UNIVERSITY OF SOUTH CAROLINA

CSCE 611
Digital Systems Design I

2005/3/27

Weeks 3 & 4 Supplemental Notes
The Executable Algorithmic State Machine (ASM) Method

© 2004 Dr. James P. Davis

Weeks 3 & 4 - Outline

- The Executable ASM Method
  - Definition.
  - Analysis activities.
  - Design activities.
  - Comparing and contrasting Analysis versus Design.
- Using FSM State Diagrams and ASM Diagrams.
  - Diagrams provide "pictorial guide" the define the structure of the sequencing behavior of the FSM.
  - Used in 3 different ways: (1) definition/specification of sequential systems, (2) analysis of sequential circuits, (3) design of circuits.
  - Many text authors use State diagrams as a means to model simple FSM circuits, but present ASM diagrams for modeling more complex control architectures.

Introduction to the Executable Algorithmic State Machine (EASM) Method

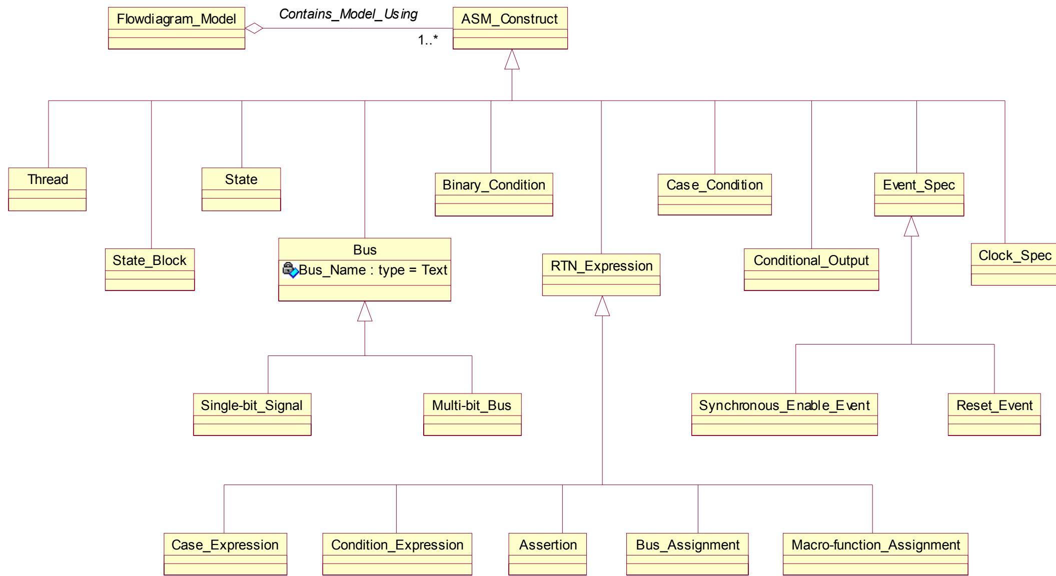
or

How do we bridge the gap between the systems level and circuits?

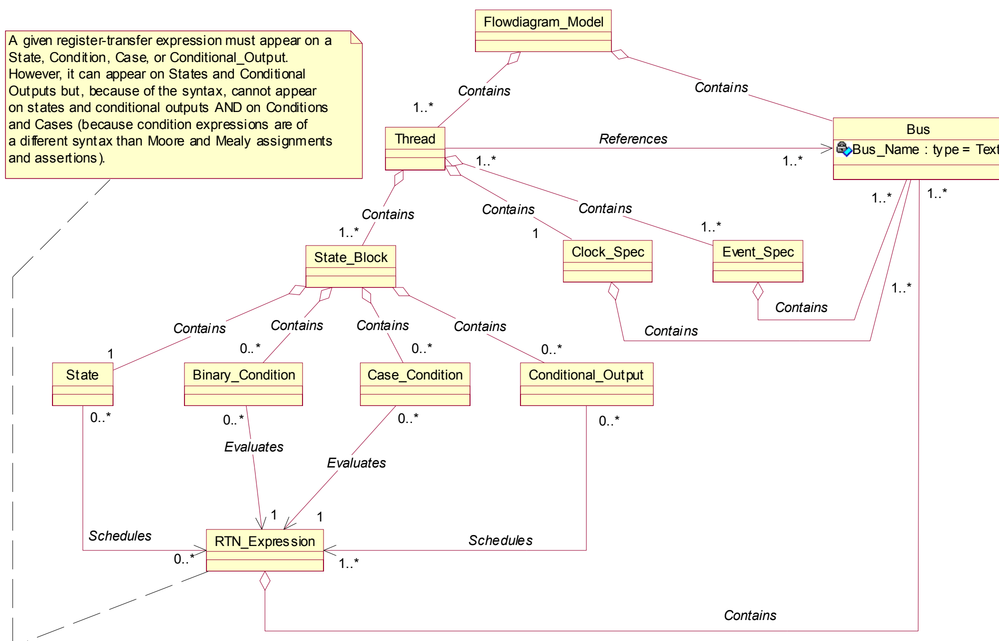
Executable ASM Diagrams - Objective

- 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 analysis model and architecture for our larger designs—UART and 802.11 MAC.
- Now, we focus on Systems-level architecture and Register Transfer Level (RTL) design using the Algorithmic State Machine.
  - ✓ Discussion of methods, tools and processes we'll be using for executing the custom-logic design activities for applications.
  - ✓ Focus on Nimbus and the use of FSM (finite state machine) and RTN (register transfer notation) for control path and data path architecture and design, using the Executable ASM notation.
  - ✓ Discussion of methods for design evaluation and tradeoff analysis of designed micro-architectures using VLSI custom-logic.

Taxonomy of ASM Diagram Concepts

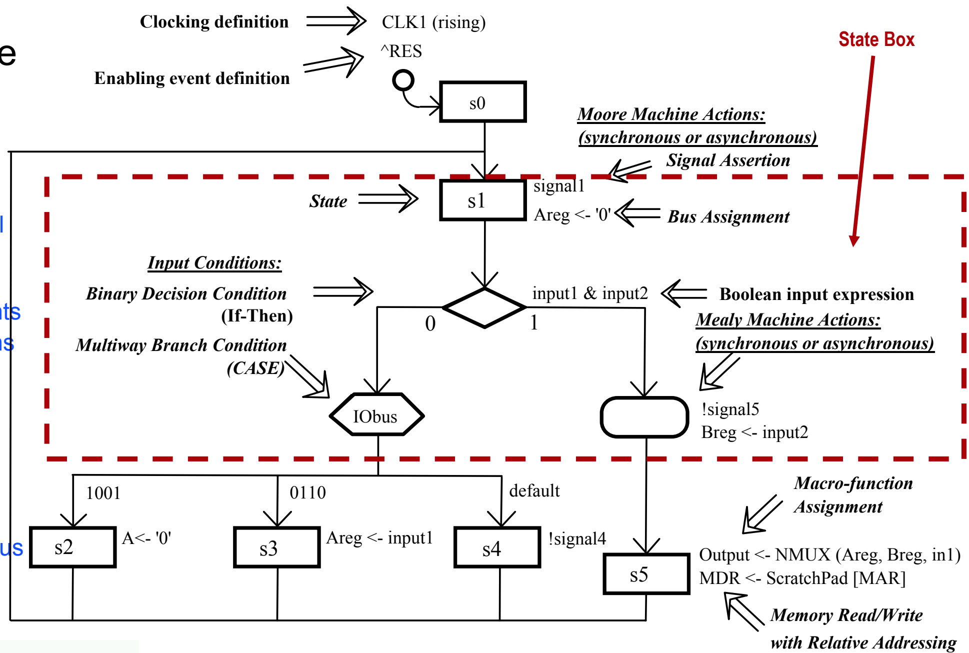


Composition of ASM Diagram - Concepts-1

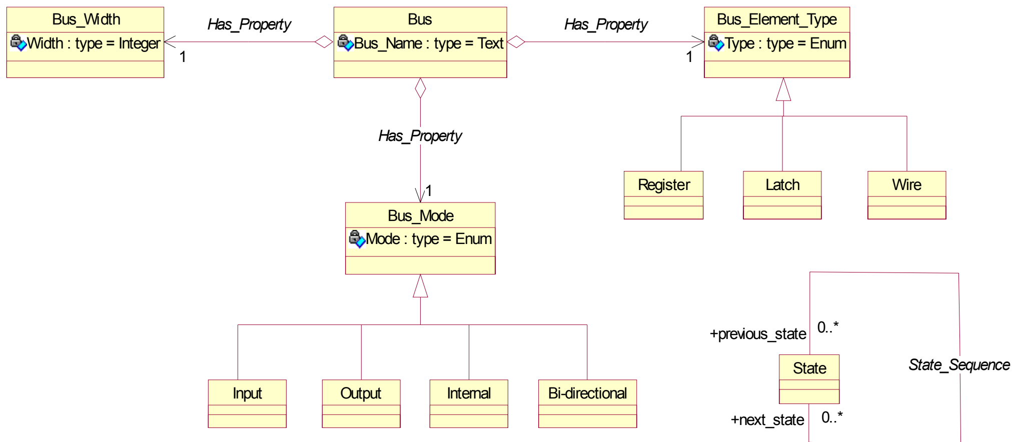


Composition of ASM Diagram - Example

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



Composition of ASM Diagram - Concepts-2

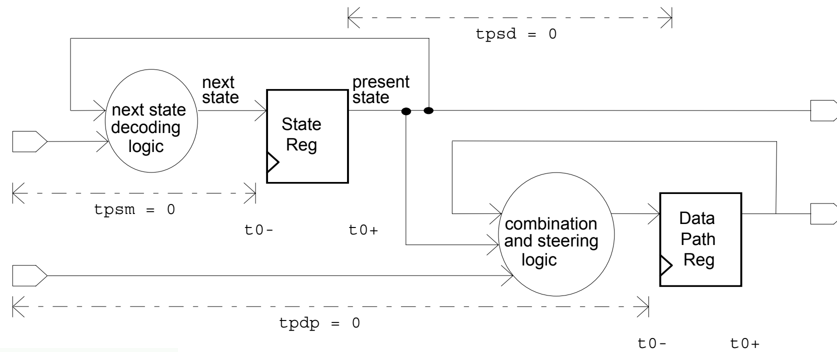


Delay Assumption for Register-Transfer Level

In Executable 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.



© 2002 Dr. James P. Davis

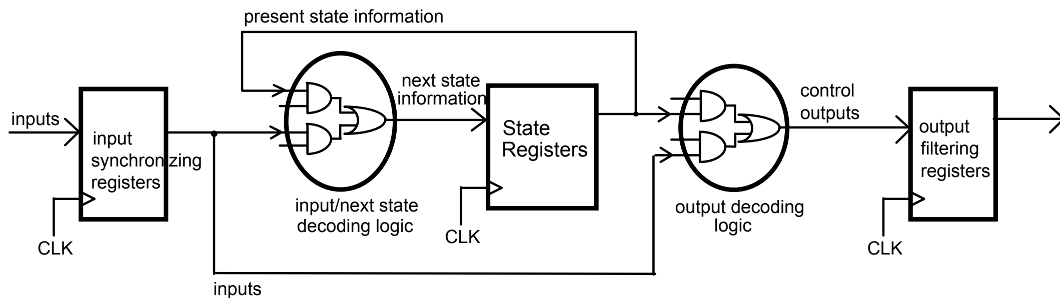
Executable ASM Method

State Machines

The timing semantics of Moore and Mealy machine modeling

© 2002 Dr. James P. Davis

Finite State Machine Model - Introduction



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.

Relationship of State Machines to Datapath

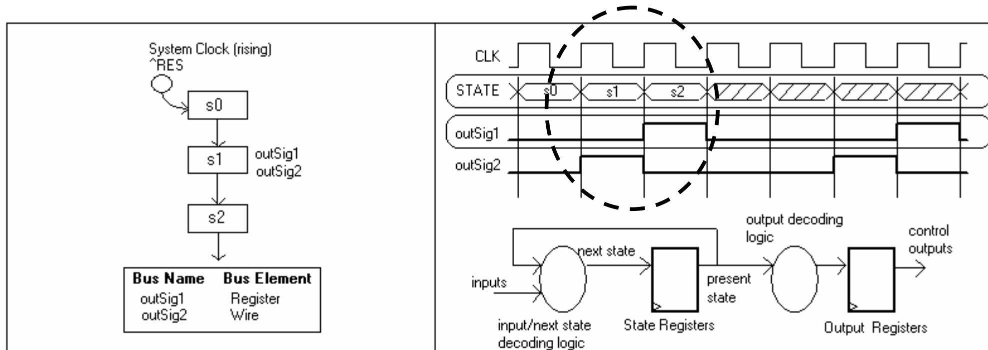
ASM diagrams incorporate information about control path and data path into a single representation. Using this notation, a design can express different design styles for both synchronous and asynchronous behavior of both the control and datapath.

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.

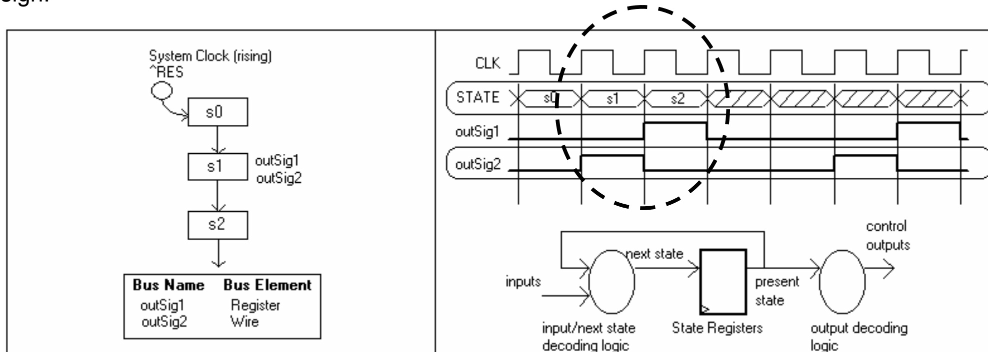


Moore Machine - Unregistered Assertions

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.

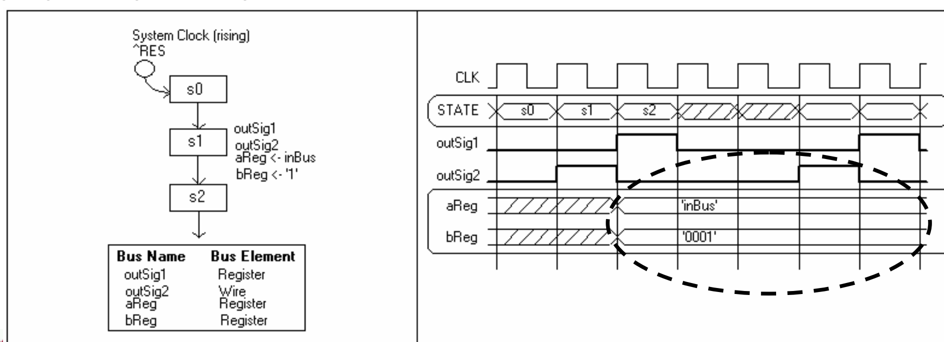


Moore Machine - Registered Bus Assignments

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.

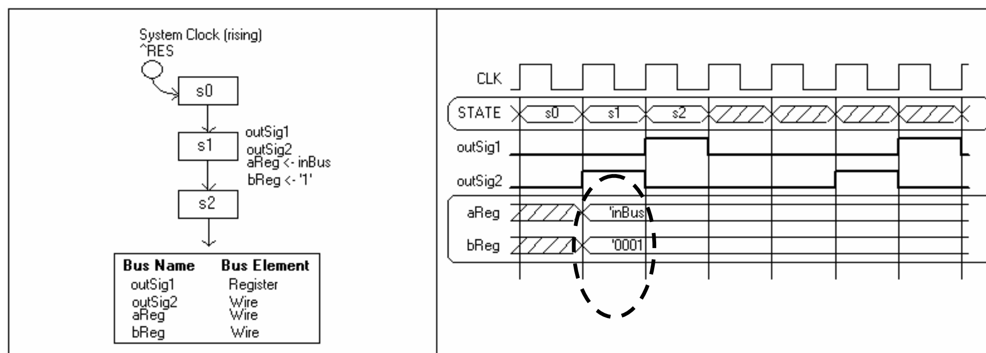


Moore Machine - Unregistered Bus Assignments

Unregistered bus assignments may be used by selecting bus Elements as "Wire" in Nimbus' 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.

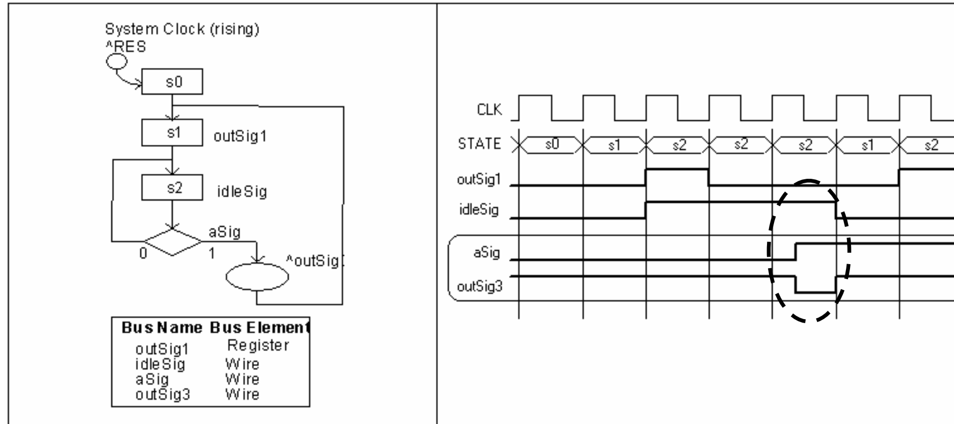
Using unregistered bus assignments reduces gate count of the datapath, and reduces circuit delay. However, special care must be taken to avoid race conditions, signal transients, and unwanted feedback loops in the datapath that can cause "metastability" (not settling to a specific value) or oscillation.



Mealy Machine - Unregistered Assertions

The output ^outSig3 is dependent both on the present state and the input. While in the state s2 associated with an input decision "diamond", if input signal aSig is asserted, the value of ^outSig3 will follow the value change of aSig.

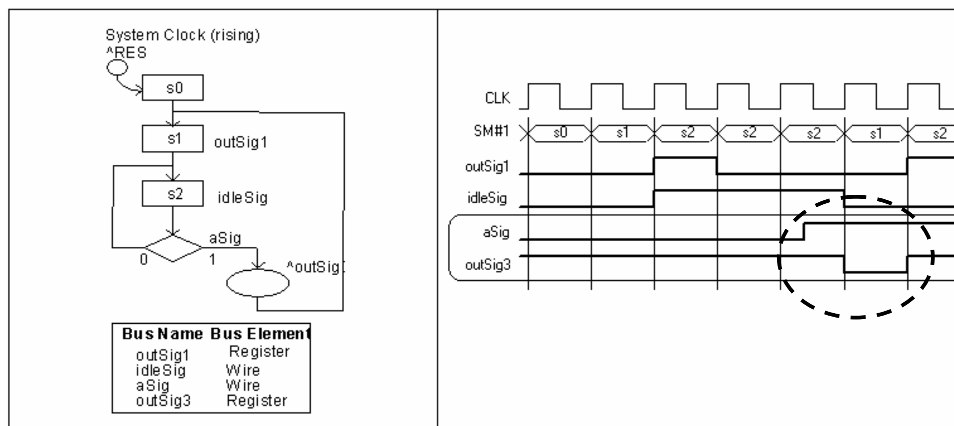
Note that ^outSig3 is an "assertion", indicating no data path information. The asserted value will be held for as long as state s2 is the current state. The output will be unregistered.



Bus Name	Bus Element
outSig1	Register
idleSig	Wire
aSig	Wire
outSig3	Wire

Mealy Machine - Registered Assertions

If the designer wishes to register a Mealy generated control output signal, the bus is defined as Registered element in the Bus Table. While in state s2, if input aSig occurs, the resultant change in outSig3 occurs on the next clock edge after the appearance of aSig.

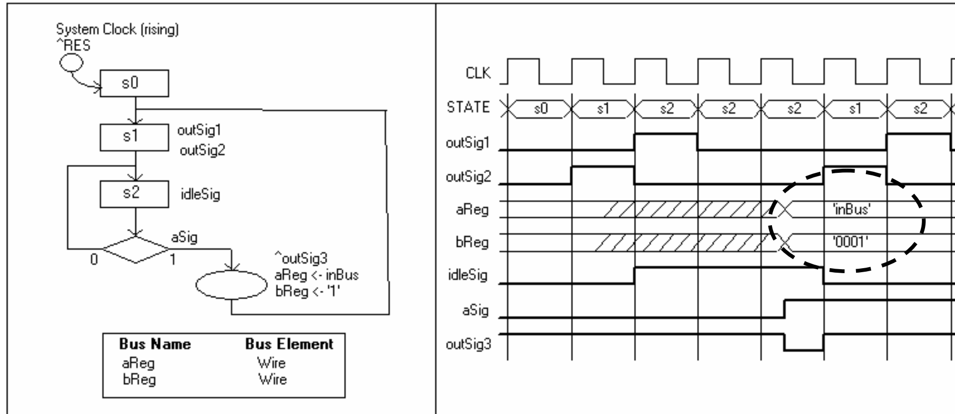


Bus Name	Bus Element
outSig1	Register
idleSig	Wire
aSig	Wire
outSig3	Register

Mealy Machine - Unregistered Bus Assignments

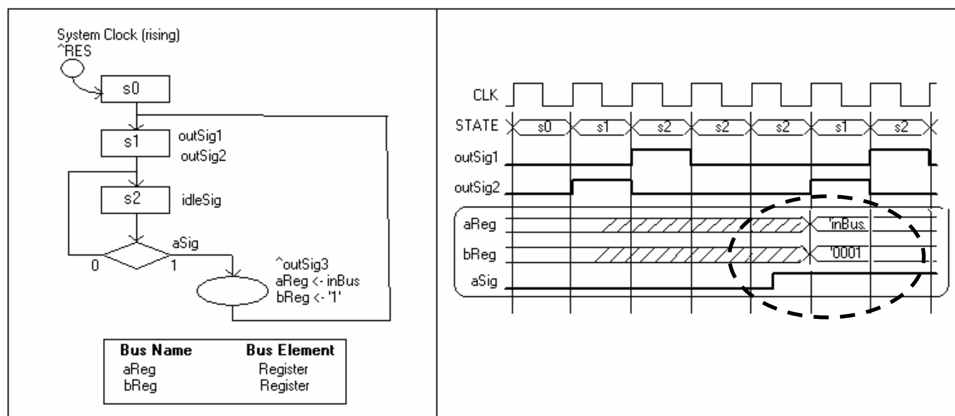
For Mealy-style outputs specifying datapath Bus assignments, the control outputs change asynchronously with the value of input `aSig`.

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.



Executable ASM Method

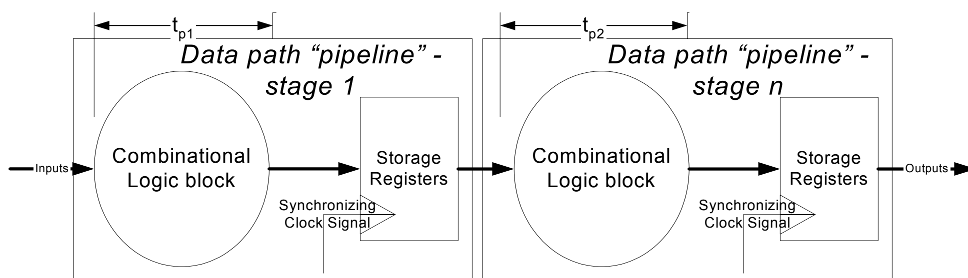
Datapath

The timing semantics of Moore and Mealy driven datapath

© 2002 Dr. James P. Davis

Datapath Logic Design

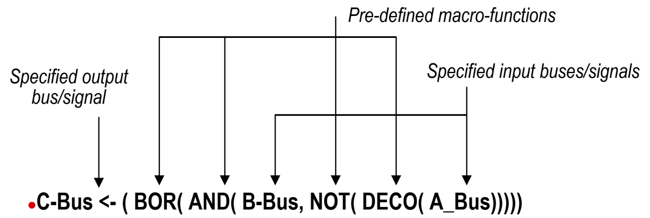
- Use of memory elements in the data path to store signal values.
  - Purpose is to synchronize the behavior of complex circuits.
  - Benefits of circuit synchronization:
    - Eliminate unpredictability of output behavior due to timing skew.
    - Create signal stability, as they must have stable values 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.

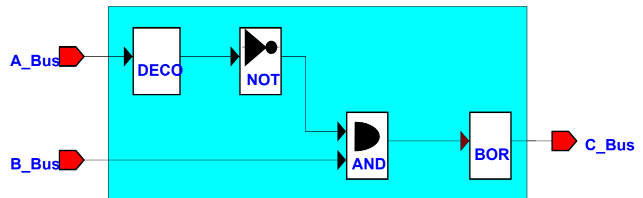


© 2002 Dr. James P. Davis

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.
  - Combinational logic elements.

Moore Machine - Macro-function Assignments

System Clock (rising)
^RES

s0: aReg <- NMUX(in1, in2, select)
bReg <- INCR(bReg)

s1

...

this will cause unwanted feedback loop in datapath!

Bus Name	Bus Element
aReg	Wire
bReg	Wire

System Clock (rising)
^RES

s0: aReg <- NMUX(in1, in2, select)
bReg <- INCR(bReg)

s1

...

Bus Name	Bus Element
aReg	Register
bReg	Register

Unregistered Output: To specify unregistered datapath for assignment of macros to buses, select Latch or Wire as Element in Bus Table.

Be careful not to use unregistered macros in a feedback loop in the datapath, as this causes metastability in the resultant circuit.

Registered Output: To specify that macro assignments in the datapath are to be registered, specify as "Register" Element in Bus Table.

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.

Mealy Machine - Macro-function Assignments

System Clock (rising)
^RES
s0
aSig
0
1
...
aReg <- NMUX(in1, in2, select)
bReg <- INCR(bReg)
this will cause unwanted feedback loop in datapatl
s1
...

System Clock (rising)
^RES
s0
aSig
0
1
...
aReg <- NMUX(in1, in2, select)
bReg <- INCR(bReg)
s1
...

Unregistered Output: Using macro assignments on Mealy outputs can be specified as using unregistered outputs in datapath by selecting Bus Table Element as "Wire"..

Be careful not to use unregistered macros in a feedback loop in the datapath, as this causes unwanted feedback in the resultant circuit.

Registered Output: Using macro assignments on Mealy outputs can be buffered using registered assignments by selecting "Register" choice for Element in Bus Table.

You can define a feedback path through datapath if you buffer with registered bus.

© 2002 Dr. James P. Davis

Avoiding Unwanted Feedback in the Datapath

s1
a <- a & b
x <- AND(x, y)
a
b
AND
a
x
y
AND
n
x

s1
c <- a & b
z <- AND(x, y)
s2
a <- c
x <- z
select
a
b
AND
c
MUX
a
select
x
y
AND
z
MUX
x

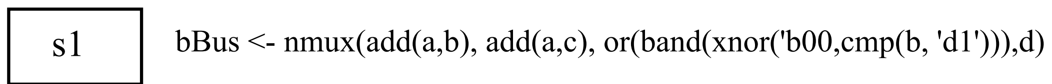
If expression or macro-function operators are specified as unregistered outputs in the ASM chart, the values seen on the output of the corresponding data path element can change values asynchronously with its inputs. If there is a feedback path to an input, an unwanted loop can be created, causing instability, metastability or even oscillation.

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.

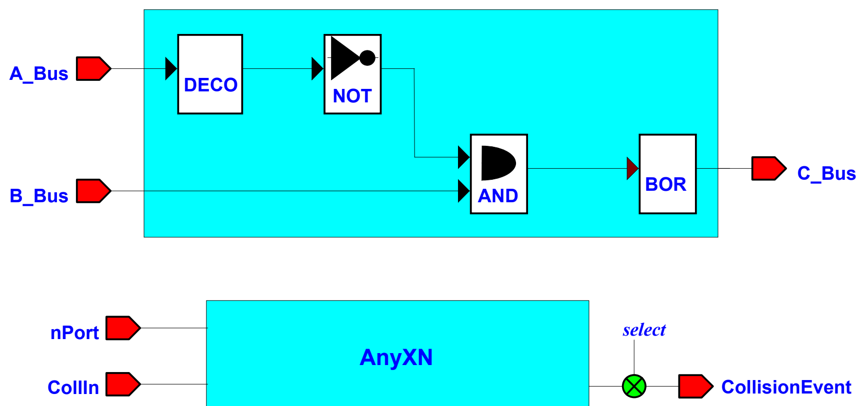
© 2002 Dr. James P. Davis

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).



Creating User-defined Macro-functions



- RTL macro-functions: contain over 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)

Using Boolean Operators vs Macro-functions

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

s1 / c <- a & b / a, b → AND → c	s1 / z <- AND(x, y) / x, y → AND → z
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.	**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.

Carry/Borrow Bit with Arithmetic Macro-functions

macro-function unit: Abus (8), Bbus (8), Carry in (this is Dbus) (1) → + → Cbus (9), Carry out (this is Cbus[8]) (1). **flowdiagram:** state2 — Cbus <- ADD(Abus, Bbus, Dbus); Cbus[8] # test for carry out (0 / 1)	**macro-function unit:** Abus (8), Bbus (8), Borrow (this is Dbus) (1) → − → Cbus (9), Overflow (this is Cbus[8]) (1). **flowdiagram:** state2 — Cbus <- SUB(Abus, Bbus, Dbus) # subtract Bbus and Dbus from Abus; Cbus[8] # test for overflow (0 / 1)
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. 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. The ADD and INCR macros work with buses up to 127 bits.	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. 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 The SUB and DECR macros are fully scalable, in that they can work with buses up to 127 bits.

Executable ASM Method

Synchronization

The timing semantics of Reset, Clock, and synchronized concurrent threads

Eliminating Latches in Asynchronous Datapath

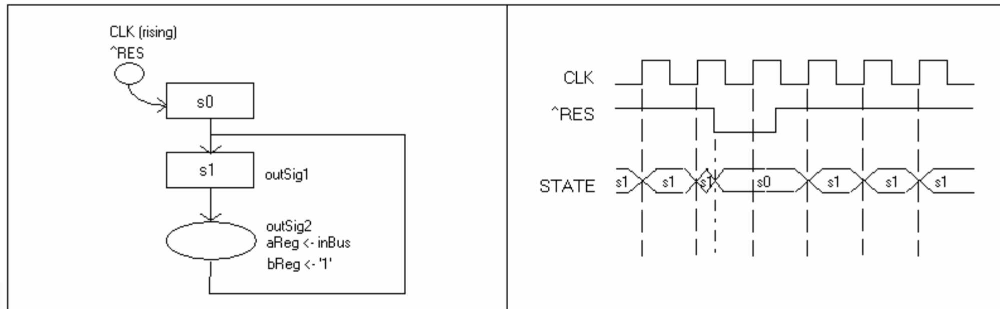
Asynchronous circuit behavior can be obtained by using buses with unregistered output. The resultant circuit created will not have register units for the buses that are assigned on the state transitions. Instead, these buses will be realized as latches or wires.	Extraneous latches can be eliminated from the design by two means. First, the "No Latch" option can be used during HDL code generation. If the resultant code isn't optimal for the application, the designer can get optimal results by completely assigning values to the buses in every unregistered state transition of the flowdiagram.

Using the Asynchronous Reset

Nimbus provides a pre-defined "Reset" signal. Reset generally defines the power-up or re-start event, on which most of the synchronous register elements are tied. It is defined asynchronous to the clock, in that it can occur at any time, and does not require synchronization before the system is to respond to the event.

Reset has some specific "pulse and hold" behavior, in that the signal is asserted or de-asserted for a certain amount of time (i.e., pulsed) before being held. During the time that the signal is in its "hold" state, the execution of the circuit is enabled. When the signal is next pulsed, the system immediately aborts its current machine cycle and transfers control to the associated Reset state.

The designer would typically use Reset to initialize each ASM "thread" in a concurrent system description. Note that only one Reset can be used in a given ASM thread, through each concurrent thread may have a Reset event defined.



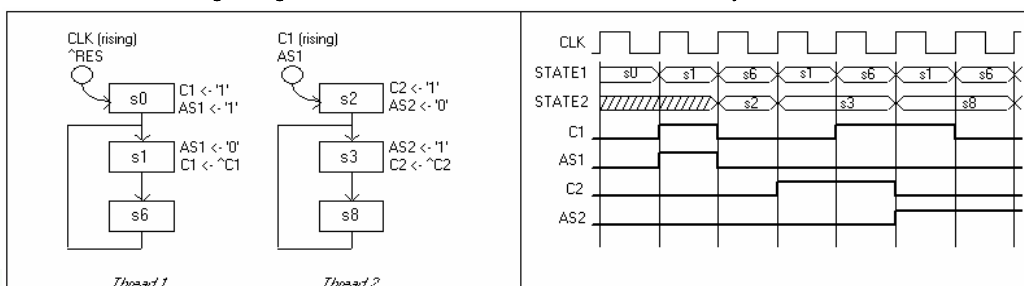
© 2002 Dr. James P. Davis

Modeling Synchronous Enabling Events

In Nimbus, designers 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.



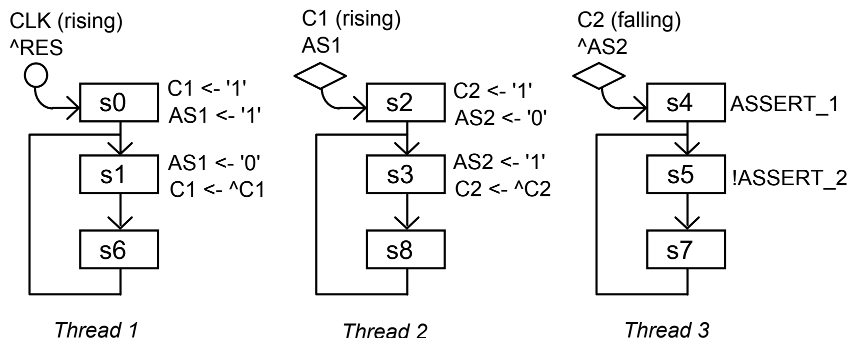
© 2002 Dr. James P. Davis

Synchronous vs Asynchronous Enable Events

Enable Events are used is 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.

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.	**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. This can be extended in v2.0 release to allow for non-standard register device types.

Enable Events for Synchronization



Modeling Concurrency:

- Multiple model FSM "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.

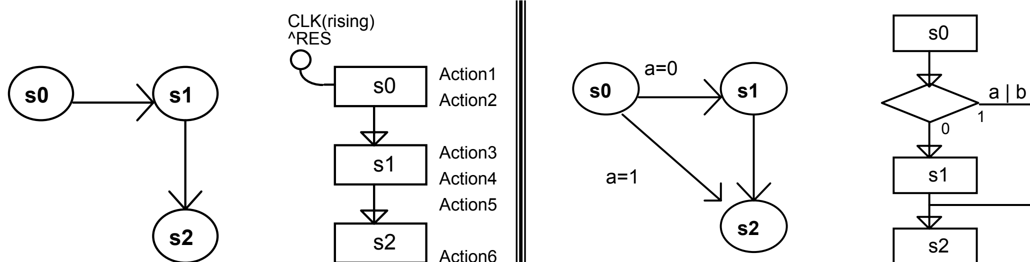
II. Competition
- shared resources (for example, bus arbitration).
- explicit use of other concurrent processes, components, or entities to model the arbitration protocol.

Executable ASM Method

Control Structures

The inventory of algorithmic control constructs and comparing them to state diagrams

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

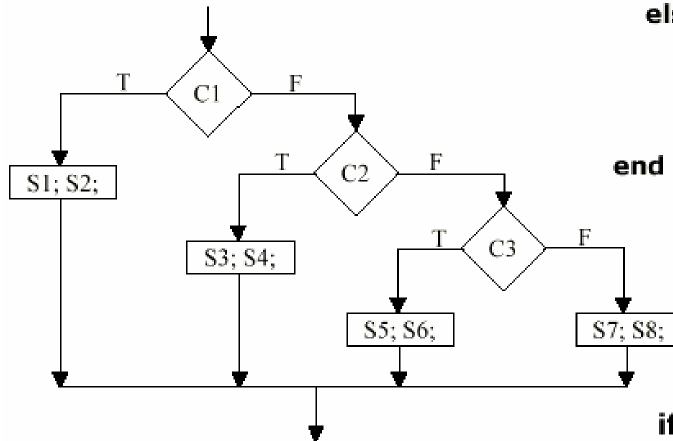
Binary branching is represented using a condition "diamond".

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

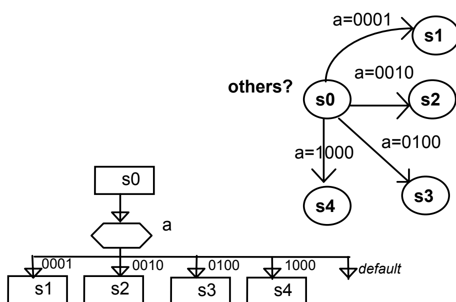


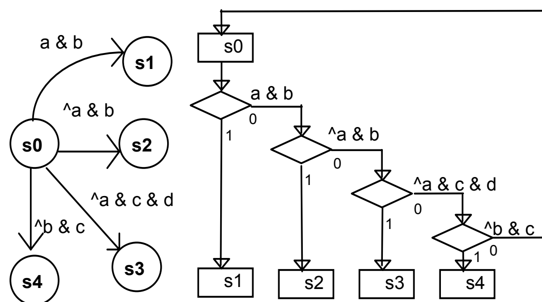

```
if (C1) then S1; S2;
   else if (C2) then S3; S4;
            else if (C3) then S5; S6;
                   else S7; S8;
                end if;
         end if;
end if;
```

Source: Roth © 1998 PWS Publishing

```
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.

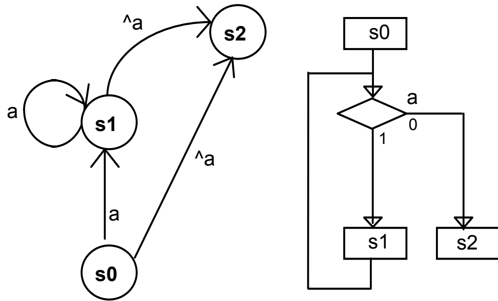
State diagrams have no such mechanism, and thus are ambiguous.

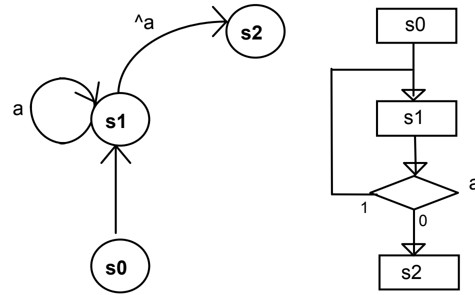
Selection: Multi-way Branching, Multi-variable

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.

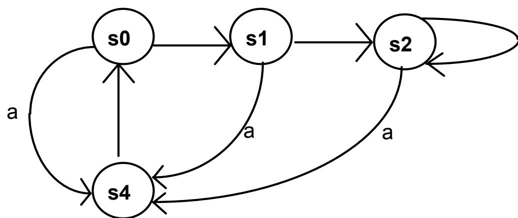
Repetition: "Repeat-Until" Control Loop

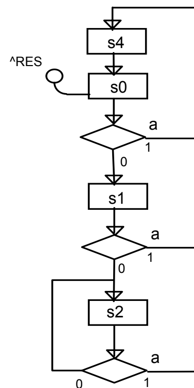
Placement of the decision "diamond" defines the type of looping construct, and the type of control behavior.

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

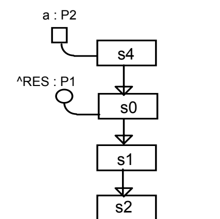
This type of control construct is used often for counting the number of loop iterations, where you test the condition *after* completing each execution of the loop, requiring execution of at least the first pass.

ASM vs. State Diagrams – Synchronous Transfer of Control





explicit conditional tests



synchronous enable event

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.

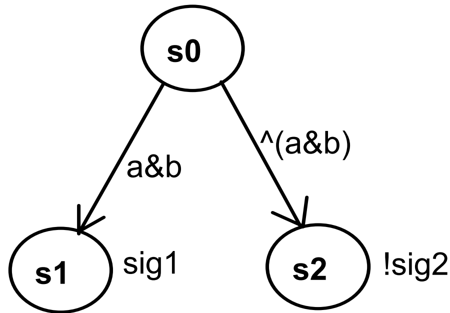
Also, if State transitions have additional conditions, it isn't clear what happens when conditions aren't met (incomplete specification). However, this may be handled by modeling "looping" on a state in some cases.

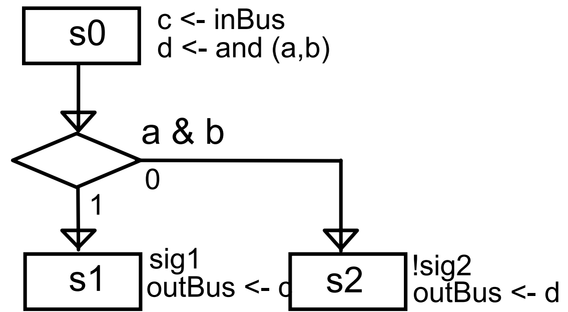
Control Interrupt Schemes – ASM Diagrams:

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.

ASM vs. State Diagrams – Moore Machine Actions

Moore machines generate control outputs that are solely dependent on the present state of the FSM.





Moore Machines in State Diagrams:

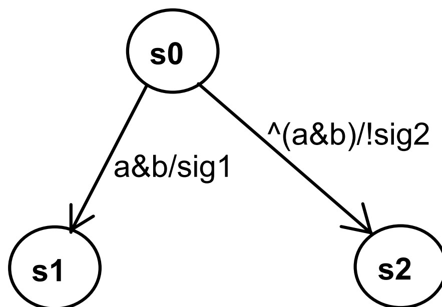
State diagrams model actions as attached to state symbols. Generally, they support control signal actions only, and not the expression of data path assignments. Also, only the statement of synchronous actions are supported.

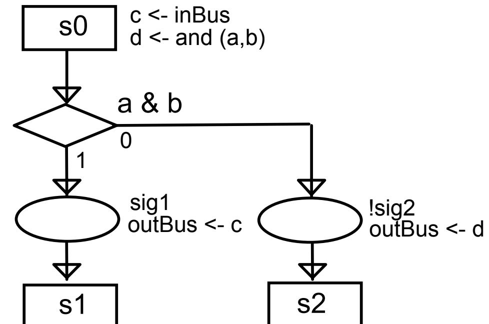
Moore Machines in ASM diagrams:

ASM diagrams model actions as expressions attached to state boxes. The actions can be *signal assertions*, *bus assignments* or *macro-function assignments*. Actions can be either synchronous or asynchronous, depending whether the buses are registered or non-registered (i.e., specified as being a “latch” or a “wire”).

ASM vs. State Diagrams – Mealy Machine Actions

Mealy machines generate control outputs that are dependent both on input values to the system and on the present state of the FSM.





Mealy Machine in State Diagrams:

State diagrams model actions that are gated by both the present state and inputs. This is represented by attaching conditional outputs on the transition arcs between states.

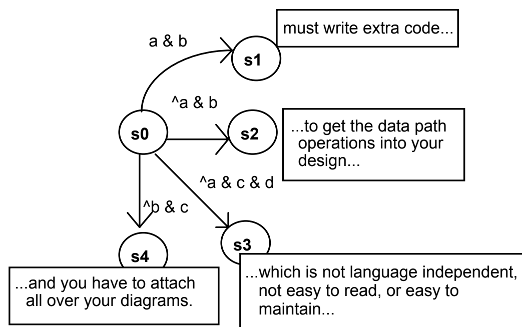
This is somewhat limiting because you can't accurately distinguish behavior when inputs change during the time the state is active, rather than during the state transition, since the FSM spends some amount of time in the state.

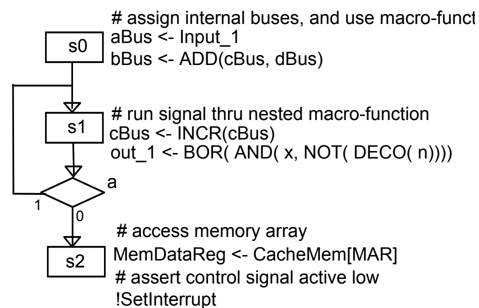
Mealy Machine in ASM diagrams:

ASM diagrams model actions as dependent on a condition being satisfied, represented by a decision "diamond" followed by an output "oval".

Once the condition is satisfied, the action is scheduled (whether an *assertion* or an *assignment*), and executed. The timing of execution is dependent on whether the buses referenced in the actions are registered or non-registered (i.e., “latch” or “wire”).

ASM vs. State Diagrams – Data Path Operations





Modeling Data Path in State Diagrams:

Most State diagram approaches don't support data path operators, but only support control through FSM model, generating explicit control signals for circuit.

Some approaches provide some operators (such as assignment), but don't support general-purpose data path macros.

Other approaches only allow you to express data path by writing it directly in HDL code; but then, what's the point of having the graphics, since the idea is to not have to write code?

Modeling Data Path in ASM diagrams:

Executable ASM diagrams (that we will use with Nimbus) directly support the modeling of data path and control path in a unified representation. The control path information is captured in the topology of the ASM diagram. The data path is captured in the RTL expressions attached to states and conditional outputs.

Nimbus provides a library of scalable and nested macro-functions for common data path functions. These map to both VHDL and Verilog, and are fully synthesizable/mappable into library elements.

© 2002 Dr. James P. Davis