

# Digital System Design

by

Dr. Lesley Shannon

Email: [Ishannon@ensc.sfu.ca](mailto:Ishannon@ensc.sfu.ca)

Course Website: <http://www.ensc.sfu.ca/~Ishannon/courses/ensc350>



*Simon Fraser University*

Slide Set: 2

Date: January 12, 2009

# Slide Set Overview

---

- Sequential Circuits in VHDL
  - Different constructs for describing sequential circuits in VHDL
    - For now, what are they and how are they used?
    - Our next lecture set will focus on how to ensure your HDL is **synthesizable**
  - Different Finite State Machine (FSM) types
  - Asynchronous vs Synchronous Sets and Resets

# Sequential Circuits in VHDL

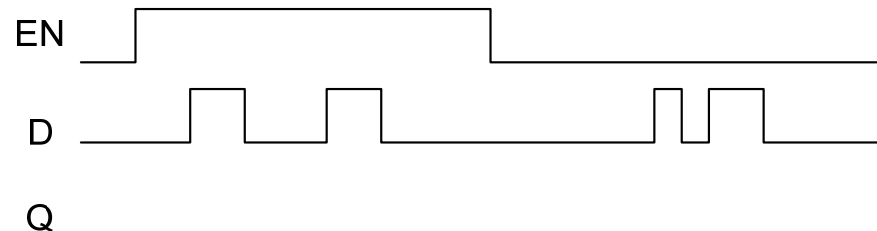
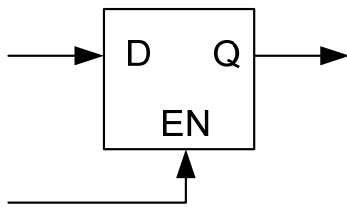
---

- Three ways to describe sequential circuits in VHDL:
  - Structural: Define a system in terms of its components
    - Can do this with the constructs we've already talked about
  - Dataflow: Need a few more constructs: BLOCK, Guarded
  - Behavioural: Need a new construct: PROCESS
- First talk about BLOCK and GUARDED, then talk about PROCESS

# Guarded and Block

---

- Often used to define small sequential structures, like latches
  - Consider a level sensitive latch:



- When  $EN = 1$ , changes on D appear on Q
- When  $EN = 0$ , Q does not change

# A Level-sensitive latch

---

```
entity LSL is
  port ( D, EN : in STD_LOGIC;
        Q : out STD_LOGIC);
end LSL;

architecture DATAFLOW of LSL is
begin
  B1: BLOCK (EN='1')
  begin
    Q <= GUARDED D;
  end BLOCK B1;
end DATAFLOW;
```

# Another Example

---

...

```
B1: BLOCK (CONTROL_SIGNAL='1')  
begin  
    X <= GUARDED A or B;  
    Y <= A and B;  
end BLOCK B1;
```

In this example, the X assignment only occurs when the CONTROL\_SIGNAL is 1 (i.e. the block condition is true). The Y assignment, however, occurs whenever A or B changes, regardless of CONTROL\_SIGNAL. This is because the Y assignment does not have the keyword GUARDED.

# Edge-Sensitive Behaviour

---

- Edge-sensitive flipflops change whenever the clock changes from 0 to 1 (rising edge) or from 1 to 0 (falling edge)
  - To describe edge-sensitive behaviour, we need the attribute 'STABLE

SIGNAL 'STABLE:            This a condition that is true whenever SIGNAL has not changed during the current simulation cycle.

Given, the condition:

$\text{CLK} = '1' \text{ and not } (\text{CLK} \text{ 'STABLE})$

This is true when?

# Example: Edge Sensitive Flipflop

---

**entity DFF is**

```
port (D, CLK : in STD_LOGIC;  
      Q : out STD_LOGIC);
```

```
end LSL;
```

**architecture DATAFLOW of DFF is**

```
begin
```

```
  B1: BLOCK (CLK='1' and not (CLK'STABLE))
```

```
  begin
```

```
    Q <= GUARDED D;
```

```
  end block B1;
```

```
end DATAFLOW;
```



# Some VHDL Terminology

---

- Transaction:
  - If an assignment is made to a signal at  $t$ , a transaction is said to occur at time  $t$ . Note that the assignment may or may not change the value of the signal (if you assign a '0' to a signal that is already '0', the signal doesn't change, but it is still a transaction).
- Event:
  - If a transaction does change the value of a signal, the transaction is also called an event
- Question: Are all events transactions?

# Other signal attributes

---

- **SIG'STABLE:**
  - True when an event does not occur on SIG during the current simulation cycle
- **SIG'QUIET**
  - True when a transaction does not occur on SIG during the current simulation cycle
- **SIG'STABLE(T):**
  - True if SIG has been stable for at least T time units
- **SIG'QUIET(T):**
  - True if SIG has been quiet for a least T time units

# Other signal attributes

---

- **SIG'ACTIVE:**
  - True if a transaction has occurred on SIG in the current simulation cycle
- **SIG'EVENT:**
  - True if an event has occurred on SIG in the current simulation cycle
- **SIG'LAST\_ACTIVE:**
  - The amount of time since the last transaction on SIG
- **SIG'LAST\_EVENT:**
  - The amount of time since the last event on SIG

# Other signal attributes

---

- **SIG'DELAYED(T):**
  - A signal identical to SIG but delayed by time T

SIG

SIG'DELAYED(T)

# Problems with using GUARDED and BLOCK

---

- Problems with using GUARDED and BLOCK:
  - Difficult to describe complex units
  - Not supported by some synthesis tools (it is supported in Quartus II)
- So we need a new method to describe sequential behaviour:
  - In fact, the method we will discuss is general enough to describe any sort of behaviour!
- The “Process”:
  - But first a summary of what sequential stuff we’ve covered so far ...

# Sequential circuits so far...

---

- We talked about flipflops (the fundamental building blocks of sequential systems). Remember:
  - Level sensitive vs Edge sensitive
  - Rising edge vs Falling edge
- We've talked about setup and hold times
  - This will matter when we talk about timing analysis
- Also talked about GUARDED and BLOCK in VHDL
  - Rarely used
  - Normally people use a "Process" , the focus of an upcoming slide set

---

# Processes

---

# The Process

---

- The process construct allows us to describe the function or behaviour of a circuit or subcircuit without describing the actual hardware
  - We have already seen some sorts of behavioural descriptions:
    - $X \leq (B \text{ nand } A) \text{ or } C;$
  - This describes the behaviour of the circuit driving  $X$ , but not necessarily the actual hardware implementation
    - e.g. Nand and or, a few nands, or a few nors, or ...
  - But we can describe circuits at a much higher level than this



# How can we describe behaviour?

---

- Writing an “english” description would be nice
  - But that would leave too much room for ambiguity
- We could create a new language that would ensure no ambiguity
  - But that would be a bit of a bother
- A better way: describe the behaviour using something that **looks** like software
  - Writing software is easy and ...
  - There’s no room for ambiguity

# How can we describe behaviour?

---

- Note:
  - This “software” or “behavioural” description only describes the behaviour of the circuit, not how the circuit is constructed

# Fundamental difference between HW & SW

---

- Hardware: A number of hardware elements that “operate” in parallel
- Software: A set of statements that are executed sequentially
- This distinction will be important soon.

# The Process

---

**process (sensitivity\_list)**

**begin**

**software-like statements that are executed sequentially**

**end process;**

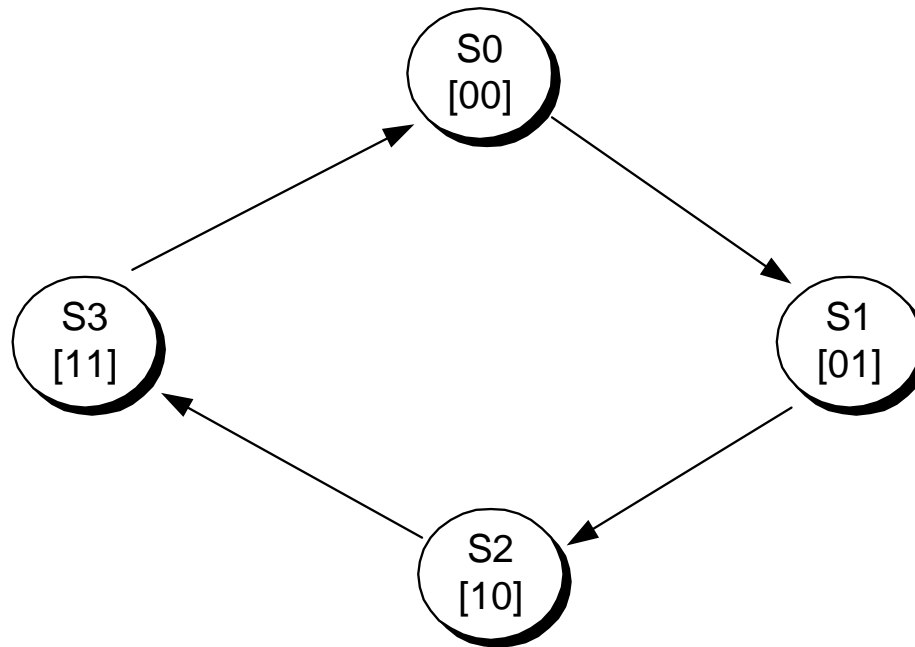
Each process describes the function of one piece of hardware.

-If the statements in the process are executed sequentially, the overall result is the same as if the result was produced by hardware

Note: The algorithmic description in the process may have no bearing on the actual hardware implementation. They only have the same behaviour.

# Dataflow method of describing an FSM

---



architecture **DATAFLOW** of FSM is

**begin**

**q0 <= some function of q0 and q1;**

**q1 <= some function of q0 and q1;**

**end DATAFLOW**

Problem: a pain if there are many state variables and/or inputs

# An easier way to describe an FSM

---

architecture BEHAVIOURAL of FSM is

begin

  process (clk)

    variable PRESENT\_STATE : bit\_vector(1 downto 0) := "00";

    begin

      if ((clk = '1') and (clk'event)) then

        case PRESENT\_STATE is

          when "00" => PRESENT\_STATE := "01";

          when "01" => PRESENT\_STATE := "10";

          when "10" => PRESENT\_STATE := "11";

          when "11" => PRESENT\_STATE := "00";

        end case;

        Z <= PRESENT\_STATE; -- update output Z

      end if;

    end process;

# More details on processes

---

- `process(clk)`
  - Define the start of a process that is executed whenever `clk` changes
    - Therefore, do we need the `clk'event`?
- `variable PRESENT_STATE ...`
  - Defines a variable `PRESENT_STATE` of type `BIT_VECTOR(2 bits)` and initializes it to “00”. Note: In this case, `PRESENT_STATE` has a clear meaning with respect to the final circuit. In general, however, variables are used only as an aid to describe the behaviour algorithmically

# More details on processes

---

- If ((clk = '1') and (clk'event)) then ...
  - The process will only execute when clk changes (see previous slide). But, when we want the state variables to change only when clk changes from 0 to 1 (when clk changes from 1 to 0, we want to do nothing)
- Case PRESENT\_STATE is ...
  - This is like a case statement in any other HLL. The value of the variable PRESENT\_STATE is set depending on the old value
- Z <= PRESENT\_STATE;
  - Z (an output of the architecture) is updated to be the current value of the variable PRESENT\_STATE



# More Details on VHDL

---

- Enumerated types make code more readable

```
type state_types is (StateLive, StateWait, StateSample,  
                    StateDisplay);
```

```
variable CURRENT_STATE: state_types;
```

- Here we have defined a variable called `CURRENT_STATE` that can take on one of four values: `StateLive`, `StateWait`, `StateSimple`, `StateDisplay`

# More Details on VHDL

---

- Here we have defined a variable called `CURRENT_STATE` that can take on one of four values: `StateLive`, `StateWait`, `StateSimple`, `StateDisplay`
- We can then use this variable as before, in a case statement

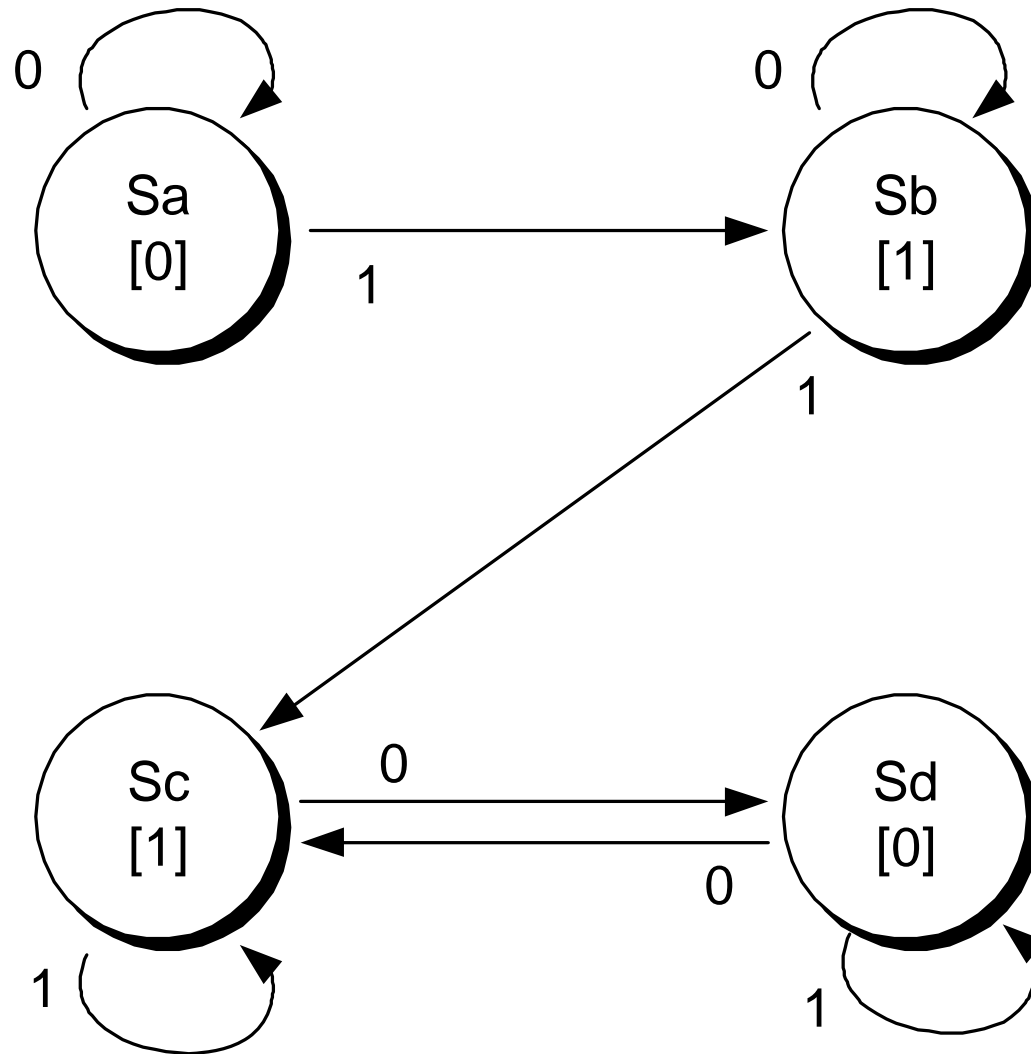
```
case PRESENT_STATE is  
  when StateLive => ...  
  ....  
  when StateWait => ...  
etc...
```

# Recall a Moore Machine

---

# An FSM example: A Moore Machine

---



# Moore FSM sample code

---

```
entity FSM2 is
  port (      INPUT, CLK : in bit;
          Z : out bit);
end FSM2;

architecture BEHAVIOURAL of FSM2 is
begin
  process(CLK)
  type state_types is (Sa, Sb, Sc, Sd);
  variable PRESENT_STATE : state_types := Sa;
  variable NEXT_STATE : state_types;
```

.... Continued on next slide

# Moore FSM sample code

---

```
begin
  if (clk = '1') then
    case PRESENT_STATE is
      when Sa => if (INPUT = '0') then
        NEXT_STATE := Sa;
      else
        NEXT_STATE := Sb;
      end if;
      when Sb => if (INPUT = '0') then
        NEXT_STATE := Sb;
      else
        NEXT_STATE := Sc;
      end if;
      ... same for states Sc and Sd
    end case;
  end if;
end;
```

# Moore FSM sample code

---

```
--architecture
  --begin
    --if (clk = '1') then

      PRESENT_STATE := NEXT_STATE;

      case PRESENT_STATE is
        when Sa => Z<='0';
        when Sb => Z<='1';
        when Sc => Z<='1';
        when Sd => Z<='0';
      end case;
    end if;
  end process;
end BEHAVIOURAL;
```

## Another way to describe a Moore FSM

---

architecture behavioural of FSM is  
begin

--next state combinational logic

process (Sel, A, B, C, D)

begin

if (Sel="00") then

next\_state <= A;

elsif (Sel="01") then

next\_state <= B;

elsif (Sel="10") then

next\_state <= C;

else

next state <= D;

end if;

end process;

--present state register logic

process(clk)

begin

if (clk = '1' and rst = '1')

pres\_state <= A;

else

pres\_state <= next\_state;

end if;

end process;

end behavioural;



# Recall a Mealy Machine

---

# One Hot Encoding

---

# Gray Encoding

---

## Processes can also be used to define combinational logic

---

```
entity simple_mux is
  port (
    Sel : in bit_vector(1 downto 0);
    A, B, C, D : in bit;
    Y : out bit);
end simple_mux;
```

# Processes can also be used to define combinational logic

---

architecture behavioural of simple\_mux is

begin

  process (Sel, A, B, C, D)

  begin

    if (Sel="00") then

      Y <= A;

    elsif (Sel="01") then

      Y <= B;

    elsif (Sel="10") then

      Y <= C;

    else

      Y <= D;

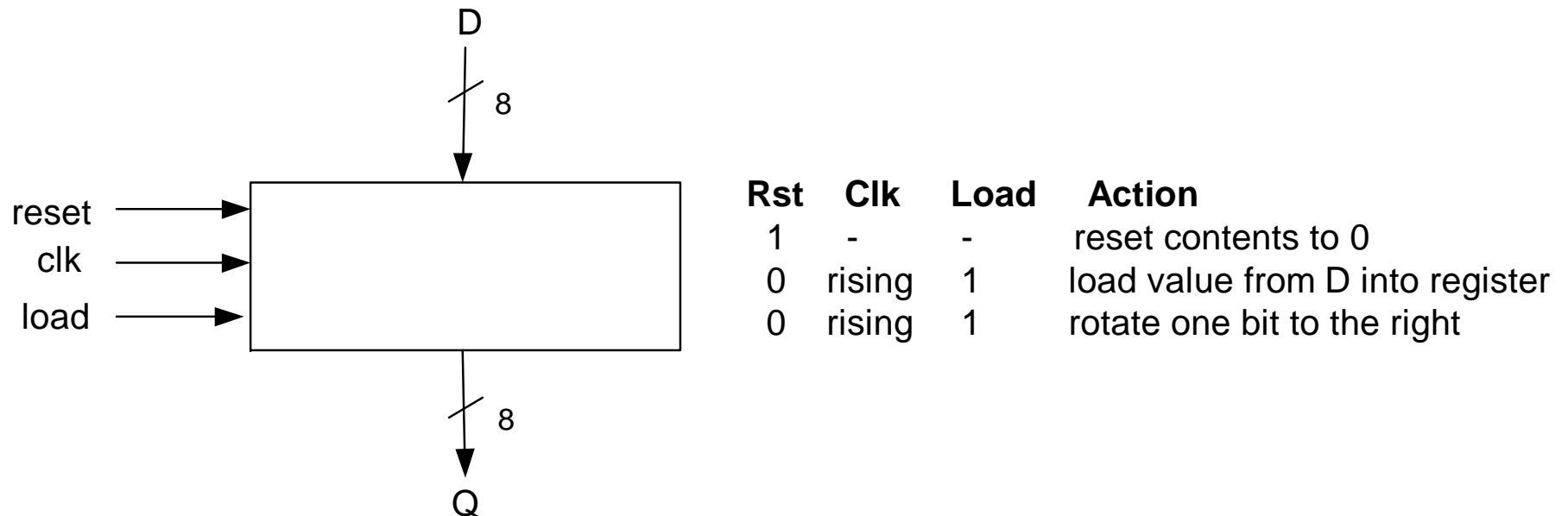
    end if;

  end process;

end behavioural;

# Can use a process to define something more complex

---



- Operation is as follows:
  - If Reset is 1, the value in the register is set to 0, regardless of clock (asynchronously)
  - Otherwise, on each rising edge of the clock if load is 1, load in value from D into register otherwise, rotate value in register one bit to the right

# Implementing the parallel-load resetable rotator

---

1. We could draw the circuit containing a multiplexor and eight flip-flops. This could then be described structurally or using dataflow techniques.
2. We could use a single process to define the behaviour of the entire unit.

# Implementing the parallel-load resetable rotator

---

**entity rotate is**

```
port(      clk, reset, load : in std_logic;  
          d : in std_logic_vector(7 downto 0);  
          q : out std_logic_vector(7 downto 0));
```

**end rotate;**



# Implementing the parallel-load resetable rotator

---

architecture behavioural of rotate is

begin

  process (reset, clk)

  variable int\_value : std\_logic\_vector (7 downto 0);

  begin

    if (reset = '1') then

      int\_value := "00000000";

    elsif (clk='1' and clk'event) then

      if (load='1') then

        int\_value := d;

      else

        int\_value := int\_value(0) & int\_value(7 downto 1);

      end if; --else int\_value := int\_value;

    end if;

    q <= int\_value;

  end process;

end behavioural;

# Concurrent Statements

---

- Note: any concurrent statements not inside a process are considered a process themselves:

**entity COMB\_BLOCK is**

**port (A, B,C,D : in BIT;**

**X,Y,Z : out BIT);**

**end COMB\_BLOCK;**

**architecture MY\_DEFN of COMB\_BLOCK is**

**begin**

**X <= A and B and C;**

**Y <= C and not D;**

**Z <= A xor B xor D;**

**end MY\_DEFN;**

**Three processes**

## Is your whole design a single process?

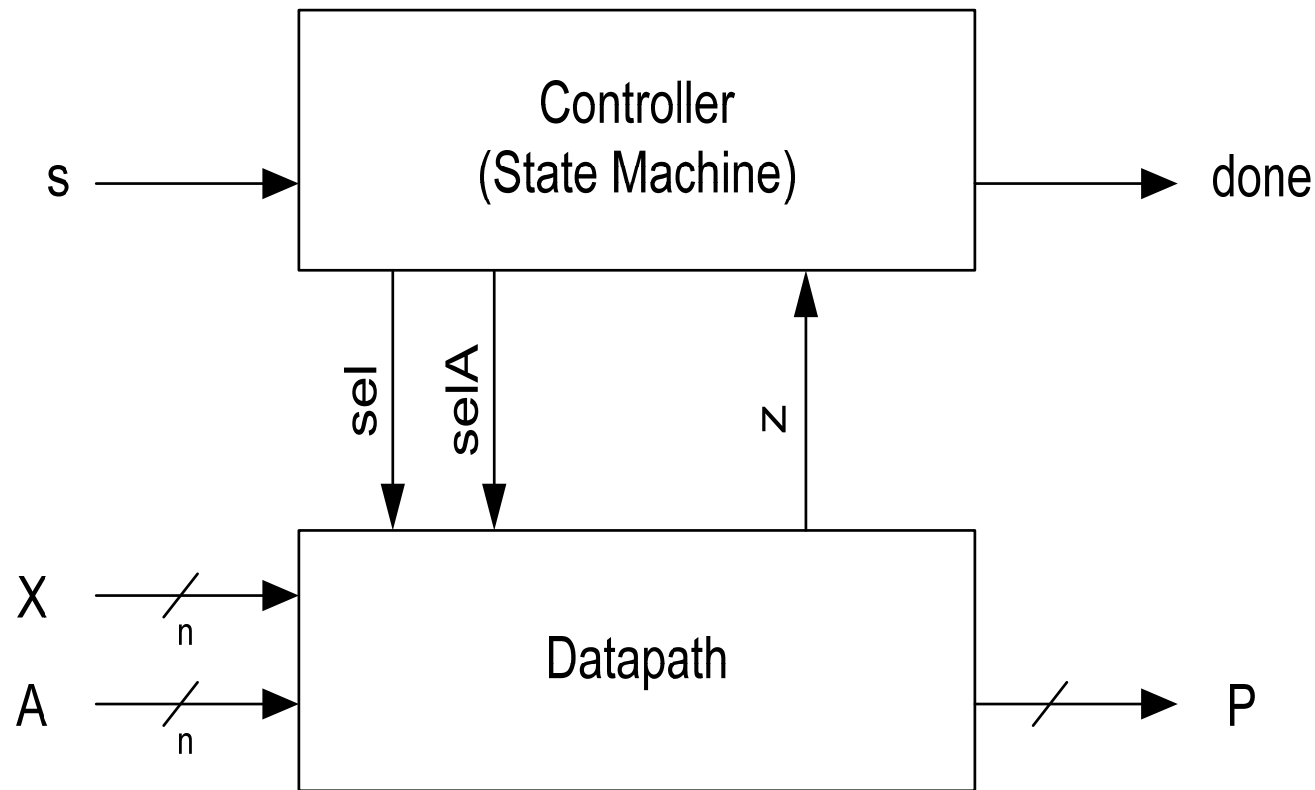
---

- A design is usually a collection of processes:
  - Each process is a single combinational or sequential block
    - More precise definition in the next slide set
- Each process is “compiled” independently (in most tools)
- One entity can have several processes.

# Is your whole design a single process?

---

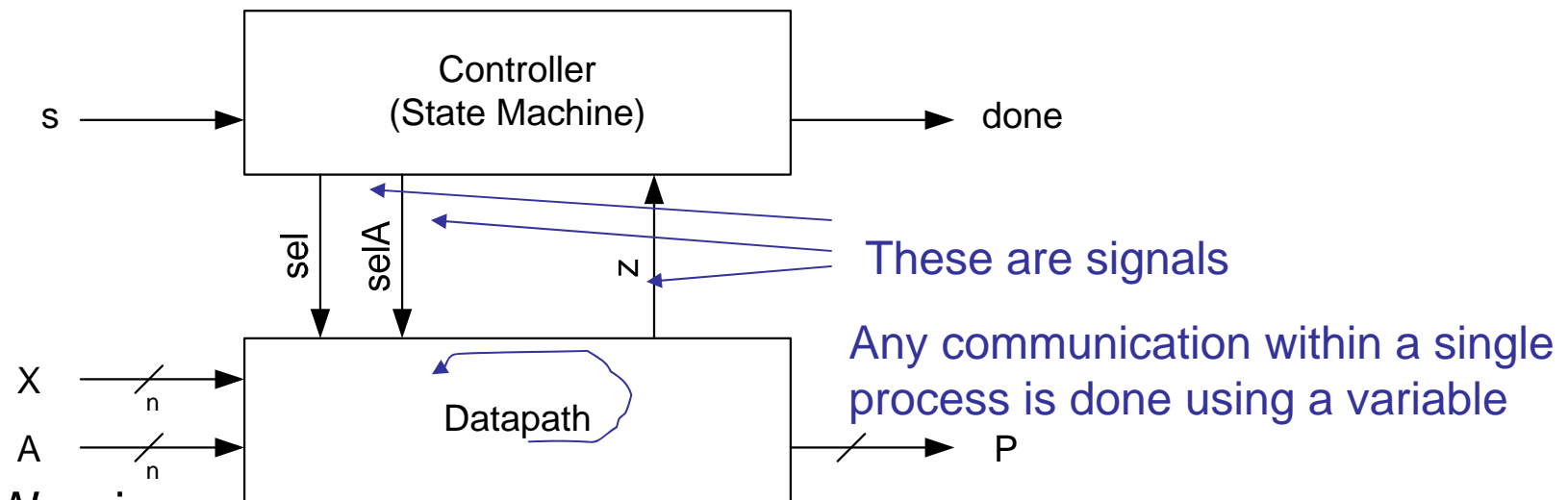
- Normal design process: divide your design into small blocks, and write a process for each one.



We will see this design later in the course. Each block is implemented as a “process”

# Variable versus Signal

- Signals: Used to transmit data between processes
- Variables: Used within a process to help describe behaviour



- Warning:
  - There are things called “shared variables” but don’t use them.
    - If you want to communicate between processes, use a signal
    - Makes the intent a lot clearer, and makes your design more portable

---

NOTE: All inputs and outputs of a process are *signals*

---

## An example from earlier

---

The variable PRESENT\_STATE could not be accessed outside this process. But Z could be accessed from outside the process.

**architecture BEHAVIOURAL of FSM is**

**begin**

**process (clk)**

**variable PRESENT\_STATE : bit\_vector(1 downto 0) := "00";**

**begin**

**if (clk = '1') then**

**case PRESENT\_STATE is**

**when "00" => PRESENT\_STATE := "01";**

**when "01" => PRESENT\_STATE := "10";**

**when "10" => PRESENT\_STATE := "11";**

**when "11" => PRESENT\_STATE := "00";**

**end case;**

**Z <= PRESENT\_STATE; -- update output Z**

**end if;**

**end process;**

---

Now let's talk about how variables and signals are “updated”:

Note: in this discussion, we will pretend that we are “executing the VHDL” code, but of course, what really happens is that the code is translated to hardware with the same behaviour.



# Variables are updated immediately

---

architecture BEHAVIOURAL of FSM is

begin

process (clk)

variable PRESENT\_STATE : bit\_vector(1 downto 0) := "00";

begin

if (clk = '1') then

case PRESENT\_STATE is

when "00" => PRESENT\_STATE := "01";

when "01" => PRESENT\_STATE := "10";

when "10" => PRESENT\_STATE := "11";

when "11" => PRESENT\_STATE := "00";

end case;

Z <= PRESENT\_STATE; -- update output Z

end if;

end process;

As soon as one of these is "executed", PRESENT\_STATE is updated

So this uses the updated value of PRESENT\_STATE

# Signals are updated at the end of the process

---

architecture BEHAVIOURAL of FSM is

begin

process (clk)

variable PRESENT\_STATE : bit\_vector(1 downto 0) := "00";

begin

if (clk = '1') then

case PRESENT\_STATE is

when "00" => PRESENT\_STATE := "01";

when "01" => PRESENT\_STATE := "10";

when "10" => PRESENT\_STATE := "11";

when "11" => PRESENT\_STATE := "00";

end case;

Z <= PRESENT\_STATE; -- update output Z

end if;

end process;

Z is updated once we finish the process (in this case, it is the last statement in the process)

# Updating Signals

---

```
signal x, y, z, w: std_logic;
```

```
process(x,y)
```

```
begin
```

```
    z <= x and y;
```

```
    w <= x or y;
```

```
end process;
```

Signals z and w get updated at the same time, at the end of the process

# Updating Signals

---

```
signal x, y, z, w: std_logic;
```

```
process(x,y)
```

```
begin
```

```
    z <= x and y;
```

```
    w <= z and y; -- w <= z_old and y
```

```
end process;
```

This is strange: the assignment to w uses the old value of z, not the new value! That is because z is not updated until the end of the process.

## If z were a variable

---

```
signal x, y, w: std_logic;
```

```
process(x,y)
```

```
variable z: std_logic;
```

```
begin
```

```
    z := x and y;
```

```
    w <= z and y;
```

```
end process;
```

In this case, w would use the “new” value of z

## A common error

What happens in our previous example if we used signal instead of variable for PRESENT\_STATE?

signal variable PRESENT\_STATE : bit\_vector(1 downto 0);

architecture BEHAVIOURAL of FSM is

begin

process (clk)

begin

if (clk = '1') then

case PRESENT\_STATE is

when "00" => PRESENT\_STATE <= "01";

when "01" => PRESENT\_STATE <= "10";

when "10" => PRESENT\_STATE <= "11";

when "11" => PRESENT\_STATE <= "00";


end case;

Z <= PRESENT\_STATE; -- update output Z

end if;

end process;

Uses old value of  
PRESENT\_STATE! Probably  
not what was intended



---

Moral: Don't use a signal to communicate *within* a process!!

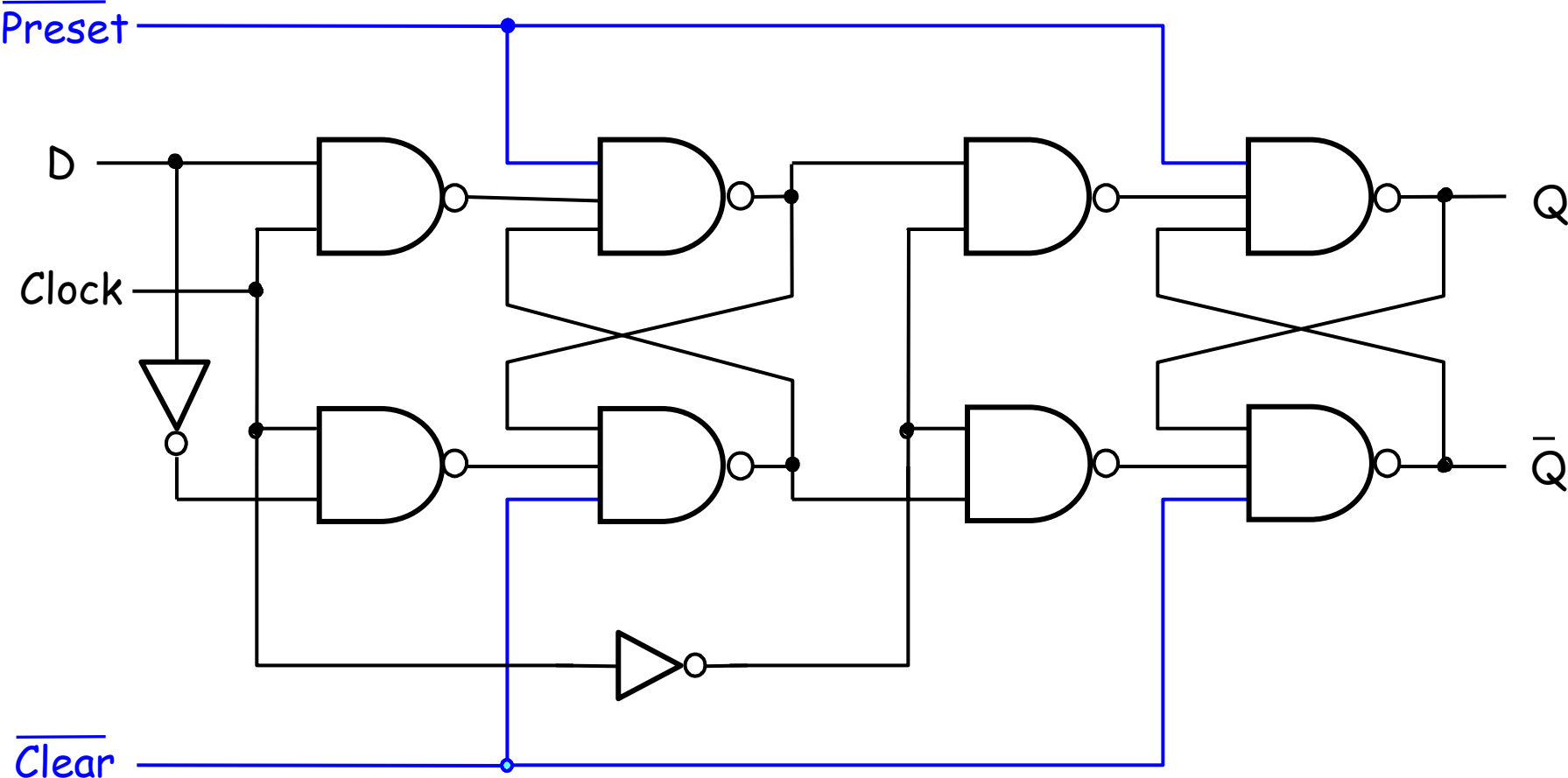
# Asynchronous vs Synchronous Sets and Resets

---

- A flip-flop can have either a synchronous or asynchronous reset
  - Technically, it could have both, but this is never done
- **Asynchronous Reset:** When the reset signal is high, the flip-flop is reset (forced to '0') immediately, regardless of the clock.
- **Synchronous Reset:** On a rising clock edge, if the reset signal is high. The flip-flop is reset (forced to '0').
- The difference: with a synchronous reset, the flip-flop is not reset until the next rising clock edge. With an asynchronous reset, it is reset immediately.



# Circuit with an Asynchronous Reset (and preset)



# Asynchronous Reset in VHDL

---

architecture behavioural of DFF is

**begin**

**process (clk, reset)**

**begin**

**if (reset = '1') then**

**Q <= '0';**

**elsif (clk='1' and clk'event) then**

**Q <= D;**

**else**

**Q <= Q; -- implied**

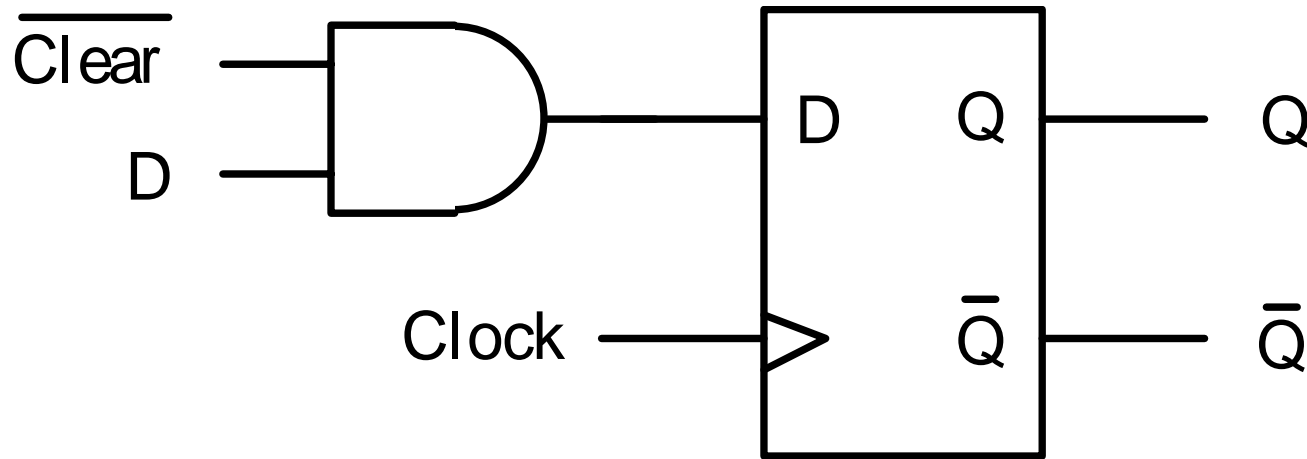
**end if;**

**end process;**

**end behavioural;**

## Circuit with a Synchronous Reset

---



# Synchronous Reset in VHDL

---

architecture behavioural of DFF is

begin

  process (clk)

  begin

    if (clk='1' and clk'event) then

      if (reset = '1') then

        Q <= '0';

      else

        Q <= D;

      end if;

    end if;

  end process;

end behavioural;

## Recall the sensitivity list

---

```
process(A, B, C)  
  begin
```

```
    ....
```

```
end process;
```

- means that the process is executed whenever A, B, or C changes

- But what if there is no sensitivity list?

```
process  
begin
```

```
    ....
```

```
end process;
```

- As soon as process finishes, it re-executes from the top
  - like an infinite loop in software!
  - surely this doesn't correspond to any real piece of hardware

# The WAIT statement

---

- Consider a process with signal A in its sensitivity list:

```
process(A)  
begin
```

```
....
```

```
end process;
```

- An equivalent definition:

```
process  
begin
```

```
    wait until A'event;
```

```
....
```

```
end process;
```

- The WAIT statement waits until there is an event on A



# A common use of the WAIT statement for a synchronous system

---

```
process
begin
    ....
    wait until (clk'event and clk='1');
end process;
```

- Can do something like:

```
process
begin
    .... do something for cycle 1
    wait until (clk'event and clk='1');
    .... do something for cycle 2
    wait until (clk'event and clk='1');
    .... do something for cycle 3
    wait until (clk'event and clk='1');
    etc...
```

---

**Note:** A process may have either a WAIT statement or a sensitivity list, but never both!

**Also Note:** Most patterns with WAIT are not synthesizable. Need to be very careful. We'll talk more about synthesizable code next.



# The highlights

---

We talked about “Process”: the most important concept in VHDL.

Each “piece of hardware” corresponds to a process

- Use signals to communicate values between processes
- Within a process, you can use variables to store temporary results

Variables only have lifetime within a process. Signals are only used to communicate between processes.

Most important thing: When you are describing hardware in VHDL, you are only describing the behaviour. The actual circuit will be synthesized (by the tools, eg. Quartus II) to gates. The FPGA then implements the gates. The FPGA does **NOT** execute the VHDL code directly!!!!

# Questions

---

- Are all transactions events?
- What are three ways we can describe sequential circuits in HDL?

# Questions

---

- What are the problems with using GUARDED and Blocked:
  - 1.
  - 2.

# Questions

---

- What is the architecture (circuit) described below?

**B1: BLOCK (SIG\_X='1')**

**begin**

**OUTA <= GUARDED (A xor B xor C);**

**OUTB <=(A and B) or (A and C) or (B and C);**

**OUTC <=GUARDED (OUTA xor D xor E);**

**OUTD <=(E and OUTA) or (D and OUTA) or  
(D and E)**

**end BLOCK B1;**

# Questions

---

- What does SIG'EVENT mean?

# Questions

---

- When are variables updated in a process?
  
  
  
  
  
  
  
  
  
  
- When are signals updated in a process?

# Questions

---

- You should not use variables to communicate between processes. Why?
- What does it mean when there are no signals in the sensitivity list?

# Questions

---

- When using a WAIT statement, what do you need to worry about?
  
  
  
  
  
  
  
  
  
  
- What's the difference between Moore and Mealy FSMs?



# Questions

---

- What type of construct is a process (i.e. dataflow?, structural?)?
  
  
  
  
  
  
  
  
  
  
- Can processes be used to implement combinational logic?

# Questions

---

- What is one-hot encoding?
  
  
  
  
  
  
  
  
  
  
- What is the difference between a synchronous and asynchronous reset?