EECE 353: Digital Systems Design
Lecture 5: Making your VHDL synthesizable

Cristian Grecu

grecuc@ece.ubc.ca

Course web site: http://www.ece.ubc.ca/~elec353

Introduction to lecture 5

This is likely the most important slide set of all.

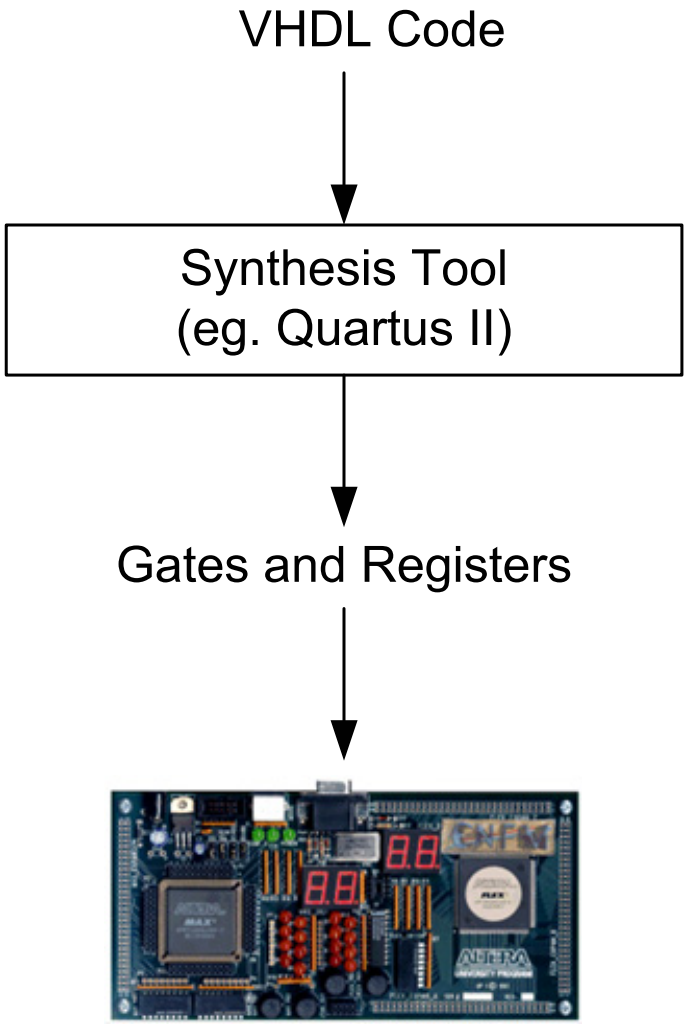
If you learn VHDL from a book, write your code using the constructs they describe, and try to compile it to a hardware (FPGA, ASIC), it probably won't work!

Why?

I'll tell you in this slide set, and also talk about how to make sure it does work.

Why is this the most important slide set of all?

Because, it will save you tons of debugging time in the lab if you understand it.

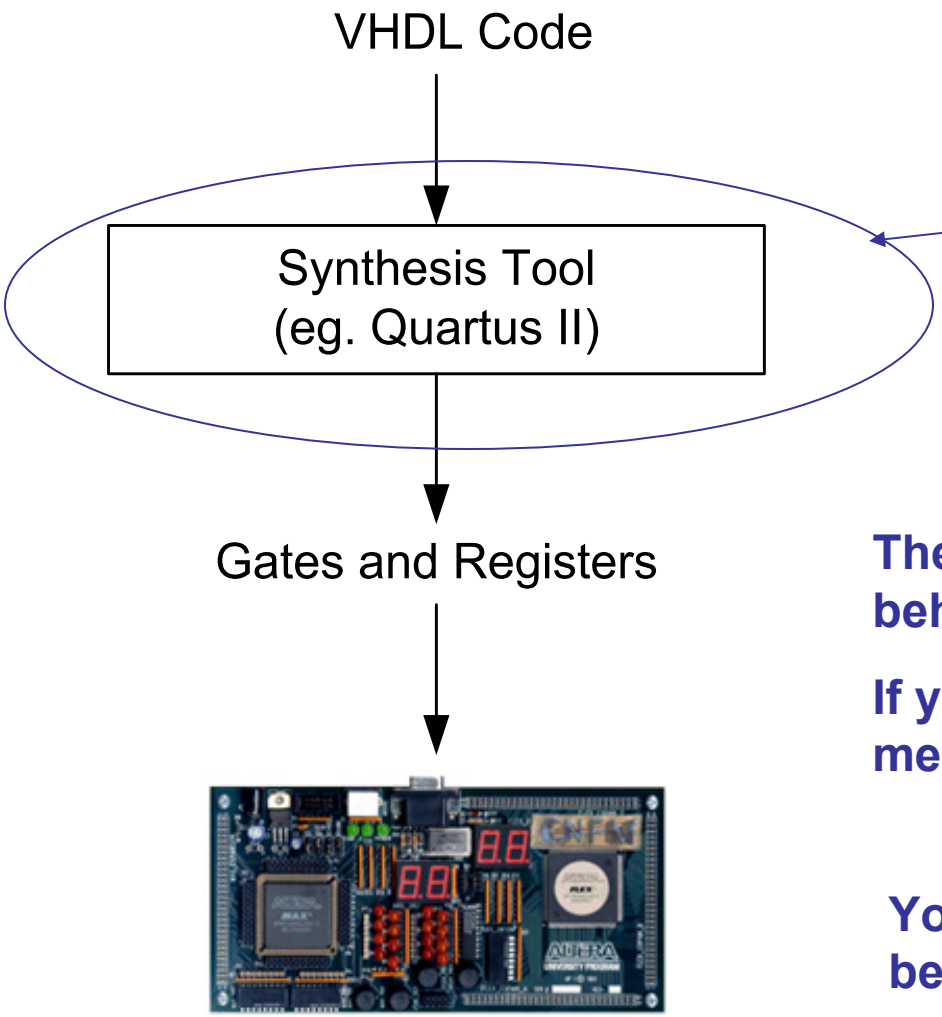


The synthesis tool "compiles" the VHDL into gates. Makes hardware out of VHDL code

The gates are implemented on a chip (in our case, on an FPGA)

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!!!!

Q: How many times have your heard this in class?
A1: What?
A2: Few times.
A3: Enough times.
A4: Too many.



What do you think happens if the synthesis tool can not make hardware for your VHDL?

The gates will not implement the behaviour you specified…

If you are lucky, you get an error message

Your circuit won't implement the behaviour you specified

“Synthesizable” VHDL

Not all VHDL code can be synthesized by current tools.

- This isn’t limited to our tools (Quartus)

Synthesizable VHDL is a subset of VHDL that can be synthesized by current tools.

If you write VHDL that is not synthesizable:

- Tools will not be able to create hardware
- Sometimes it will try, but end up with something that is “not quite right”
- Sometimes you get an error message, sometimes you don’t!

Moral: if you are going to synthesize, always write Synthesizable VHDL!

But what sort of VHDL is Synthesizable?

In general, it depends a bit on the tools.

RTL-level Synthesis Tools: can handle a fairly small subset

Behavioural Synthesis Tools: larger subset

In the next few slides, I am going to show you the minimum set that is synthesizable by all tools. If you restrict your VHDL to this small set, your code will be synthesizable by all tools.

To make sure your VHDL is synthesizable, every process must be one of three types:

1. Type 1: Purely Combinational: all outputs are a function only of the current inputs (not on the previous inputs)

```
process (SEL, A, B)
begin
   if (SEL = '0') then
      Y <= A;
   else
      Y <= B;
   end if;
end process;
```

For purely combinational processes:

Rule 1: Every input (that can affect the output(s)) must be in the sensitivity list.

Rule 2: Every output must be assigned a value for every possible combination of the inputs

This would *not* be synthesizable (violates Rule 1):

```
process (A, B)
begin
   if (SEL = '0') then
      Y <= A;
   else
      Y <= B;
   end if;
end process;
```

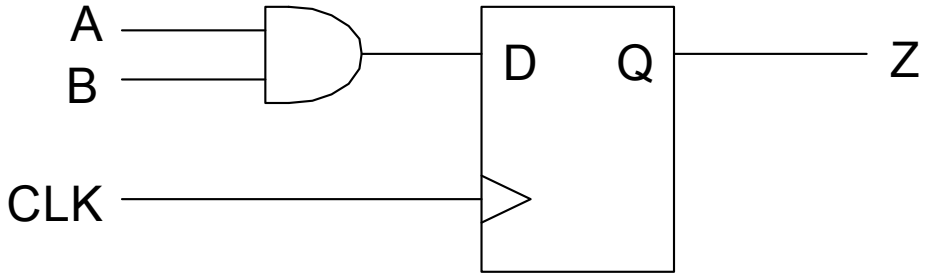
This would *not* be synthesizable (violates Rule 2):

```
process (SEL, A, B)
begin
      if (SEL = '0') then
            Y <= A;
      end if;
end process;
```

Type 2: Purely Sequential: Each output changes *only* on the rising *or* falling edge of a single clock

```
process (CLK)
begin
   if (CLK'event and CLK='1') then
      Z <= A and B;
   end if;
end process;
```

optional



Aside: std_logic type

- 'U': uninitialized. This signal hasn't been set yet.
- 'X': unknown. Impossible to determine this value/result.
- '0': logic 0
- '1': logic 1
- 'Z': High Impedance
- 'W': Weak signal, can't tell if it should be 0 or 1.
- 'L': Weak signal that should probably go to 0
- 'H': Weak signal that should probably go to 1
- '-': Don't care.

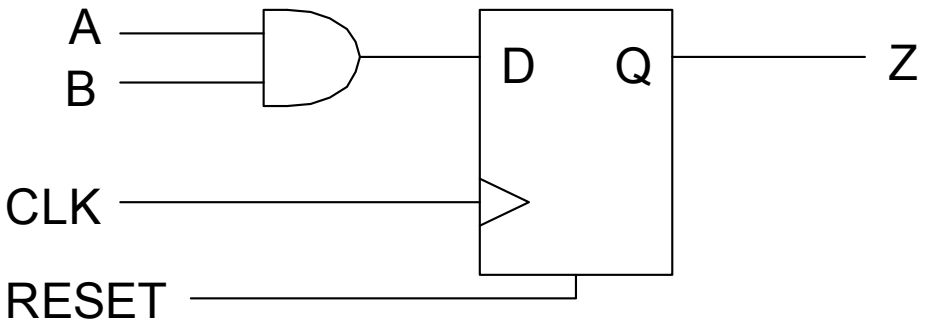
For a purely sequential process:

Rule 1: Only the clock should be in the sensitivity list

Rule 2: Only signals that change on the same edge of the same clock should be part of the same process

Type 3: Purely Synchronous with asynchronous set or reset

```
process (CLK, RESET)
begin
   if (RESET = '1') then
       Z <= '0';
   elsif (CLK'event and CLK='1') then
       Z <= A and B;
   end if;
end process;
```



For a purely sequential with asynchronous set/reset:

Rule 1: Sensitivity list includes clock and set/reset signal

Rule 2: Need the *clk'event* clause (why?)

Rule 3: Inside the first part of the *if* statement, must assign either 0 or 1 (not anything else).

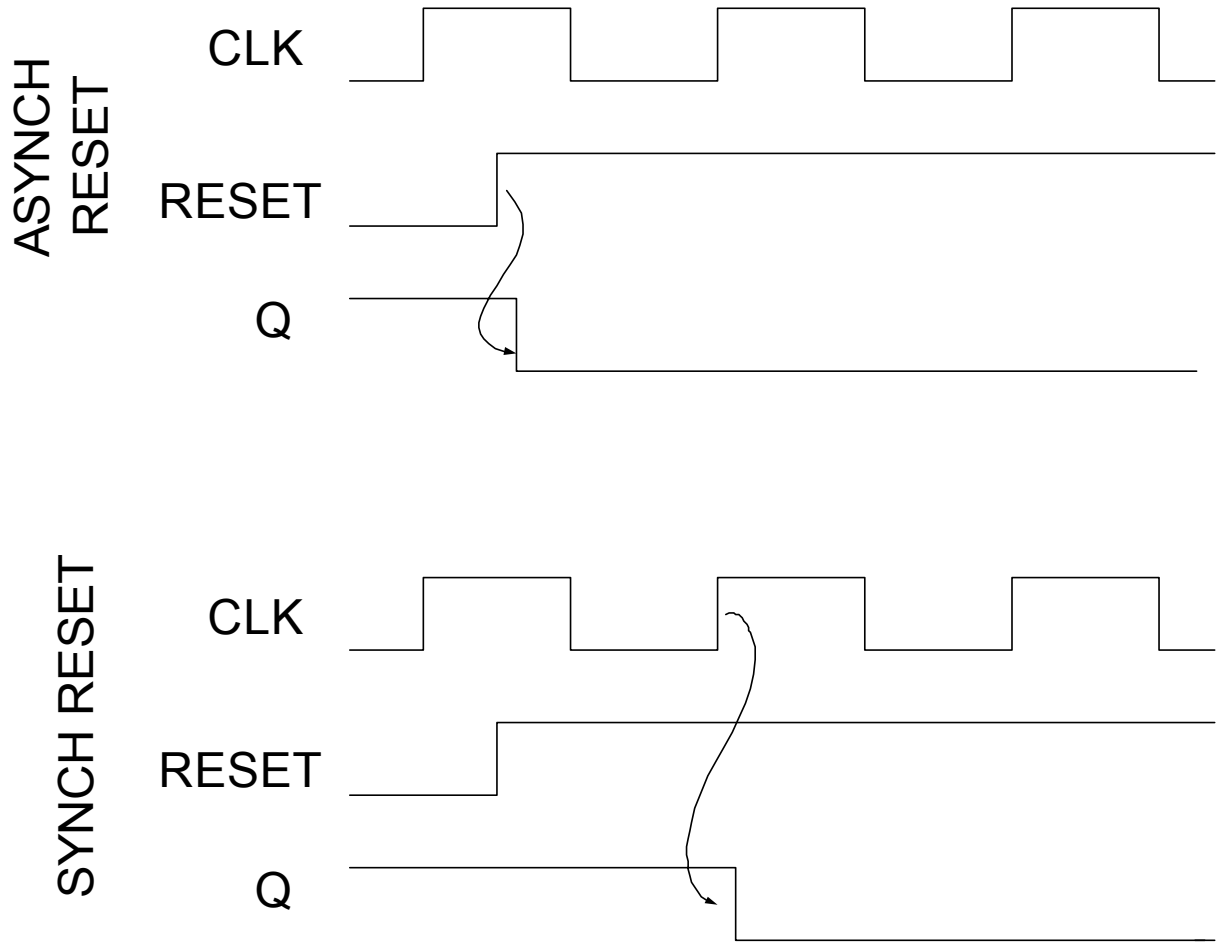
Final Rule (the most important rule of all):

If you want to synthesize your circuit, *every* process must fall *exactly* into one of these categories. Every process. Every single one. No exceptions.

If one of your processes doesn't, you need to break it up into blocks, where each block does fit into one of these categories.

(note: I am being a little bit conservative here… some synthesizers will handle a few patterns not described here. But don't count on it).

Consider a flip-flop with a synchronous reset.



Consider a flip-flop with a synchronous reset.

Question: Can we describe this sort of flip-flop using any of the three types of processes described earlier?

If so, which type?

Remember these are the types:

Type 1: Purely Combinational
Type 2: Purely Synchronous
Type 3: Purely Synchronous with async set/reset

```
process (CLK)
begin
   if (CLK'event and CLK='1') then
       if (RESET = '1') then
           Q <= '0';
       else
           Q <= D;
       end if;
end process;
```

Question 2: What about a Moore State machine:

Remember, a Moore machine has outputs that depend only on the current state.

Can we define this in one process? If so, what type?

```
process(clk)
variable PRESENT_STATE : bit_vector(1 downto 0) := “00”;
begin
        if (clk’event and clk=‘1’) then
          case PRESENT_STATE is
                  when “00” => if (in_sig = ‘0’) then
                                        PRESENT_STATE := “01”;
                                  else
                                        PRESENT_STATE := “00”;
                                  end if;
                  when “01” => PRESENT_STATE := “10”;
                  when “10” => PRESENT_STATE := “11”;
                  when “11” => PRESENT_STATE := “00”;
           end case;
           Z <= PRESENT_STATE(0) and not PRESENT_STATE(1);
        end if;
    end process;
```

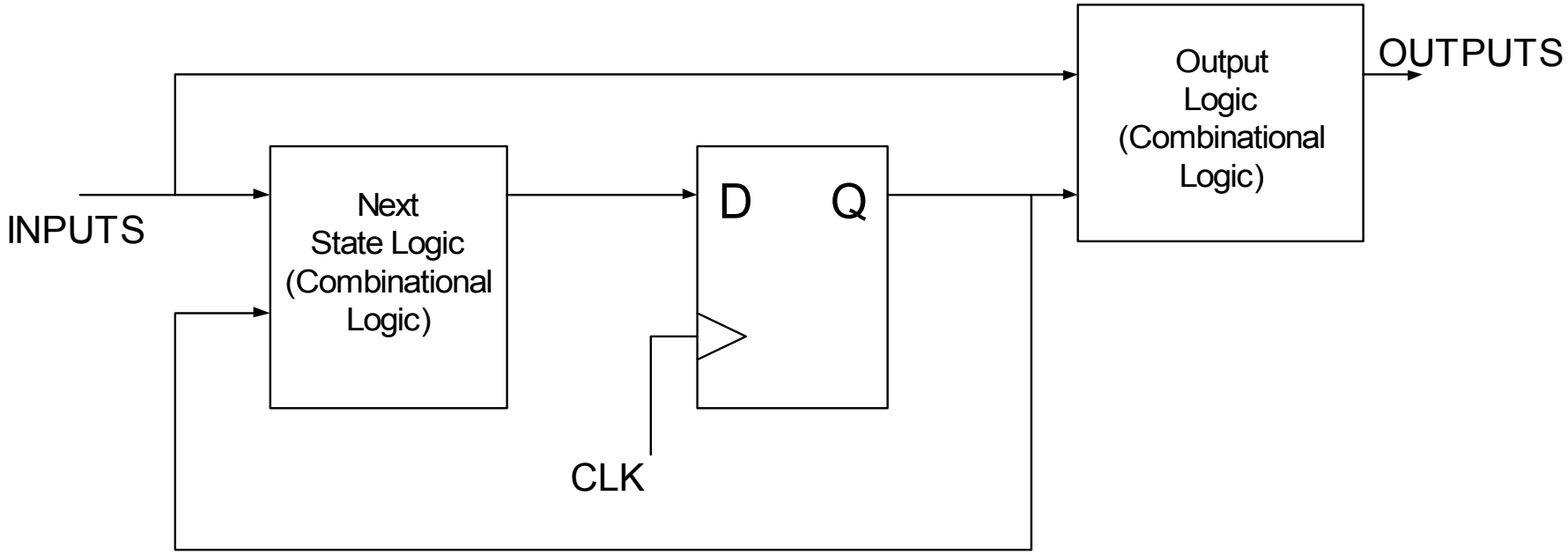

Anything wrong or missing?

Question 3: What about a Mealy State Machine?

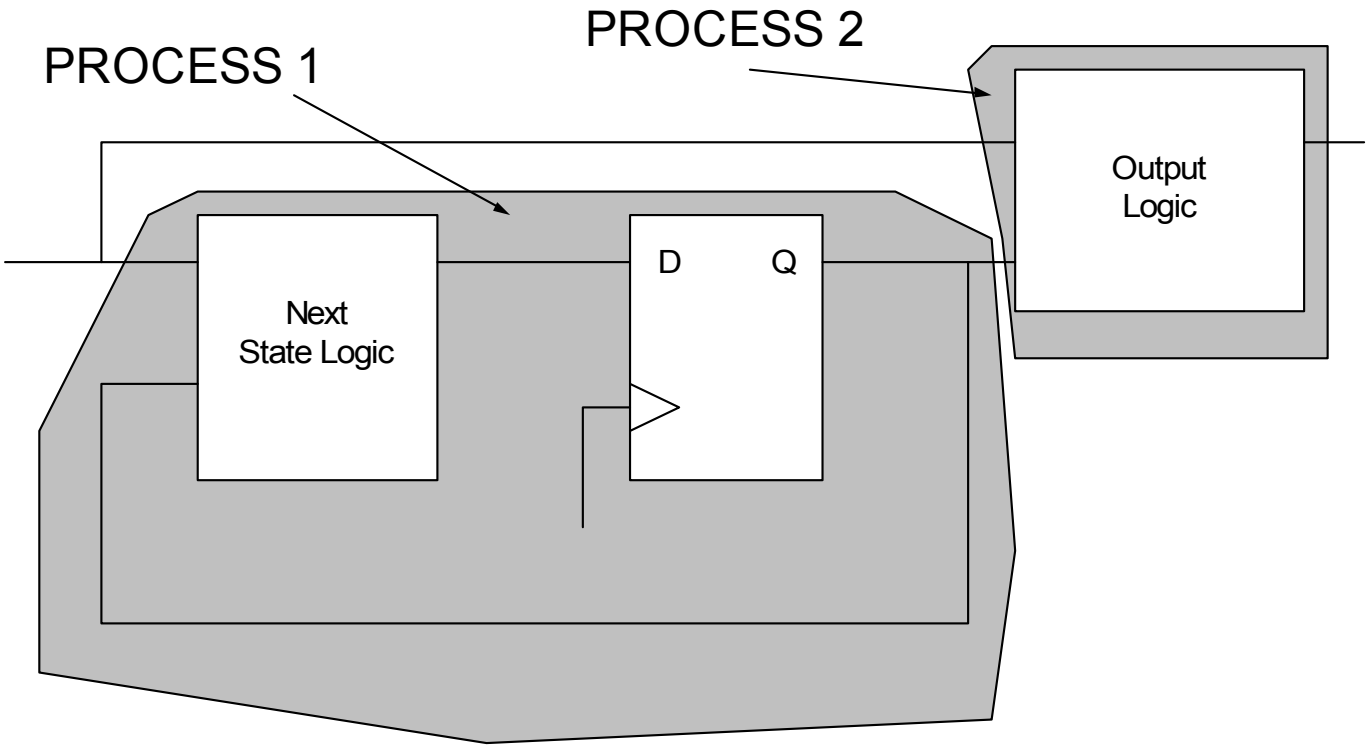
Remember, a Mealy machine has outputs on both the current state and the current input(s).

Can we define this in one process? If so, what type?

A Mealy state machine:



Break into two processes:



```
architecture behavioural of mealy is
signal current_state : bit_vector(1 downto 0) := “00”;
begin

        process (clk)
        begin
           if (clk’event and clk=‘1’) then
                case current_state is
                        when “00” => if (in_sig=‘0’) then current_state <= “01”;
                                                     else current_state <= “00”;
                        when “01” => ….
                 end case;
             end if;
        end process;

        process (current_state, in_sig)
        begin
              out_sig <= some function of current state and in_sig
        end process;

end behavioural;
```

So for state machines, remember this:

Moore Machines: can do in one process

Mealy Machines: must break into two processes (some people use three processes, by separating next state logic from flip-flops)

Summary of this lecture:

Make sure every process is one of these three types:

Type 1: Purely Combinational
Type 2: Purely Synchronous
Type 3: Purely Synchronous with async set/reset

If you do so, your code will be synthesizable.

If you don't, there is a good chance your circuit won't work.

Remember this when debugging: for each process:

- Identify which of the three types it is
- For that type, follow the pattern in this slide set, exactly.