

CSCE 491

Computer Engr. Design Project

2005/9/6

Week 4

Modeling & Design Applications-1

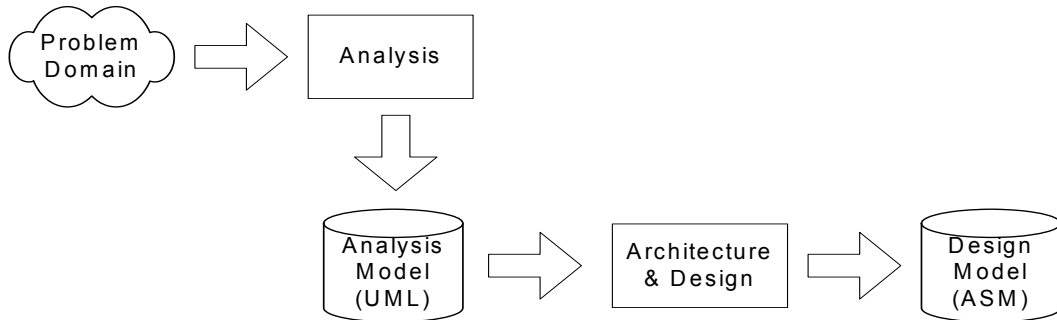
ALU Subtraction, a Counter, and Model Stimulus

© 2003 Dr. James P. Davis

Week 4 - Outline

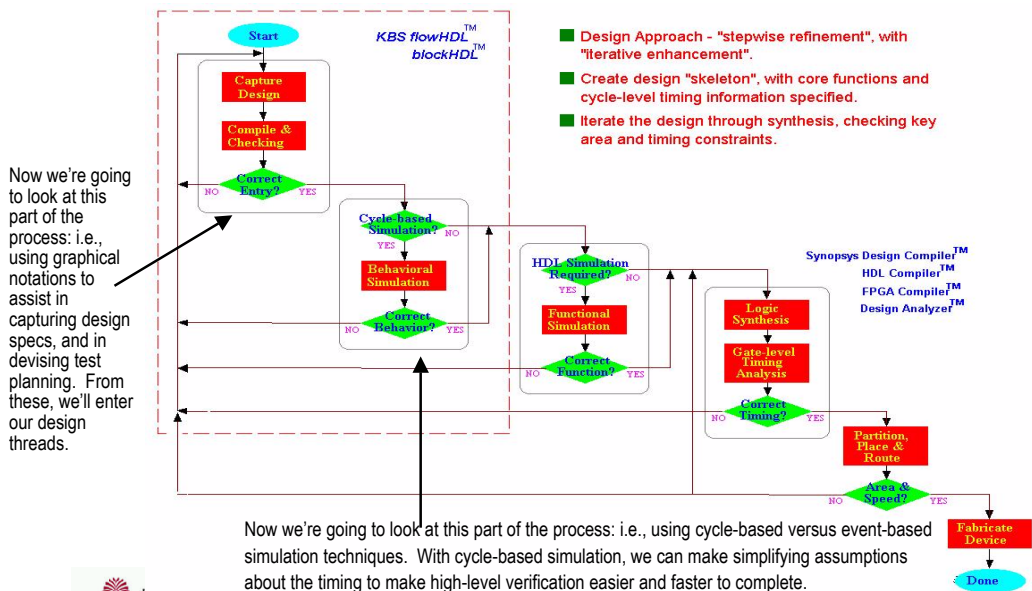
- Objective #1: Work examples to give you more insight on:
 - How to start a design model from an analysis model
 - How to think about digital systems modeling using ASM charts.
 - To explore the connection between the *analysis* and *design* activities of Computer Engineering discipline.
- Objective #2: To discuss the topic of design verification to give you more insight on:
 - How to test and debug a design model once the model has been created.
 - How to think about verification of digital systems using UML and ASM charts as test planning aids, not only to create the design model, but also to create the model for the “test harness” as well. (We’ll create test models as part of our project work later with the 802.11 design.)
- Means:
 - ✓ The Subtract function for the ALU.
 - ✓ Binary Up/Down Counter.
- Result:
 - You should be able to create these designs from scratch for HW Assignments #3 & #4.

Analysis, Synthesis, Verification



Systems Design Method – Design Process

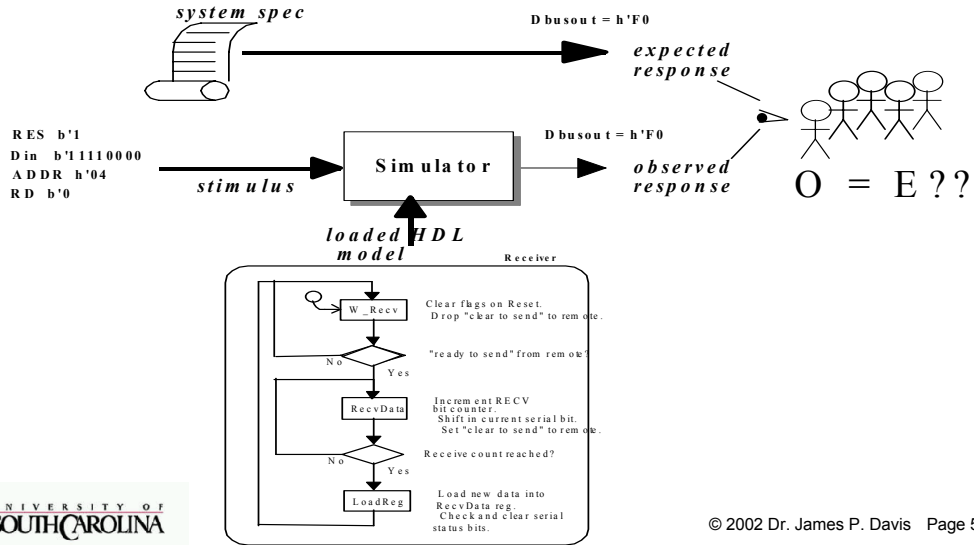
- Conceptual Approach to Architecture and Design.
 - ✓ Create a design model, and iterate on its definition, incorporating debug and verification through simulation runs.



Systems Design Method – Test Process

- Approach to Verification

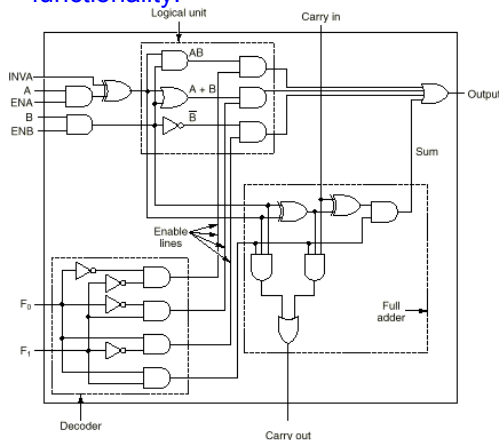
- ✓ Create a set of stimuli that can be used to verify observed behavior against expected behavior.
- ✓ We'll use different levels of design and test specification, to take advantage of fast turnaround in gaining functional and cycle-level timing verification, then obtaining gate-level timing and load analysis, post-layout. The same test harness will be used at both levels.



The Multi-function ALU

- This ALU specification comes from the Tanenbaum, 4th ed., *Computer Architecture* © 1999, Prentice-Hall, text:

- ✓ The gate-level schematic of the circuit is shown to the left.
- ✓ The truth table representation of the functionality to be supported in the Arithmetic Logic Unit is shown to the right.
- ✓ We'll be adding functionality by creating a "wrapper" around this model to add new functionality.



F ₀	F ₁	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	1	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	-1



ALU – Extension for Subtraction

- Our ALU has 6 functions currently supported:
 - ✓ 4 functions via the function lines F0, F1.
 - ✧ AND, OR, ADD – 2 operands (A and B)
 - ✧ NOT – 1 operand (B)
 - ✓ 2 functions via explicit input control lines.
 - ✧ INCA, INVA – 1 operand (A)
 - ✓ But we have no way to subtract, as in $A - B$.
- Strategy:
 - ✓ To obtain subtraction, using the existing working (and verified) model, we'll add another thread that uses the existing ALU functions in such a way to obtain subtraction capability.
 - ✓ This is analogous to creating an object “wrapper”, or extending a “component”, where we extend the capability of an existing block without modifying its internals. It is the essence of constructing reusable model libraries.



Review – How to Subtract

- One method for realizing the subtraction operation is to change the operands into appropriate 2's complement representation, and then add.
- Example (from MacKenzie, CSCE 313 text):

Minuend:	X	1001	<i>This works as long as</i>
Subtrahend:	-Y	0100	<i>operand $Y \leq X$.</i>
	---	-----	<i>It is like adding a positive</i>
Difference:	Z	0101	<i>number and a negative one.</i>
- General case (positive and negative numbers): We take the 2's Complement of the *subtrahend* operand and Add.

$$\begin{array}{r} 1001 \\ 1011 + 1 \\ \hline \end{array} \Rightarrow \begin{array}{r} 1001 \\ 1100 \\ \hline \end{array}$$

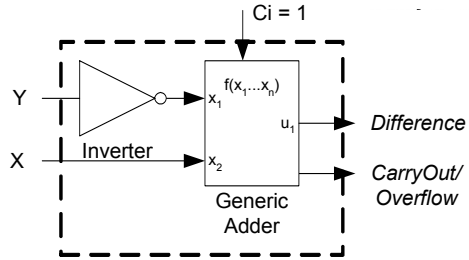
0101 (we ignore the final Carry Out bit and any Sign bit).



Architecture of Subtractor

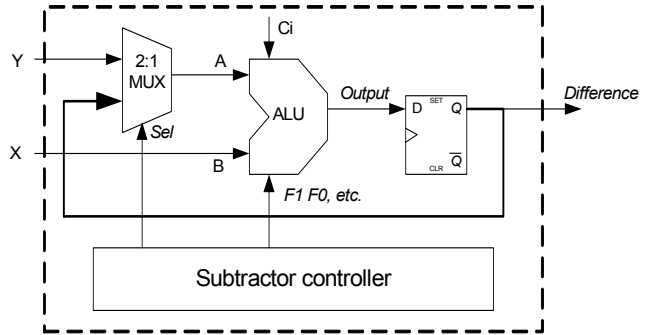
- General structure:

- ✓ Inverter on the subtrahend operator (Y) feeds one input to the generic Adder.
- ✓ The Carry In input = '1' to allow the completion of the 2's complement during the addition.



- ALU “wrapper” structure:

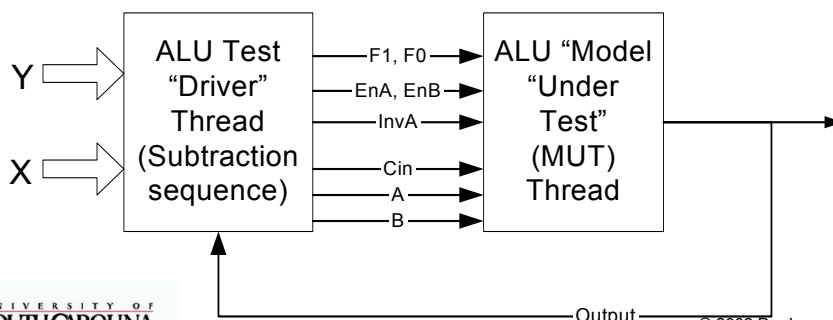
- ✓ How would you feed the subtraction operands to the ALU?
- ✓ Would you need multiple passes through the ALU to complete the subtraction?
- ✓ Can X come into ALU on same cycle as Y which is undergoing 2's Comp op?
- ✓ Do we need a “controller block” to sequence steps?



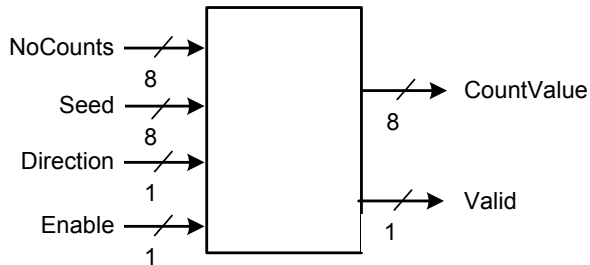
Creating an ALU Test Driver

- Define a 2nd ASM state machine thread.

- ✓ If there is a sequence of ALU operations that must be set up, do so with one ALU run in each state in the Test Driver state sequence.
- ✓ For **HW #3**, re-define all the ALU's “input” buses as “internal” buses in the flowHDL Bus Table.
- ✓ Initialize all ALU inputs by assigning the buses in the Driver thread at the start of each ALU operation run. Read the Output result and feed back around to the input for subsequent run, if needed.
- ✓ If you can get the complete 2's Comp Add done in a single ALU operation, then your Test Driver thread will be much simpler.
- ✓ Make multiple ALU subtraction runs with a range of data operand values.



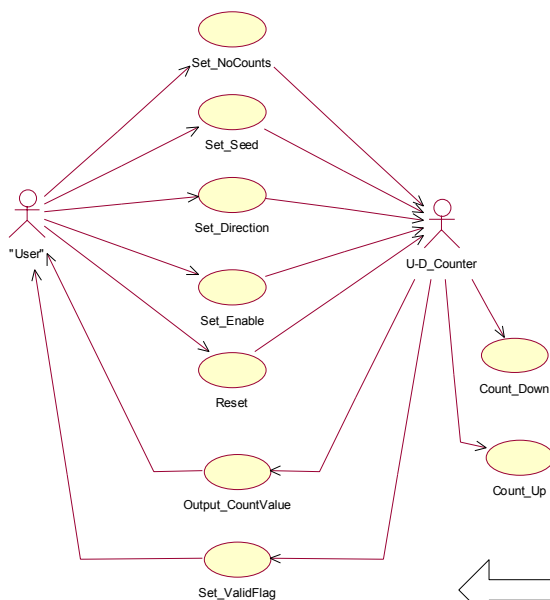
The Binary Up/Down Counter – Register Level



- The Binary Up/Down Counter
 - ✓ This multi-mode counter block takes an input “seed” value, and counts up or down from this value, based on the direction of count. Each count cycle the Enable pulse is set, the Direction bit can be set to count in one or the other direction, and the Seed and *NoCounts* values are set.
 - ✓ The counter will run for the number of cycles indicated by input *NoCounts*.
 - ✓ With an “enable” signal defined, we know our Counter will have a “poll loop”.

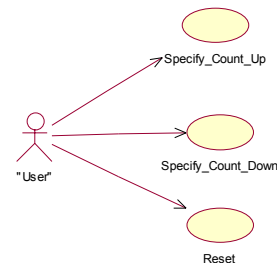


Up/Down Counter Model – Use Case Diagram



2nd Pass Analysis Model.

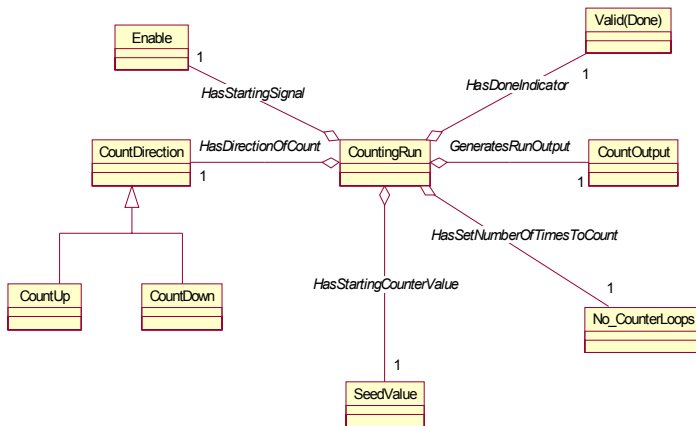
- Use cases and actors:
 - ✓ We have two different Use Case models, one that is more abstract than the other.
 - ✓ The more abstract one is created from the 1st pass analysis of the Counter, the 2nd one is created from a more detailed look.



1st Pass Analysis Model.



Up/Down Counter Model – Class Diagram

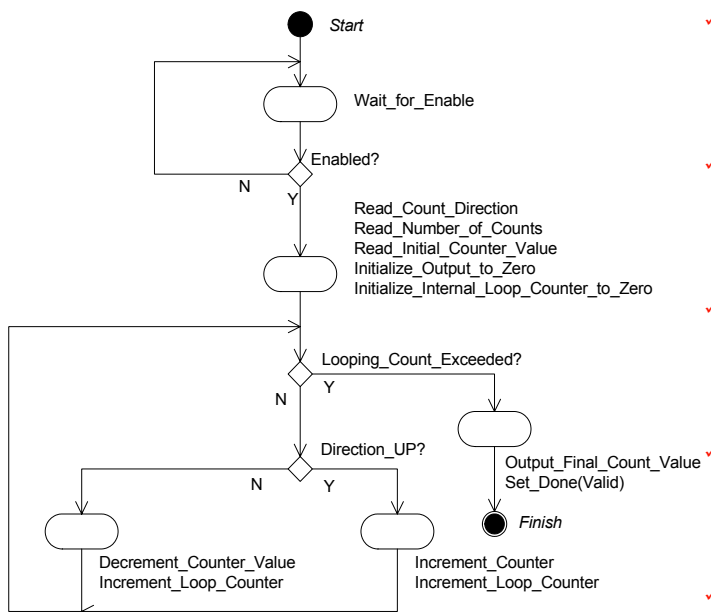


Classes:

- ✓ We start out process of understanding the components of the Counter by abstracting their relationships with the execution of the counting function (a “run”).
- ✓ Although this seems trivial use of the Class Diagram notation, it provides useful organizing information about the semantics of the counter domain.
- ✓ The same purpose could be served by simply listing the hardware signal names and their purpose in a Table, but here, it is in a more easily obtainable form.



Up/Down Counter Model – Pseudocode

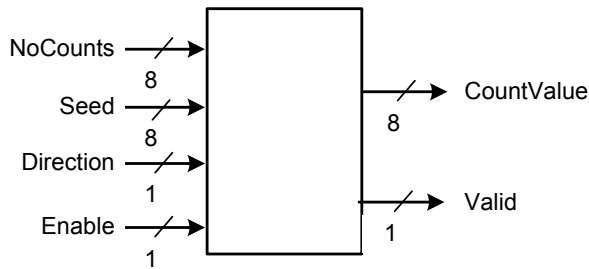


Activities:

- ✓ We use the Activity Diagram notation to get a sense of the algorithmic description of the counter’s functionality.
- ✓ We use this rather than a Statechart, as there isn’t a well articulated abstract state-based model for the counter’s overall function.
- ✓ However, we could model the explicit state of the counter at a lower level of abstraction (not to be considered here).
- ✓ We consider Activity Charts when looking at the control flow within a single Class (i.e., a hardware Block or Thread).
- ✓ In **HW #4**, we take this model and create a working design.



The Binary Up/Down Counter – Test Planning

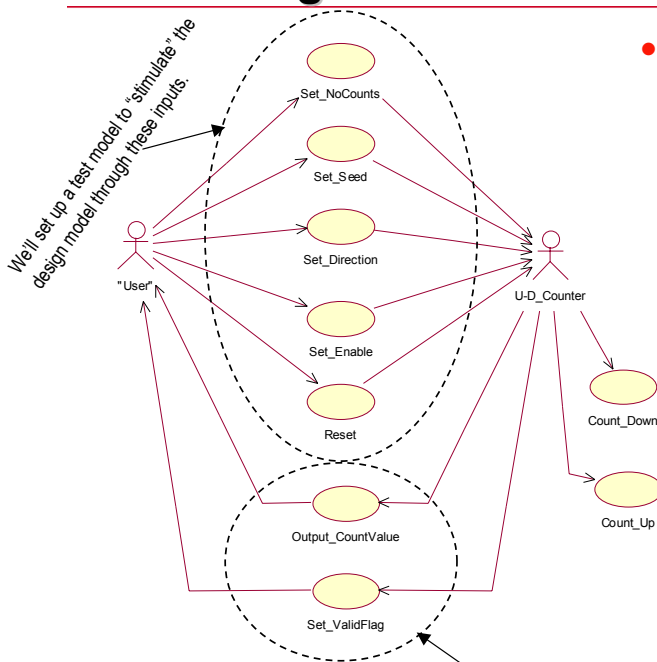


• The Binary Up/Down Counter

- ✓ We have discussed the basic function and operation of the counter.
- ✓ We have created a design model for its behavior.
- ✓ Now, we want to devise a means to test its correctness and accuracy:
- ✓ *Question 1:* does it perform the function we have intended for it to do?
- ✓ *Question 2:* how “robust” is the design in the face of different patterns of inputs?
- ✓ *Question 3:* how do we evaluate the design to insure that it does what it should do?



Creating a Test Plan – Use Case Diagram

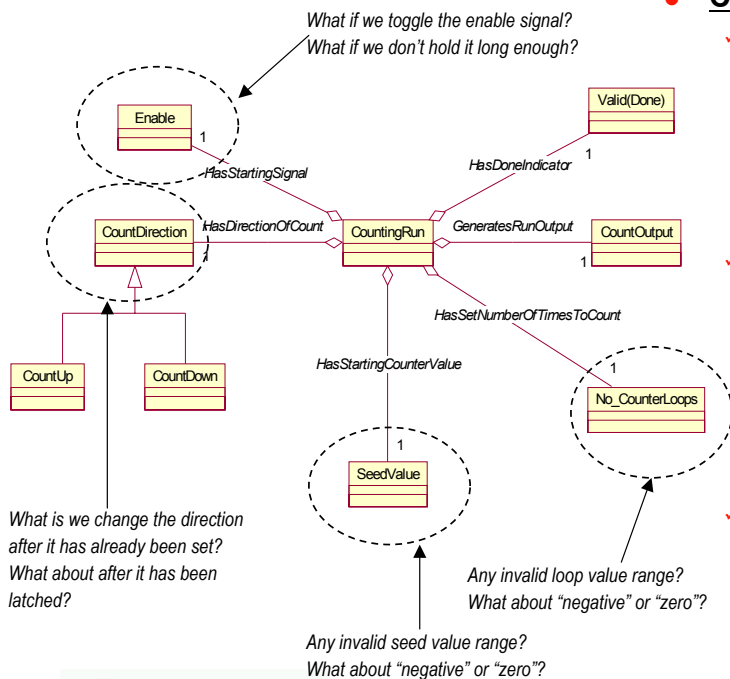


• Use cases and actors:

- ✓ Use Cases have been used in the analysis part of our process.
- ✓ We now want to employ them in helping us to identify and articulate specific test scenarios, and localize test “points” in the design model.
- ✓ In the analysis model, we identified the “user” as an entity external to the Counter; we’ll create a test model that provides some of its functionality for purposes of feeding test data to our design model.
- ✓ The Use Case model gives us the set of actions and interactions we’ll need to be concerned with.



Creating a Test Plan – Class Diagram

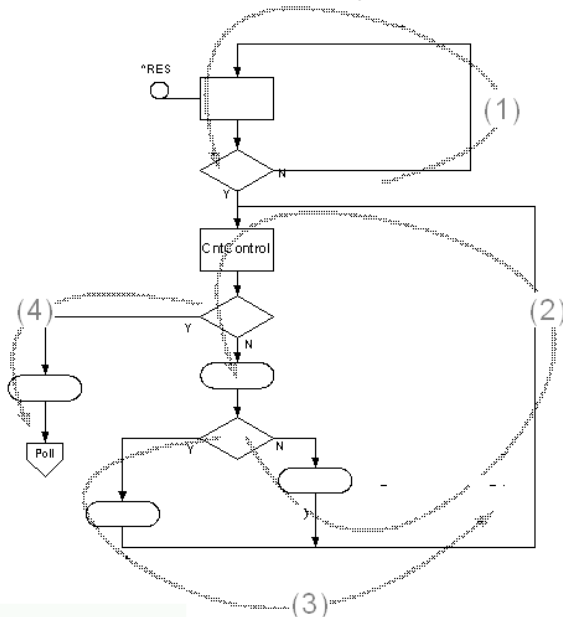


Classes:

- ✓ We start our process of examining the specific “interface points” to our design model by looking at the data definitions at the boundaries of the design.
- ✓ The Class diagram for the design model allows us to identify specific boundary values and conditions for evaluating whether the design operates correctly under “benign” and “error” conditions.
- ✓ Characteristics such as: (1) order of inputs, (2) input values, (3) changing values once set, and (4) value dependencies are places to look.

Creating a Test Plan – ASM Design Model

- In examining an ASM “thread”, we want to evaluate which logic paths we can test by defining a set of input stimuli to excite the design, so that each run allows different logic to be tested.



Test points:

- ✓ We start by looking at specific points allowing us to check different logic paths through the design.
- ✓ We look at the signals and the value ranges that will cause our design to choose a different logic path.
- ✓ We then create a set of stimulus “steps” that will allow us to trace the design in simulation.

Week 4 Summary

- We want to create an algorithmic, clock cycle-driven specification of a sequence of operations that carry out meaningful computation in digital custom logic.
- Often, we may have a block that we want to reuse, and can do so by creating a “wrapper”, passing the input sequence, and providing external control of the operation sequence.
- Always, we want to efficiently verify the function and timing (at the clock cycle level), which we can do by creating a “Test Driver” that treats the “model under test” (MUT) as a “black box”.
- In providing stimulus to a thread, we must be careful about the sequencing of the model under test. Test drivers require “debugging and verification as well.

