

CSCE 491

Computer Engr. Design Project

2005/9/1

Week 3

Digital VLSI Design Methods-2

ASM Modeling

© 2004 Dr. James P. Davis

Week 3 - Outline

- We have briefly discussed System-level specification, analysis and architecture design.
 - ✓ Unified Modeling Language (UML) as the means for expressing the specifications and exploring design architecture.
 - ✓ We'll come back to this when we start discussing details of 802.11 analysis model and architecture.
- Now, we focus on Systems-level architecture and Register Transfer Level (RTL) design using the Algorithmic State Machine.
 - ✓ Discussion of basic drivers behind VLSI design
 - ✓ Discussion of the process, methods and tools we'll be using for executing the VLSI-based hardware design activities for the 802.11 WLAN application.
 - ✓ Focus on Nimbus and the use of FSM (finite state machine, CSCE 211) and RTN (register transfer notation, CSCE 212) for control path and data path architecture and design, using extended ASM notation.
 - ✓ Discussion of methods for design evaluation and tradeoff analysis of designed micro-architectures.

I. Design Heuristics:

Questions to Ask as You Start Modeling a Digital Circuit or System



© 2002 Dr. James P. Davis Page 3

Five Heuristic “Principles” of Digital Systems Modeling

- We express the inquiry of design as “probing” questions capturing a set of heuristic principles we want to apply in algorithm transformation.
- Each principle concerns a different aspect of the architecture analysis and design problem-solving activities.
- Each is used to create a set of candidate architectures that may differ along one or more of the principle dimensions.
- Exploring the design space involves creating candidate models, verifying their correct behavior, then conducting logic synthesis and layout in target technology platform.



© 2002 Dr. James P. Davis Page 4

The Five “Principles” – Question 1

- Question 1: What control and data path functionality should be included in each partitioned block?
 - ✓ Partitioning can be carried out according to “function”, or by “responsibility”.
 - ✓ “Responsibility-driven design” is used in Object-Oriented Analysis & Design practice.
 - ✓ Partitioning according to clear responsibilities, and clear collaboration between modules, minimizes “coupling” and increases “cohesion” in partitioned modules.
 - ✓ Each partitioned block has a range of control and data path capabilities.
 - ✓ Some blocks are only controllers, sending out control signals.
 - ✓ Some blocks are only datapath, providing single or multistaged (pipelined) computation.
 - ✓ Most blocks of computation are a mixture of both control and datapath.



The Five “Principles” – Question 2

- Question 2: What amount of computation can be carried out within a given clock cycle? (i.e., “computing step”).
 - ✓ We assume the design of synchronous sequential systems under the control of one or more “synchronizing” signals.
 - ✓ These signals can be the system clock, or other internally generated signals.
 - ✓ Clocking can be “periodic” or “aperiodic” (to approximate “asynchronous” circuits and “isochronous” coordination in distributed systems).
 - ✓ We will decompose application-specific computation across some number of computing “cycles”, generally synchronized to one or more clocking signals.
- Alternate question: What should length of the clock cycle be in order to accommodate (“service”) our computation?



The Five “Principles” – Question 3

- Question 3: For a specific unit of computation, when do we need to have the data available on the computing unit’s output?
 - ✓ Units of computation involve some datapath function followed by an assignment operation.
 - ✓ For example: **Out <- ADD (A, B)**
 - ✓ Signals are registered, latched or wired on completion of computation.
 - ✓ Concerns the “cycle delay” or “latency” of a specific computation, and when the value is available for use in downstream computations.
 - ✓ Assigning the result of computation to a wire is available immediately. In general, $t_{prop} \ll t_{clk}$, so we ignore wire delay for now.
 - ✓ Assigning computation to a register incurs a one-cycle delay from when the computation is “scheduled” by the state machine to when the result is available on the registered output for the next computation.



The Five “Principles” – Question 4

- Question 4: For a given stream of computation, what must be computed serially, and what can be computed in parallel?
 - ✓ Algorithmic computation may be *strictly ordered*, *partially ordered*, or *unordered*.
 - ✓ The ordering of computation is based on the data dependencies between each stage or “unit” of computation.
 - ✓ What forms of parallelism? We use basic “patterns”.
 - ✓ Pipelining of the datapath.
 - ✓ Coordinated control via “handshaking” or “arbitration” in the control path.
 - ✓ Concurrency in either the control or datapath, or both.
 - ✓ The form of parallelism governs computational “latency” and “throughput”.
 - ✓ The “granularity” of computation of each parallel unit, and topology of interconnect between units, affects circuit complexity.



The Five “Principles” – Question 5

- Question 5: For a given stream of computation, what is the nature of “looping” in the computation?
 - ✓ Algorithmic computation may require iteration (FOR loop).
 - ✓ Iteration requires control of the looping process using feedback (using a Counter and/or Comparator circuit).
 - ✓ Feedback may exist in the datapath, where an internal bus may recycle a newly computed value back through a computing element.
 - ✓ In hardware, feedback without sufficient “delay” unit causes oscillation, metastable, bistable or other non-converging behavior. A register or latch is needed to store values prior to feedback through the circuit.
 - ✓ Looping for iteration can be “unrolled” to minimize logic depth, cost of comparator logic.

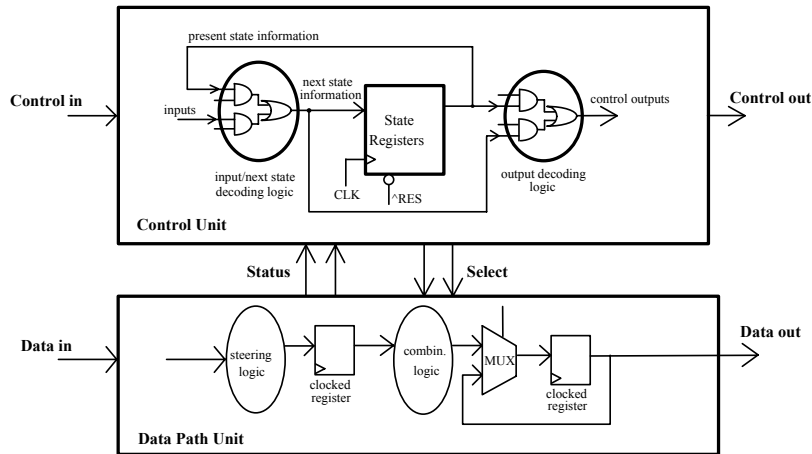


II. ASM Design Notation:

Specifying Signal Assertions and Register Assignments



Functional Partitioning of Control and Datapath



<u>Control Unit</u>	<u>Data Path Unit</u>
<ul style="list-style-type: none"> ❑ modeled using FSM model ❑ defines the clock-based sequencing of actions in the data path or external to the block 	<ul style="list-style-type: none"> ❑ modeled using RTL model ❑ defines both synchronous and asynchronous transformations of data as it moves through the block

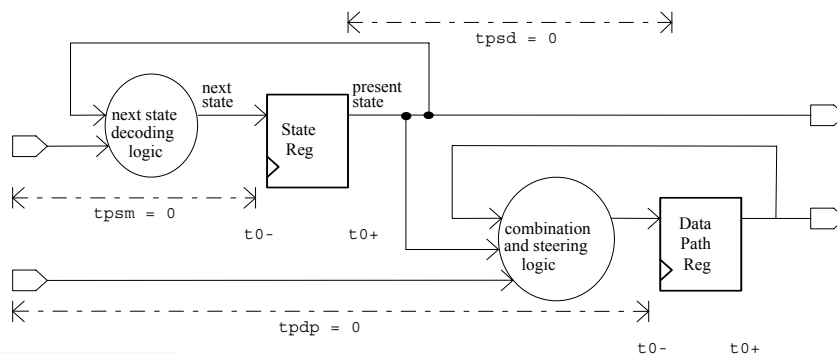


Delay Assumption for Register-Transfer Level

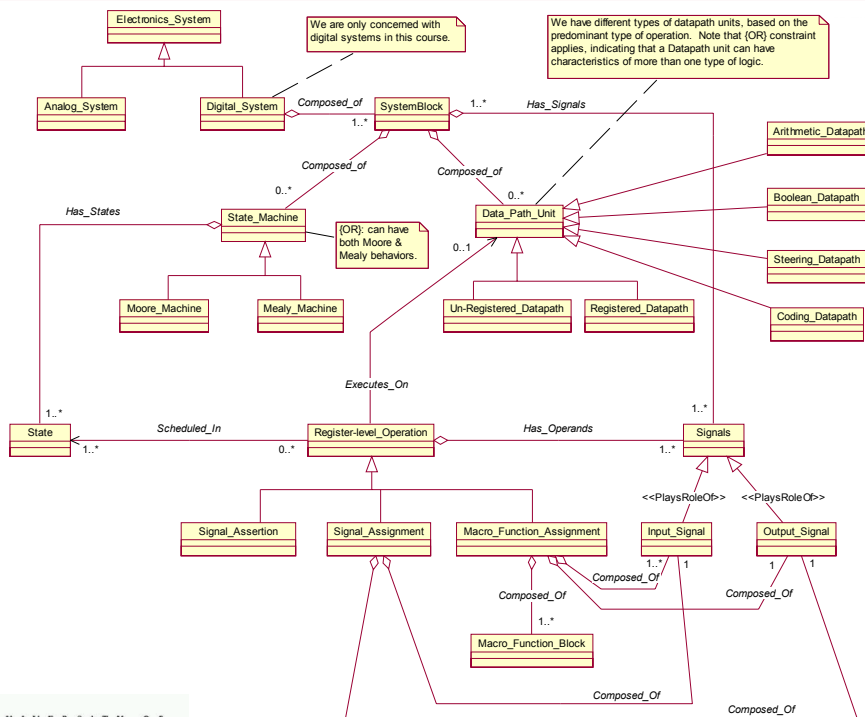
In ASM, we model the behavior of registers using the "limit" assumption. First, at some time t_n corresponding to the active edge of a clock, there is a different value on the input of a register than on its output.

The time between when the register input is "sampled" and when the value appears on the register output cannot be zero. We need to consider the change in "state" of the design on the clock edge, where we are assigning values to the input and wanting to see the results that appear on the output.

We assume the mathematical limit of t_n from both sides of the clock transition, which we refer to as times t_{n-} and t_{n+} . The time t_{n-} is when the register input is being "sampled", and the time t_{n+} is when the sampled value appears on the register output.



Abstract Model of System Concepts



Relationship of State Machines to Datapath

ASM diagrams incorporate information about control path and data path into a single representation. Using this notation, a design can express different design styles for both synchronous and asynchronous behavior of both the control and datapath.

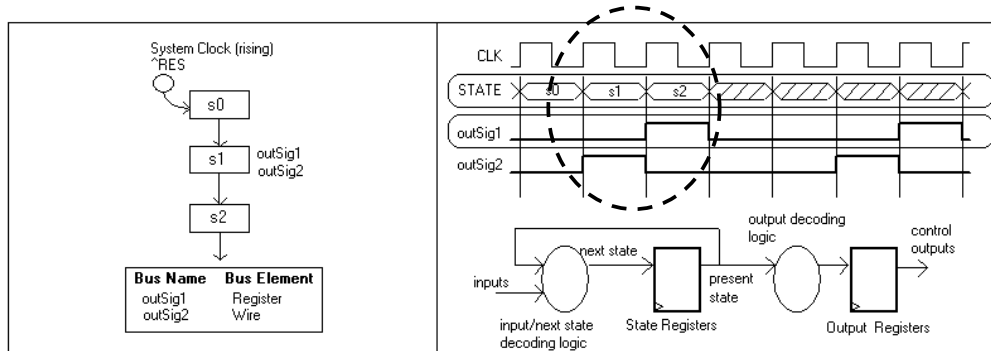
FSM Type	Registered Datapath	Unregistered Datapath
Moore Machine	Assignments placed on state "box" in flowdiagram, bus defined as "Register" in Bus Table.	1. Assignments placed on state "box" in flow-diagram, bus defined as "Wire" in Bus Table. 2. Assignments placed on output "oval" result in either wires or latches in datapath, buses defined as "Context" in Bus Table.
Mealy Machine	Assignments placed on output "oval" after condition "diamond", buses defined as "Register" element in Bus Table.	Assignments placed on output "oval" after condition "diamond", buses defined as "Latch" or "Wire" result in either latches or wires in data path.



Moore Machine - Registered Assertions

The designer may specify that `outSig1`, dependent only on present state, be registered by selecting the bus Element as a Register in the Bus Table. Selecting bus elements as registers provides good isolation from signal transients and race conditions, by synchronizing the state machine output to the clock.

The output action `outSig1` is scheduled upon entry to state `s1`. However, the asserted signal will be delayed on the output of the `outSig1` bus by 1 clock cycle. This is because of 1 cycle delay due to additional output register in the control path. The assertion of `outSig1` is held for a 1 clock cycle duration, corresponding with the present state is state being `s2`. Upon leaving `s2`, because the state registers and output registers are tied to the same clock, `outSig1` is de-asserted.

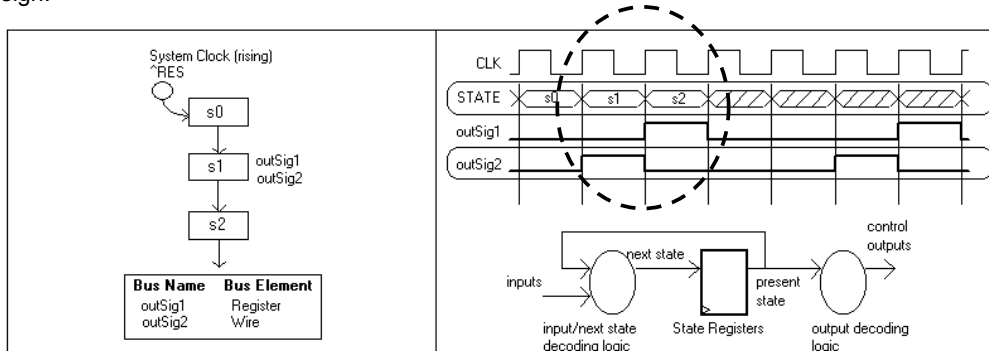


Moore Machine - Unregistered Assertions

The designer may specify that control signal `outSig2`, be unregistered by selecting the Element as a Wire in the Bus Table.

The output `outSig2` is asserted upon entry to state `s1`, and is held for the duration while in state `s1`. Upon leaving `s1`, `outSig2` is de-asserted. Contrast this with the fact that `outsig1`, which is “registered”, does not actually assert until the next clock cycle (it is a registered assertion).

Unregistered Moore-style of design saves registers and reduces delay. Care must be taken to provide good isolation from signal transients and race conditions, by synchronizing somewhere else in the design.

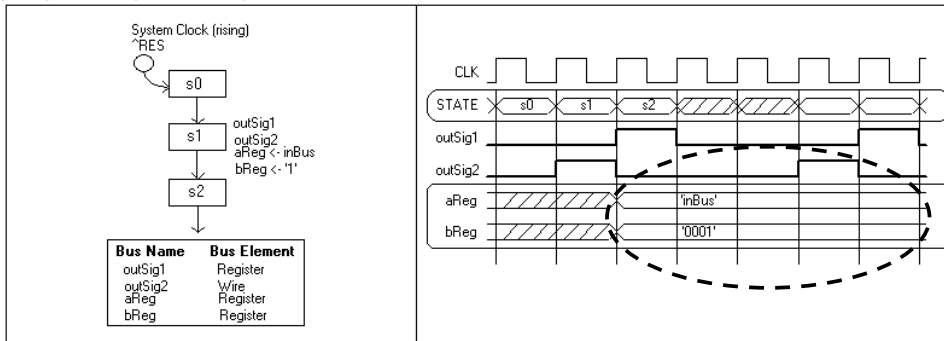


Moore Machine - Registered Bus Assignments

Registered bus assignments may be used by placing expressions on buses that are defined as “Register” in the Element field of Bus Table. The buses in the datapath will be realized using registered logic.

The buses aReg and bReg are realized by using additional layer of registers. This imposes a 1 clock cycle delay from when the operation is scheduled by the state machine in state s1 and when the updated values are propagated to outputs of aReg and bReg. However, the resultant assignment value is preserved in the registered output until it is explicitly modified.

Using registered bus assignments increases gate count and circuit delay of the datapath. However, the designer avoids race conditions, signal transients, and unwanted feedback conditions by designing with registered logic.

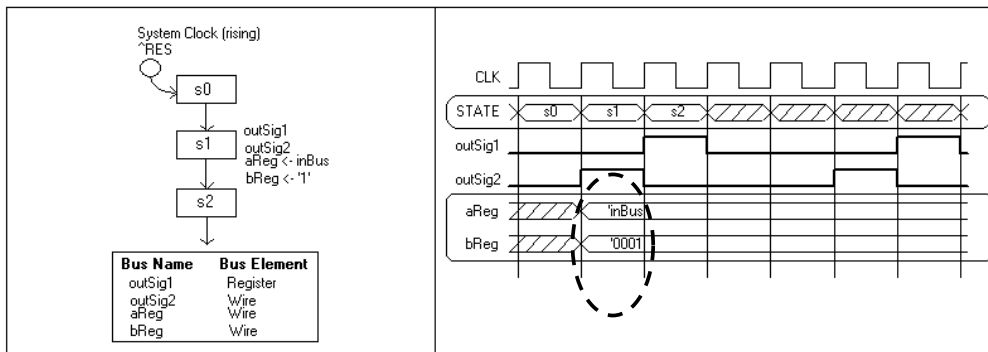


Moore Machine - Unregistered Bus Assignments

Unregistered bus assignments may be used by selecting bus Elements as “Wire” in flowHDL’s Bus Table. This implies that the buses in the datapath will not be realized using registers, but with wires.

Unregistered datapath operations, though causing the datapath buses to be realized without registers, are still synchronized by the clock driving the Moore-style state register outputs.

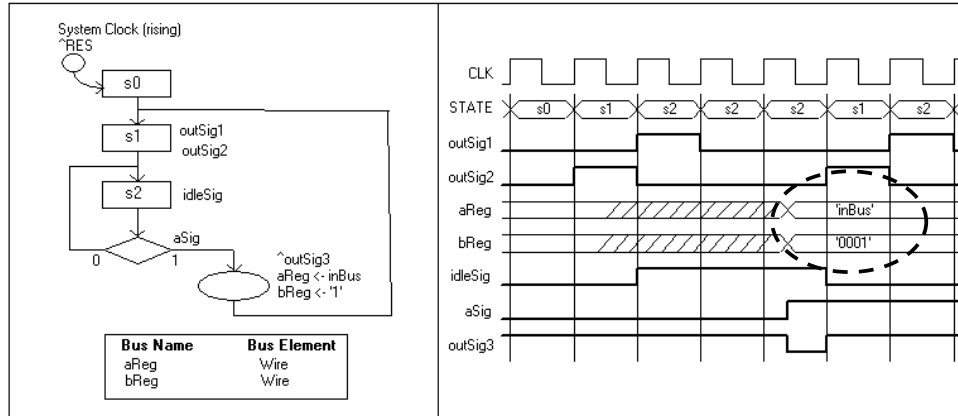
Using unregistered bus assignments reduces gate count of the datapath, and reduces circuit delay. However, special care must be taken to avoid race conditions, signal transients, and unwanted feedback loops in the datapath that can cause “metastability” (not settling to a specific value) or oscillation.



Mealy Machine - Unregistered Bus Assignments

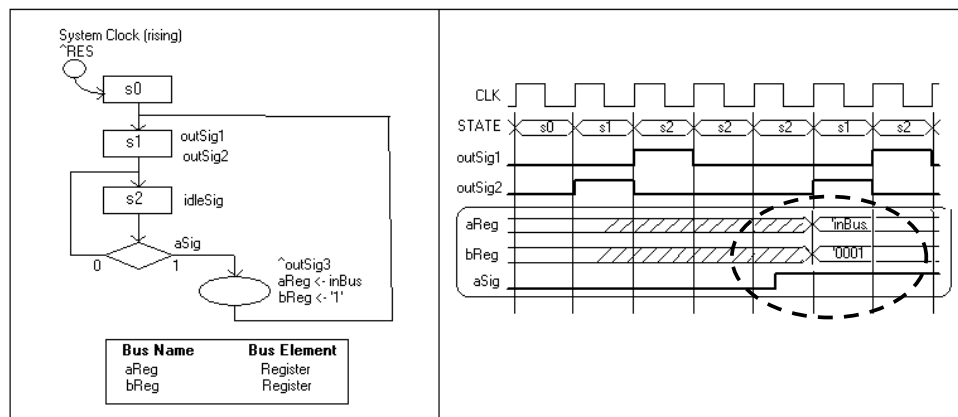
For Mealy-style outputs specifying datapath Bus assignments, the control outputs change asynchronously with the value of input aSig.

The use of Bus assignments with Mealy outputs implies that the datapath buses with be unregistered (either latches or wires, depending on Element value specified in the Bus Table). Thus the result of the assignment will appear on the output of the datapath immediately (assuming no propagation delay, since it is assumed this is subsumed by the clocking around the other elements of the design unit).



Mealy Machine - Registered Bus Assignments

Specifying registered assignments of buses used in Mealy outputs is done by selecting "Register" as bus Element type in Bus Table. The values to buses aReg and bReg are assigned synchronous to the next active clock edge.



Moore Machine - Macro-function Assignments

<p style="text-align: center;">Bus Name Bus Element</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>aReg</td><td>Wire</td></tr> <tr><td>bReg</td><td>Wire</td></tr> </table>	aReg	Wire	bReg	Wire	<p style="text-align: center;">Bus Name Bus Element</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>aReg</td><td>Register</td></tr> <tr><td>bReg</td><td>Register</td></tr> </table>	aReg	Register	bReg	Register
aReg	Wire								
bReg	Wire								
aReg	Register								
bReg	Register								
<p>Unregistered Output: To specify unregistered datapath for assignment of macros to buses, select Latch or Wire as Element in Bus Table.</p> <p>Be careful not to use unregistered macros in a feedback loop in the datapath, as this causes metastability in the resultant circuit.</p>	<p>Registered Output: To specify that macro assignments in the datapath are to be registered, specify as "Register" Element in Bus Table.</p> <p>Note: the resultant output of datapath operation is delayed one clock cycle after the operation is scheduled by the state machine, because of register delay inserted in the circuit.</p>								



Mealy Machine - Macro-function Assignments

<p style="text-align: center;">Bus Name Bus Element</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>aReg</td><td>Wire</td></tr> <tr><td>bReg</td><td>Wire</td></tr> </table>	aReg	Wire	bReg	Wire	<p style="text-align: center;">Bus Name Bus Element</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>aReg</td><td>Register</td></tr> <tr><td>bReg</td><td>Register</td></tr> </table>	aReg	Register	bReg	Register
aReg	Wire								
bReg	Wire								
aReg	Register								
bReg	Register								
<p>Unregistered Output: Using macro assignments on Mealy outputs can be specified as using unregistered outputs in datapath by selecting Bus Table Element as "Wire"..</p> <p>Be careful not to use unregistered macros in a feedback loop in the datapath, as this causes unwanted feedback in the resultant circuit.</p>	<p>Registered Output: Using macro assignments on Mealy outputs can be buffered using registered assignments by selecting "Register" choice for Element in Bus Table.</p> <p>You can define a feedback path through datapath if you buffer with registered bus.</p>								



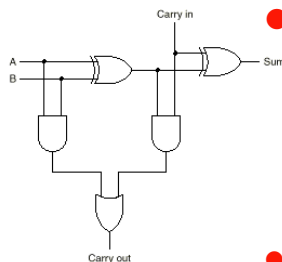
Avoiding Unwanted Feedback in the Datapath

<p>If expression or macro-function operators are specified as unregistered outputs in the ASM chart, the values seen on the output of the corresponding data path element can change values asynchronously with its inputs. If there is a feedback path to an input, an unwanted loop can be created, causing instability, metastability or even oscillation.</p>	<p>The designer should use an intermediate bus to hold the result of the asynchronous operation, or should specify a latch to hold the value, with the latch enable set in a different state, or should perform the operation synchronously with register logic.</p>



Modeling Combinational Logic in ASM

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Modeling styles

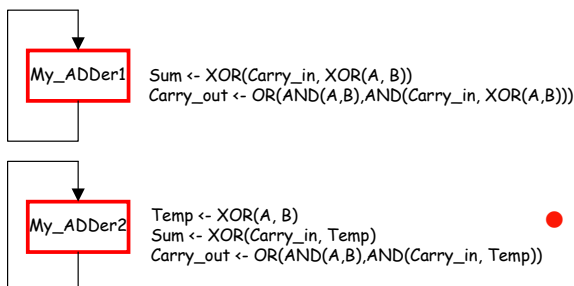
✓ There are two ways to model nested computation of data path: (1) nesting computations following the structure of the dataflow; (2) explicitly defining "temp" signals to hold intermediate results, and having each assignment as a separate macro statement.

Implicit modeling of data computation

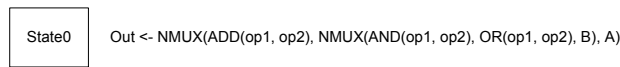
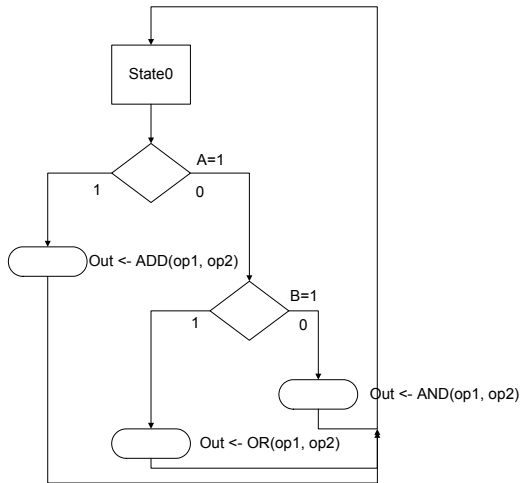
✓ This is used to embed the intermediate computations and simply generate the final result. Using this style assumes that the computation can be completed within the timing constraints of the logic and clocking scheme.

Explicit modeling of data computation

✓ This is used when representing the operation's intermediate results is of interest. Often it is easier to debug the design, and also to modify the timing behavior by changing an intermediate bus from "wire" to register.



Modeling Combinational Logic in ASM



• Modeling styles

- ✓ There are two ways to model behavior: (1) explicitly using ASM control constructs; (2) implicitly using nested macro-functions.

• Explicit modeling of control

- ✓ This is used to make clear the nature of decoding logic, based on combinations of control or data inputs.

• Implicit modeling of control

- ✓ This is used when representing the operation selection through a MUX-based scheme, using the Multiplexer macro-functions.

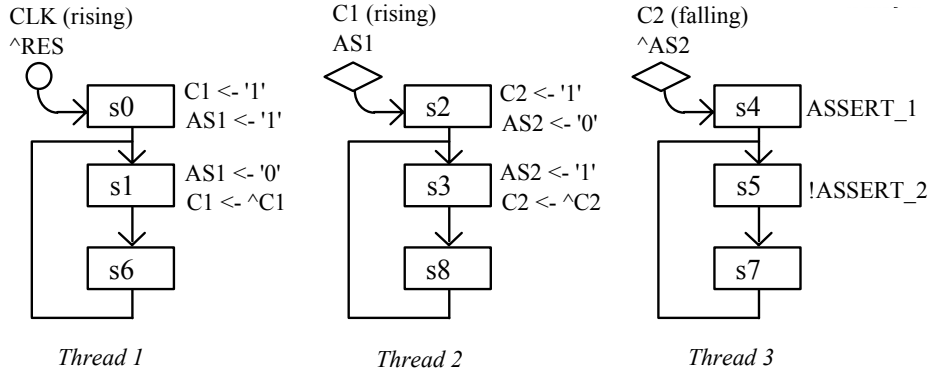


III. ASM Design Notation:

Modeling Concurrent Components in Digital Systems



ASM - Modeling Thread-level Concurrency



Modeling Concurrency:

- Multiple model ASM "threads" having shared buses.
- Independent clocking schemes and enabling events (e.g., ^RES).
- Types of concurrent interaction:

I. Synchronization

- coordinated activities (e.g., handshaking, pipelining).
- implicit references to shared buses.

II. Competition

- shared resources (for example, bus arbitration).
- explicit use of other concurrent processes, components, or entities to model the arbitration protocol.

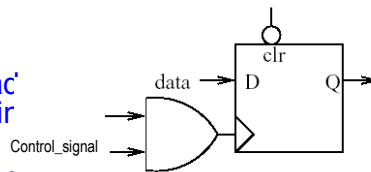


ASM- Control Exerted Across Threads

Figures: Lee © 2000 Prentice-Hall Publishers, Inc.

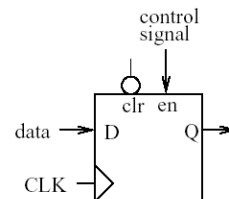
• Gated/User-defined clocking:

- ✓ We use some combination of signals to derive the clock signal for an ASM thread governing its state transitions and clockir of its data path registers.
- ✓ Nimbus only lets you specify a single use defined signal on the clock.



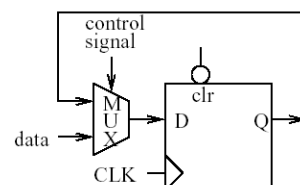
• Gated/User-defined resets:

- ✓ We use some control signal to reset the registers of a control block and its data path, either from the data path or from another controller.



• MUXed data path inputs:

- ✓ We use control signals to "gate" the data into a register or data path stage by "implying" a MUX by the specification of an expression assignment to a particular bus in one or more states.



ASM - Synchronous Enabling Events

Enable Events are used in specifying pre-emptive control behaviors of the design, where the normal control flow in the design is interrupted. Enable Events can either be synchronous or asynchronous.

<p>Synchronous Enable Events: Signal events that are capable of interrupting the control flow, but are synchronized to the clock. They have priority over "next state" values generated in next state decoding logic of state machine. Defines priority to pre-empt the next state value. The new "next state" generated from synchronous enable event is sampled on the next active clock edge to state registers, just as for normally encoded next state values.</p>	<p>Asynchronous Enable Events: Events capable of interrupting the control flow of the state machine block immediately, without synchronizing to the next clock edge. Their use is limited by the underlying device logic. Most design libraries use either D or JK flip flops for state registers. The only valid asynchronous enable event is the Reset.</p>

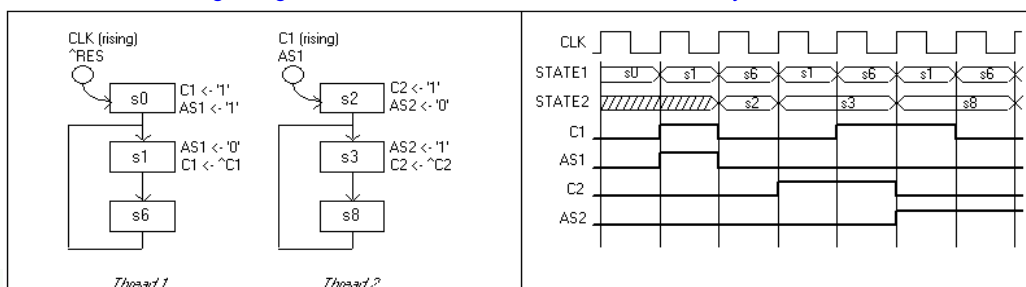


ASM - Modeling Synchronous Events

Using ASM, we can model special signaling situations, where a design component can be enabled by the presence of an "enabling" signal event. Rather than just defining a strobe for setting up a particular sequence of operations, an "enabling event" defines when a particular ASM "thread" is to be executed.

When the specified event is "activated" by the signal attaining the defined value, the control flow transitions to the state with the event attached to it. Control remains in that specified state while the event is "active".

Similar to the Reset pulse, an Enable Event is a pulse that causes a synchronous change of state, on the next active clock edge, to the specified state. The pulse can be active for any length of time. As long as the pulse is active, the ASM does not change to a new state. Once the defined enable event is "deactivated", the design will follow the normal state transitions. Single event triggers can be defined, if the signaling event is driven inactive on the next clock cycle.



IV. Modeling Problems:

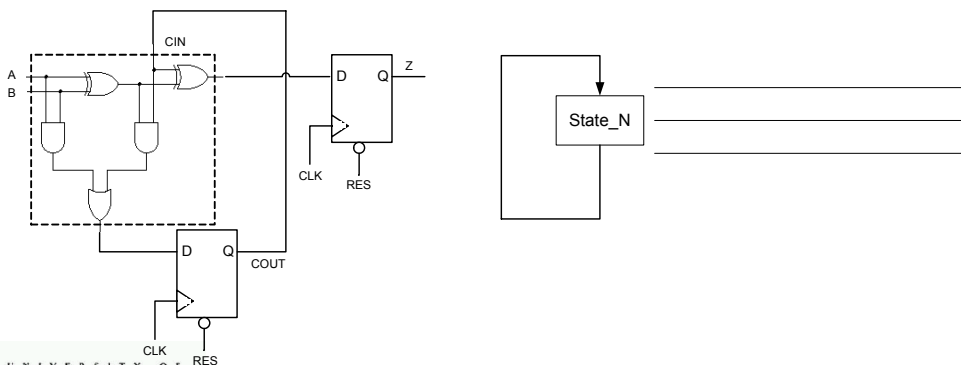
Modeling Combinational Logic Circuits using ASM



© 2002 Dr. James P. Davis Page 35

In-Class Exercise – Bit-serial Adder

- Data path circuit schematic for a bit-serial Adder architecture.
- Indicate the appropriate “nested” macro-function and assignment sequence on the ASM state that captures this data path operation.
- Hint: you should have three different assignment statements.
- Macro-functions that might be relevant include $AND(x, y)$, $OR(x, y)$, $XOR(x, y)$, remembering that macro input buses and outputs in the assignment of the macro assignment expressions must agree in width and data type.



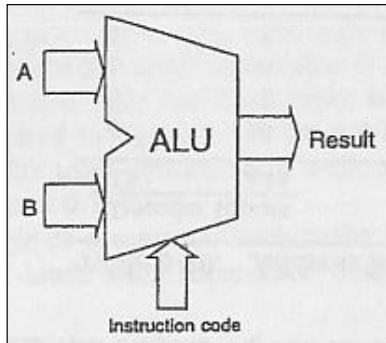
© 2002 Dr. James P. Davis Page 36

The Arithmetic Logic Unit

MacKenzie © 1995 Prentice Hall Publishing

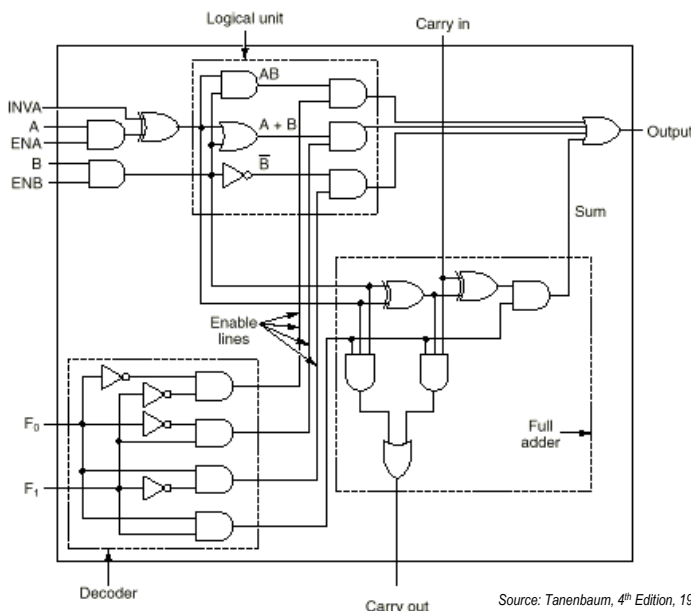
- The Arithmetic Logic Unit (ALU)

- ✓ The ALU provides many functions that are selectable using control signals.
- ✓ The ALU has an arithmetic circuit, a logic circuit, and decoding logic to determine which function to select.
- ✓ Only one ALU function can be selected at a time.
- ✓ ALUs are integral component of a CPU.



© 2002 Dr. James P. Davis Page 37

The Arithmetic Logic Unit



- 1-bit Arithmetic Logic Unit (ALU)

- ✓ The ALU function is provided as a single-bit operation, built up of gate-level combinational logic.
- ✓ The multi-bit ALU is created as a result of combining many single-bit ALUs in a cascaded configuration.
- ✓ This ALU can be analyzed by first constructing a truth table to obtain the Boolean equations.

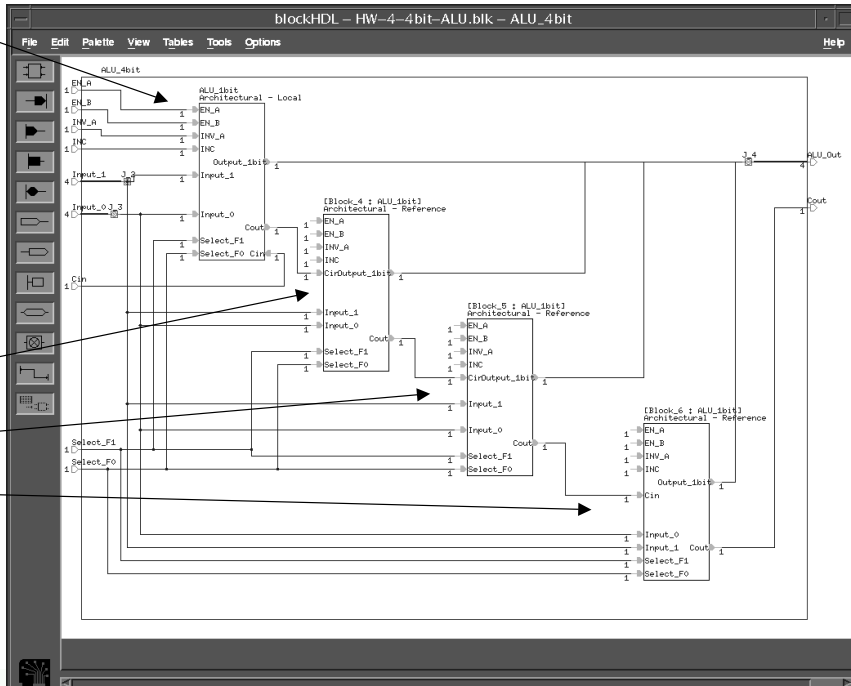


© 2002 Dr. James P. Davis Page 38

The ALU Model – Block Diagram

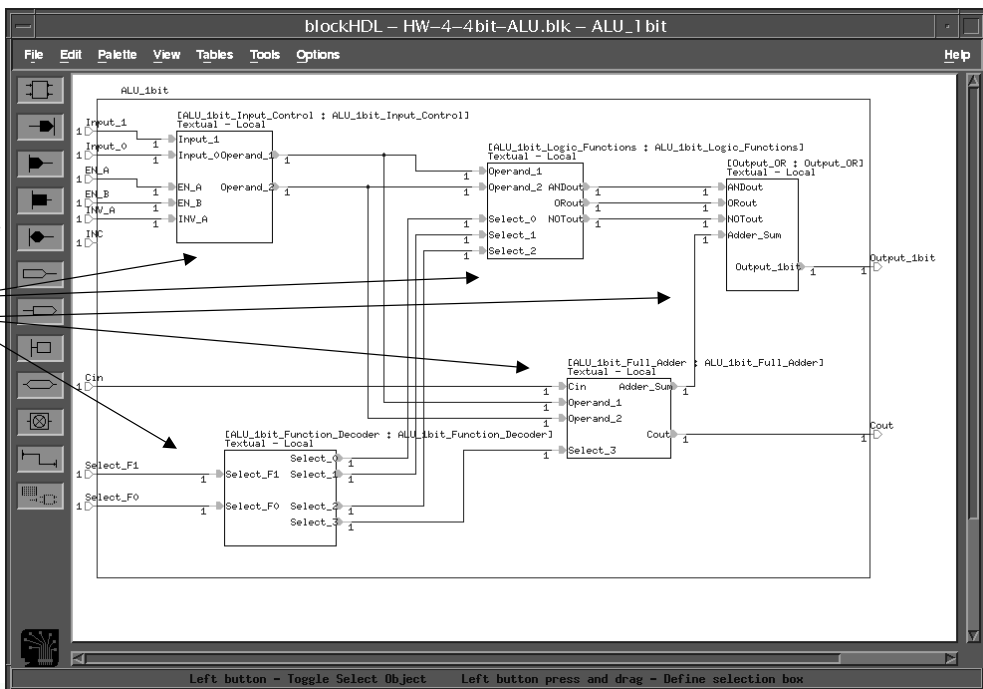
This model is created in terms of its hierarchy and behavior (see its decomposition).

These models are simply instances of the original model, with appropriate net connections (such as from Cout to Cin).



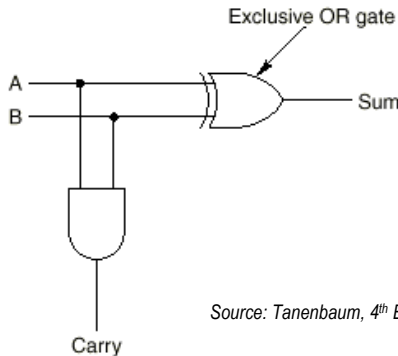
The ALU Model – Block Diagram

These blocks are leaf nodes in the block hierarchy, some of whose function is trivial.



ALU Components – The Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Source: Tanenbaum, 4th Edition, 1999.



• Half Adder circuit:

- ✓ This circuit takes two single-bit inputs and adds them together to produce a Sum as output.
- ✓ The adder also generates a Carry signal, which can be used to connect to another adder element.
- ✓ Multiple adders are connected together to implement a multi-bit adder circuit, and more complex arithmetic functions.

✓ Using the ASM macro-functions:

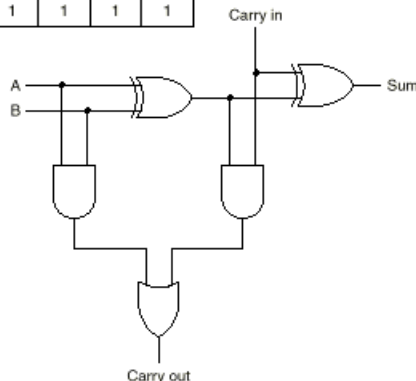
- ✓ Sum[1:0] <- ADD(A,B)
- ✓ Sum[0] <- ADDNC(A,B)
- ✓ The first ADD() assumes a carry_out signal will be stored in MSB.

© 2002 Dr. James P. Davis Page 41

ALU Components – The Adder

Source: Tanenbaum, 4th Edition, 1999.

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



• Full Adder circuit:

- ✓ This circuit takes two single-bit inputs and adds them together to produce a Sum as output.
- ✓ The Full Adder also has a Carry (called a Carry Out) like the Half Adder.
- ✓ The Full Adder also has a Carry In signal, allowing Carry Out from earlier adder stage to be connected.
- ✓ This allows a multi-bit, multi-stage adder circuit to be constructed.

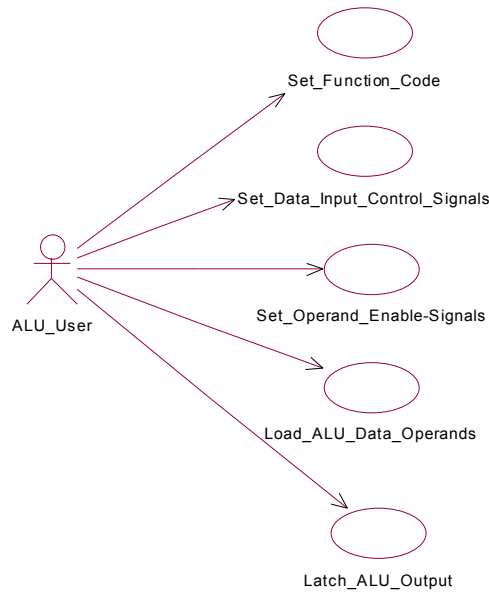
✓ Using the ASM macro-functions:

- ✓ Sum[1:0] <- ADD(A,B, Carry_in)
- ✓ Sum[0] <- ADDNC(A,B, Carry_in)
- ✓ The first ADD() assumes a carry_out signal will be stored in MSB of Sum[1:0].



© 2002 Dr. James P. Davis Page 42

The ALU Analysis Model – Use Cases



• Abstract ALU:

- ✓ This circuit takes two inputs and operates on them to produce an output.
- ✓ The output is determined by the selection of function codes, data input control, and operand enable signals.
- ✓ There is an implied ordering of tasks for setting up the ALU (not shown in the Use Case diagram): all the control signals must be stable before applying the data inputs; then the device is enabled.
- ✓ The ALU operates as a combinational logic device; so its output is latched into a register when the operation is complete.



The ALU Function Model – Truth Table

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

Inputs

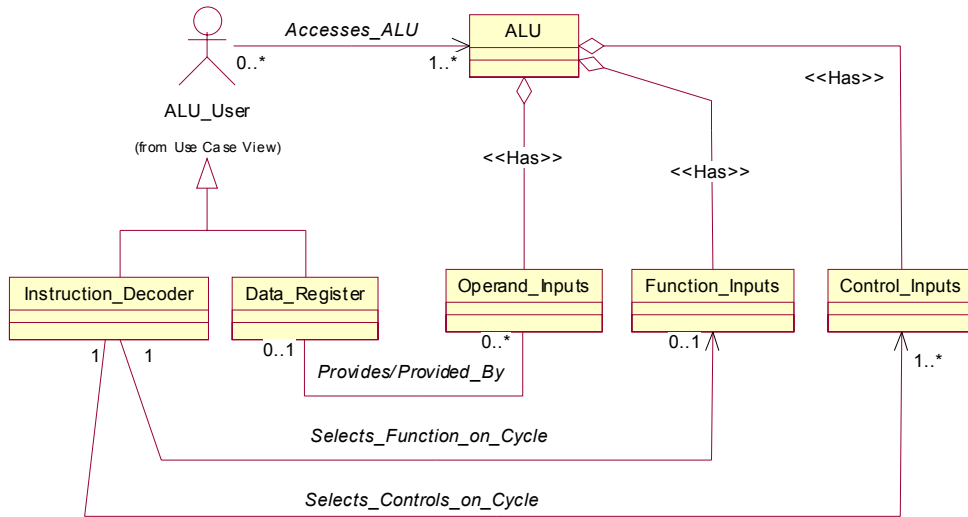
Output
Function

Functional specification:

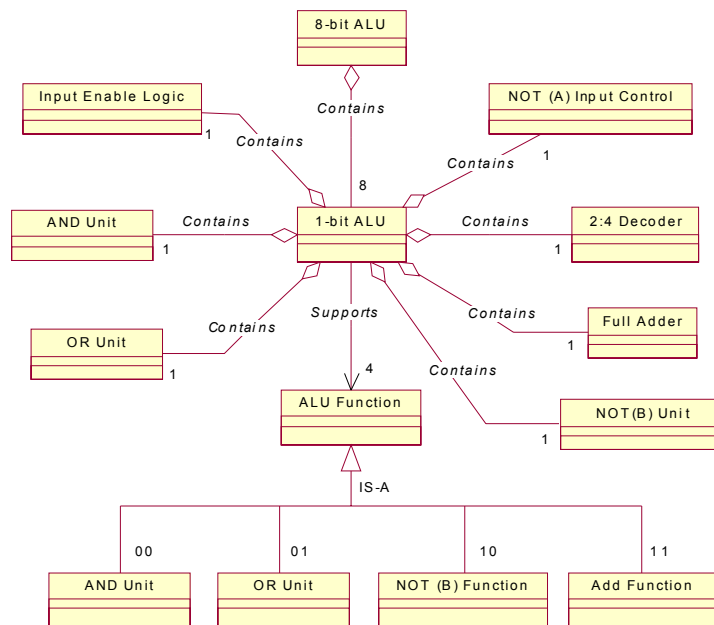
- ✓ This is a technique to represent a purely functional result.
- ✓ You list a column in the table for each input, and a column for each possible output combination.
- ✓ The table is used as a “look-up”, given some input combination, to determine the output.
- ✓ The outputs are a function of the inputs.
- ✓ Truth table results are derived based on rules of Boolean Logic, and complex functions can be built up from understanding more primitive ones.



The ALU Analysis Model – Class Diagram-1



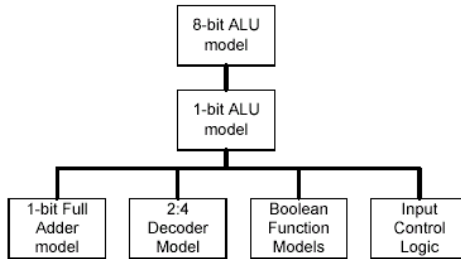
The ALU Analysis Model – Class Diagram-2



The ALU – Function versus Structure View

Source: Tanenbaum, 4th ed. © 1999, Prentice-Hall

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



Which form of specification?

- ✓ There are several choices for the level at which we'll model the control flow of the ALU.
- ✓ We could model it as one monolithic unit, where we simply implement all of the output combinations indicated in the truth table.
- ✓ Alternately, we could model the ALU hierarchically, as a collection of concurrent threads. Each thread implements a piece of the ALU (function decoding logic, adder, etc.).
- ✓ Which is easier to model?
- ✓ Which would be a more efficient design? (We'll have to generate a circuit to determine the answer.)
- ✓ We'll work both models and compare.

© 2002 Dr. James P. Davis Page 47

Homeworks #1 & 2 - Assignment

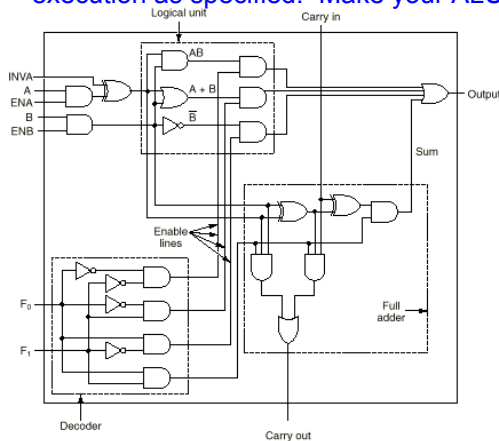
- Objective: To start some high-level circuit modeling to give you more insight on:
 - How to start a design model from an analysis model
 - How to think about digital systems modeling using ASM charts.
- Means:
 - ✓ The ALU example (taken from Tanenbaum, 4th edition, 2000).
- Result:
 - You should be able to create this design in Nimbus or flowHDL and verify its functionality through simulation.
 - You will generate a number of simulation test cases, indicating the "expected" behavior of the design. (Then, in Lab, we'll learn how to run the simulator to verify the correctness of the design functionality).



© 2002 Dr. James P. Davis Page 48

Assignment – 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.
 - ✓ Create an ASM thread that realizes the input decoding control and operation execution as specified. Make your ALU accept 4-bit operands.



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



Lab – Multi-function ALU

- Lab purpose:
 - ✓ This lab introduces the student to modeling *combinational logic circuits* using an algorithmic style, while also using the appropriate arithmetic and logic functions in the macro-function library.
- Lab discussion:
 - ✓ This lab problem can be modeled in two different ways using ASM notation:
 - ✓ (1) *algorithm style*, where the “decoding logic” is represented as a collection of nested If-Then conditionals and Case multi-way branch; and
 - ✓ (2) *function style*, as a set of nested macro-function expressions attached to the single state or as conditional “Mealy-style” outputs that are dependent on a particular combination of Function and control signal inputs.
 - ✓ You’ll want to use the macro-functions: **ADD(a,b)**, **AND(a,b)**, **OR(a,b)**, **NOT(a,b)**. For modeling the ALU as functional expressions, you might also use the NMUX(x,y,sel) macro for selecting among multiple ALU functions.
 - ✓ You will want to define a set of test cases to evaluate your model in the Nimbus simulator. Some things to think about:
 - ✓ (1) *data combinations*: which data values for inputs reasonably test circuit?
 - ✓ (2) *control combinations*: what is the precedence of the control inputs? Does the circuit behave as expected under different combinations of control inputs?
 - ✓ (3) *function inputs*: does the ALU circuit function as expected, for the supported functions?



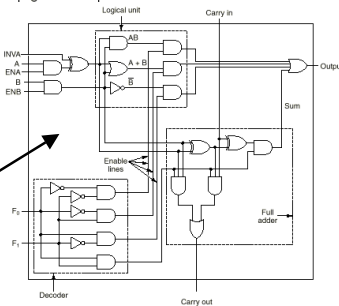
ALU Design – In-class Part

Source: A. Tanenbaum, 4th ed., 1999.

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	0

Functional description of "system" model (truth table).

Structural description of "system" model (gate schematic).



Part 1:

- ✓ We will take the basic specification for the ALU functions, as given by the truth table, and we'll discuss an ASM design model in class.

Part 2:

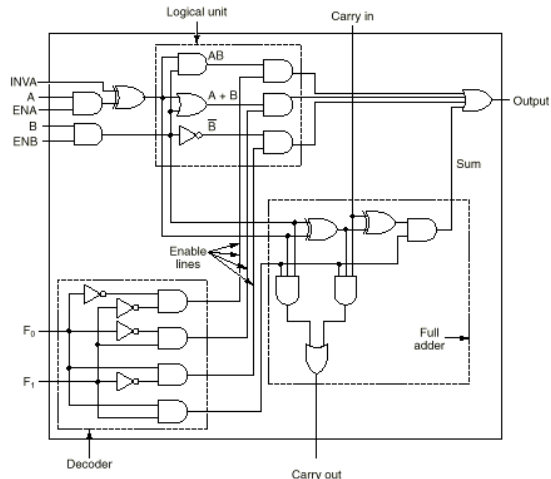
- ✓ You will use Nimbus to create your own version of the ALU model.
 - ✦ Define/declare signals and buses.
 - ✦ Create the flow diagram of the state machine.
 - ✦ Check the model and compile it into its internal form.
 - ✦ Simulate the model to verify the correctness of the design model. (You will come up with 5 different test cases as part of HWs #1 & 2).

© 2002 Dr. James P. Davis Page 51

Lab – Arithmetic Logic Unit

ALU structure

- ✓ A single-bit ALU consists of logic for (1) function decoding, (2) performing 3 logic functions AND, OR, NOT (B), (3) Full Adder unit, (4) Input enable logic, (5) NOT (A) function from separate input.
- ✓ Creating a multi-bit ALU unit involves scaling the bit-widths of the input operands A, B and the Output.
- ✓ We will start out by creating a single-bit ASM model for this circuit, then scaling it to 4 bits.
- ✓ There are two styles of ASM model that can be created: (1) a structural model mimicking the gate schematic, and (2) a functional model independent of the structure.



Source: Tanenbaum, 4th Edition, ©1999 Prentice Hall Publishers, Inc.



© 2002 Dr. James P. Davis Page 52

Lab - ALU Model (Continued)

- ALU Function:

- ✓ F1, F0 define selection of 4 functions: '00' AND, '01' OR, '10' NOT (input B), '11' ADD.
- ✓ These functions are “qualified” by other control signals input to the ALU: (1) A,B input Enable lines, (2) INVA – NOT (input A) function, (3) INC increment function (which has special meaning, depending on the status of the other control inputs).
- ✓ The outputs are defined in terms of the two inputs, but some outputs are dependent on a single input (if the control signals are configured appropriately).

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



Lab – Single-bit ALU

- You will use Nimbus™ or flowHDL® to create an ASM model of the design as described. The design entry consists of the following parts:
 - ✓ Declare the signals and buses in the Bus tree view dialog box. You'll need to declare all buses to have Element type = “wire”.
 - ✓ Note we are starting with a single-bit ALU unit in this part of the workshop.
 - ✓ Create a model for the ALU block, as discussed. A combinational circuit will consist of an ASM “thread” with a single state. Thus, the design consists of modeling the operations using the Nimbus macro-functions, nested together, or of modeling the decision logic as conditional output operations using the ASM control constructs (Case, Condition), all looping on the single state of the ALU thread.
 - ✓ Note: you are also including declarations for CLK and RES signals, although this is a combinational logic design unit, so they will not be referenced in the actual design.
 - ✓ Compile your model: If you get errors during model compilation, you should go to the state where errors are flagged (indicated by an error code), and click the object while holding down the ‘CTRL’ key (CTRL + Left Mouse). This will bring up and explanation of the error in a dialog box.
 - ✓ Usually, the cause of errors is that you are using signals in an assignment expression that have different bus widths, different signal types, or signal modes (input, internal, internal_out, versus output). So, you'll have to debug the design a bit. Nimbus uses strongly typed signal definitions.

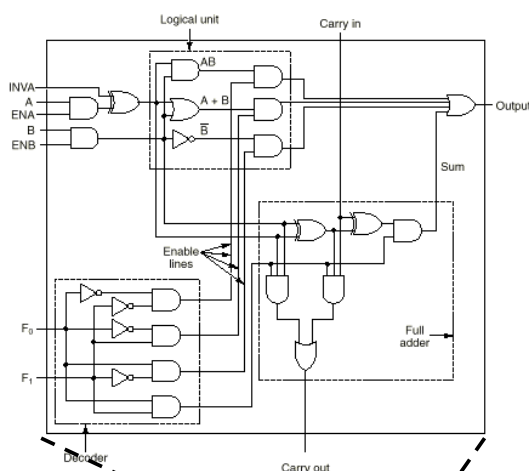


Lab – Scaling the ALU Model

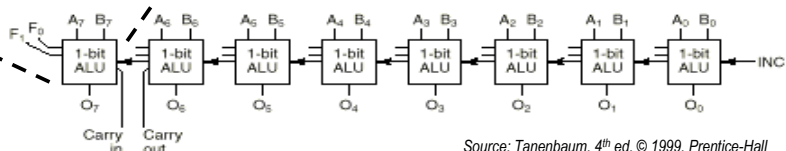
- For this part of the workshop, you will take the specification for the single-bit ALU (using the original truth table only), and devise an ASM model that meets the requirements of a combinational logic model, namely that the operation can be completed in a single cycle.
- In addition, the input buses A and B will be 4-bits rather than 1-bit in width. This means the output bus will also be 4-bits. (Note that the control inputs and the carry-in and carry-out signals will be similar to those in the first version of the ALU model.)
- You can use nested conditionals, case constructs and macro-functions to implement the arithmetic and logical functions required for generating the outputs in the presence of the inputs and controlling signals.



Lab - The Multi-bit Arithmetic Logic Unit



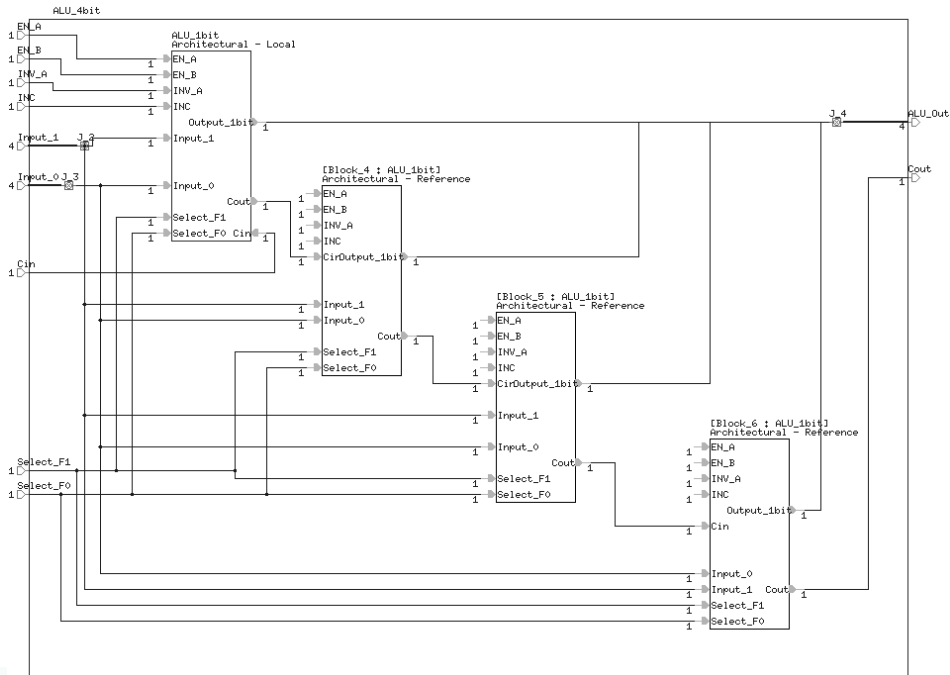
- ALU description
 - ✓ Provides the arithmetic, logic and other data functions in a single package.
 - ✓ ALU functions are selectable using control signals.
 - ✓ Individual gate-level elements are cascaded together to form an ALU of the computer's word length.
 - ✓ NOTE: The ALU incorporates the Full Adder model.



Source: Tanenbaum, 4th ed. © 1999, Prentice-Hall



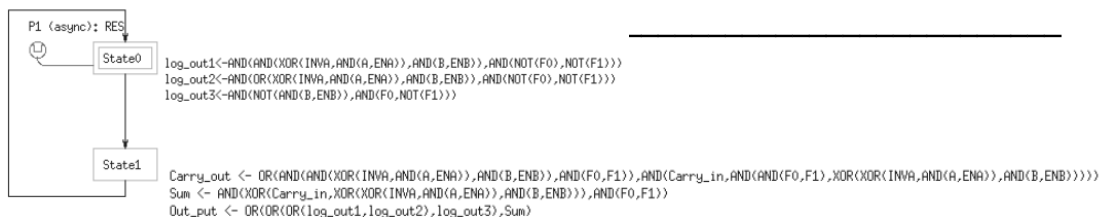
Lab – Structural 4-bit ALU



Lab – Alternate ALU Model

- Some questions about the ALU model shown below:
 - ✓ This model has two states, and has operations scheduled on each state. Is there any problem with this design, in regards to its specification in the truth table shown earlier?

- ✓ If you see a problem with how the design block has been modeled, how would you fix the problem (i.e., how would you model the ALU differently so that the problem goes away)?



Laboratory ASM Worksheet

ALU_version_1

- Use this worksheet to hand-draw your initial ASM design.
- The single-thread ASM model should only have one state, as the circuit to be modeled is a combinational logic circuit.
- Can you model this version using a set of nested ASM control structures (If-Then and Case constructs)?



© 2002 Dr. James P. Davis Page 61

HW #3 – Testing Computation of the ALU

- For this assignment, you will take your 4-bit ALU and create a second ASM “test thread” that uses the ALU thread to implement a Subtraction function for signed integers.
- Subtraction is carried out as follows: (1) convert the second operand into two’s complement (invert and add 1), (2) perform addition of the operands, and (3) convert sum back from two’s complement (depending on the value of the “sign” bit).
- This will require consecutive operations using the ALU: (1) NOT(B), (2) B<-INCRNC(B), (3) ADD(A, B), (4) test of MSB (sign bit).

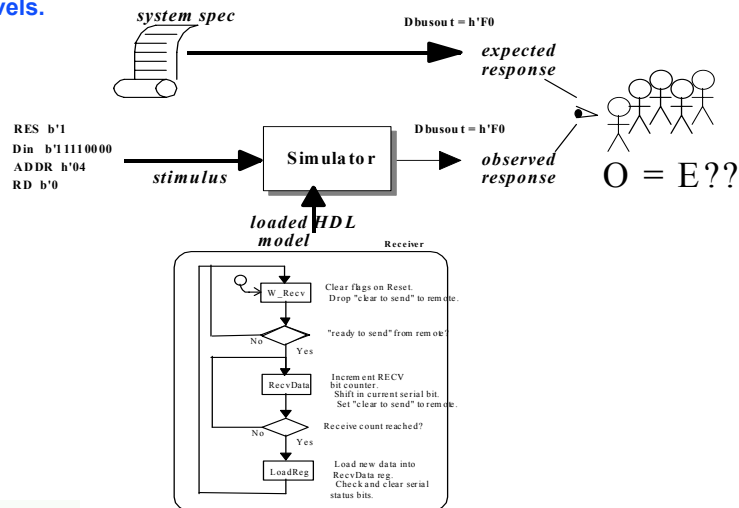


© 2002 Dr. James P. Davis Page 62

Debug Test & Verification Process

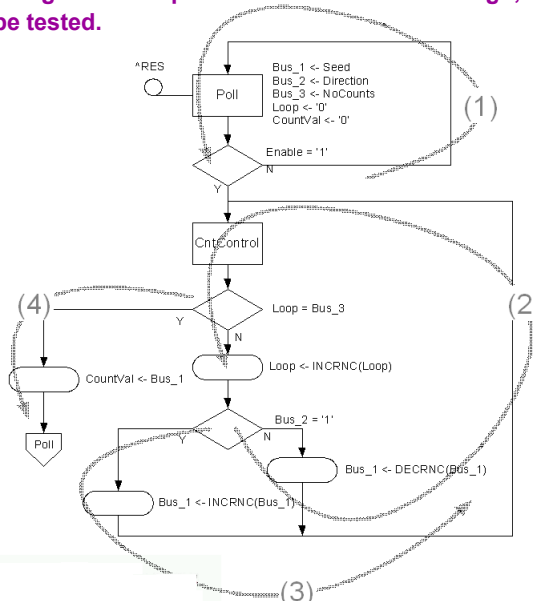
- Approach to Verification

- ✓ 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.



Debug Test & Verification Process

- 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.



