

CSCE 491

Capstone Computer Design Project

2005/8/31

Week 2

Digital Systems Design Methods-1

© 2003 Dr. James P. Davis
Some figures from Tanenbaum, 4th ed., ©1999 Prentice-Hall.

Week 2 - Outline

- Digital Systems Design Methods.
 - ✓ Structural – decomposition, refinement
 - ✓ Functional – input/output response specification.
 - ✓ Behavioral/ASM – sequencing and scheduling of operations in the data path.
 - ✓ Behavioral/Timing – input/output response over some time frame.
- Clocking specification.
 - ✓ Use waveform specification as a means for defining characteristic system behavior, based on arrival of input signals and the resultant response of the system, and its internal, intermediate actions.
- ASM Diagrams.
 - ✓ We can also start with a truth table to define the equations for the outputs based on value combinations of the inputs.
 - ✓ If we include all inputs—both data and control—then the truth tables can become quite large.
 - ✓ However, what we are looking to identify from the equations is the Sum of Products (SoP) form, that can be used to identify which operations are to be scheduled in which states of the state machine (should one be required).
 - ✓ Note: we use truth tables for combinational logic function specification, but sometimes these combinational logic functions are under the sequencing control of a state machine.

Week 2 - Objective

- Objective: To gain comprehensive enough understanding of Algorithmic State Machine notation in order to use it to create digital circuit and systems architecture.
- Means:
 - ✓ Modeling the *control path* as a Finite State Machine (FSM).
 - ✓ Modeling the *data path* operations using the Register Transfer Notation (RTN).
 - ✓ Applying the underlying assumptions about timing behavior based on whether we use *registered* or *non-registered* elements in the data path. Registered elements have a one-cycle delay from when they are scheduled in the FSM and when the result appears on the output of data path registers.
 - ✓ Assuming “unit delay” in the scheduling of data path operations from the control path of the state machine.
- Result: Construct architectures easily from abstract system models described in UML, where these models are “cycle accurate”, but not necessarily accurate in terms of a set of specific VLSI circuit delays.



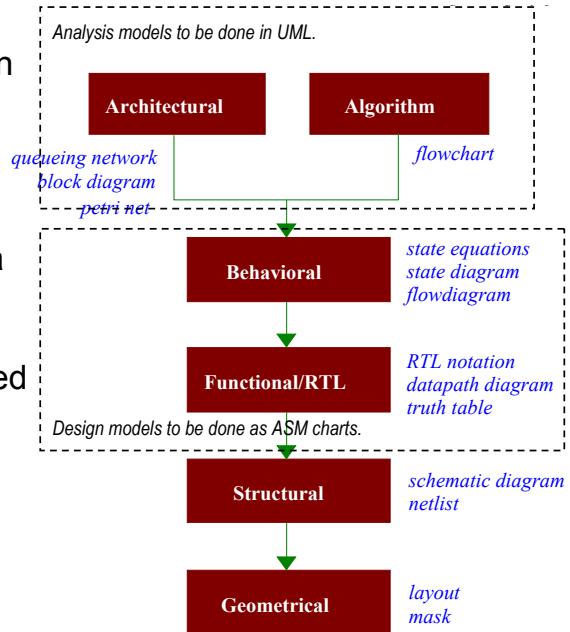
I. Overview:

Review of Logic Concepts and Basic Device Functions



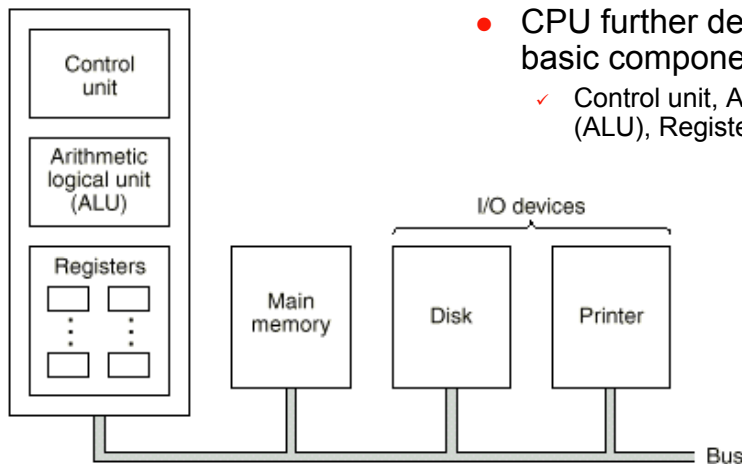
Levels of Abstraction in VLSI System Design

- A design transforms from "concept" to "implementation" in a series of ordered levels.
- From the highest level to lower levels of design "abstraction", a design is iteratively refined.
- The design description is verified and validated at each level, often cycling between levels of abstraction.
- Design descriptions are described using one or more domain representations (Behavior, Structure, Physical).



Systems Design – Structure of Computers

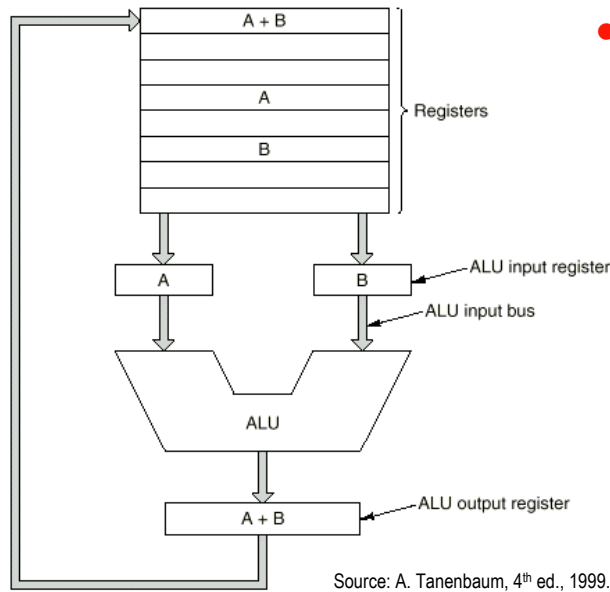
Central processing unit (CPU)



- Basic systems components of Computer:
 - ✓ CPU, Memory, I/O Devices, Bus
- CPU further decomposes into basic components:
 - ✓ Control unit, Arithmetic Logic Unit (ALU), Registers



Systems Design – Structure of Computers



• Von Neumann computer architecture:

- ✓ Developed in the late 1940s by John Von Neumann (Princeton U.)
- ✓ Combined the elements of *stored program machine*: one machine could run many programs (CSCE 212).
- ✓ Program instructions stored in memory and fetched, in sequence, to be executed by CPU.
- ✓ Results of execution stored in Registers, later written back to memory.
- ✓ The Motorola 68000 follows this style of architecture (CSCE 313).



Review of Boolean Logic Functions

2^q functions of q Boolean variables $\Rightarrow (q = 2 \Rightarrow |F| = 16; F = \{f_0, f_1, \dots, f_{15}\})$

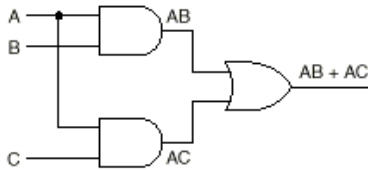
x_2	x_1	$f_0(x)$	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$	$f_5(x)$	$f_6(x)$	$f_7(x)$	$f_8(x)$	$f_9(x)$	$f_{10}(x)$	$f_{11}(x)$	$f_{12}(x)$	$f_{13}(x)$	$f_{14}(x)$	$f_{15}(x)$
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

$AND(x_1, x_2)$ $PASS(x_1)$ $PASS(x_2)$ $NOR(x_1, x_2)$ $NOT(x_1)$ $NOT(x_2)$ $NAND(x_1, x_2)$
 $AND(NOT(x_1), x_2)$ $AND(x_1, NOT(x_2))$ $XOR(x_1, x_2)$ $OR(x_1, x_2)$ $XNOR(x_1, x_2)$ $OR(NOT(x_1), x_2)$ $OR(x_1, NOT(x_2))$ $Tautology(x_1, x_2)$



Review of Boolean Representation

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



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

$$f(A,B,C) = AB + AC$$



© 2003 Dr. James P. Davis Page 9

• Boolean algebra formulation:

- These laws are used to transform Boolean equations into appropriate forms.
- They were formulated in the 1930's by mathematician George Boole.
- There is an equivalence between a Boolean expression, a Truth table and a Gate-level diagram representation. They are all used to convey different forms of the same information.
- We start with a mathematical formulation of the desired function, and we want an efficient circuit to realize this function.
- We apply a set of “well formed” logical operators that are mathematically “sound” and “complete”.

Review of Boolean Representation

• Laws and Theorems of Boolean algebra:

- ✓ Used to transform Boolean equations into appropriate forms.
- ✓ Used to simplify or expand Boolean expressions, depending on the form of the expression.

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

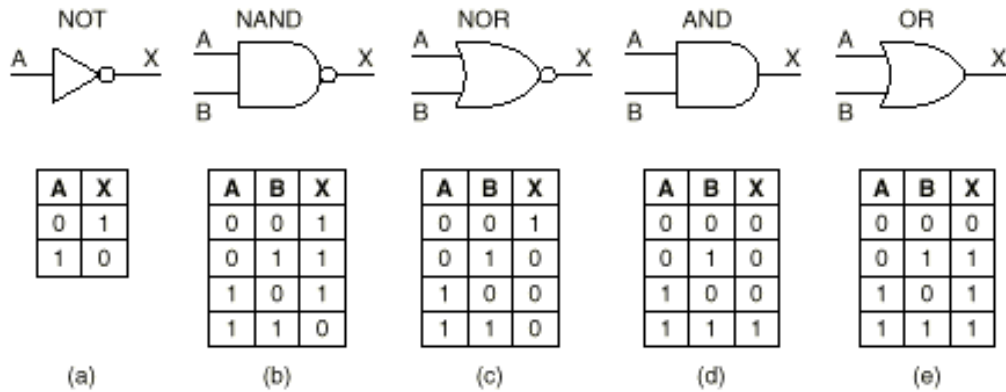
Source: Tanenbaum, 4th ed. © 1999, Prentice-Hall



© 2003 Dr. James P. Davis Page 10

Review of Gate-level Devices

- Symbols and Truth tables for 5 basic gate-level logic gates: NOT, NAND, NOR, AND, OR.



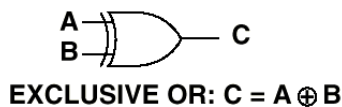
Source: Tanenbaum, 4th ed. © 1999, Prentice-Hall



Review of Gate-level Devices

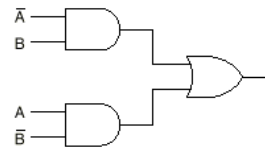
• XOR Logic Function

- ✓ There is generally no XOR gate in a technology library. It is realized using other gate structures.
- ✓ The Truth table for XOR and 3 different circuits that realize the function for two input signals.

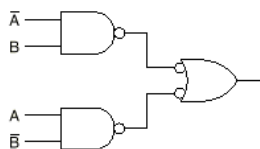


A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

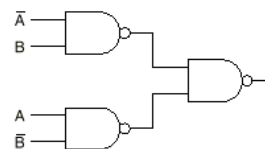
(a)



(b)



(c)



(d)

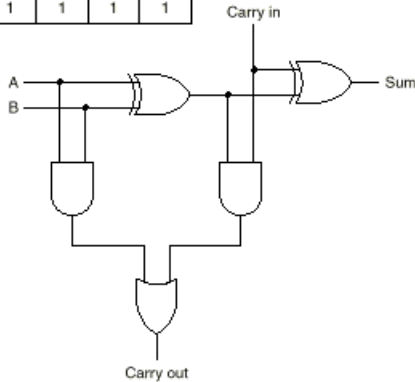
Source: Tanenbaum, 4th ed. © 1999, Prentice-Hall



Review of Gate-level Devices

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

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



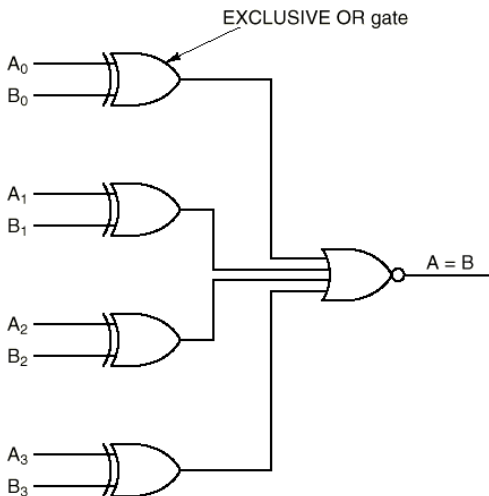
• Full Adder circuit:

- ✓ 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.

© 2003 Dr. James P. Davis Page 13

Review of Gate-level Devices

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



• Comparator (EQ):

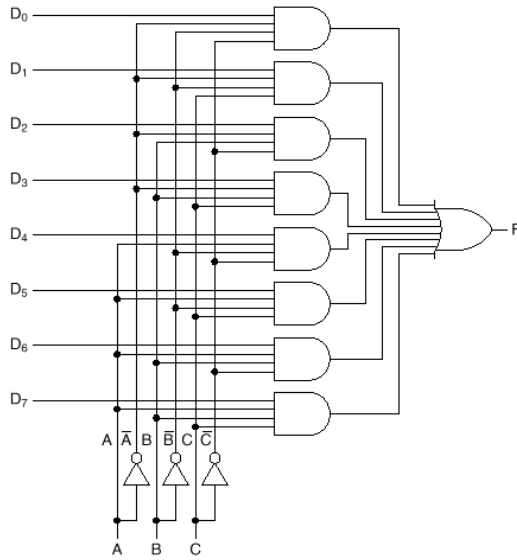
- ✓ COMP allows comparison of two different inputs to check if they are equal. Alternate circuits evaluate the relative magnitude of two inputs.
- ✓ If they are equal, the output is HIGH, but if they are not equal, the output is LOW.
- ✓ The comparison must be done with two signal inputs of equal width.
- ✓ The output of the Comparator operation is a single-bit signal.



© 2003 Dr. James P. Davis Page 14

Review of Gate-level Devices

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



• 8-input Multiplexer (MUX):

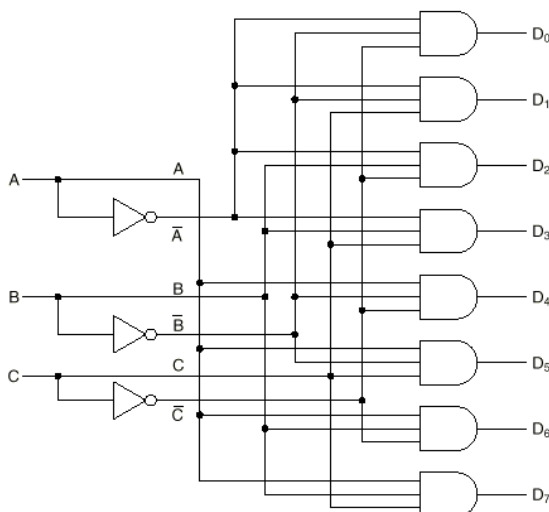
- ✓ MUX allows the signal value of one of its data inputs (D0 – D7) to pass to the output F.
- ✓ The selection of the signal to be passed is controlled by SELECT lines (A, B, C).
- ✓ The number of select lines, n , is based on a power of 2 for the number of inputs, m . So, if we have m inputs, we'll need n select lines so that $2^n = m$.
- ✓ The MUX inputs and output must be the same width, and the SELECT lines are 1-bit each.



© 2003 Dr. James P. Davis Page 15

Review of Gate-level Devices

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



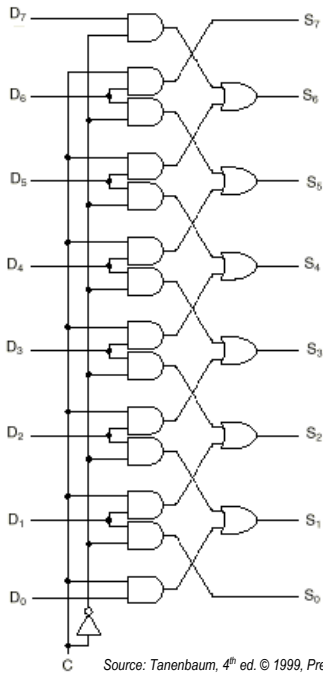
• 3:8 Decoder (n to 2^n DECO):

- ✓ DECO takes a binary encoded input of n data bits, and decodes it into individual data output lines, where one output (D0 – D7) is enabled, depending on whether the encoded value corresponds to the data line number.
- ✓ A Decoder input with n lines means we can encode 2^n possible binary encoded values.
- ✓ With 2^n possible encoded values on the input, we'll need exactly n output lines, one for each possible encoded input value.
- ✓ The output line corresponding to the decoded value will be enabled.



© 2003 Dr. James P. Davis Page 16

Review of Gate-level Devices

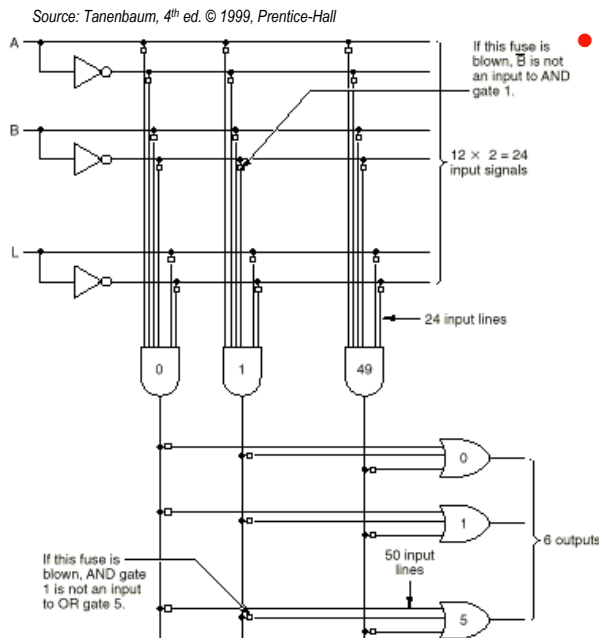


• 1-bit Left-Right Shifter (LSHFT, RSHFT):

- ✓ The n -bit input signal ($D_0 - D_7$) is shifted either Left or Right by one bit.
- ✓ The shifted value is propagated to the output ($S_0 - S_7$).
- ✓ The bit-width of the input must equal the bit-width of the output.
- ✓ If we shift Left (downward in the figure), the value on input signal D_0 is lost.
- ✓ If we shift Right (upward in the figure), the value on input D_7 is lost.



Programmable Logic Devices - PLA



• Programmable Logic Array (PLA):

- ✓ PLA is a structure with an AND and OR array of gates.
- ✓ The fuses allow specific connections (signal paths) to be burned so that a signal can pass through the AND or the OR part of the array.
- ✓ The PLA is used to implement custom logic functions using a standard circuit package.
- ✓ The fuses are arranged in two matrix structures: top one for AND, bottom one for OR.

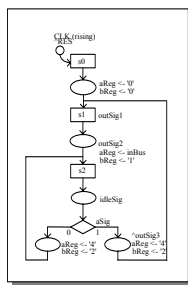


II. Overview:

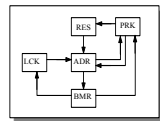
Digital Systems Modeling & Design



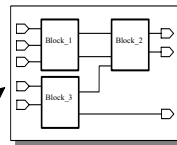
Summary - Specifying a Digital System



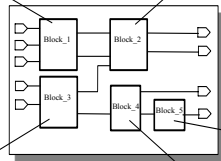
Concurrent Flowdiagrams



State Diagrams

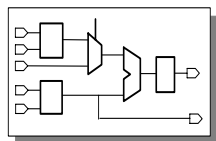


Hierarchical Block Diagrams

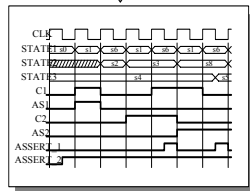


Truth Tables

A	B	C	D	f(A,B,C,D)
00	00	00	11	
00	00	01	10	
00	01	00	01	
00	00	10	10	
00	10	00	01	



Hierarchical Datapath Diagrams



Timing Diagrams

```
entity DRAM_Controller is
port (
CLK      : in bit;
RES      : in bit;
WORD_IN  : in bit_vector (7 downto 0)
);
end DRAM_Controller;
architecture ARTIFACT of DRAM_Controller is
signal CYCLES_DONE : in bit;

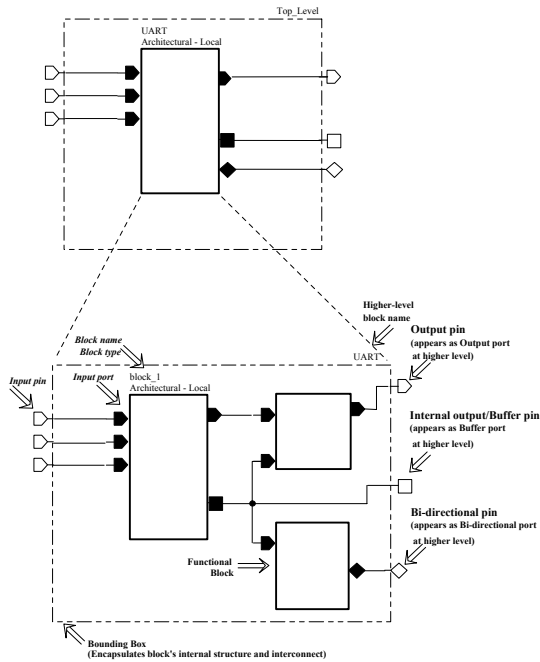
```

VHDL & Verilog

- Design structure
 - ✓ Hierarchical block diagrams
 - ✓ HDL code
- Design function
 - ✓ Data path diagrams
 - ✓ Truth tables
 - ✓ HDL code
- Design timing
 - ✓ Timing diagrams
- Design behavior
 - ✓ State diagrams
 - ✓ ASM/flow diagrams



Systems Design - Structure

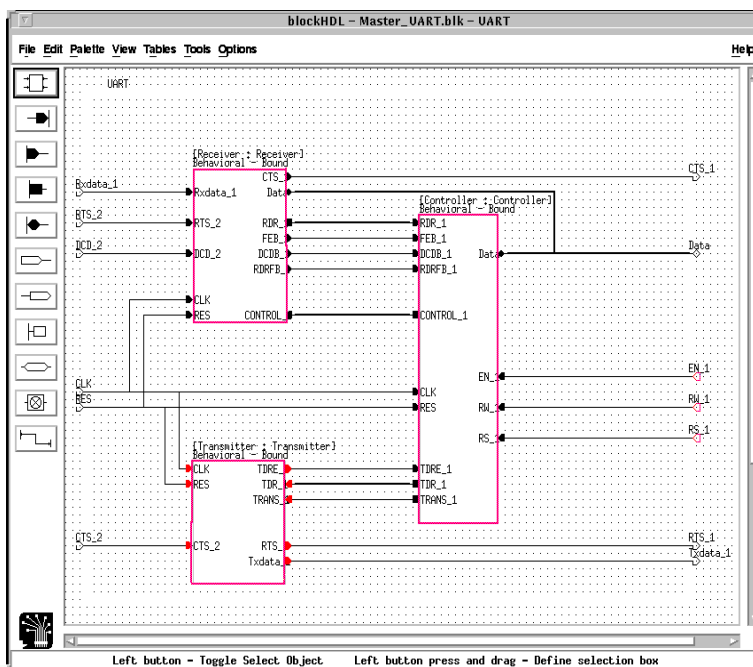


● **Block diagram:** Used to partition and decompose a design into its functional units.

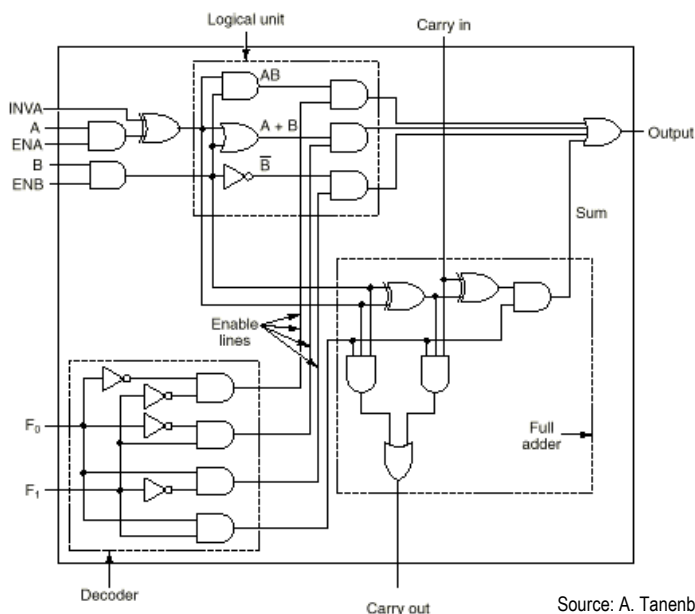
- ✓ Step #1: Identify "top level" and all functional operations that transform the application's data.
- ✓ Step #2: Group the functions by how the data is manipulated, or by what resources are needed.
- ✓ Step #3: Separate different functions from each other by "interconnect" that shows the flow of data.
- ✓ Step #4: Decompose more "abstract" functions into more "primitive" sets of functions that operate on data.
- ✓ Step #5: Repeat Step #4 until all the functions are fully decomposed into a hierarchy of "primitive" units.



802.11 WLAN – MAC Block Diagram



Gate Level Design – Circuit Structure



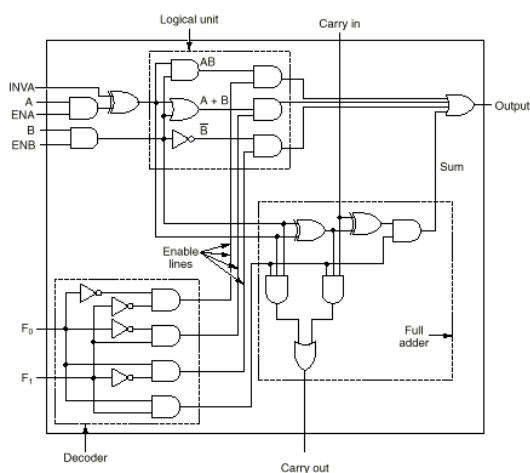
Source: A. Tanenbaum, 4th ed., 1999.

1-bit 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.

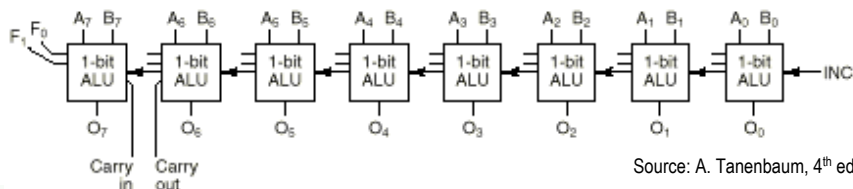


Register Level Design – Circuit Structure



Arithmetic Logic Unit (ALU)

- ✓ 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.



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



Circuit Design – Function

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

Truth Table:

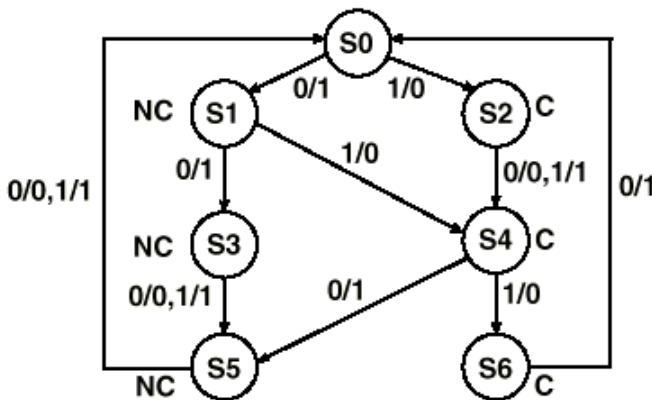
- ✓ This is a technique to represent a purely functional result.
- ✓ You list a column in the table for each input, and a column for each possible output combination.
- ✓ The table is used as a “look-up”, given some input combination, to determine the output.
- ✓ The outputs are a function of the inputs.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

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



Circuit Design – Behavior

• Bubble Diagram / State chart:

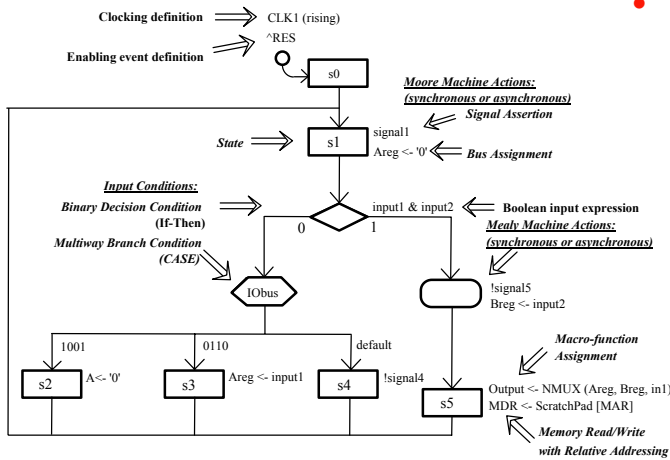


- ✓ Used to describe the state machine behavior, sequencing, and concurrent actions of the control logic.
- ✓ Step #1: Identify discrete states, and label them.
- ✓ Step #2: Enumerate state sensitivity and transition conditions for changing from a state to a “next” state.
- ✓ Step #3: Identify any outputs associated with a particular state. Also specify whether the outputs occur during the state or on transition either into or out of the state.
- ✓ Identify redundant states and use a hierarchy of states to model common transitions.

Source: Roth, 2000.



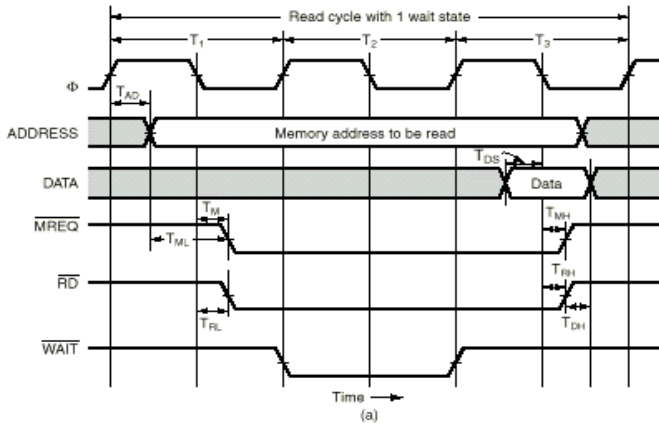
Systems Design – Behavior



- **ASM Diagram:**
- Algorithmic State Machine (ASM)
- Used to decompose behavior within a function block into an ordered sequence of operations.
 - ✓ Step #1: Define sequence of steps within control algorithm, and specific operations to occur in each step, using abstract "flow chart".
 - ✓ Step #2: Insert the necessary control to respond to events, using Case branch and Conditional branch.
 - ✓ Step #3: Allocate each abstract data operation to RTL macro operator.
 - ✓ Step #4: Bind individual buses for function variables to RTL bus units (registers, latches, wires).
 - ✓ Step #5: Define control scheduling with declaration of clocking.
 - ✓ Step #6: Decompose into concurrent "threads" for resource sharing.

© 2003 Dr. James P. Davis Page 27

Systems Design – Behavior (Response)



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

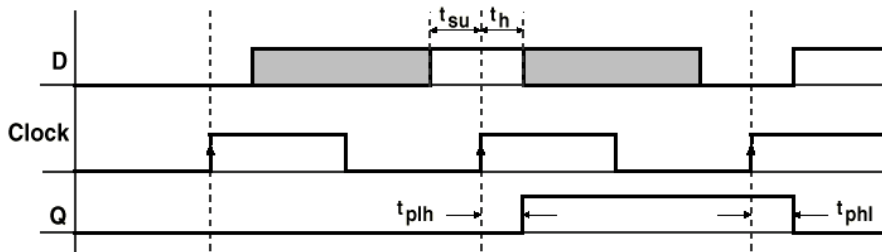
- **Timing Diagram:**

Used to describe the behavior in terms of changes in data values or state information over time. Each signal or quantity of interest has its value history recorded (or anticipated) along the time axis. Horizontal axis: advancing in time (such as during simulation). Vertical axis: a row for each signal or quantity of interest. Vertical bars allow specific time markets to be set (usually according to a system clock signal).



© 2003 Dr. James P. Davis Page 28

Gate Level Design – Device Timing



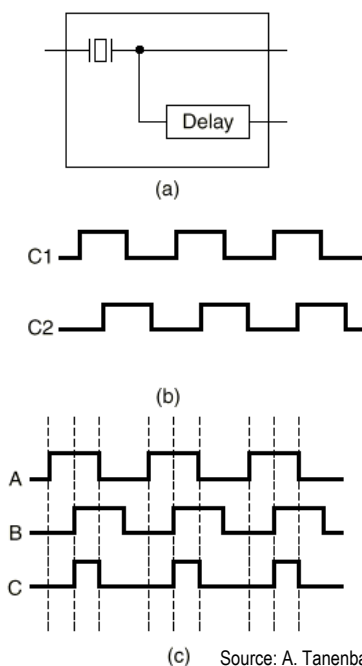
Source: A. Tanenbaum, 4th ed., 1999.

- Setup Time
 - ✓ Time t_{su} is the amount of time we must keep stable data on the input prior to sampling at the active clock edge.
- Hold Time
 - ✓ Time t_h is the amount of time the input signal value must be stable after the active clock edge, so that it can be sampled by the flip flop correctly.
- Example: D flip flop



© 2003 Dr. James P. Davis Page 29

Register Level Design – Clocking Schemes



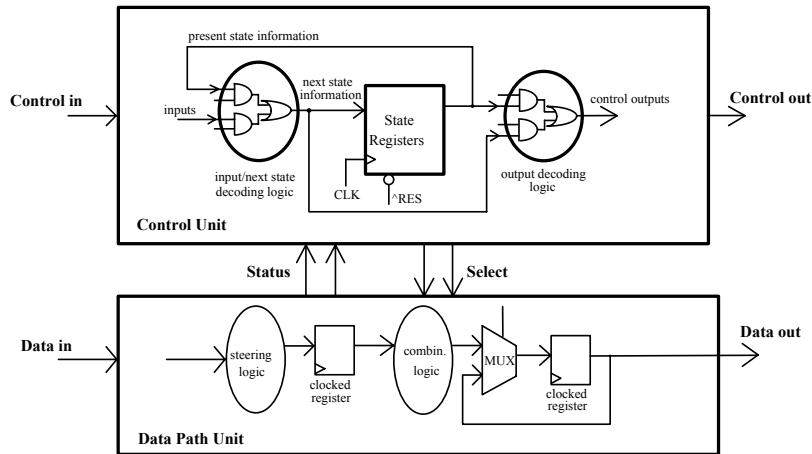
(c) Source: A. Tanenbaum, 4th ed., 1999.

- A clocking element
 - An oscillating crystal with properties that allow it to serve as a synchronizing element in digital designs.
- Timing diagram
 - Graphical means to indicate expected or observed timing behavior of a design.
 - Represented by a waveform of significant signals, their timing and values at each instant.
- Multiple clocks
 - Many designs operate on the clock to generate some multiple or fraction of the whole clock signal, which could be symmetric or asymmetric to the system clock.



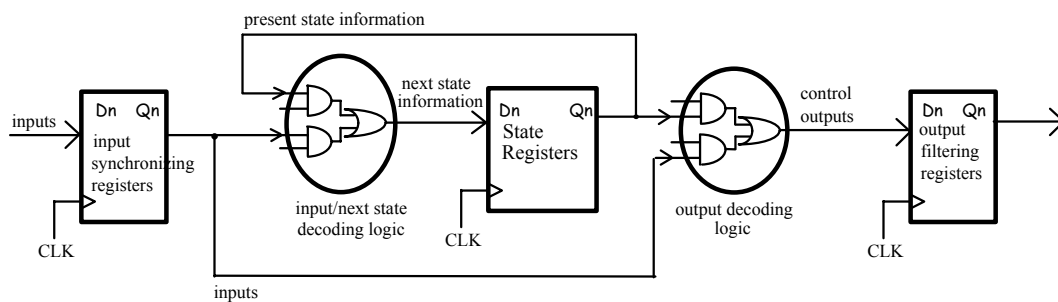
© 2003 Dr. James P. Davis Page 30

Partitioning into 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

Finite State Machine Model

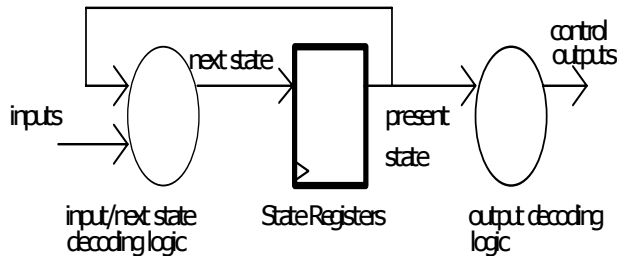


Components of FSM Model:

- ✓ State registers, input synchronization registers (optional) and output filter registers (optional).
- ✓ Next state decoding logic, and output decoding logic - combinational logic blocks.
- ✓ Input signals to the state machine, which are inputs to the next state and output decoding logic blocks (could be synchronized to clock with input registers).
- ✓ Next state information, which is generated as a result of input/next state decoding logic.
- ✓ Present state information, output from the state registers, which is fed back as an input to both next state and output decoding logic blocks.
- ✓ Outputs from the state machine - either generated synchronously from the output of the state registers (also used as present state information), or asynchronously as output of the output decoding logic block (which takes input and present state information to produce outputs). Could be filtered using output registers to eliminate possible signal transients.

Finite State Machine Model

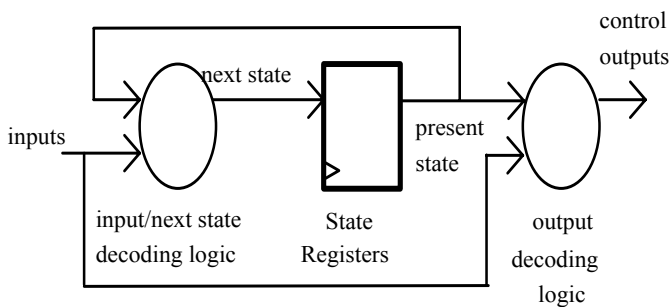
• Moore machines:



- ✓ *Control outputs* generated by the state machine are dependent only on the *present state information*.
- ✓ The *control outputs* are synchronized to the clock that controls state transitions.
- ✓ Moore machines are used when it is important to synchronize all control actions with the change in state, and thus, by the clock.
- ✓ Moore machines effectively filter out transients, and can be used to eliminate race conditions when inputs are unfiltered.

Finite State Machine Model

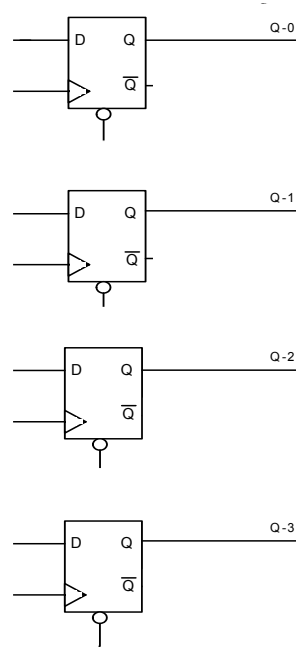
• Mealy machines:



- ✓ The *control outputs* of the state machine are dependent on the *inputs* and *present state information*.
- ✓ The *control outputs* can be asynchronous, in that outputs can change value as the *inputs* change value, provided the appropriate *present state information* is maintained.
- ✓ The *control outputs* are gated by the *present state*.
- ✓ Mealy machines are used to create control blocks that respond quickly to external signal changes.
- ✓ Care must be taken to isolate the design from transients and race conditions.

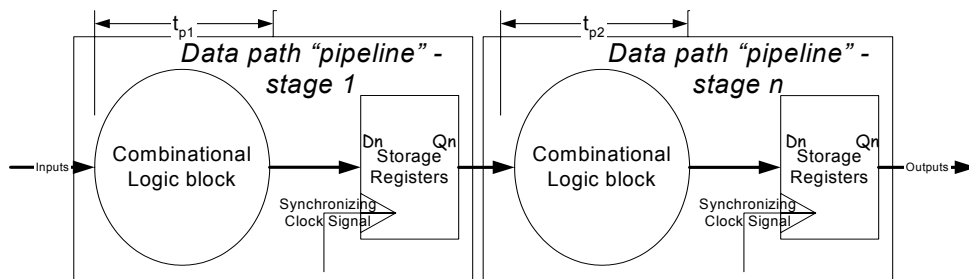
State Machine - FSM Encoding

- **Encoding Techniques** – we are interested in encoding the output lines from state registers:
 - **Binary** – ascending count of binary numbers representing each state in the sequence. Control output signals require decoding (combinational logic) to generate a specific control signal.
 - **Gray** – sequencing of the states so that only one bit between adjacent values changes at a time. This minimizes “glitches”, since ascending Binary encoding involves transitioning through intermediate values when more than a single bit changes value.
 - **One-hot** – use one D-FF for each state output line, where only a single line is active for each state. No encoding of the lines, so output control signals can be derived directly off the state lines.

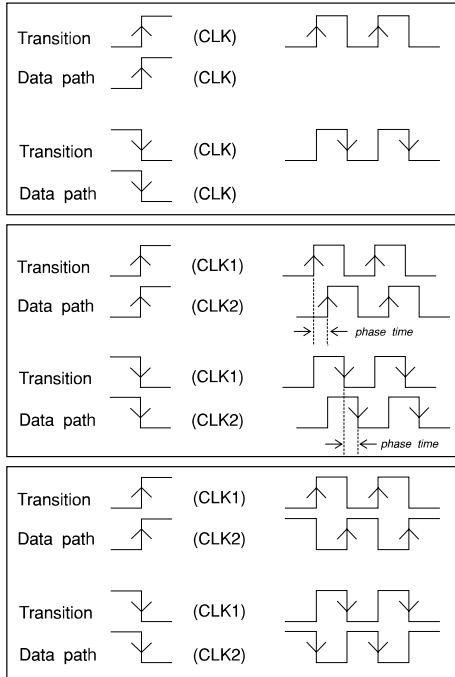


Pipelined Datapath Model

- **Use memory elements in the data path to store signal values.**
 - ✓ Purpose: synchronize behavior of complex circuits.
 - ✓ Benefits of circuit synchronization:
 - ✦ Eliminate unpredictability of output behavior due to timing skew.
 - ✦ Create signal stability, as they must have stable values for certain period of time (setup, hold).
 - ✦ Better isolate signals from noise transients.
- **Use of memory to create complex control structures.**
 - ✓ Controller sequences operations in the data path.



Register Level Design – Clocking Schemes



- **Scheme 1**
 - ✓ Single-phase, rising or falling edge clocks.
- **Scheme 2**
 - ✓ Two-phase overlapping, rising or falling edge clocks.
- **Scheme 3**
 - ✓ Two-phase non-overlapping, rising and/or falling edge clocks.

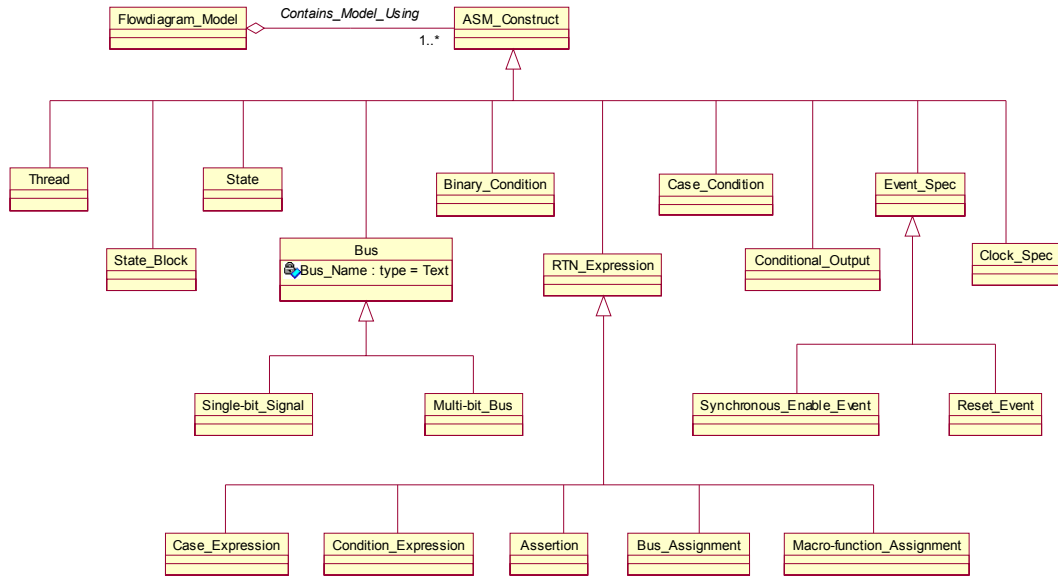


III. Lecture:

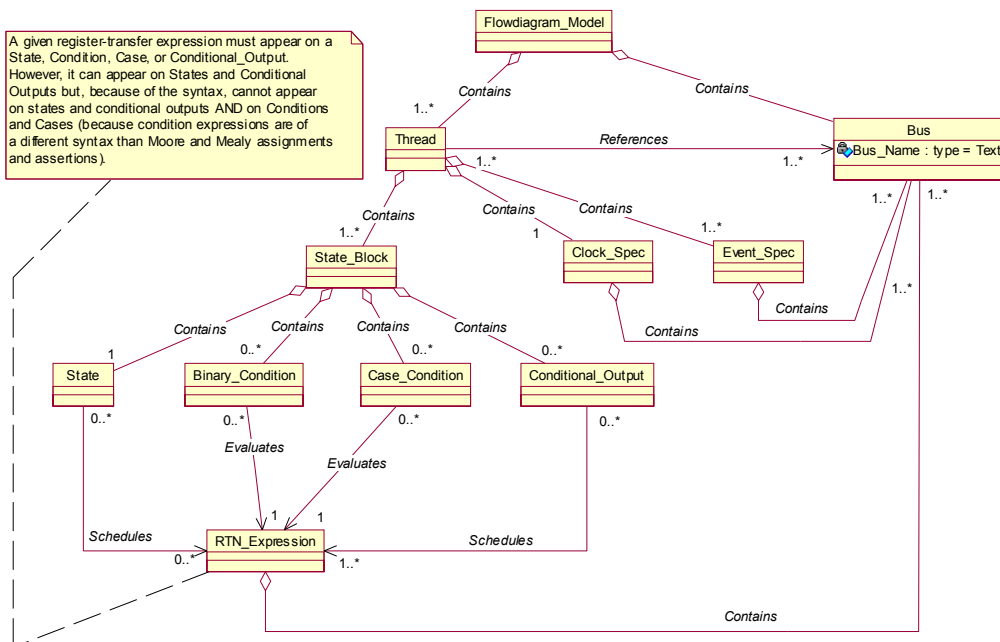
Model-Based Design Notation: The Algorithmic State Machine



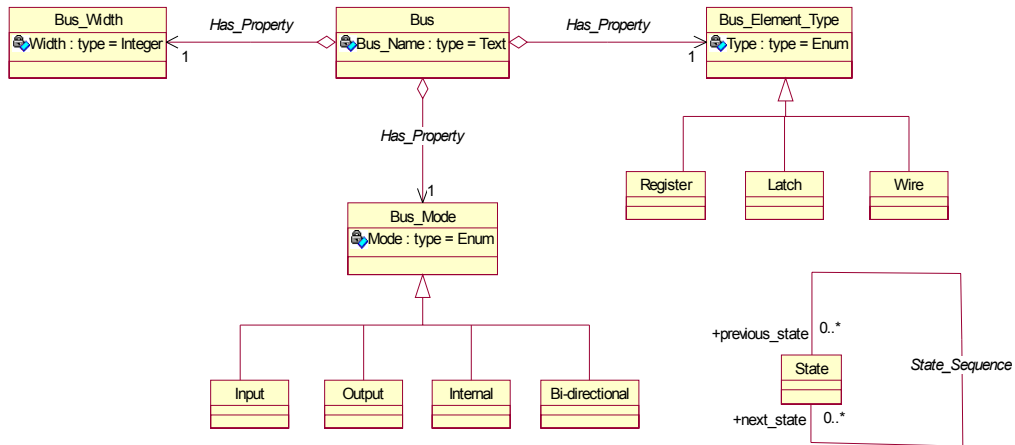
Taxonomy of ASM Diagram Concepts



Composition of ASM Concepts-1



Composition of ASM Concepts-2



Executable ASM Method

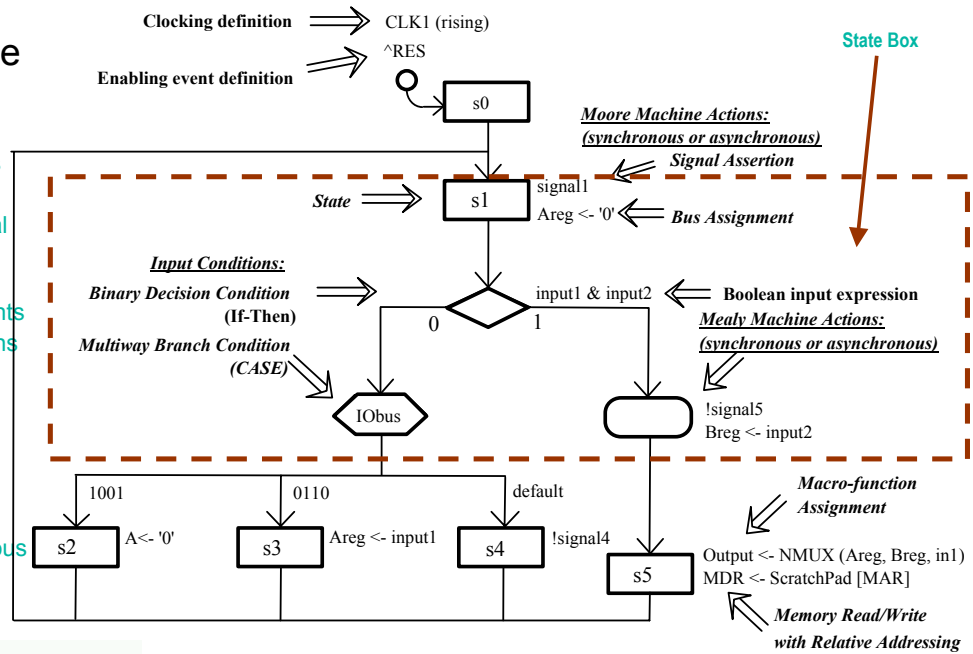
State Machines

The semantics of Moore and Mealy machine modeling

Constructs of Executable ASM Diagram

• Executable ASMs

- ✓ States
- ✓ Conditions
- ✓ Cases
- ✓ Conditional Outputs
- ✓ Assertions
- ✓ Assignments
- ✓ Expressions
- ✓ Macro-functions
- ✓ Memory Indexing
- ✓ Clocking
- ✓ Reset
- ✓ Synchronous events



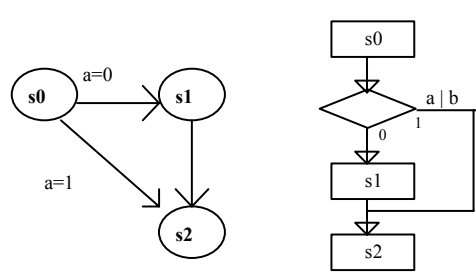
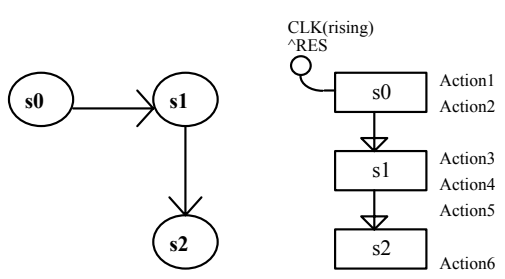
Relationship of State Machines to Datapath

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.



ASM vs. State Diagrams - Control Structures



Sequence: Series of States

States follow the sequencing indicated by direction of state transitions. ASM diagrams have *actions* attached to the right side of state "boxes" for clarity.

In ASM diagrams, transitions to next state are triggered by the system clock or any single-bit signal.

States with no actions are called *delay states*, indicating a delay of one clock cycle.

Selection: Binary Branching

Binary branching is represented using a condition "diamond".

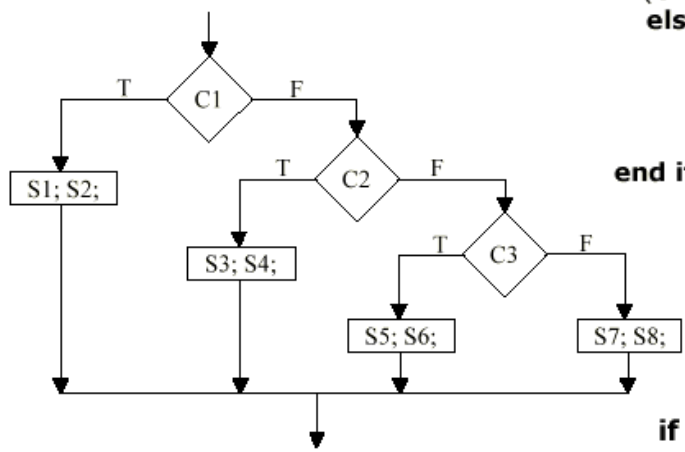
Boolean expressions on conditions can be of arbitrary complexity, with many terms and variables.

For simple branching situations, both representations are equally suited to the task.



ASM Diagram – Nested Control Structures

- If-Then-Else: Nested Statements



```

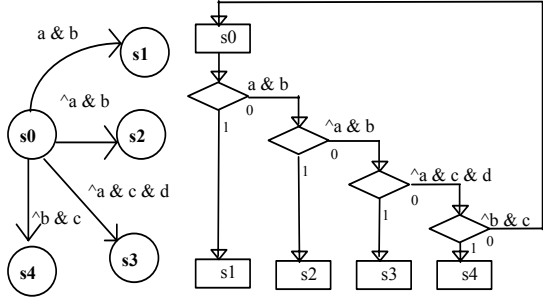
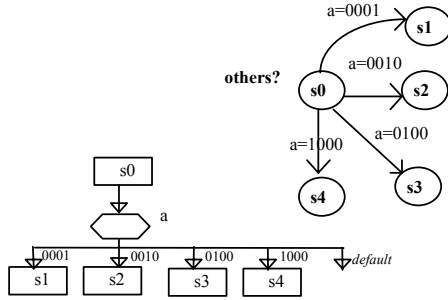
if (C1) then S1; S2;
  else if (C2) then S3; S4;
    else if (C3) then S5; S6;
      else S7; S8;
    end if;
  end if;
end if;
  
```

```

if (C1) then S1; S2;
  elsif (C2) then S3; S4;
  elsif (C3) then S5; S6;
  else S7; S8;
end if;
  
```



ASM vs. State Diagrams – Branching Control Structures



Selection: Multi-way Branching, Single Variable

ASM diagrams use *case* construct for multi-way branch conditions of a single, multi-bit variable/bus. Branch conditions are binary or enumerated values.

In ASM diagrams, all undefined transitions are tied to a default transition. This eliminates possible transitions to unspecified states, a common cause of design failure in the field.

State diagrams have no such mechanism, and thus are ambiguous.

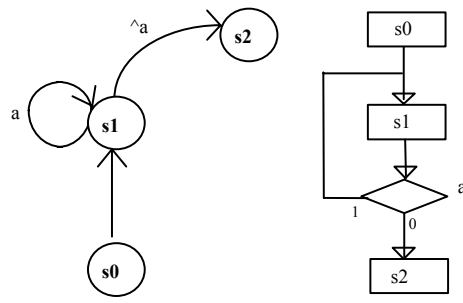
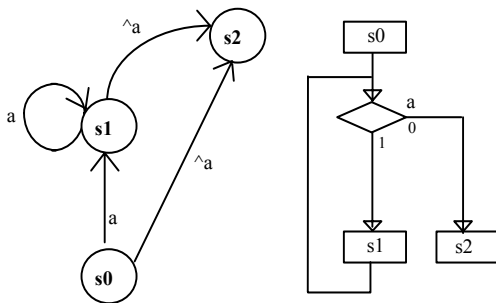
Selection: Multi-way Branching, Multi-variable

In State diagrams: (1) the ordering of transitions is ambiguous; (2) all transitions aren't specified (device problems likely in the field); (3) behavior is unpredictable under all conditions.

In ASM diagrams: (1) ordering of transitions is explicit; (2) transitions specified for all possible input combinations (including MVL values); (3) behavior is predictable.



ASM vs. State Diagrams – Loop Control Structures



Repetition: "While-Do" Control Loop

While-Do control structure is more apparent in the ASM diagram, and is consistent with hardware description language (HDL) constructs.

Single decision point in ASM diagram handles complexity more easily when multiple terms and variables are used in looping condition expression.

This type of control construct is used often for counting the number of loop iterations, where you test the condition *prior to* each execution of the loop, including the first pass.

Repetition: "Repeat-Until" Control Loop

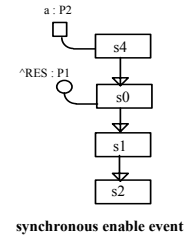
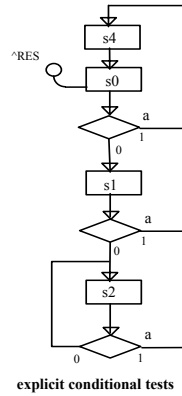
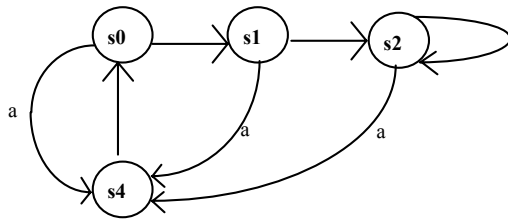
Placement of the decision "diamond" defines the type of looping construct, and the type of control behavior.

Repetition control structures are used in various styles of polling loops for implementing handshaking protocols.

This type of control construct is used often for counting the number of loop iterations, where you test the condition *after* completing each execution of the loop, requiring execution of at least the first pass.



ASM vs. State Diagrams – Synchronous Transfer of Control



Control Interrupt Schemes - State Diagrams:

The State diagram models state transitions, but the prioritization information--when multiple transitions are possible--isn't clear in the notation, without adding some additional symbol to indicate priority of the transitions.

Also, if State transitions have additional conditions, it isn't clear what happens when conditions aren't met (incomplete specification). However, this may be handled by modeling "looping" on a state in some cases.



Control Interrupt Schemes – ASM Diagrams:

In ASM diagrams, you can either define a test on each state transition for the specific event using condition "diamonds", or you can use the Enable Event construct, indicating that the specified event has precedence over the normally-specified next state transition. This works like a priority encoder on the next state decoding logic of the state machine. At any time when the input is sampled for determining next state, the transition for the Enable Event 'a' will take precedence.

© 2003 Dr. James P. Davis Page 49

Executable ASM Method

Datapath

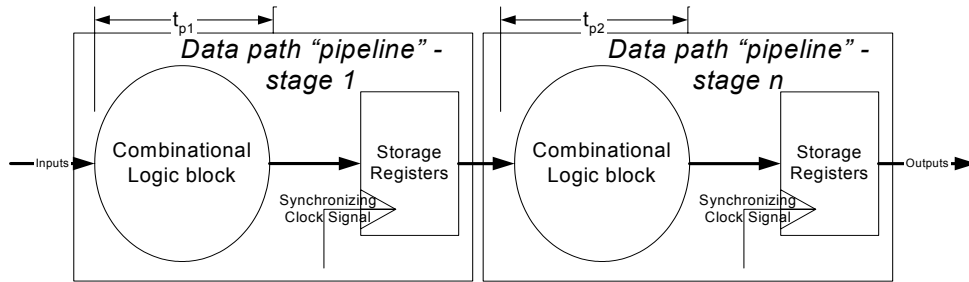
The timing semantics of Moore and Mealy driven datapath



© 2003 Dr. James P. Davis Page 50

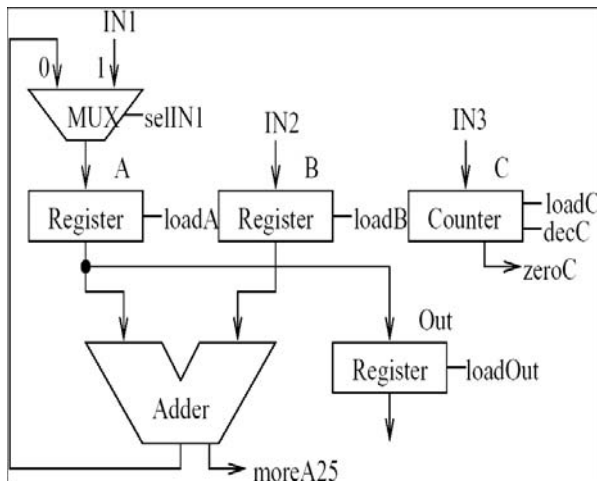
Datapath Logic Design

- Use of memory elements in the data path to store signal values.
 - ✓ Purpose is to synchronize the behavior of complex circuits.
 - ✓ Benefits of circuit synchronization:
 - ✧ Eliminate unpredictability of output behavior due to timing skew.
 - ✧ Create signal stability, as they must have stable values for certain period of time.
 - ✧ Better isolate signals from noise transients.
- Use of memory to create complex control structures.
 - ✓ Controller sequences operations in the data path.

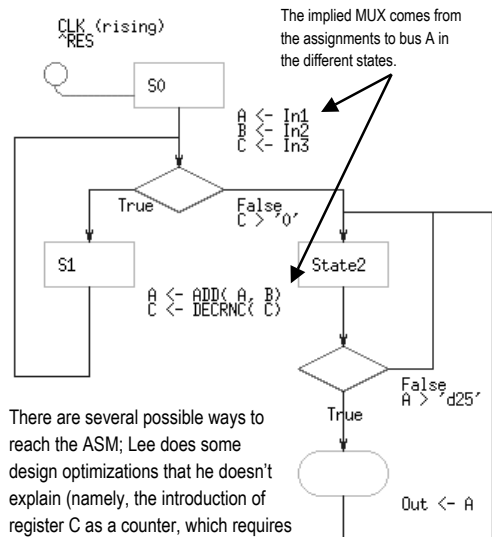


Data Path Description to Executable ASM

- Given the register-level data path circuit, below, what is the basic structure of the associated ASM model?



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

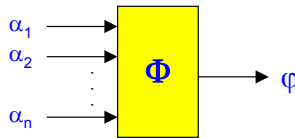


There are several possible ways to reach the ASM; Lee does some design optimizations that he doesn't explain (namely, the introduction of register C as a counter, which requires its own control thread in our ASM). Figure 4.4 done in Nimbus, cf. Fig 4.8 p. 130, Lee, which has control signals.



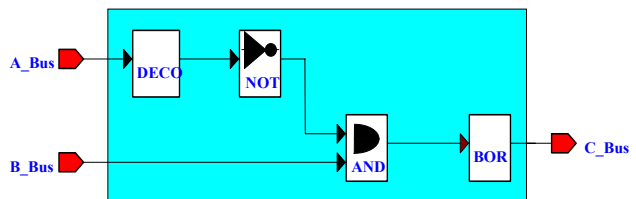
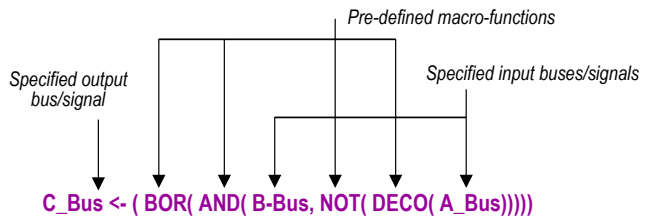
ASM – Datapath Macro-functions

- We can extend the expressive capability of basic ASM by adding macro-functions representing datapath computations.
 - Pure functional representation that is mathematically sound (i.e., no “side effects”).
 - Notation of the form: $\varphi \leftarrow \Phi(\alpha_1, \alpha_2, \dots, \alpha_n)$
 - φ : output signal into which macro assignment is made.
 - \leftarrow : macro-function assignment operator.
 - Φ : macro-function operator, representing combinational logic functionality (with no internal delay elements (i.e., computation should complete in a single clock cycle).
 - $\alpha_1, \alpha_2, \dots, \alpha_n$: input operands, signals that are fed into logic function.



ASM - Datapath Logic Operations

- **Pre-defined executable datapath operator macros.**
 - ✓ Are used in LHS of macro assignment statements.
 - ✓ Datapath macro operations are “scheduled” in states on entry to the state.
 - ✓ Attached as a text expression to the state.
 - ✓ LHS: assigned bus/signal.
 - ✓ RHS: input args, macro function calls, possibly nested (like C functions).

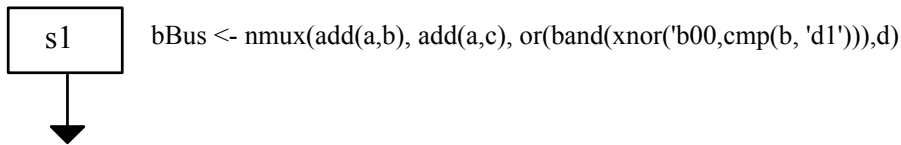


- **Macro types**
 - ✓ Arithmetic: ADD, SUB, INCR, MUL, DIV, REM.
 - ✓ Boolean logic: AND, OR, NOT.
 - ✓ Steering logic: MUX, DECO, PENCO, DMUX.

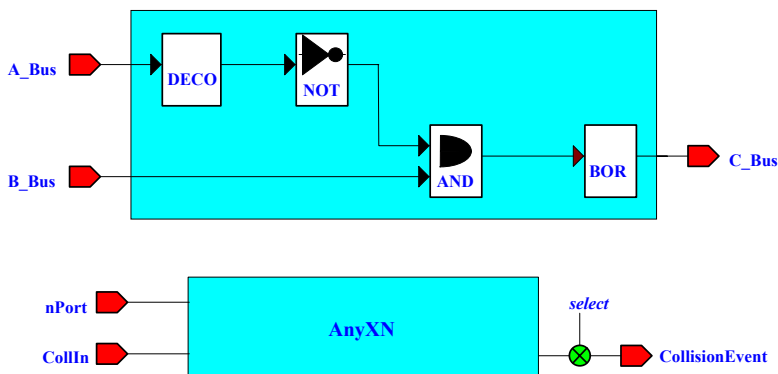


ASM - Modeling Complex Macro-functions

- Create complex datapath descriptions that are efficient, in terms of gate count and delay.
 - Pure functional representation that is mathematically sound.
 - Lends itself to compact annotation without requiring operator precedence rules.
- Things to remember:
 - Macro arguments should match, in terms of bus *width*, signal *type* (binary, MVL9, etc.) and signal *mode* (input, output, etc.).
 - Macros should not have an output argument fed back as an input without proper buffering (with register or latch definition for signal/bus specified on LHS of expression).
 - Don't drive a signal/bus as both *registered* and *unregistered* (as this causes a *multi-drive error*, unless using MVL data types with resolution tables).



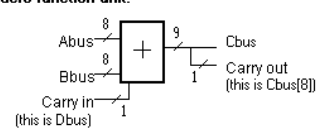
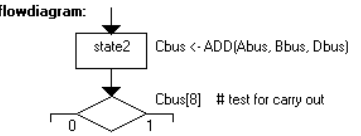
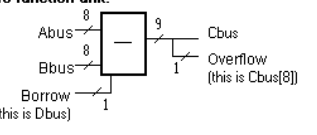
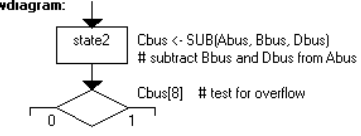
ASM - User Defined Datapath Operations



- RTL macro-functions: approx. 30 primitive data path elements in a library.
- Macros are "scalable" - with any number of buses and any bus widths.
- Macros can be used to construct more complex user-defined data path functions.
 - ✓ **Macro-function definition:** `AnyXN(A_bus,B_bus) ::= BOR(AND(B_bus,NOT(DECO(A_bus))))`
 - ✓ **Macro-function binding:** `CollisionEvent <- AnyXN(nPort,CollIn)`

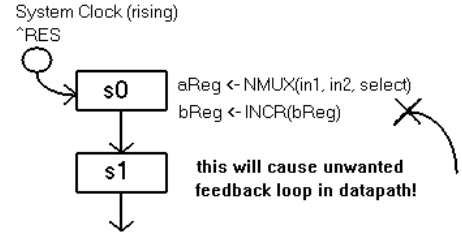
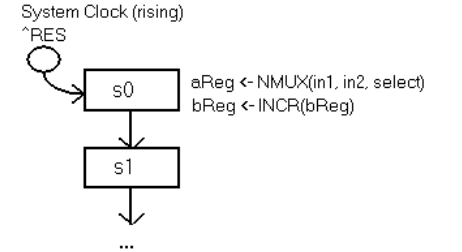


ASM - Arithmetic Macro-functions

<p>macro-function unit:</p>  <p>flowdiagram:</p> 	<p>macro-function unit:</p>  <p>flowdiagram:</p> 
<p>Carry In is optional input argument to the ADD macro. Carry Out can be accessed as the most-significant-bit (MSB) of the output bus.</p> <p>ADD uses Carry In and provides Carry Out. INCR provides Carry Out. Carry Out bit is MSB of output bus Cbus if width of Cbus is equal to Abus width + 1.</p>	<p>Borrow is optional input argument to the SUB macro. Overflow flag can be accessed as the MSB of the output bus to which the result of the operation is assigned.</p> <p>SUB uses Borrow and generates Overflow. DECR also generates Overflow. Overflow bit is MSB of output bus Cbus if width of Cbus is equal to Abus width + 1.</p>
<p>The ADD, INCR, SUB, DECR macros scale well up to 32 bits; beyond that, it is better to create custom arithmetic designs that scale more efficiently [Parhami, 2000].</p>	



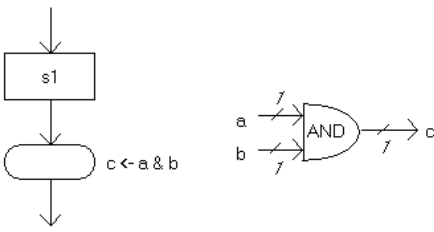
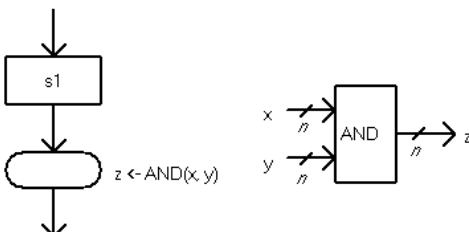
Moore Machine - Macro-function Assignments

<p>System Clock (rising)</p>  <p>Bus Name Bus Element</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td>aReg</td> <td>Wire</td> </tr> <tr> <td>bReg</td> <td>Wire</td> </tr> </table>	aReg	Wire	bReg	Wire	<p>System Clock (rising)</p>  <p>Bus Name Bus Element</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <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>								



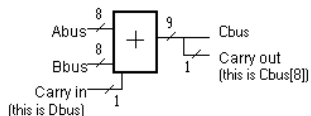
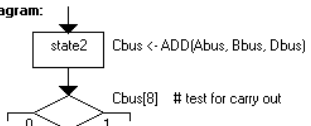
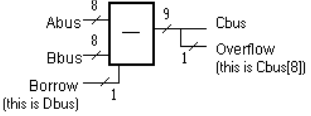
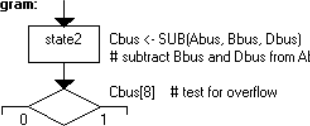
Using Boolean Operators vs Macro-functions

Specifying simple data path operations can be done using either *expression operators* or *macro-function operators*.

	
<p>Expression operators: these are provided by the flowHDL expression language. These are unary, binary and relational operators for the designer to compactly construct descriptions of data path behaviors. The restriction is that they can only be used with single-bit buses.</p>	<p>Macro-function operators: these are provided as part of the macro-function library. There are no general restrictions on the use of macro-functions with buses. However, for single-bit buses, the use of expression operators is more efficient in usage of gates.</p>

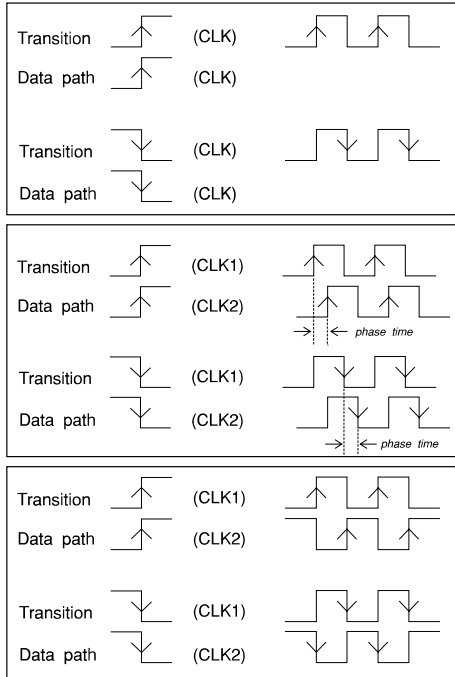


Carry/Borrow Bit with Arithmetic Macro-functions

<p>macro-function unit:</p>  <p>flowdiagram:</p> 	<p>macro-function unit:</p>  <p>flowdiagram:</p> 
<p>Carry In is optional input argument to the ADD macro. Carry Out can be accessed as the most-significant-bit (MSB) of the output bus.</p> <p>ADD uses Carry In and provides Carry Out. INCR provides Carry Out. Carry Out bit is MSB of output bus Cbus if width of Cbus is equal to Abus width + 1.</p> <p>The ADD and INCR macros work with buses up to 127 bits.</p>	<p>Borrow is optional input argument to the SUB macro. Overflow flag can be accessed as the MSB of the output bus to which the result of the operation is assigned.</p> <p>SUB uses Borrow and generates Overflow. DECR also generates Overflow. Overflow bit is MSB of output bus Cbus if width of Cbus is equal to Abus width + 1.</p> <p>The SUB and DECR macros are fully scalable, in that they can work with buses up to 127 bits.</p>



Register Level Design – Clocking Schemes



- **Scheme 1**
 - ✓ Single-phase, rising or falling edge clocks.
- **Scheme 2**
 - ✓ Two-phase overlapping, rising or falling edge clocks.
- **Scheme 3**
 - ✓ Two-phase non-overlapping, rising and/or falling edge clocks.



IV. Examples:

Mapping ASM Algorithm-level models to Register-level and Gate-level representations



Examples - Manual Analysis of ASM

- **Objective:** To understand various methods for mapping the logic of the *controller* and *datapath* blocks of a digital component.
- **Approach:** We take a couple of ASM thread examples, and discuss the following mappings:
 - ASM to register-level datapath diagram, and from data flow/datapath diagram to ASM.
 - Starting from a control flow-centric or dataflow-centric point of view with regards to your design units.
 - Obtaining decoding circuits for *next state* and *output* equations derived by inspection from the ASM threads for all FSMs.
 - ASM model mapping to concurrent state machines, including how we generate both the combinational and register logic using LUT and register structures found in Xilinx® FPGA CLB resources.



© 2003 Dr. James P. Davis Page 63

Example 1 – ASM Controller Model

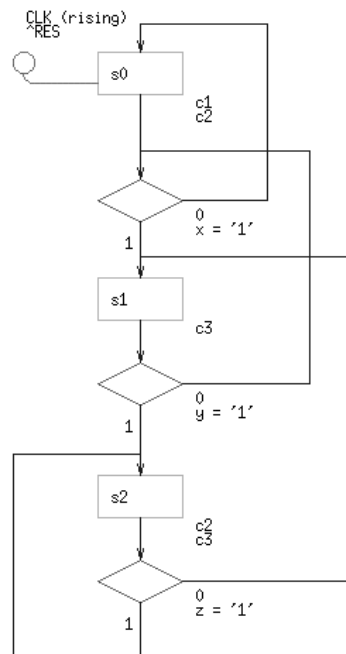
• Structure of an ASM Thread:

- ✓ We specify clocking and reset on initial state.
- ✓ The single-bit signals 'c1', 'c2', and 'c3' are *asserted* rather than *assigned*, which implies that we will use only a state machine, and not define any registers in the data path for RTL assignments.
- ✓ Note the state transition path from state 's1' to either 's0' or back into 's1' through the two conditions (*next state* equ.):

$$s1: (s1 \cdot (\sim(y=1) \cdot (x=1))) + (s0 \cdot (x=1)) \\ + (s2 \cdot \sim(z=1))$$

- ✓ Note that the ^RES signal is not part of next state decoding logic. It represents the "reset" pin for the state registers, and the CLK is tied to clock pin.

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

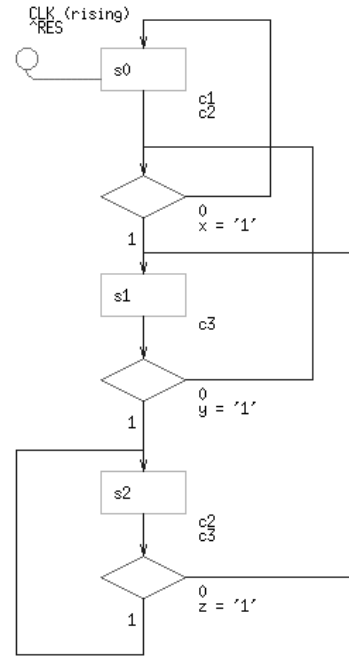


© 2003 Dr. James P. Davis Page 64

Example 1 – ASM to FSM Model

- Derive the *next state* equations for the FSM:

- Derive the *output* equations for the FSM:



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

© 2003 Dr. James P. Davis Page 65



Example 1 – ASM to FSM Model

- Derive the *next state* equations for the FSM:

$$s1: (s1 \cdot (\sim(y=1) \cdot (x=1))) + (s0 \cdot (x=1)) + (s2 \cdot \sim(z=1))$$

$$s2: (s1 \cdot (y=1)) + (s2 \cdot (z=1))$$

$$s0: (s0 \cdot \sim(x=1)) + s1 \cdot (\sim(y=1) \cdot \sim(x=1))$$

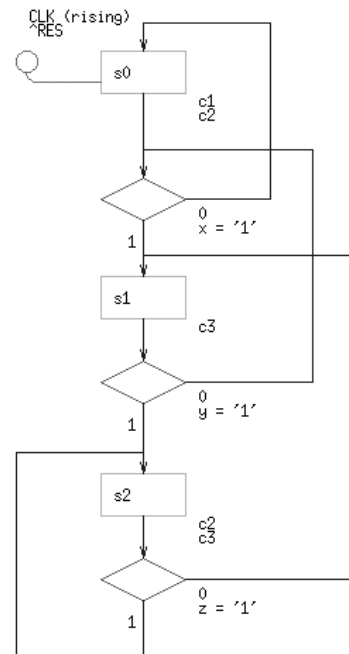
- Derive the *output* equations for the FSM:

$$c1: s0$$

$$c2: s0 + s2$$

$$c3: s1 + s2$$

(These are Moore-style outputs, dependent only on present state.)



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

© 2003 Dr. James P. Davis Page 66



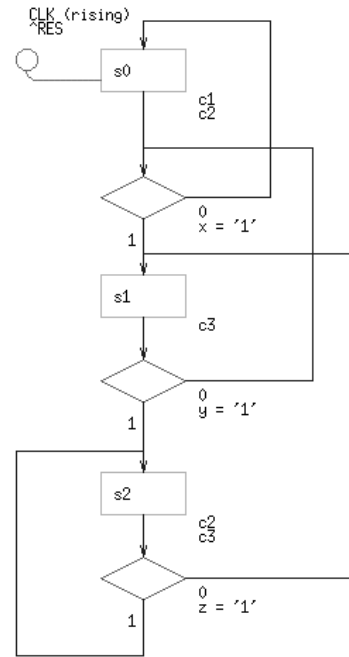
Example 1 – ASM to FSM Model

- Make the *state assignments* for the FSM:



- State assignment policies:

- Binary
- Gray code
- One-hot



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

© 2003 Dr. James P. Davis Page 67



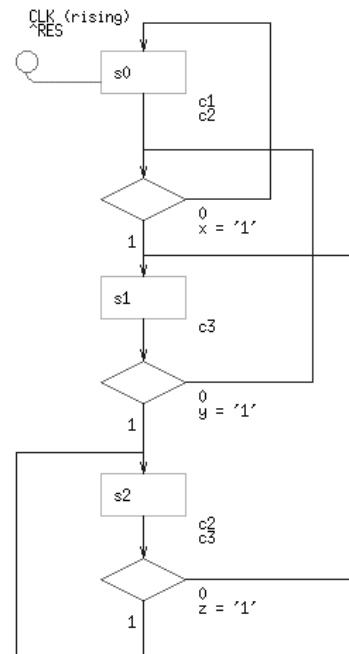
Example 1 – ASM to FSM Model

- Make the *state assignments* for the FSM:

s0 : Q1 = 0, Q0 = 0 (reset state)
s1: Q1 = 0, Q0 = 1
s2: Q1 = 1, Q0 = 1
 Use Gray code for encoding adjacent state assignments with only on state variable changing at a time.

- State assignment policies:

- Binary
- Gray code
- One-hot
- If P = number of states, we'll require $\lceil \ln(P) \rceil$ state register flip flops for binary state encoding.



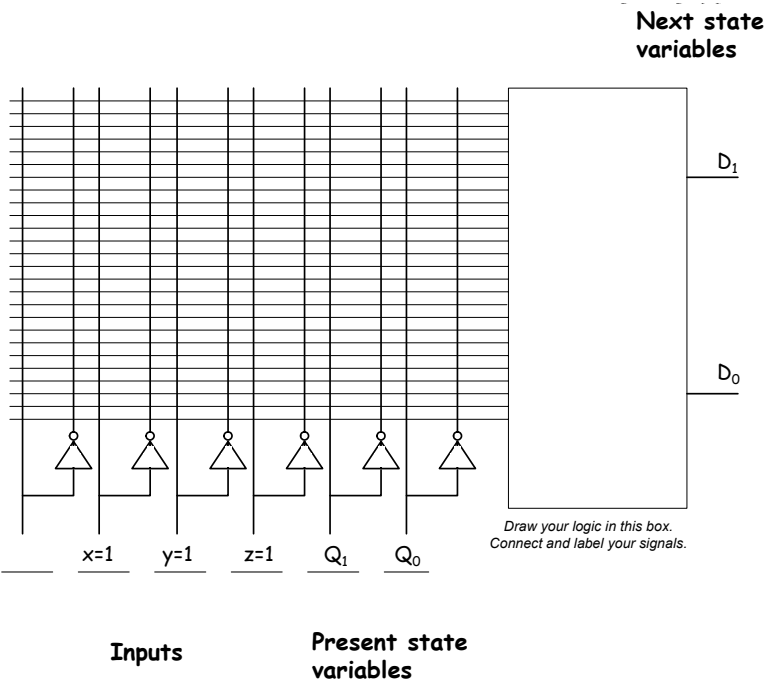
Example: Lee © 2000 Prentice-Hall Publishers, Inc.

© 2003 Dr. James P. Davis Page 68



Example 1 – ASM to Digital Logic

- Construct the schematic circuit diagram for the *next state decoding logic block* of the design, in the space provided below. Label the signals and buses appropriately.
- Take your next state equations (previous exercise) and "realize" them on this grid; draw the SoP (AND and OR gates) for each next state register input (D_1 and D_0).
- Draw the connections between buses to show which inputs feed into which parts of your logic circuit (i.e., use a blackened circle to show the connection between the vertical plane and the horizontal plane).



Example 1 – ASM to Digital Logic

$$s1 (01): (s1 \cdot (\sim(y=1) \cdot (x=1))) + (s0 \cdot (x=1)) + (s2 \cdot \sim(z=1))$$

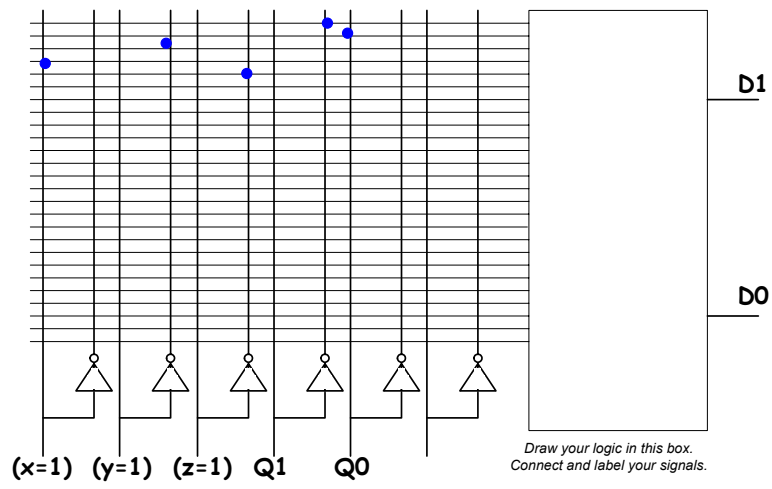
$$s2 (11): (s1 \cdot (y=1)) + (s2 \cdot (z=1))$$

$$s0 (00): (s0 \cdot \sim(x=1)) + s1 \cdot (\sim(y=1) \cdot \sim(x=1))$$

$$D0: \text{equ. } (s1 + s2)$$

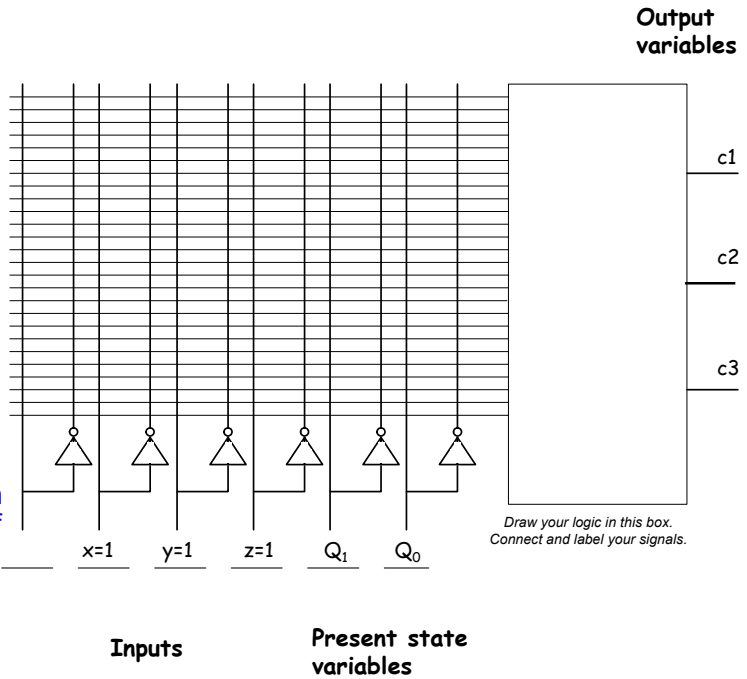
$$D1: \text{equ. } s2$$

To create SoP, we need to minimize equations.



Example 1 – ASM to Digital Logic

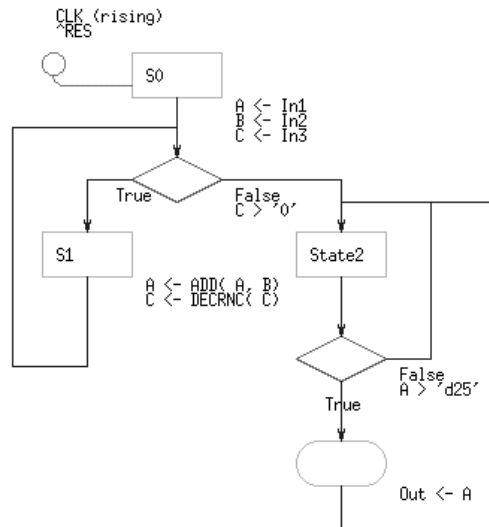
- Construct the schematic circuit diagram for the output decoding logic block of the design, in the space provided below. Label the signals and buses appropriately.
- Take your next state equations (previous exercise) and “realize” them on this grid; draw the SoP (AND gates) for each next state register input (D_1 and D_0).
- Draw the connections between buses to show which inputs feed into which parts of your logic circuit (i.e., use a blackened circle to show the connection between the vertical plane and the horizontal plane).



Example 2 – ASM to FSM Model

- Derive the *next state* equations for the FSM:

- Derive the *output* equations for the FSM:



Example 2 – ASM to FSM Model

- Derive the *next state* equations for the FSM:

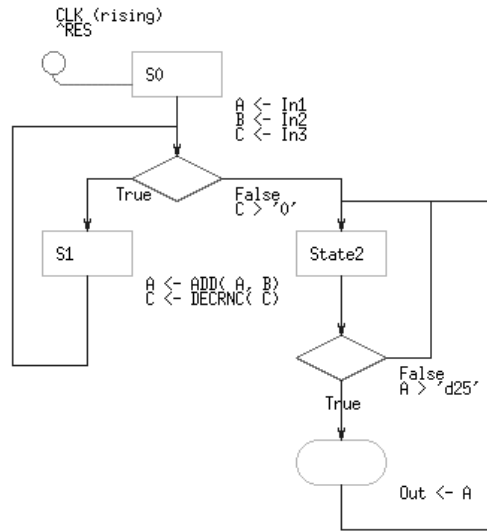
```

s0: // we only enter this state on reset.
s1: s0·(C>0) + s1·(C>0)
s2: s0·^(C>0) + s1·^(C>0) + s2
    // while in state s2, we circulate here.
    
```

- Derive the *output* equations for the FSM:

```

A<-In1: s0
A<-ADD: s1
B<-In2: s0
C<-In3: s0
C<-DECRNC: s1
Out<-A: s2 · (A>25)
// Each output assignment represents separate
// output equations. The first 6 are Moore
// assignments, the last is a Mealy assignment.
    
```



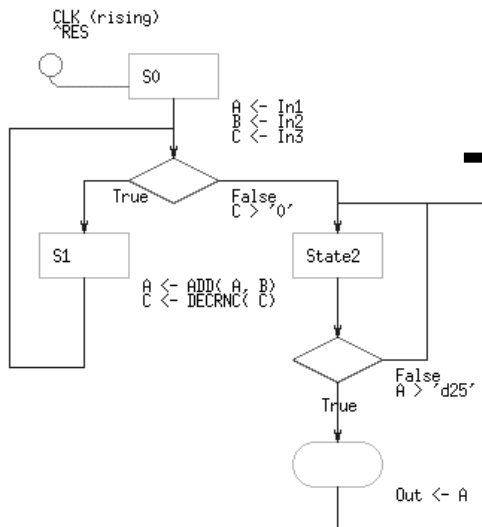
Example: Lee © 2000 Prentice-Hall Publishers, Inc.

© 2003 Dr. James P. Davis Page 73

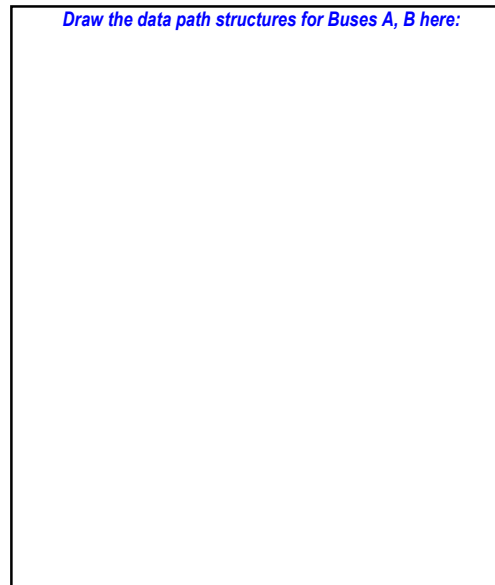


Example 2 – ASM to Data Path diagram

- Given the ASM model below, what is the basic structure of the associated register-level data path circuit?
- Can you draw a RTL diagram of the data path elements represented by the expressions on the model?



Draw the data path structures for Buses A, B here:



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



© 2003 Dr. James P. Davis Page 74

Example 2 – ASM to Data Path diagram

- **Using the implied Selection MUX:**
 - If you are controlling the setting of a value on a data path bus, by setting the value in more than one state, the implied logic is to control the data path operation using a MUX, whose 'select' line is derived from the FSM output decoding logic.
- **Using the implied Circulation MUX:**
 - If you are controlling the setting of a bus in more than one state, but do not explicitly set this bus in all referenced states in the thread, then the implied logic is to control the value of this register by "circulating" the value back into itself while the FSM is in any state where the value is not explicitly assigned.

