FYSIKUM

Digital System Construction - 1

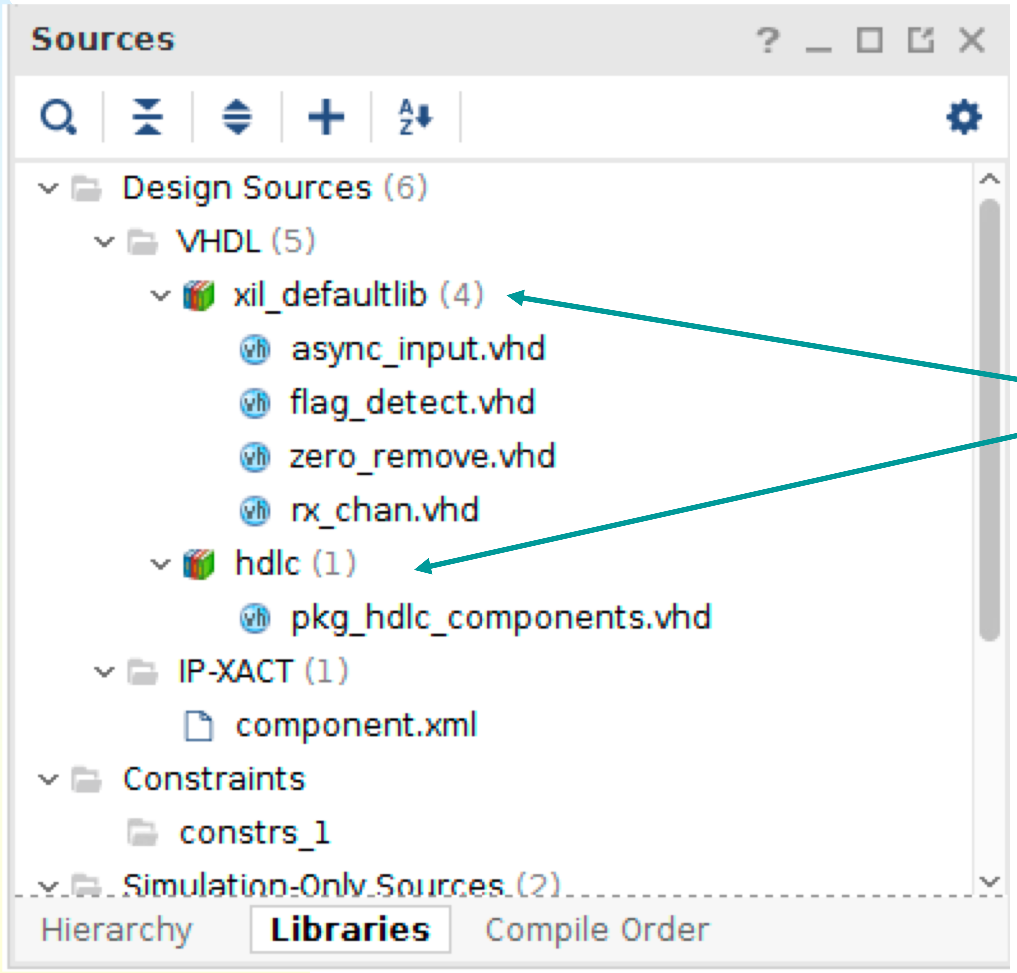
Lecture 3:
More VHDL, intro to sequential logic

- Instantiation/generics/generate
- Operators in VHDL
- Sequential logic
- Introduction to Lab 2

Component libraries

- FPGA design software organizes your VHDL modules (and other design units) in libraries
  - User-written design units are usually placed in a default library
    - In many design enviroments: “work”
    - In Xilinx Vivado: “xil_defaultlib”
  - In larger or complex projects, multiple libraries from different sources can be imported to your project
    - Good for combining custom/shared code

Libraries in Vivado



```
Design Sources (6)
  VHDL (5)
    xil_defaultlib (4)
      async_input.vhd
      flag_detect.vhd
      zero_remove.vhd
      rx_chan.vhd
    hdlc (1)
      pkg_hdlc_components.vhd
  IP-XACT (1)
    component.xml
Constraints
  constrs_1
Simulation-Only Sources (2)
```

Two VHDL libraries

Instantiating components

- The synthesis tool needs to know where to find a component.
- There are several options:
  - Declare a component, use a configuration statement to specify which library to use for it
    - (optionally entity/architecture as well)
  - Declare component without a configuration
    - Synthesis tool assumes the default library
  - Don’t declare component, instantiate directly from library
    - This is called “Entity instantiation”

With configuration

```
architecture arch of MuxExample is
   component Mux
       port(  a:     in std_logic;
              b:     in std_logic;
              sel:   in std_logic;
              y:     out std_logic
       );
   end component;
   for all : Mux use entity work.Mux; -- Configuration
   begin
       mux0: Mux
          port map (a=>a(0), b=>b(0), sel=>sel, y=>y(0));
end arch;
```

Library Entity

More complete configuration

```
architecture arch of MuxExample is
   component Mux
      port(  a:     in std_logic;
             b:     in std_logic;
             sel:   in std_logic;
             y:     out std_logic
      );
   end component;
   for all : Mux use entity work.Mux(arch1);-- Configuration
   begin
      mux0: Mux
         port map (a=>a(0), b=>b(0), sel=>sel, y=>y(0));
end arch;
```

Library Entity Architecture

“Entity instantiation”

```
architecture arch of MuxExample is

    -- No component instantiation

  begin
       mux0 : entity work.Mux -- use MUX from library work
          port map (a=>a(0), b=>b(0), sel=>sel, y=>y(0));
end arch;
```

Without configuration

```
architecture arch of MuxExample is
   component Mux
       port(  a:      in std_logic;
              b:      in std_logic;
              sel:    in std_logic;
              y:      out std_logic
       );
   end component;
   begin
       mux0: Mux  -- Use default library (e.g. work)
          port map (a=>a(0), b=>b(0), sel=>sel, y=>y(0));
end arch;
```

Generics

- Generics are <u>static</u> values or parameters
  - Only used to guide synthesis
    - Nothing that changes in run time
  - Can be any valid type
    - Integer, Boolean, time, floating point, text…
    - User defined types may also be used
- How to declare and use generics
  - Declare in the entity
    - One or more, in a list format similar to ports
  - Must be declared with a default value
    - Can override that value during instantiation
  - Can be used in both the entity port declaration and the architecture

Using generics

- Real-world examples:
  - Design units with customizable size/width
    - n-bit Adder/multiplexer/comparator/etc.
  - Register timing options
    - Choose rising/falling clock edge
  - Define initial register values, memory contents
    - Look-up tables
    - Random number generator “seed”
  - Set options for e.g. generate statements
    - (“if [generic] then….”)
- Very useful for writing portable/reusable code!

Generic declaration

- Example from ATLAS trigger upgrade:

```
component ClusterSort is
   generic (output_width        : integer:=  5; --width of the sort output
            input_chan_width    : integer:= 20; --number of input channels
            input_tobs          : integer:= 120 );  --number of TOBs
   port   (clk40            : in std_logic; -- 40 MHz clock
           Iso_Mask         : in std_logic_vector (4 downto 0);
           InputArr         : in  ClusterArray (input_tobs-1 downto 0);
           OutputArr        : out TOBArray (output_width-1 downto 0) );
end component:
```

Sorting algorithm with adjustable sizing

Generate statement

- Generate is a <u>concurrent loop </u>statement
  - Used to replicate concurrent statements
- Has two forms
  - for: generates a fixed number of concurrent statements (most commonly used form)
    - **for** i **in** 0 to 15 **generate**
  - if: conditionally selects concurrent statements to be generated
    - **if** i = 2 generate
- Generate loops can be <u>nested</u>

Example: n-bit MUX

- Use a **generic** integer in entity declaration to define the width of the inputs/outputs
- Use **generate** in the architecture to instantiate the right number of one-bit MUXes

nx2 multiplexer entity

Generic nmux_width sets the port widths

```
entity MuxN2 is
generic (nmux_width : integer := 4;
        );                                variable width
port(a:  in std_logic_vector (nmux_width-1 downto 0);
     b:  in std_logic_vector (nmux_width-1 downto 0);
     sel: in std_logic;
     y:  out std_logic_vector (nmux_width-1 downto 0)
    );
end MuxN2;
```

Nx2 mux architecture

Generic nmux_width used with generate

```
architecture arch of MuxN2 is
   component Mux
       port(  a:     in std_logic;
              b:     in std_logic;
              sel:   in std_logic;
              y:     out std_logic
       );
   end component;

 begin
       g0 : for i in 0 to (nmux_width-1) generate
         u1: Mux
           port map (a=>a(i), b=>b(i), sel=>sel, y=>y(i));
       end generate g0:
end arch;
```

1-bit MUX

4-bit adder (complicated)

```
architecture arch1 of add4 is
   component full_adder
        port(    a, b, cin :       in std_logic;
                 cout, s: out std_logic);
   end component;
   signal carry_bit: std_logic_vector(3 downto 1);
   begin
        four_adder : for i in 0 to 3 generate
             bottom_adder: if i = 0 generate
                 a_bot : full_adder
                 port map (a=>a(0), b=>b(0), cin=>carry_in,
                    cout=>carry_bit(1), s=>s(0));
                 end generate bottom_adder;
             middle_adders: if (i>0 and i<3) generate
                 a_mid : full_adder
                port map (a=>a(i), b=>b(i), cin=>carry_bit(i),
                       cout=>carry_bit(i+1), s=>s(i));
                 end generate middle_adders,
             top_adder: if i = 3 generate
                 a_top : full_adder
                 port map (a=>a(3), b=>b(3), cin=>carry_bit(3),
                    cout=>carry_out, s=>s(3));
                 end generate top_adder;
        end generate four_adder;
end arch1;
```

FOR loop

Conditional

Conditional

Conditional

4-bit adder (simpler)

```
architecture arch1 of add4 is

   component full_adder
         port(    a, b, cin :       in std_logic;
                  cout, s: out std_logic);
   end component;

   signal carry_bit: std_logic_vector(4 downto 0);

   begin
         four_adder : for i in 0 to 3 generate
             add_one_bit: full_adder
                  port map (a=>a(i), b=>b(i), cin=>carry_bit(i),
                     cout=>carry_bit(i+1), s=>s(i));
         end generate four_adder;

         carry_bit(0) <= cin;
         cout <= carry_bit(4);

end arch1;
```

} FOR statement

Operators in VHDL

(Not an exhaustive list)

- Logical (for Boolean or bit operations)
  - and, or, nand, nor, xor, not
- Relational
  - =, /=, <, <=, >, >=
- Arithmetic
  - +, -, abs, mod, rem, *, /
- Concatenate
  - & (example c <= a & b;)

Concatenation operator (&)

- Used for linking together bits and vectors to build a longer vector
- Examples:

```
signal a: std_logic_vector(3 downto 0) := “1001”;
signal b: std_logic_vector(6 downto 0);

b <= “101” & a;                -- Result: b = “1011001”

a <= ‘1’ & a(3 downto 1);      -- Result: a = “1100”

b <= ‘1’ & “11” & “0000”;      -- Result: b = “1110000”;
```

Two assignment operators:

- Variable assignment (:=)
  - Syntax: variable := value;
  - Immediately assigns new value to a variable within a process
  - Also to initialize signals, variables, constants
- Signal assignment (<=)
  - Syntax: signal <= value [after time];
  - Delayed assignment of new value to a signal
  - If used in a process, signal value changes at the end of the process.

Assignment operators used in a process (example)

```
parity_calc : process (a,b)

   variable temp: std_logic;

   begin

       temp := '1';           -- sequential code!
       temp := temp xor a;
       temp := temp xor b;

       parity_out <= temp;   -- Assign the result to a signal

end process;
```

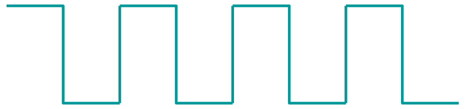
Sequential logic

Sequential logic

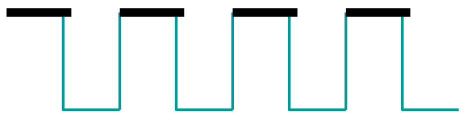
- Sequential code must be written inside processes
- Registers (flip-flops) created by making a process dependent on a “clock” signal
- Multiple ways to code conditional statements
  - "if…then” is very common
    - Good for clock statements and asynchronous behavior
  - “case” is useful for coding state machines
  - "wait until…” also possible
    - But not recommended for synthesis

Latches vs flip-flops

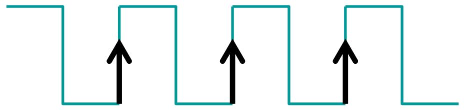
- Both rely on a <u>clock</u> signal



- Latches change outputs on a clock <u>level</u> (high or low)

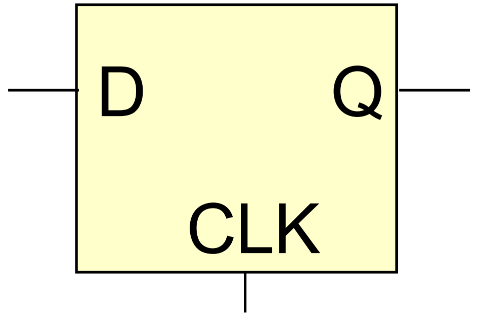


- Flip-flop outputs change on a clock <u>edge</u> (rising or falling)



Example: D-latch

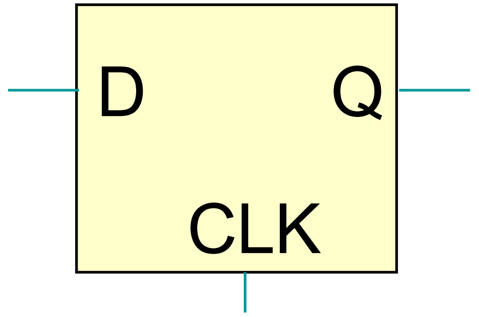
(not recommended for use in FPGAs)



```
process (CLK)
     begin
     if CLK='1' then
          Q <= D;
     else
          -- Don't do anything
     end if;
end process;
```

Or more concisely…

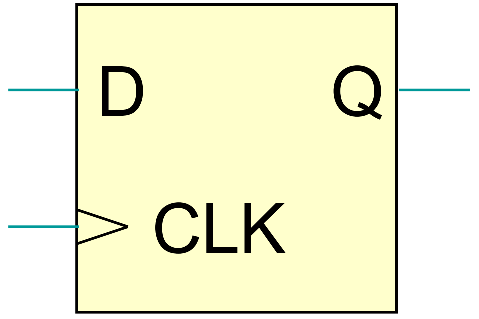
```
process (CLK)
      begin
      if CLK='1' then
            Q <= D;
      end if;
end process;
```



The empty 'else' condition is implied

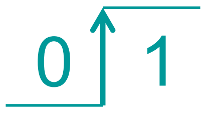
D flip-flop

Using native syntax:



```
process (CLK)
     begin
     if CLK'event and CLK='1' then
          Q <= D;
     end if;
end process;
```

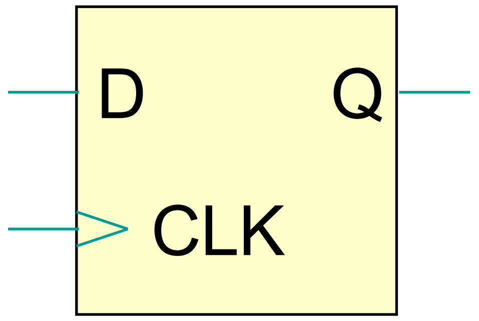
Rising edge



D flip-flop

Using IEEE 1164 std_logic syntax:
(recommended)

```
process (CLK)
      begin
      if rising_edge(CLK) then
            Q <= D;
      end if;
end process;
```

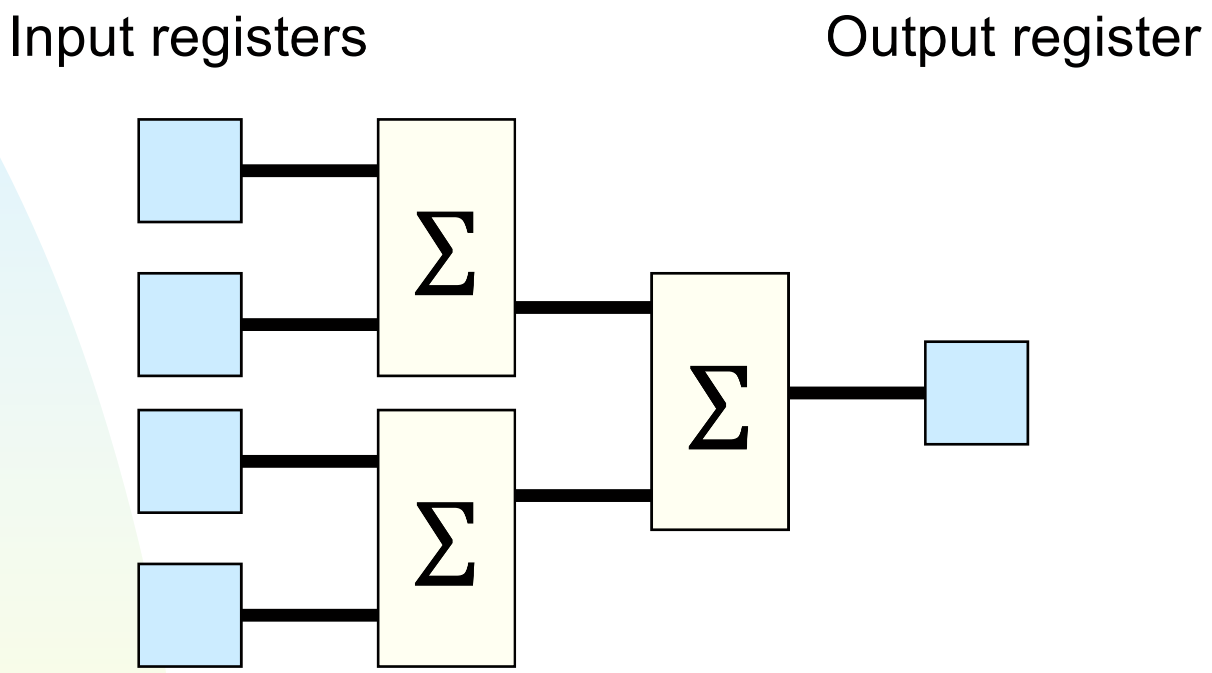


(can also use `falling_edge`)

Common sequential designs

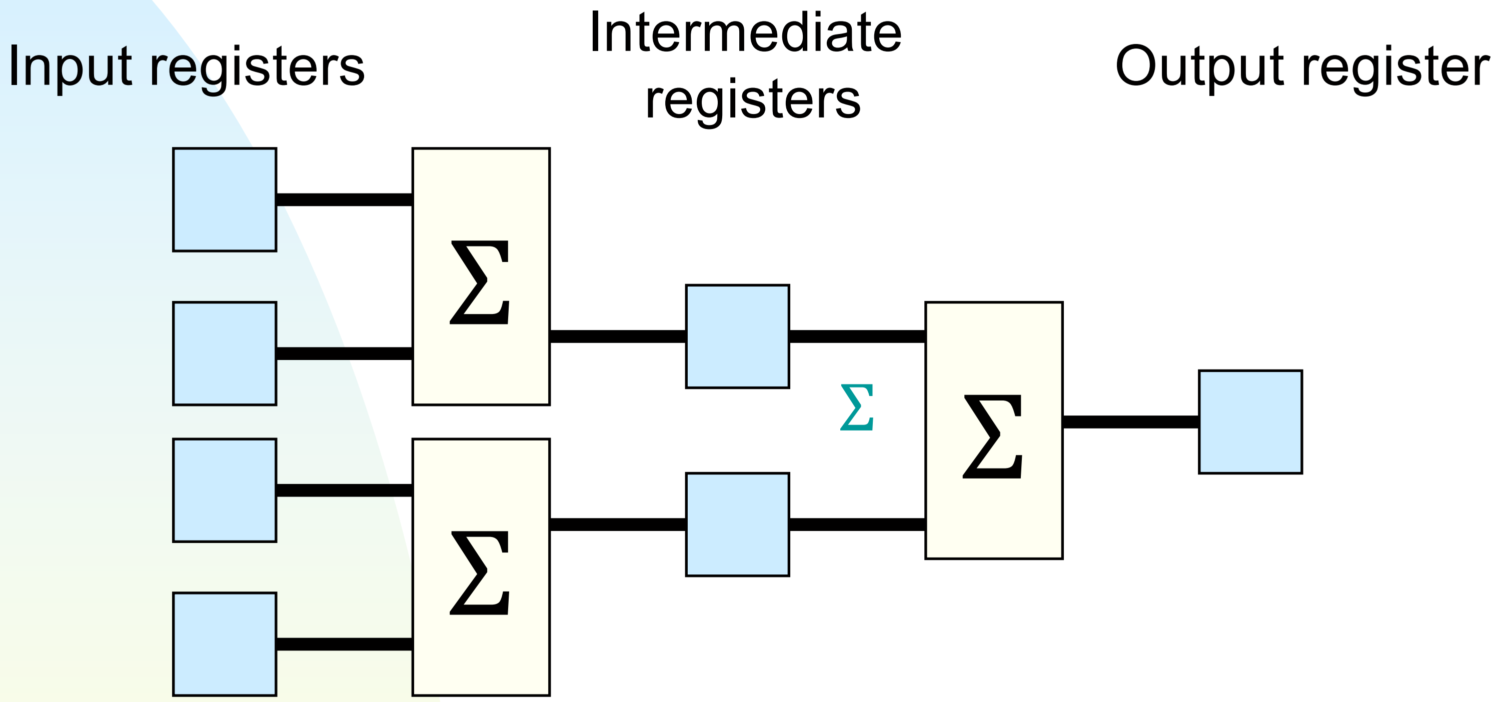
- Finite State Machine (FSM)
  - State information saved from previous operations, used in next operation
- Pipelined algorithms
  - Sequential operations performed on data in multiple steps

Pipelined algorithms



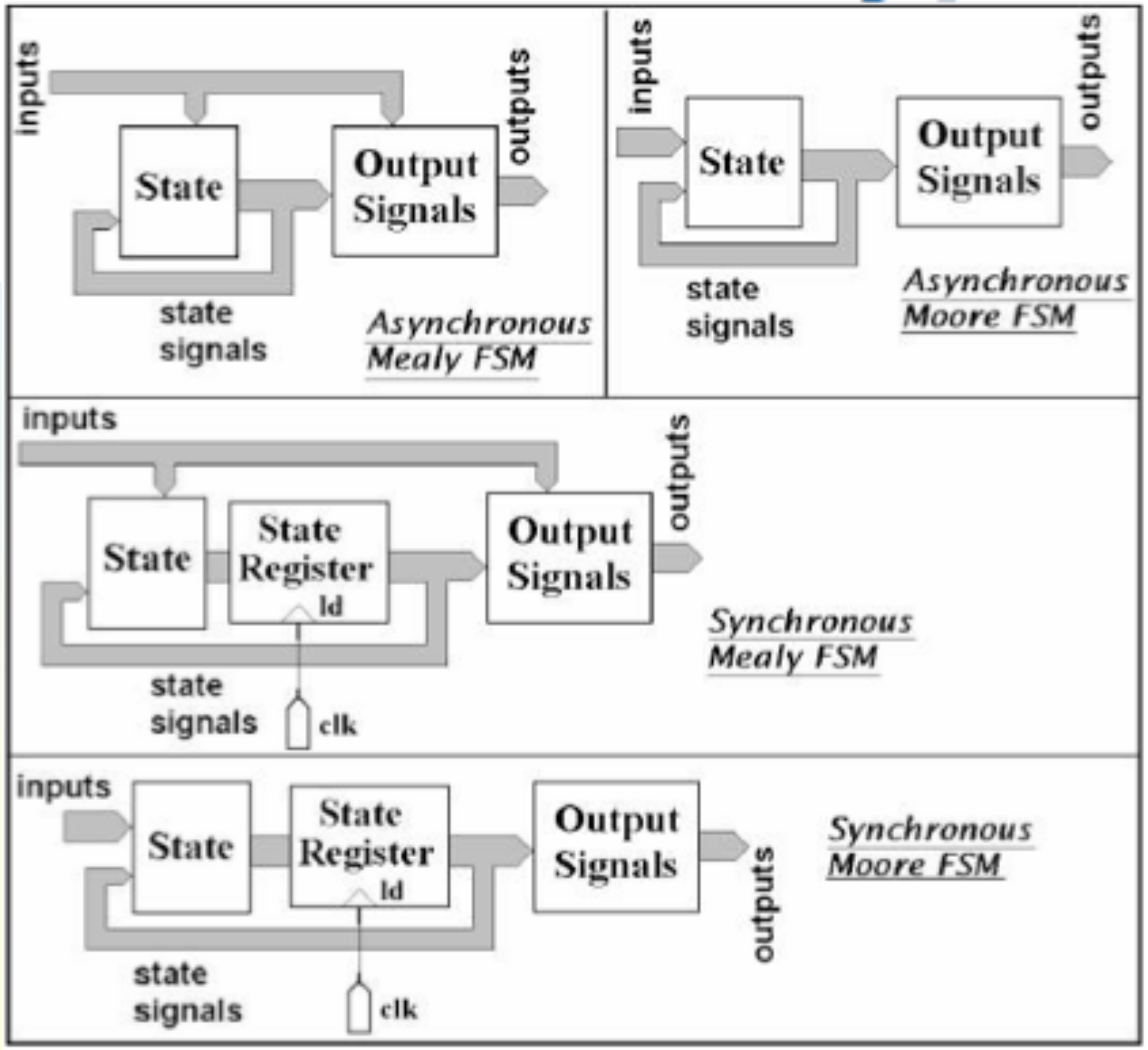
Single-stage summation

Pipelined algorithms



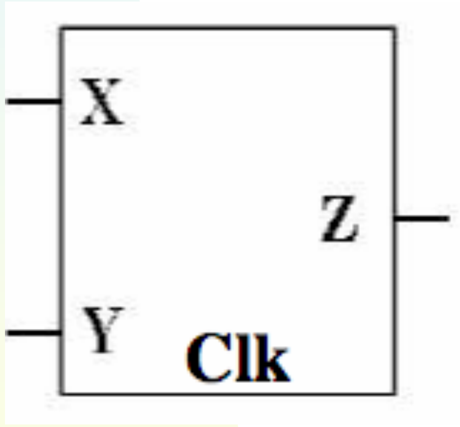
Multi-stage summation:
Faster clock speed
(but longer latency)

FSM types


inputs
State
Output Signals
outputs
state signals
Asynchronous Mealy FSM
inputs
State
Output Signals
outputs
state signals
Asynchronous Moore FSM
inputs
State
State Register
ld
Output Signals
outputs
state signals
clk
Synchronous Mealy FSM
inputs
State
State Register
ld
Output Signals
outputs
state signals
clk
Synchronous Moore FSM

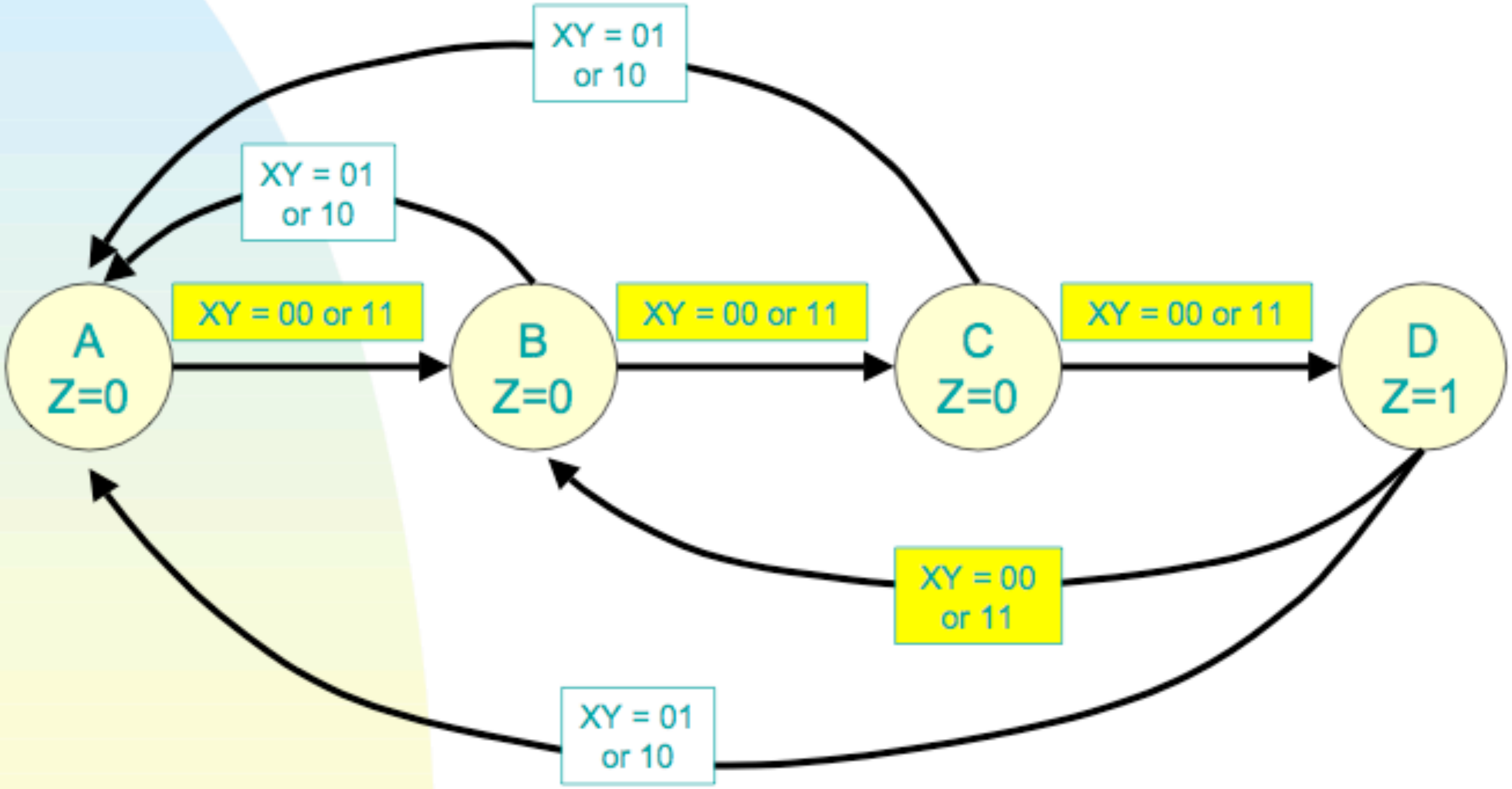
Review: synchronous Moore FSM

- Let Z equal 1 after inputs X and Y are equal 3 times in a row:



```
x= 01100111010001010110100001
y= 11101111011001010110101011
z= 00010001000001001001000000
```

Flow chart for the FSM



Entity for the FSM:

```
entity FSM is
port(x:   in std_logic;
     y:   in std_logic;
     clk: in std_logic;
     z:   out std_logic
    );
end FSM;
```

Behavioral architecture

```
architecture behav of FSM is

signal state: std_logic_vector (1 downto 0);

begin

reg1 : process (clk, x, y, state)
   begin
     if rising_edge(clk) then
       case state is
          when "00" =>
             if (x = y) then state <= "01"; else state <= "00";
             end if;
          when "01" =>
             if (x = y) then state <= "10"; else state <= "00";
             end if;
          when "10" =>
             if (x = y) then state <= "11"; else state <= "00";
             end if;
          when others =>
             if (x = y) then state <= "01"; else state <= "00";
             end if;
       end case;
     end if;
   end process;

   z <= state(1) and state(0);

end behav;
```

If..then with a clock edge implements the flip-flops

Clear description using “case”

} Output

Assignment operator: <=

- Implies a time delay from right-hand side to the left-hand side of the operator
  - Signal value is updated at “end process”
- If signal assignment depends on a clock edge, a flip-flop is implemented.
  - state (after clock edge) <= state (before clock edge)

Order is not important for concurrent statements

```
architecture behav of FSM is

signal state: std_logic_vector (1 downto 0);

begin

 z <= state(1) and state(0);
```

} Output is <u>concurrent</u> with the process

```
reg1 : process (clk, x, y, state)
   begin
     if rising_edge(clk) then
       case state is
          when "00" =>
             if (x = y) then state <= "01"; else state <= "00";
             end if;
          when "01" =>
             if (x = y) then state <= "10"; else state <= "00";
             end if;
          when "10" =>
             if (x = y) then state <= "11"; else state <= "00";
             end if;
          when others =>
             if (x = y) then state <= "01"; else state <= "00";
             end if;
       end case;
     end if;
   end process;

end behav;
```

Order not important in concurrent code

```
architecture rtl of FSM is

signal state : std_logic_vector (1 downto 0);

begin
   z <= state(1) and state(0);

   reg1: process (clk, x, y, state)
   begin
     if (rising_edge(clk)) then
         state(1) <= (not (x xor y)) and (state(1) xor state(0));
         state(0) <= (not (x xor y)) and (state(1) or not (state(0)));
     end if;
   end process;

end rtl;
```

} Concurrent with the process

Example: 4-bit counter (entity)

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; -- New library!

entity counter is

  generic(n: natural := 4); -- Width of the counter
  port(clock:      in std_logic;
       clear:      in std_logic;
       Q:          out std_logic_vector(n-1 downto 0)
  );
end counter;
```

4-bit counter (architecture)

```
architecture behv of counter is

    signal Count: std_logic_vector(n-1 downto 0);

begin

    process(clock, clear)
    begin
       if clear = '1' then
           Count <= Count - Count;
       elsif rising_edge(clock) then
           Count <= Count + 1;
       end if;
    end process;

    Q <= Count;

end behv;
```

Asynchronous reset

Synchronous with clock

Unsigned + and - operations

Output

Another way to write it:

```
architecture behv of counter is

    signal Count: std_logic_vector(n-1 downto 0);

begin

    process(clock, clear)
    begin
       if clear = '1' then
           Count <= (others => ‘0’);
       elsif rising_edge(clock) then
           Count <= Count + 1;
       end if;
    end process;

    Q <= Count;

end behv;
```

Fill all elements of vector with ‘0’

Unsigned + and - operations

Output

Why use Count internally, instead of just using Q?

- Q is an <u>output</u> of the entity
- An architecture can only <u>write</u> to an output
  - Not read from it!
- So here we use an internal signal (Count)
  - Count can be both written and read.
  - Output Q is then set to the value of Count
  - Signals like count are sometimes called <u>buffers</u>

Process sensitivities

- Sensitivity list is part of a process declaration
- Process outputs can change when at least one signal in the sensitivity list changes
- Safest to include all input signals in the list
  - Not necessary, but synthesis may give fewer warnings…
- Missing (important) sensitivities are discovered most often in simulation

Sensitivity example: FSM

```
reg1: process (clk)
begin
   if (rising_edge(clk)) then state <= next_state;
   end if;
end process;
```

This works because “clk” is always changing in every cycle.
But “next_state” can also change the output!

Example: FSM

```
reg1: process (clk, next_state)
begin
   if (rising_edge(clk)) then state <= next_state;
   end if;
end process;
```

This is a “safer" way to do the same thing
(or at least get fewer warnings)

Example: counter

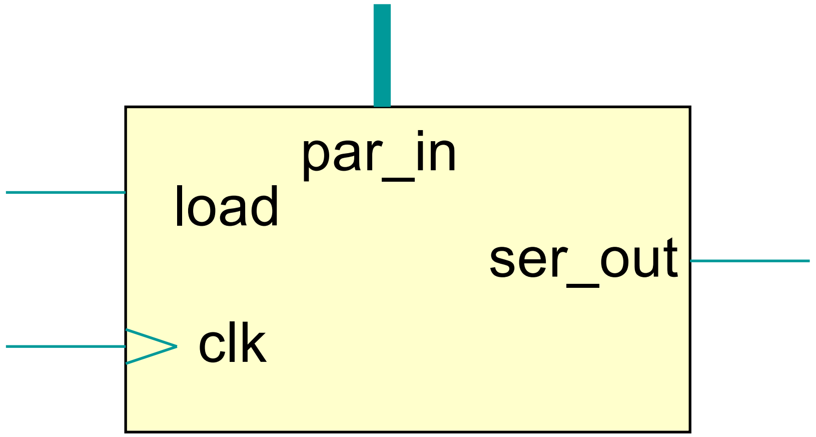
```
process(clock, clear)
    begin
     if clear = '1' then
         Count <= (others => '0');
     elsif (clock='1' and clock'event) then
         Count <= Count + 1;
     end if;
end process;
```

`clear` is a required sensitivity, because it does not have to change at the same time as `clock`.

Example: simple shift register

Function: load a parallel value, then circular shift to the right, with the least significant bit sent to ser_out

```
entity simple_sr is
  port(par_in:  in std_logic_vector(3 downto 0);
       load:    in std_logic;
       clk:     in std_logic;
       ser_out: out std_logic
      );
end simple_sr;
```



simple_sr architecture

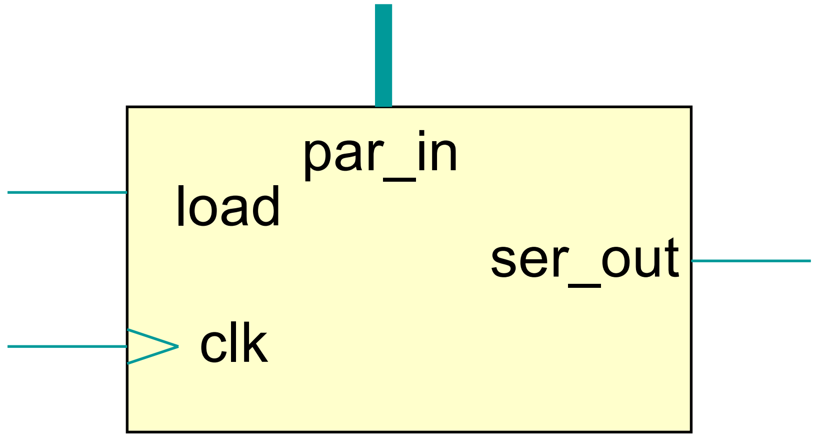
```
architecture behav of simple_sr is

signal val : std_logic_vector(3 downto 0);
begin
reg1: process (clk, load, par_in, val)
    begin
      if (rising_edge(clk)) then
        case load is
           when '1' => val <= par_in; -- load parallel value
           when others => val <= val(0) & val(3 downto 1); -- shift
        end case;
      end if;
    end process;

  ser_out <= val(0); -- Output

end behav;
```

Use "&" operator to move the LSB up to MSB, and shift down the top three bits



Be careful to avoid <u>extra</u> registers

```
architecture behav3 of simple_sr is

signal val : std_logic_vector(3 downto 0);

begin
  reg1: process (clk)
    begin
        if (rising_edge(clk)) then
           case load is
                when '1' => val <= par_in;
                when others => val <= val(0)
                                   & val(3 downto 1);
           end case;
           ser_out <= val(0);
        end if;
     end process;

end behav2;
```

Registered

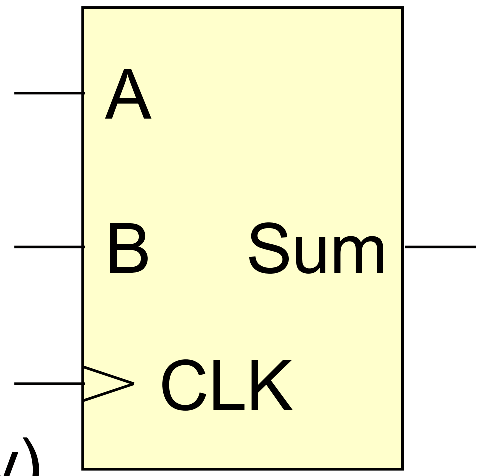
Extra register added to Output!

What happens when you do this?

- Signals `val` and `ser_out` are updated on the same rising edge of the clock (after the process is ended)
- So `ser_out` is updated with the contents of `val` from the previous clock cycle.
- Algorithm has an extra clock cycle of latency!

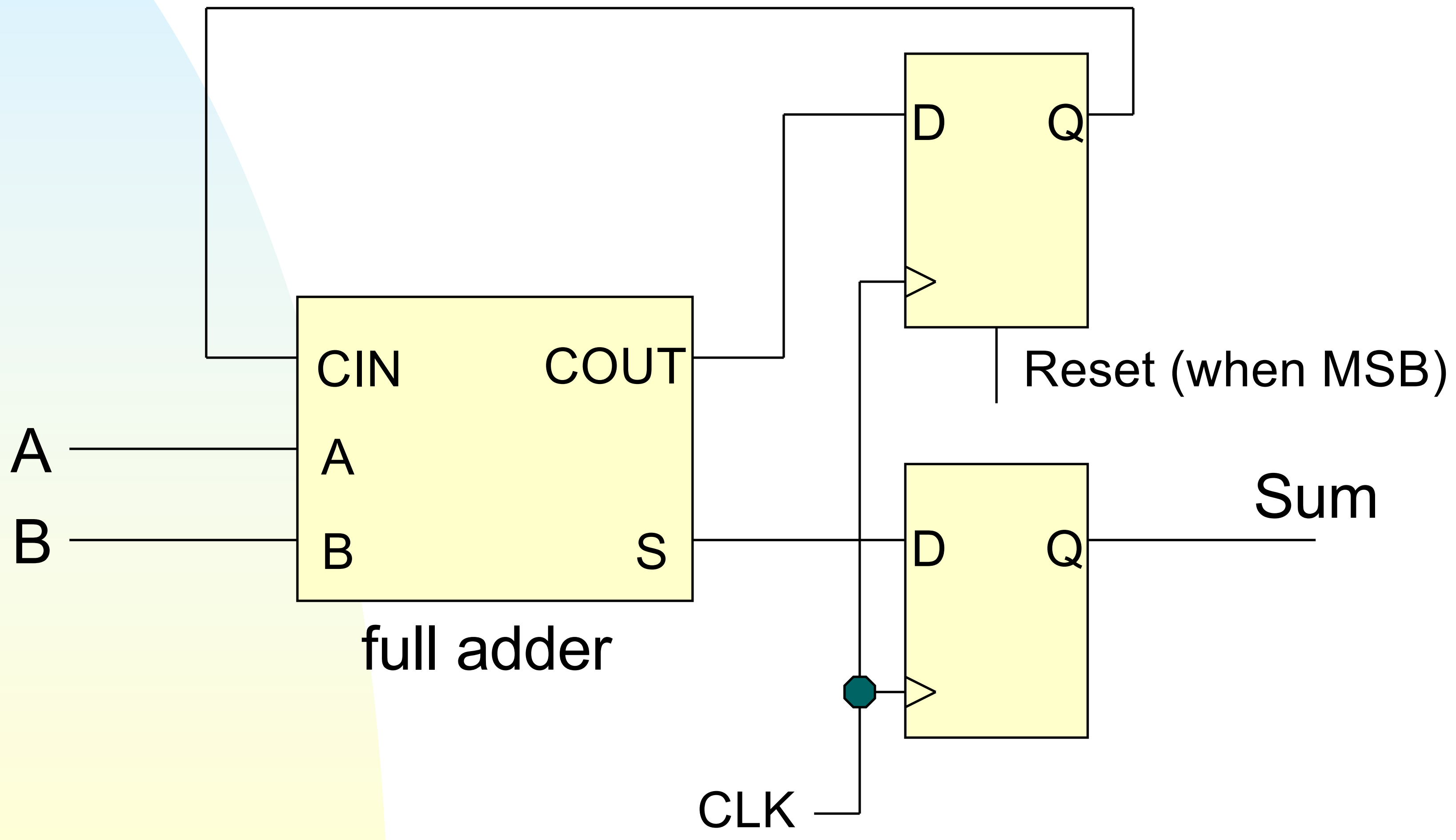
Keep track of timing!

Serial vs parallel arithmetic



- Example: n-bit Adder
  - Parallel
    - A and B are vectors with width *n*
    - Sum is vector with width *n* (plus carry)
    - Summation done in one clock cycle
  - Serial
    - A and B are <u>single-bit</u> inputs
    - Input data arrives in n cycles, LSB first
    - Output data leaves in n cycles, LSB first

Serial Adder



Why use serial arithmetic?

- Size
  - Parallel adder needs n full adders
  - Serial adder needs only 1 full adder
- Speed
  - Shorter carry chains mean less propagation delay
  - This allows serial adders to run at higher clock speeds (but longer latency)
- Serial adders can be useful for
  - Pipelined algorithms with many steps
  - Designs with serial input/output data

Lab 2 (next Monday)

- Produce a bit-serial adder
  - Design components
  - Test components individually
  - Assemble into full design
- Synthesize and implement
- Test on the developer board

Before the lab

- Read through Lab 2 writeup
- Think about components you will need, and how to implement them
- Try to write some sample code