

RTL Coding style and when we need to deviate from such a style

In FPGA/ASIC design, it is recommended that you adopt a pure synchronous design methodology as much as possible. Synthesizing tools can interpret (and map) such designs successfully.

Synchronous design

In a good synchronous design, you perform

- register transfers only on significant clock edges
- avoid latches
- avoid asynchronous loading of registers/counters using direct set/direct clear
- avoid asynchronous setting/resetting of flip-flops acting as status/flags
- avoid level sensitive (asynchronous) RAM write operations

In a synchronous design, you compute the value (to be deposited in a destination register) ahead of the arrival of the clock edge (at least setup time before the clock edge) and deposit the value at the clock edge. The computation logic is a combinational logic such as an incrementer in the case of a counter.

Two major coding choices

Two coding styles are described below. The first choice amounts to performing low level design by hand. Most engineers tend to use this style initially because they have been performing hand-designs / schematic-based designs and they are perhaps using HDL for the first time. Very soon, they realize the power of inference by the synthesis tools and start coding using the second style called RTL coding style. Experienced designers will occasionally use the first coding style to cater for special cases requiring deliberate deviation from the RTL method of designing (or to overcome the limitations of the synthesis tools).

Coding choice #1 (low-level coding, trying to translate the schematic design into HDL)

In this choice of coding, separate combinational always procedural blocks/continuous assignments are used for all combinational logic. Clocked always procedural blocks are used *just to update* registers with the values computed by the combinational always procedural blocks. For example coding NSL (Next State Logic) by hand in a combinational always procedural block and having a clocked always procedural block just to update the Current_State with the value of the Next_State (as computed by the combinational always procedural block) comes under this style of coding.

In a datapath, if a register X is loaded with Xin in the initial state, and with (X - Y) in a later computational state, one can visualize a subtractor to calculate (X - Y) and a multiplexer to select between Xin and (X - Y) and code them as combinational blocks. One also needs to think of the control signals such as X_Mux_Select and X_Load and write a combinational block OFL (Output Function Logic) to generate those signals.

Coding choice #2 (preferred RTL coding style exploiting the power of inference by the synthesis tool)

In this choice of coding, one or more *clocked* always procedural blocks are used to describe the **control unit** and **datapath**. Combinational hardware is left to the synthesis tool to infer.

This is the preferred coding style called RTL coding style.

Let us take the simple divider problem (discussed in class) as an example to illustrate the coding style.

Example “A” of coding choice #2 (Separate Control Unit and separate Data Path Unit)

CONTROL_UNIT

```
always @(posedge Clk, posedge Reset)

begin : CONTROL_UNIT
    if (Reset)
    |begin
    |    state <= INITIAL;
    |end
    else
    |begin
    |    (* full_case, parallel_case *)
    |    case (state)
    |    |INITIAL:
    |    |begin
    |    |    if (Start)
    |    |        state <= COMPUTE;
    |    |end
    |    |COMPUTE:
    |    |begin
    |    |    if (X < Y)
    |    |        state <= DONE_S;
    |    |end
    |    |DONE_S:
    |    |begin
    |    |    if (Ack)
    |    |        state <= INITIAL;
    |    |end
    |    |endcase
    |end
end // CONTROL_UNIT
```

**always block sensitive to
Clk and Reset**

DATA_PATH_UNIT

```
always @(posedge Clk)

begin : DATA_PATH_UNIT
    (* full_case, parallel_case *)
    case (state)
    |INITIAL:
    |begin
    |    X <= Xin;
    |    Y <= Yin;
    |    Quotient <= 0;
    |end
    |COMPUTE:
    |begin
    |    if (!(X < Y))
    |    |begin
    |    |    X <= X - Y;
    |    |    Quotient <= Quotient + 1;
    |    |end
    |end
    |DONE_S:
    |begin // it is best to have "begin"
    |end // and "end" so that code
    |    // can be added later
    |endcase
end // DATA_PATH_UNIT
```

**always block sensitive to
just Clk alone**

Example “B” of coding choice #2 (Combined Control Unit and Data Path Unit)

CONTROL_UNIT and DATA_PATH_UNIT combined into one always block in one case statement

```

always @(posedge Clk, posedge Reset)

begin : CU_n_DU // Control Unit and Data Path Unit combined
  if (Reset)
    begin
      state <= INITIAL;
      X <= 4'bXXXX; // to avoid recirculating mux controlled by Reset
      Y <= 4'bXXXX;
      Quotient <= 4'bXXXX;
    end
  else
    begin
      (* full_case, parallel_case *)
      case (state)
        INITIAL:
          begin
            // state transitions in the control unit
            if (Start)
              state <= COMPUTE;
            // RTL operations in the Data Path
            X <= Xin;
            Y <= Yin;
            Quotient <= 0;
          end
        COMPUTE:
          begin
            // state transitions in the control unit
            if (X < Y)
              state <= DONE_S;
            // RTL operations in the Data Path
            if (!(X < Y))
              begin
                X <= X - Y;
                Quotient <= Quotient + 1;
              end
          end
        DONE_S:
          begin
            // state transitions in the control unit
            if (Ack)
              state <= INITIAL;
            // RTL operations in the Data Path
            // In DONE_S state, there are no RTL operations in the Data Path
          end
      endcase
    end
  end
end

```

In the above CU_n_DU always procedural block (in Example “B”), both state transformations and data transformations are coded in one single block in one single case statement. While this is the preferred coding style, one should address the fact that most of the data registers do not need to be reset, however they are coded under an always procedural block sensitive to reset (besides clock).

Mr. Bruin's
Coding

```
always @(posedge Clk, posedge Reset)

begin : CU_n_DU // Control Unit and Data Path Unit combined
  if (Reset)
  | begin
    | state <= INITIAL;
  | end
end
```

Mr. Trojan's
Coding

```
always @(posedge Clk, posedge Reset)

begin : CU_n_DU // Control Unit and Data Path Unit combined
  if (Reset)
  | begin
    | state <= INITIAL;
    | X <= 4'bXXXX; // to avoid recirculating mux controlled by Reset
    | Y <= 4'bXXXX;
    | Quotient <= 4'bXXXX;
  | end
end
```

Consider the above two ways of writing the reset part of the code.

The first code is not-so-thoughtful. It is a common mistake committed by engineers (even after years of HDL coding).

Note that in the first “if (Reset) clause”, the simple-minded engineer (Mr. Bruin ?) simply did not talk about what happens to “X” and “Y” registers and “Quotient” counter under the “if (Reset) clause” because he knows that these items do not require reset, so nothing needs to be done to them.

Unfortunately, the synthesis tools do not think that way! The synthesis tools will interpret that the “X” and “Y” registers and the “Quotient” counter have to retain their current values under (Reset) and hence they provide a recirculating mux to return the current content of each of these registers for deposition into themselves on each significant edge of the clock as long as (Reset) is true. This means that we have **extra hardware**, (further means *extra cost*, and also *slower*).

In the second “if (Reset) clause”, the designer has written statements such as **X <= 4'bXXXX;**. These statements tell the synthesis tools that there is no need to retain/preserve the current value in these registers/counters. The synthesis tools do not create any extra hardware to deposit “4'bXXXX”. On the other hand, the synthesis tools remove the earlier recirculating muxes and we get **simpler faster** designs compared to the first “if (Reset) clause”.

Summary of our recommendation:

1. Avoid coding at low level. 2. Avoid coding separate combinational processes. Let the combinational process be implicitly specified in the RHS (Right Hand Side) expression of a RTL statement. 3. Use zero delay coding for the core design (for the DUT and not for the testbench) 4. Combine coding of the CU and the DPU in a case statement under one clocked always block.

If you follow the above style of coding, **you do not have to visualize the datapath components, such as comparators, subtracters, multiplexers and code them as separate combinational always procedural blocks. You do not have to think of control signals such as X_Mux_Select and X_Load and write a combinational always procedural block or a bunch of assign statements to generate the outputs.** The synthesis tool (such as the Xilinx Synthesis Tool (XST) or the Synopsys Design Compiler (DC)) would infer all those needed details. You will be able to complete the code in a shorter time. The code will be more readable and maintainable. The code is concise and it resembles a pseudo code just to describe the algorithm.

Please see the diagram on the next page depicting the HDL view of the division between the CU and the DPU. This view of HDL facilitates the use of a simple case statement [`case (state)`] to code the RTL operations.

It is common for engineers learning to write HDL for the first time to be worried about the above coding style for two reasons:

(1) Some think that they need to have additional items in the sensitivity list of the always block, for example, Xin, state, etc. But in a synchronous design, you will not take any action until the clock edge (or Reset) going active. So it is a wrong intuition to think that you need anything other than clock and reset in the sensitivity list of either a positive-edge triggered or negative-edge triggered always procedural block.

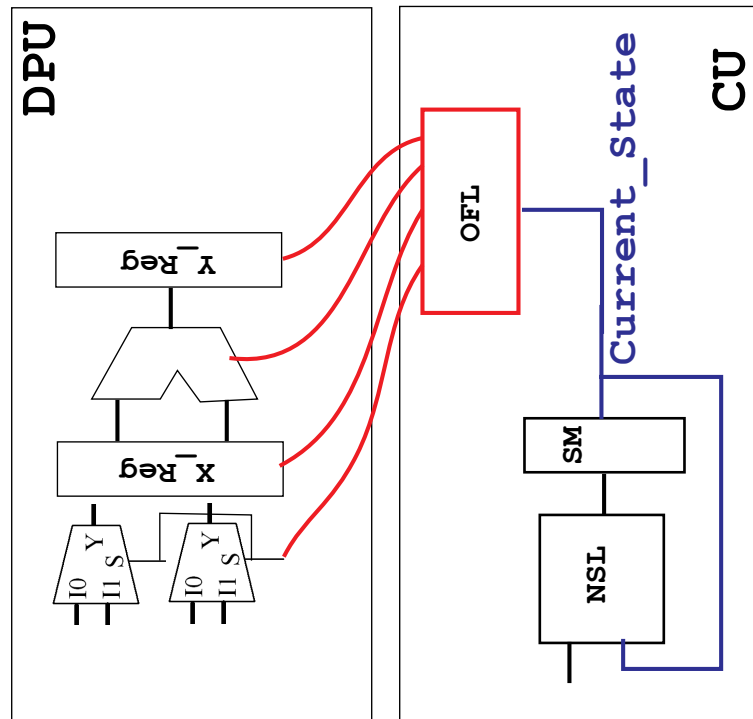
(2) Don't you need to calculate the results of the combinational logic such as the subtracter ahead of the clock?

Yes!

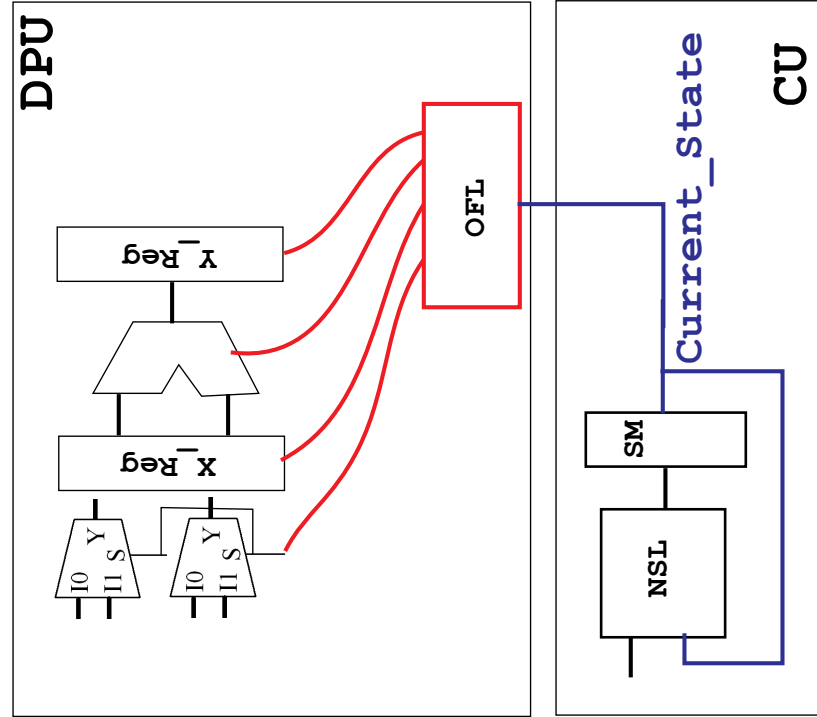
But then, the code seems to imply that we start doing the subtraction "after" the clock edge in a hurry!

Well, that is NOT the implication inferred by the synthesis tools. The synthesis tools clearly know that "the value to be deposited in a register at a clock edge" needs to be computed ahead of (before) the clock edge. The simulation tool computes the value in "no time" (because of zero-delay behavioral coding) and deposits the value into the destination register right at that edge (with no loss of time due to computation).

Traditional division
between DPU and CU
 OFL (combinational logic)
 is in the CU.



Division between DPU and
CU for HDL coding
 OFL (combinational logic)
 is moved to DPU. It is
 NOT coded explicitly.
 The OFL is implicit in the
 DPU's RTL in the CASE
 statement.



Deviation from the RTL coding style (optional material for EE201L)

So, when and why would we deviate from the above RTL coding style?

Generally RTL Coding style assumes that all register transfers occur on an edge of a free running system clock and all combinational processes complete in one clock cycle.

However, the above is not necessarily the case. Some of the common deviations are as follows. These are covered in EE560.

1. Combinational operations such as multiplication/division may take multiple clocks to finish.
2. Any real system consists of several stand-alone subsystems cooperating and communicating. You may need glue-logic between systems to facilitate this coordination and communication. This glue logic does not follow the RTL coding pattern/style.
3. Gated clocking
4. Asynchronous operations
5. Pipelined design employing data-stationary control (you do not describe this using a “`case (state)`” construct).
6. Resource sharing in special operations/architectures.
7. Logic to deal with external devices (for example, the seven-segment displays).
8. Logic to deal with instantiation of IP (intellectual property) components such as DDR memory controllers.