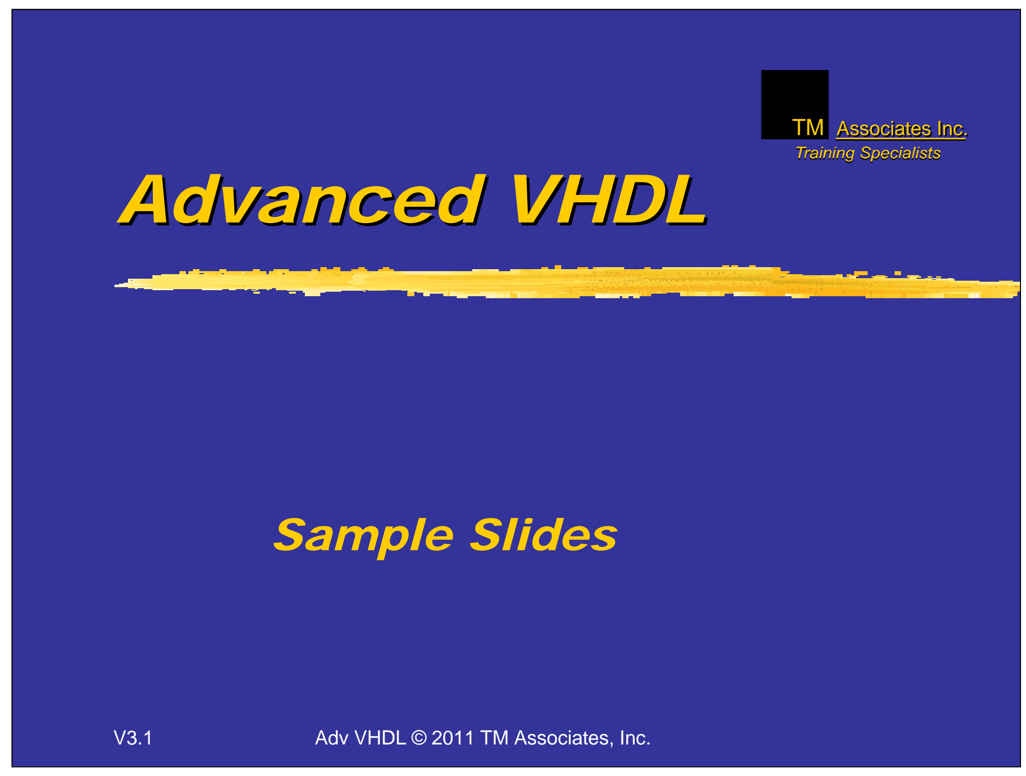
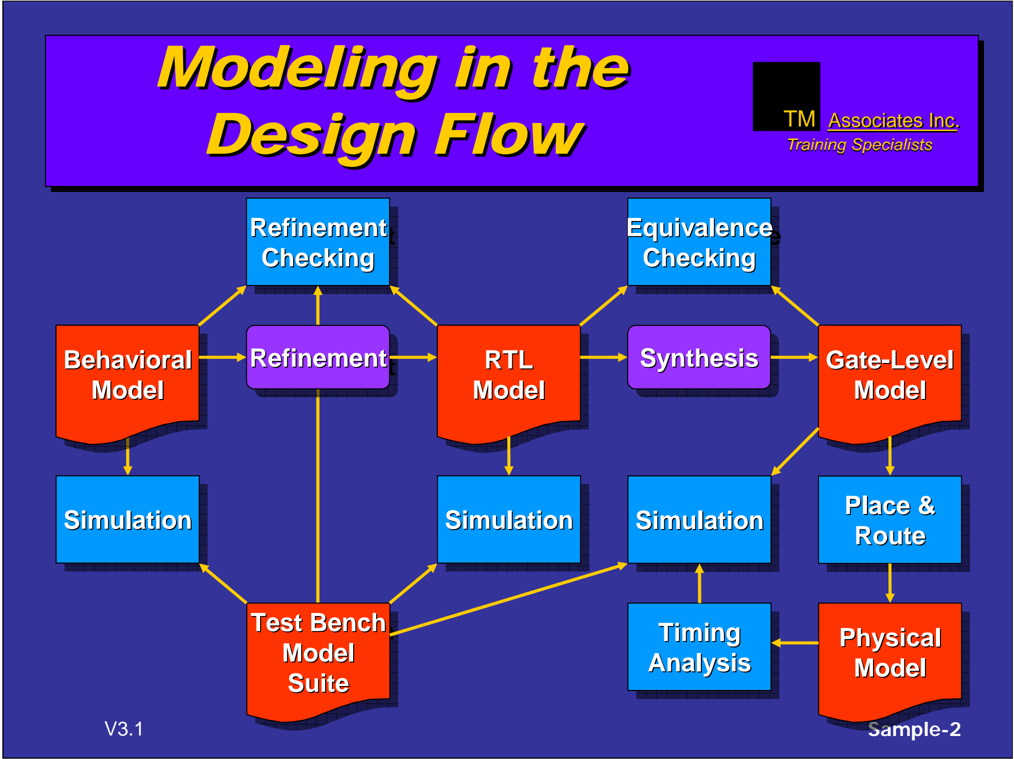


The sample slides come from a number of topics in the Advanced VHDL course.
For a complete outline visit the website

www.tm-associates.com

© 2011 TM Associates, Inc.



This is a stereotypical design flow, illustrating some of the artifacts and processes involved. As you can see, models are pervasive.

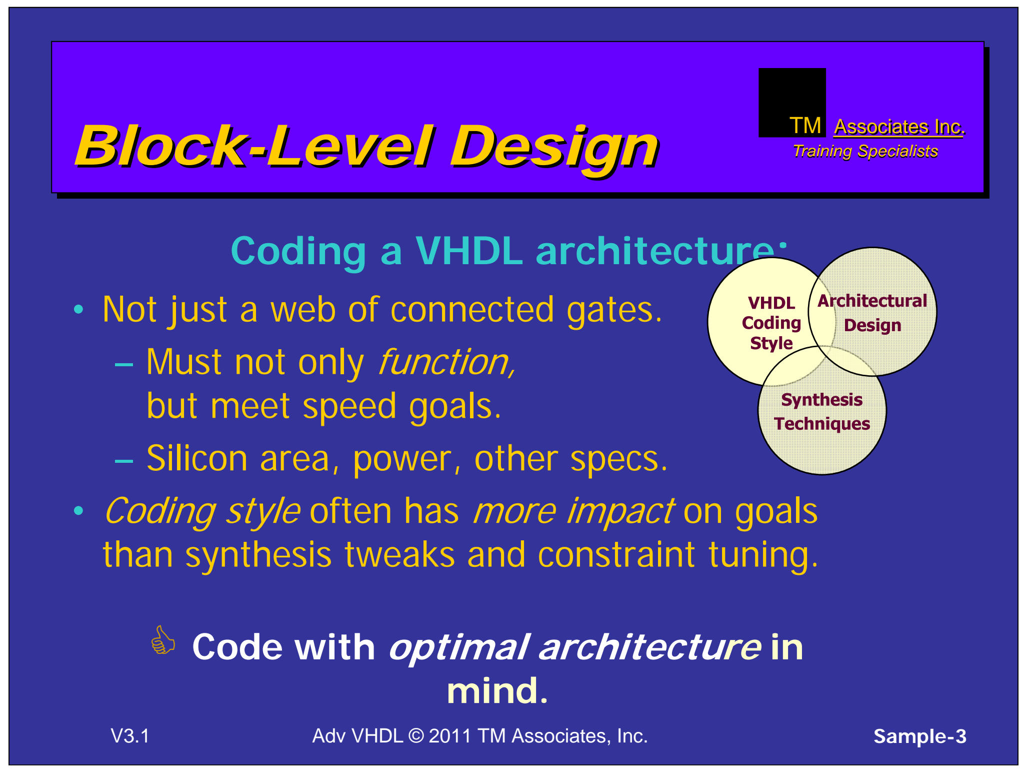
A behavioral model expresses the abstract function of the system. A test bench model suite includes descriptions of the stimuli presented to the system by its environment. We use simulation to verify that the system produces the correct response.

We refine the behavioral model (usually manually) to a register-transfer-level (RTL) model. Again, we simulate to verify faithful refinement. We can also do refinement checking by simulating both the behavioral and RTL models with the same stimuli and verifying that they produce the same results.

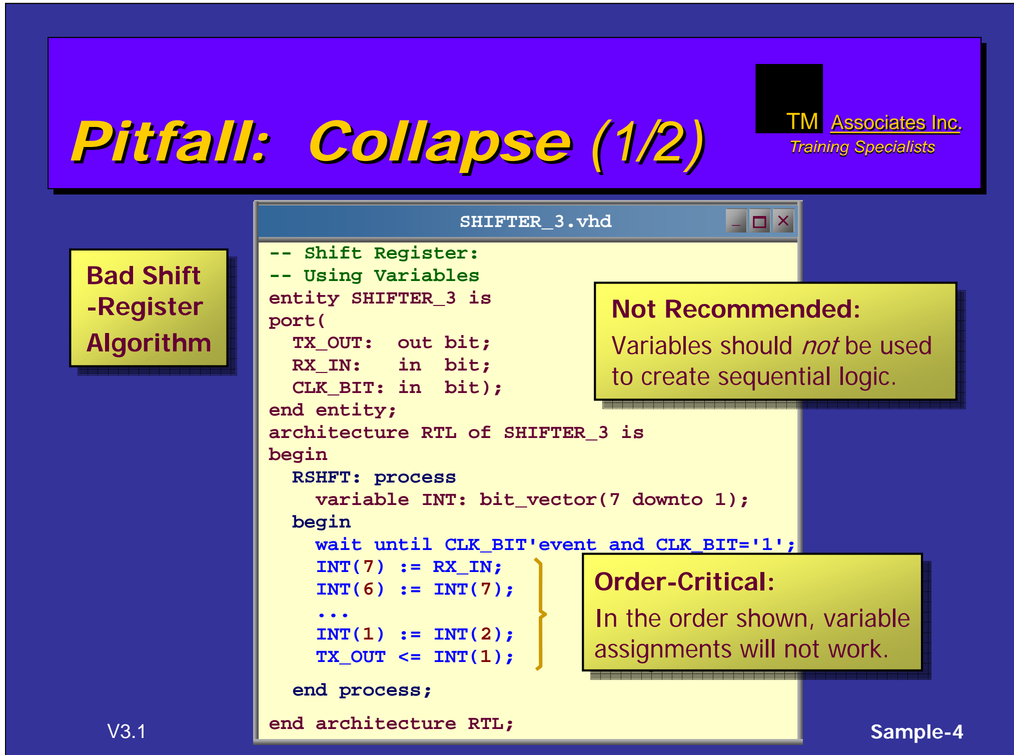
We can then synthesize (usually automatically, with lots of hand-holding!) the RTL design to get a gate-level model. Again, we simulate to verify correct synthesis. We can also do formal equivalence checking.

Finally, we place and route the gates to get a physical design model. We can do timing analysis on this model to get more accurate estimates of delays, and use the estimates to resimulate the gate-level model.

© 2011 TM Associates, Inc.



- Industry experience with HDL synthesis has shown that the most dramatic improvements in speed and other goals are made at the *architectural* level.
- For example, performing computations *in parallel* instead of in cascade; increasing the *width* of processed data; avoiding *ripple-carry* structures.
- Part of *architectural-level* design, too, is thinking through what's needed to minimally implement a function, before starting to churn out pages of code.
- Therefore, code with an *optimal architecture* in mind, relying on synthesis options and constraint fine-tuning only for the last 10—20% improvement.
- We'll apply this approach to the **`HANDSET`** design, starting with a top-down refinement of an architectural spec or state diagram, prior to writing code.
- At times, we'll consider architectural trade-offs and synthesis techniques.
- Our primary focus in this workshop, however, is VHDL coding style for efficient synthesis and effective verification.

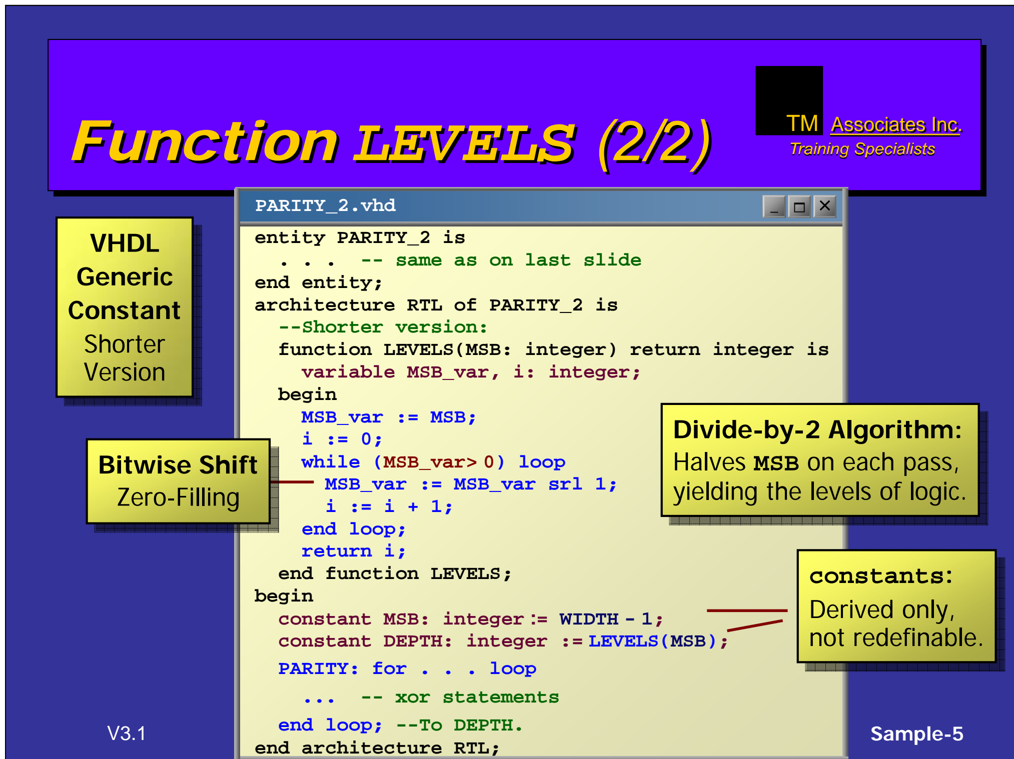


- The code on this slide is identical to **`SHIFTER_1.vhd`**, except that code block **`RSHFT`** now uses eight *variable assignments to* describe the shift register.
- In the order shown on the slide, this attempt to avoid the recommended style will not work. Here's what happens during elaboration.
- In accordance with the WYSIWYG rule, the synthesis tool will reason that bit **`RX_IN`** is *immediately* assigned to **`INT(7)`**, in zero simulation time.
- In the next line of code, **`INT(7)`** is in turn *immediately* assigned to **`INT(6)`**. This *ripple effect* continues, until the *same value* gets assigned to **`TX_OUT`**.
- The unexpected synthesized logic—one flop—is shown on the next slide.
- By *reversing* the order of the eight variable assignments, you could get this code to synthesize correctly. But order-dependent coding is *risky*.

Coding Guidelines:

- *Variables* are allowable for algorithms describing pure combinational logic in **`process`** constructs, functions, or tasks.
- The rippling effect of a chain of variable assignments then closely models the propagation of data through cascaded levels of gate logic.

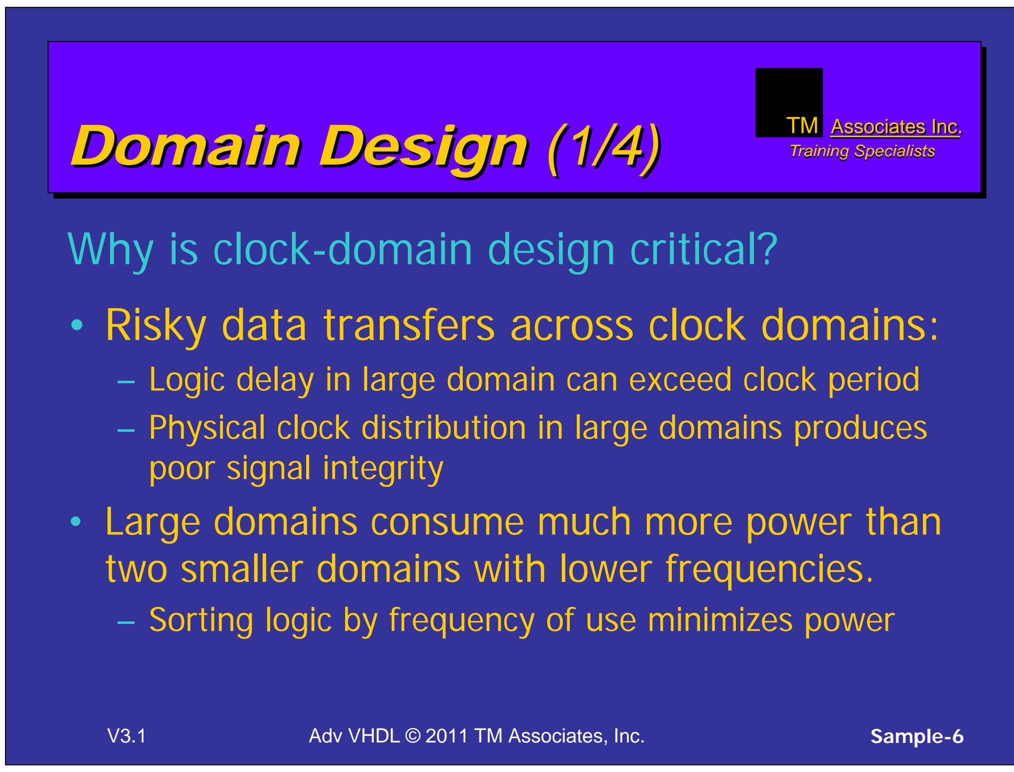
© 2011 TM Associates, Inc.



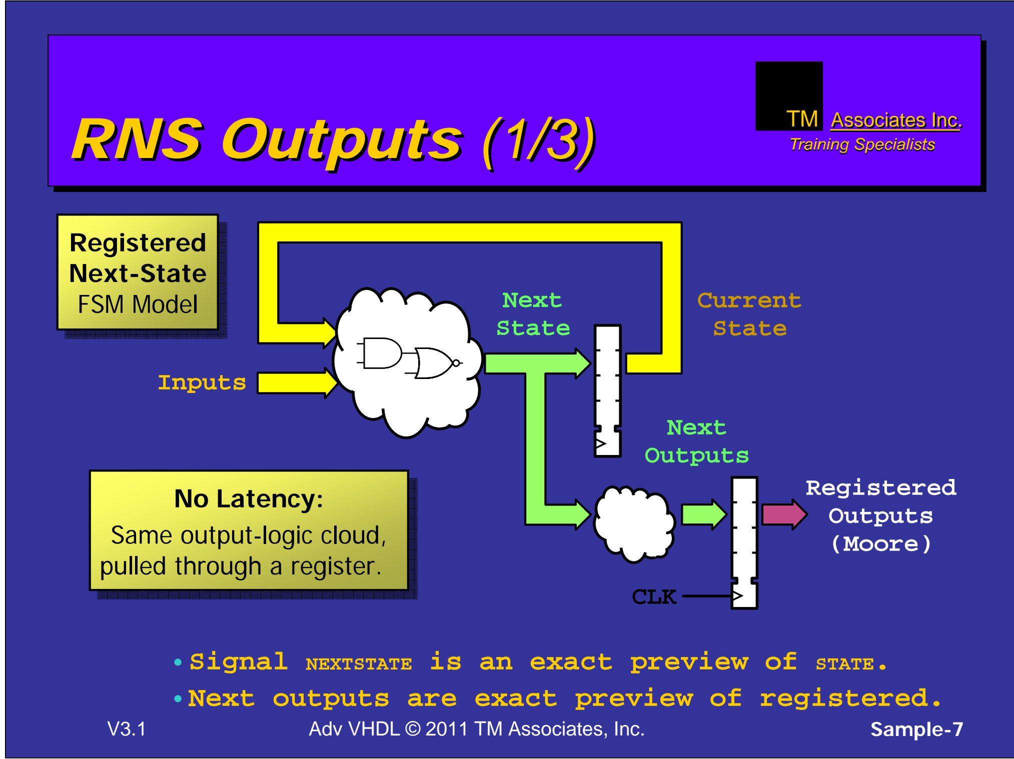
- This slide shows a simpler version of the function that calculates **`DEPTH`** from **`WIDTH`**. It uses a *divide-by-two* algorithm to compute **`LEVELS`**.
- This version requires a different constant, **`MSB`**, which would be **`7`** for eight-bit input data. Parameter **`MSB`** is now the *input* to the function.
- For each iteration, the *copy* of **`MSB`** passed to the function is right-shifted. When no **`1`**s remain, the depth is stored in the counter variable, **`i`**, and returned.
- Intuitively, assuming two-input XOR gates, we are *halving* the number of inputs going into each logic level. The total number of passes is the depth.

Function Details:

- It's the explicitly declared variable **`i`** that's being incremented on each loop iteration.
- Note that the function is non-destructive to **`MSB`**, since only a *copy* is shifted.



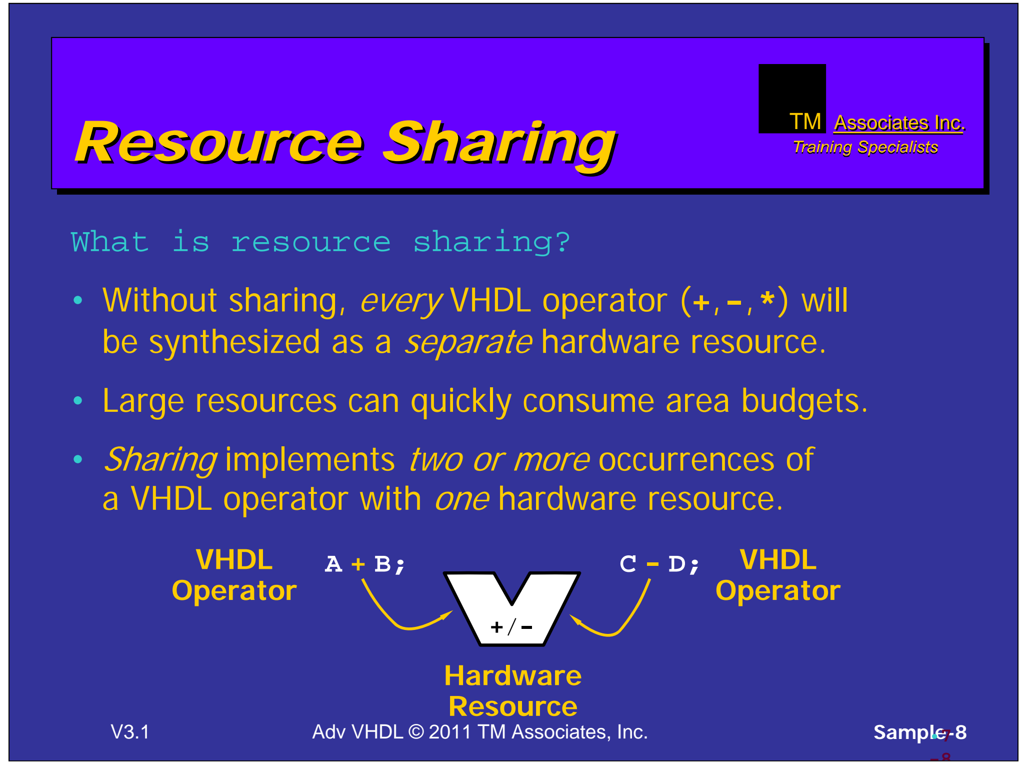
- This slide begins a review of IC clocking strategies—especially issues that can be resolved through partitioning or modification of the code.
- We'll consider a few clock domain guidelines for high performance, regarding *domain size* and *registered outputs*.
- We'll also look at power reduction *by construction,* minimizing clock frequency for each chip section or entity to reduce switching activity.
- Finally, we'll demonstrate the multilevel clocking scheme used to bring clocks to the various modules in the **`HANDSET`** chip.



- As we've seen, the classic model had an undesirable output logic cloud that *violated* the *registered outputs* partitioning guideline from **Unit 4**.
- Worse, if any unregistered outputs were Mealy, transient input changes could propagate through the output cloud, resulting in *output glitches*.
- This slide shows one solution—a *registered next-state* (RNS) FSM model. The *next* state is used to derive *next* outputs, which are then registered.
- This approach is based on Christian Green's article on *SDRAM Controllers*, (***EDN,*** 2 Feb. '98, *State Machine Theory* sidebar, p.163).
- Green called this architecture *modified Mealy*. It has the flexibility of Mealy outputs, without glitching. All FSM outputs appear at $t_{CLK\text{-}Q}$ after the clock.
- Note a minor typo: Green's *Fig. A* reverses *next state* and *next output* labels.

Alternative Solutions:

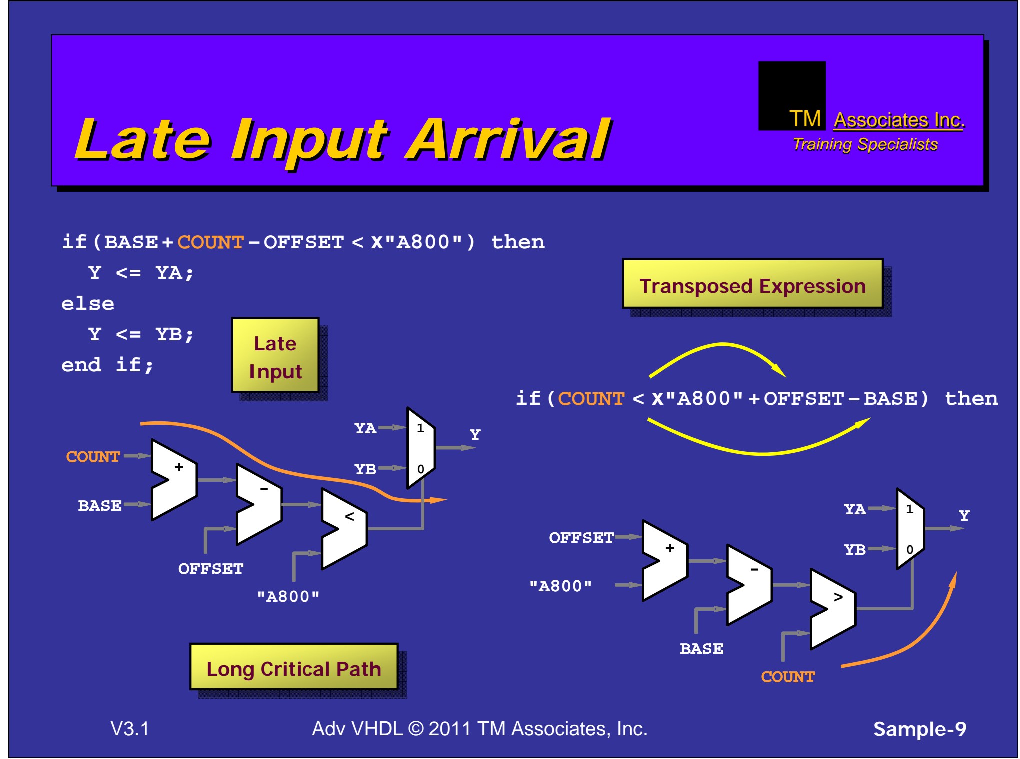
- Contrast this RNS model with a naive solution, in which an output register is simply *tacked on* in the VHDL code to the outputs of entity **CTRL_1**.
- This ensures registered outputs, but introduces one clock cycle of *latency*.
- A better solution—but one involving more recoding effort—is to choose state-encoding bits such that **CODEC** has the *same truth table* as a state bit.
- This ensures that **CODEC** is registered, but in general requires *widening* of the state register, using the redundant state bit(s) to get the right mapping.



- Sharing of resources is a powerful technique to reduce overall gate count.
- The diagram shows a *single* adder/subtractor circuit used to implement *two* mutually-exclusive operators (**+** and **-**) in a typical VHDL entity.
- The two operators are said to be *bound to* this specific hardware resource.
- Without sharing, every occurrence of the VHDL operators **+**, **-**, and ***** in the source code would be synthesized as a *distinct* hardware resource.
- For instance, every **+** sign encountered in the code—provided its operands are not just constants—would result in a *separate* synthesized adder.
- Large unshared resources, such as chained adders, 64-bit multipliers, and array comparators, can consume area budgets and raise power dissipation.
- To write VHDL code that meets specific design goals, you should be aware of what resource sharing is, and how it works for various synthesis tools.

Constraint-Driven:

- Logic optimization is, in general, driven by constraints on timing and area.
- At the gate level, AND-OR logic is often replaced by NAND-NAND logic—but the decision depends on the detailed timing and area requirements.
- In analogy, sharing of resources involves a decision based on constraints.



Synthesis tools assume that input data which originates *outside of* a module will arrive by default at the *start* of each clock cycle—i.e., at time 0.0 ns.

An input arrival time of 0.0 ns is *optimal* for synthesis, since the gate logic then has virtually the entire clock period to process the newly-arrived data.

But if one input arrives *late* in the cycle, a performance bottleneck results.

As shown at left, input **COUNT** is needed at the *beginning* of the clock cycle in order to compute an effective address, and select output data **YA** or **YB**.

But if **COUNT** arrives late—let's say, at ½ of the cycle—then it's unlikely that the cascaded add, subtract, and compare will be done by the next edge.

As shown at right, a simple VHDL code revision can often *accommodate* the late-arriving input, by moving it closer to the output side of the module.

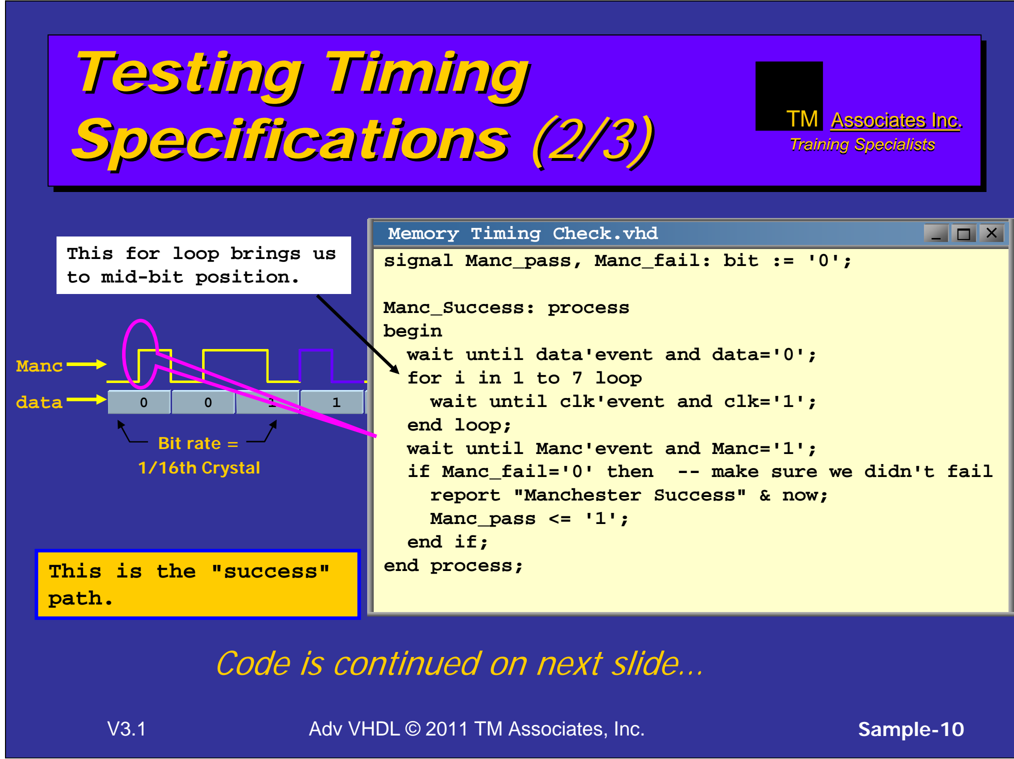
By *transposing* early-arriving inputs to the RHS of the *less-than* inequality, we defer the need for **COUNT** until roughly *halfway* through the clock cycle.

Tool-Specific Details:

Synthesis tools cannot tell when input data will arrive at a module—unless they *extract* arrival times from downstream logic that's *already* synthesized.

We assume the block designer explicitly specifies a late input arrival, using the *Design Compiler* syntax: **set_input_delay** *<time> <port>*.

The corresponding *Synplify* command is: **define_input_delay**.



This slide shows how to test for the success or failure of a specified event. This example checks timing only on falling edges of the data signal. The same checks must be performed on rising edges of data, but is not shown here.

The signal Manc denotes the boundaries of the data cell. Each positive edge marks the beginning of a data bit.

The first process checks for the success of the Manchester encoder. Success is defined as the detection of the Manc signal going active. The success criteria is that the signal must go active in mid-data bit position, after 7 clock cycles.

For more information on this and other VHDL, Verilog, and SystemVerilog training courses, contact

Tom Wille

tw@tm-associates.com

503-656-4457