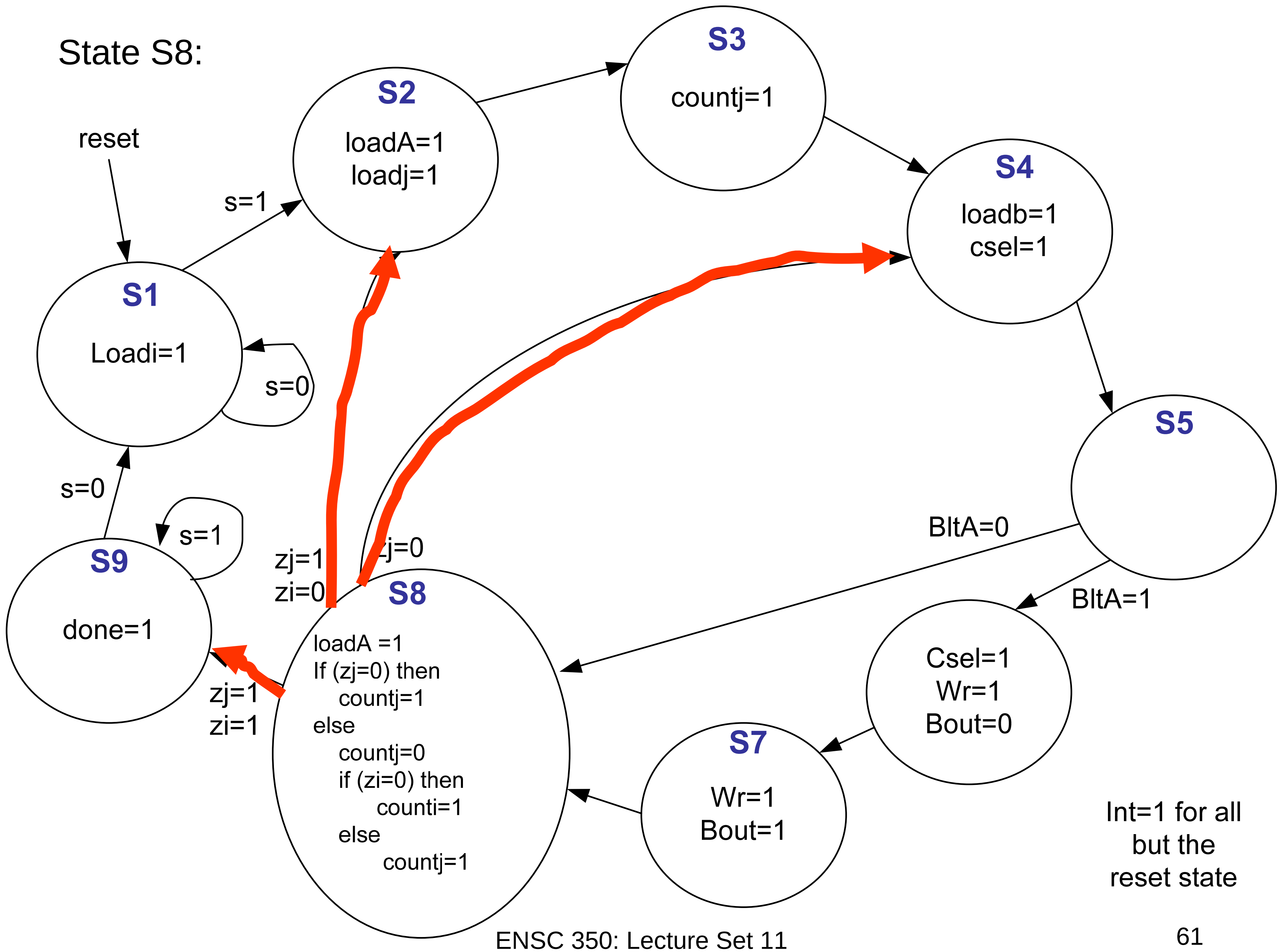
State S8: Load A from Ri



State S8: This was being done all the time, but now we will use zi and zj

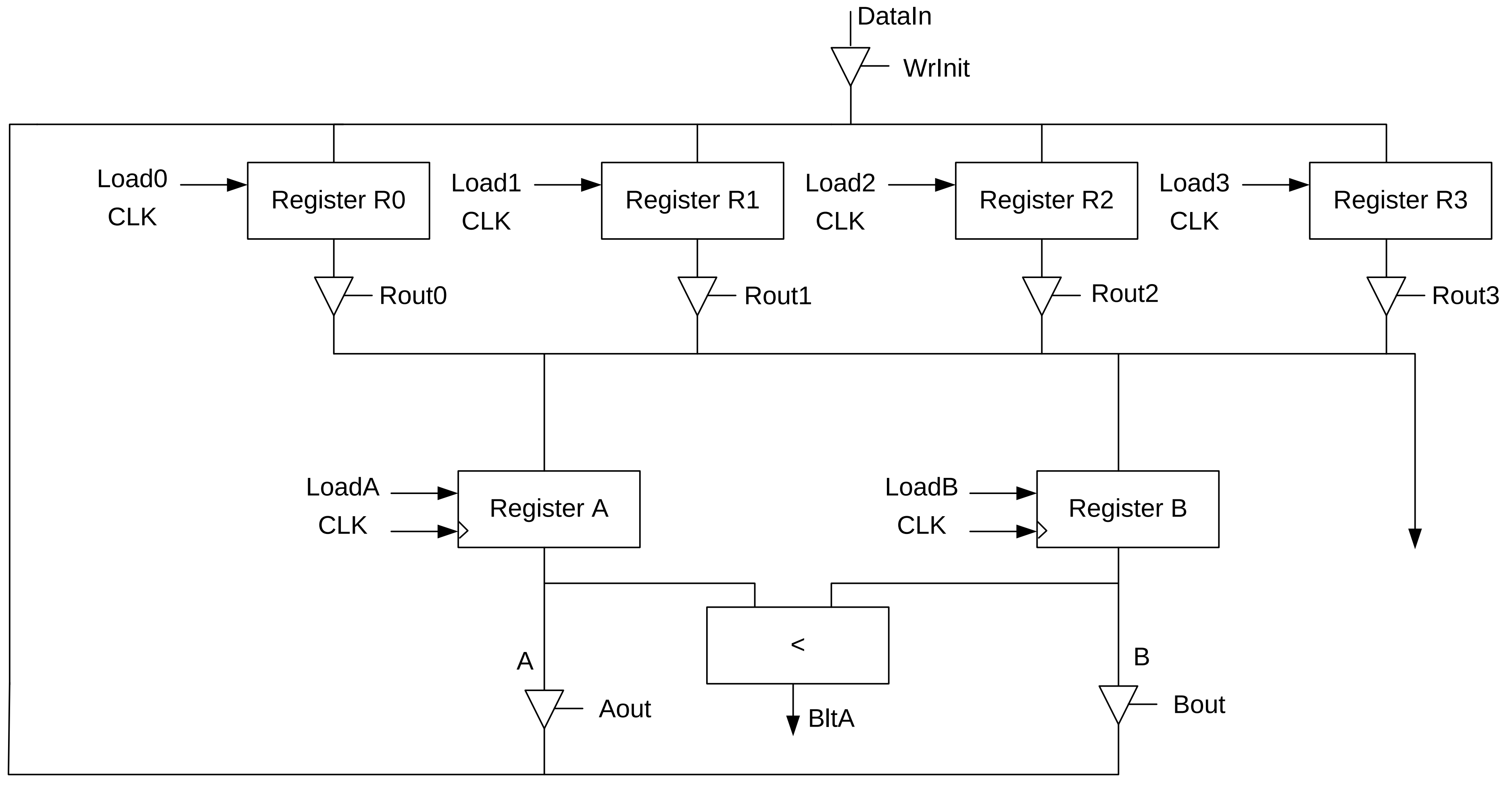


State S8:



Int=1 for all but the reset state

An alternative datapath: Tri-state buffer based datapath



Area vs. Speed

In this example, we saw two implementations:

Big and Fast

Small and Slow

In general, you can trade off area for speed. Ideally, if you double the number of functional units, then you can reduce the number of cycles by half. Rarely can you achieve this.

Which is the right implementation? Depends on how fast you need the circuit to produce results. Larger circuits cost more (more chip area, more power, higher prob. of defects), so if you don't need the speed, a small implementation is probably better.

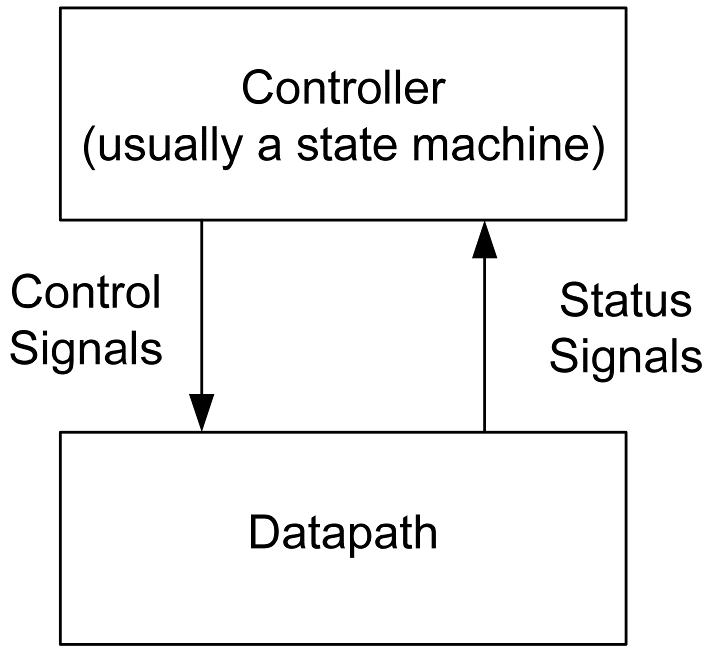
There is no general rule: as an engineer, it is up to you to choose a good implementation based on the specs you are designing to

Summary of this long Slide Set

We saw a lot of examples of datapath and control circuits

Do you need to regurgitate all the details of any of these examples on a test?

No, but you might be asked to design a simple system that contains both a datapath and controller. But, if you understand these examples, you'll be in a good position to do the design on a test, and more importantly, in the real world once you graduate (or go on co-op)



Summary of this slide set

- All of these examples – except the divider - can be found in this textbook (pages 673-712)
- They use ASM charts as opposed to FSMs
- There are no review questions for this slide set, just these examples to guide your thought process