

# CSCE 611: Sequential Logic and Finite State Machine Controllers

Instructor: Jason D. Bakos

---

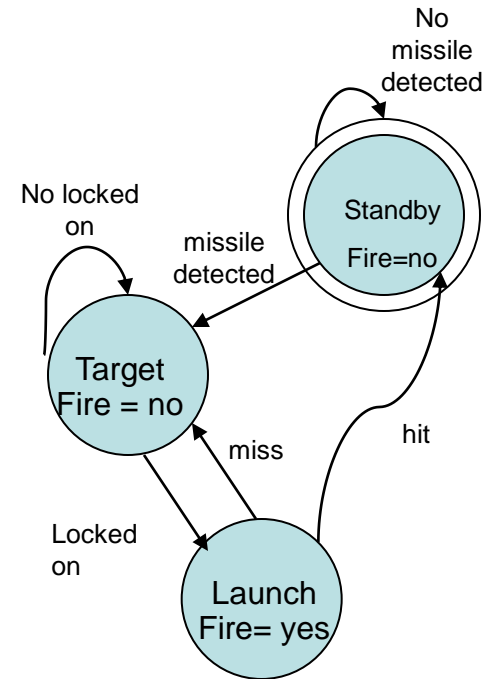
# Sequential Logic

---

- Combinational logic
  - Output =  $f(\text{input})$
- Sequential logic
  - Output =  $f(\text{input}, \text{input history})$
  - Involves use of memory elements
    - E.g. Registers
- Sequential logic can be designed as a *finite state machine*

# Finite State Machines

- FSMs are made up of:
  - input set
  - output set
  - states (one is start state)
  - transitions
- FSMs are used for controllers

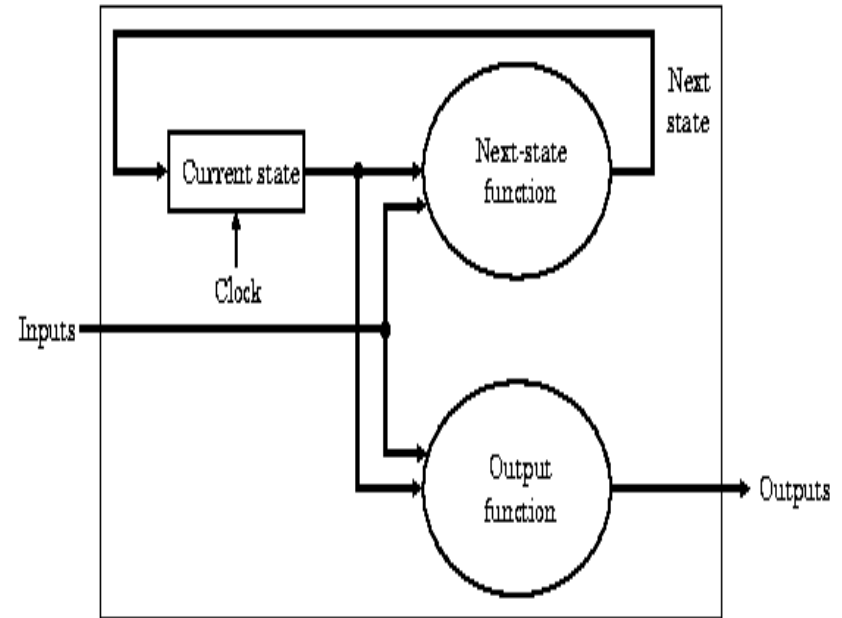


Inputs = *missile detected, locked on, hit, miss*

Output alphabet {*fire*}

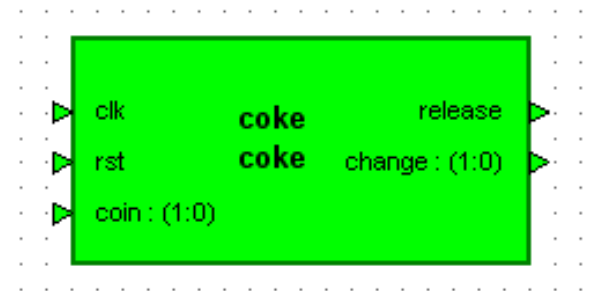
# Finite State Machines

- Registers
  - Hold current state value
- Output logic
  - Encodes output of state machine
    - **Moore-style** (synchronous output)
      - Output =  $f(\text{current state})$
    - **Mealy-style** (asynchronous output)
      - Output =  $f(\text{current state}, \text{input})$ 
        - » Output values associated with state transitions
- Next-state logic
  - Encodes transitions from each state
  - Next state =  $f(\text{current state}, \text{input})$
- Synchronous state machines transition on clock edge
- RESET signal to return to start state (“sanity state”)
- Note that state machines are triggered out-of-phase from the input and any memory elements they control



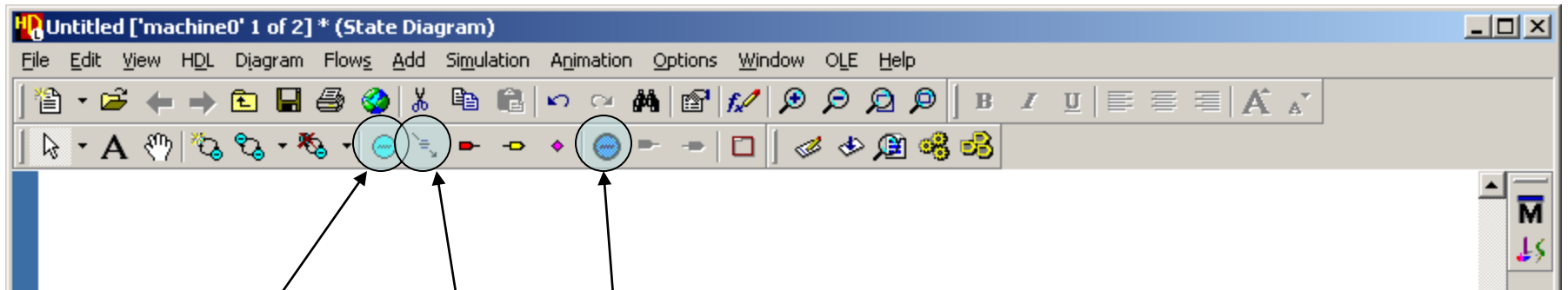
# Example

- Design a coke machine controller
  - Releases a coke after 35 cents entered
  - Accepts nickels, dimes, and quarters, returns change
  - Inputs
    - Driven for 1 clock cycle while coin is entered
    - COIN = { 00 for none, 01 for nickel, 10 for dime, 11 for quarter }
  - Outputs
    - Driven for 1 clock cycle
    - RELEASE = { 1 for release coke }
    - CHANGE releases change, encoded as COIN input



# Example

- We'll design this controller as a state diagram view in FPGA Advantage



Add new state  
(First is start state)

Add new transition

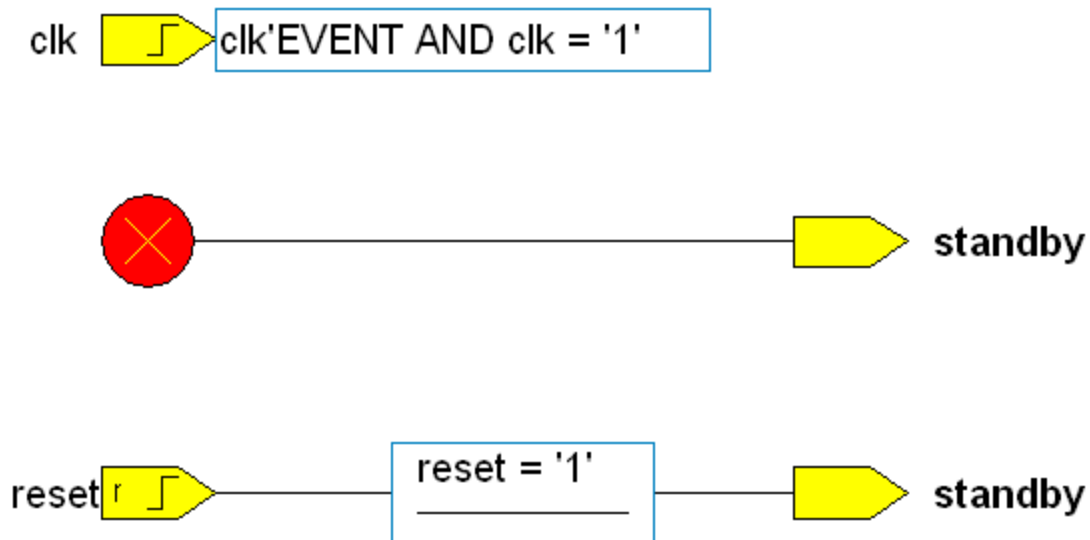
Add new *hierarchical*  
state

Note: transitions into and out of a hierarchical state are implicitly ANDed with the internal entrance and exit conditions



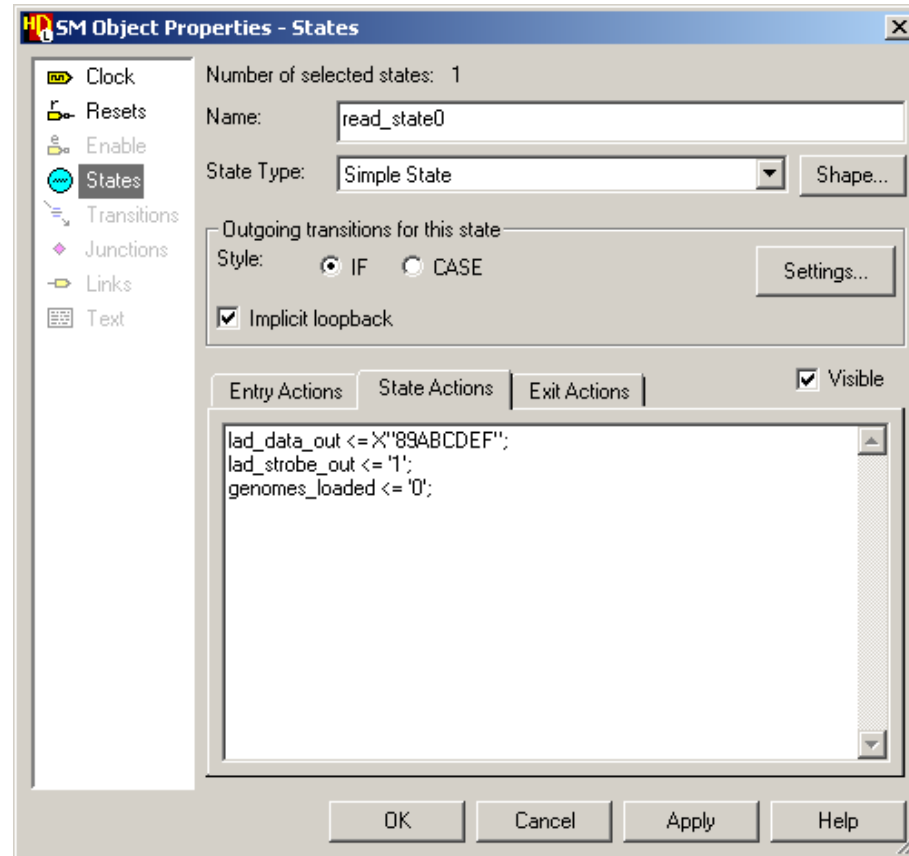
# Example

- Go to state diagram properties to setup the state machine...



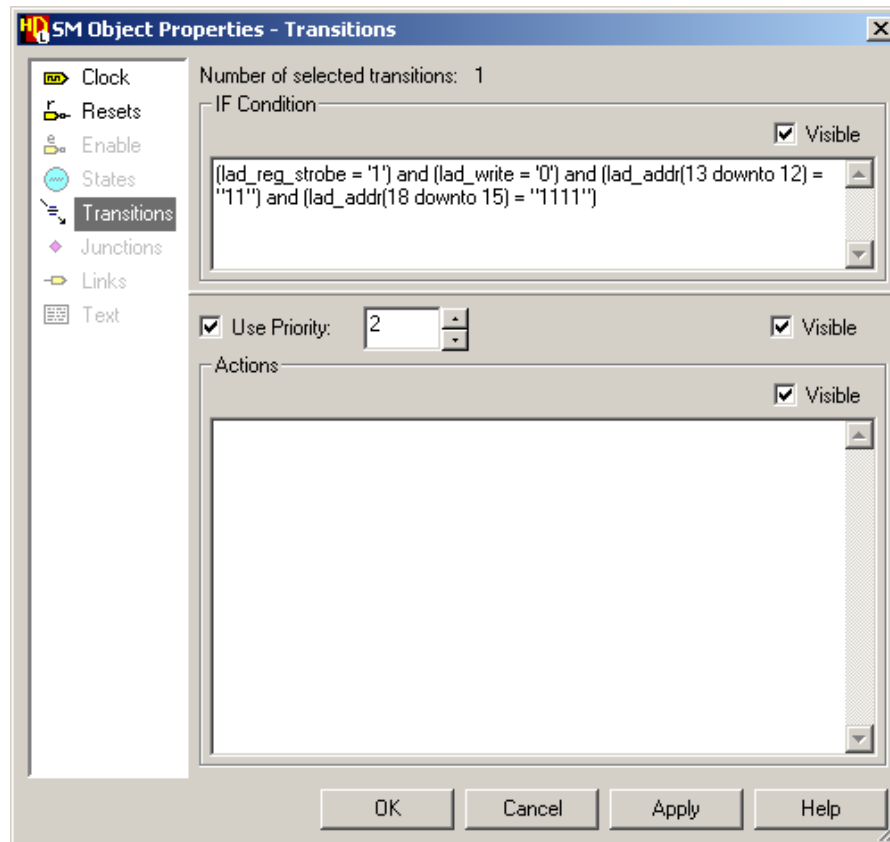
# Example

- Specify the output values for each state in the state properties

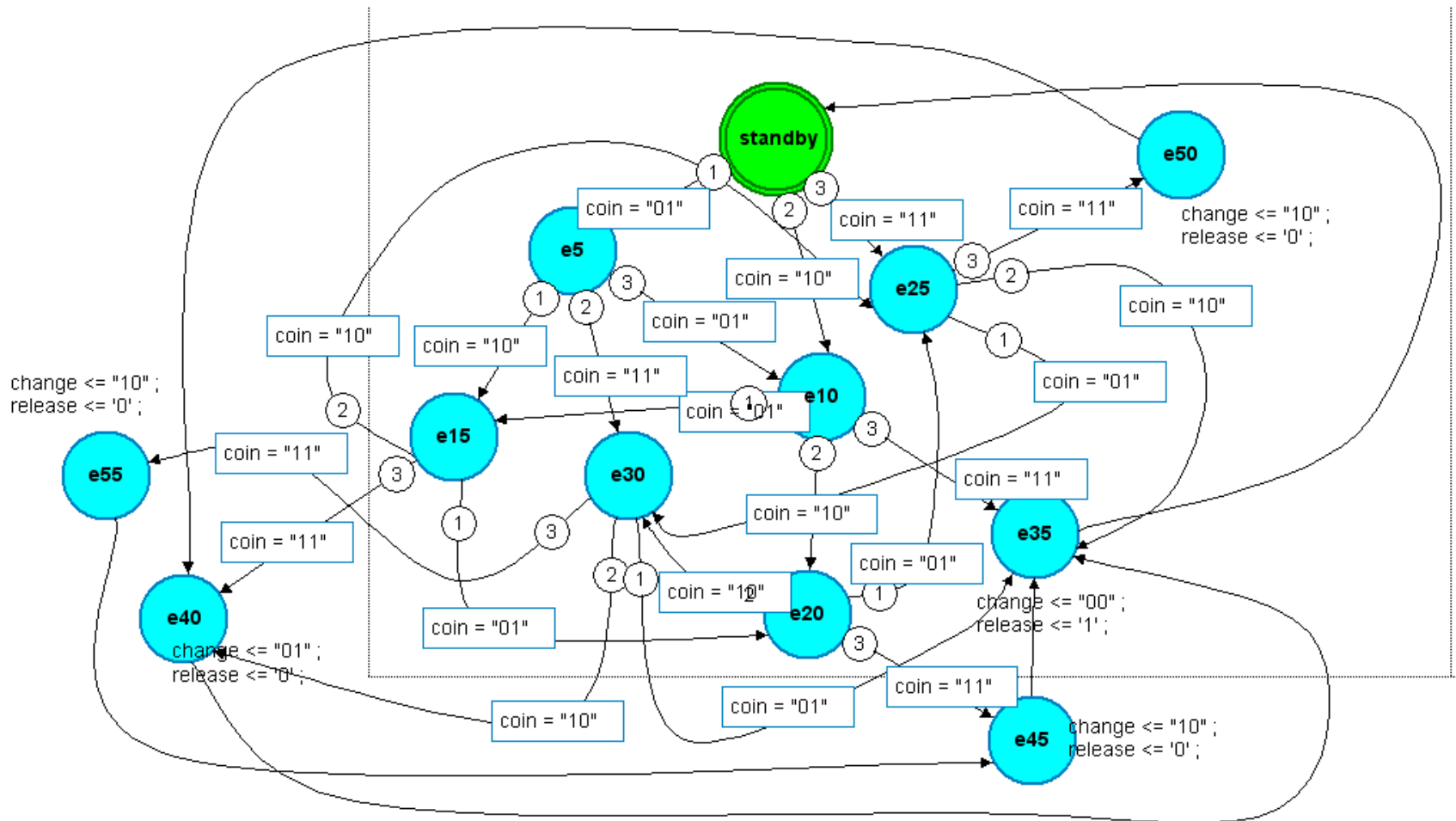


# Example

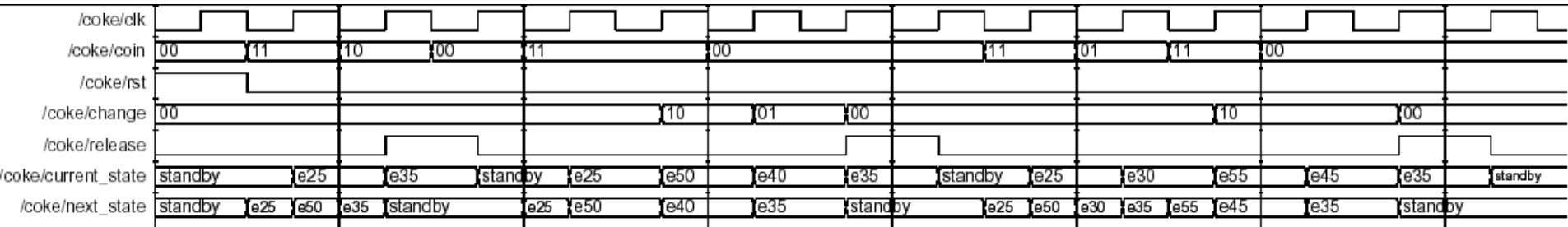
- Specify the transition conditions and priority in the transition properties



# Example



# Example



---

# State Machine VHDL

---

- Let's take a look at the VHDL for the FSM
  - Enumerated type: **STATE\_TYPE** for states
  - Internal signals, **current\_state** and **next\_state**
  - **clocked** process handles reset and state changes
  - **nextstate** process assigns next\_state from current\_state and inputs
    - Implements next state logic
    - Syntax is case statement
  - **output** process assigns output signals from current\_state
    - Might also use inputs here



---

# Types

---

```
ARCHITECTURE fsm OF coke IS

  -- Architecture Declarations
  TYPE STATE_TYPE IS (
    standby,
    e5,
    e10,
    e25,
    e30,
    e15,
    e20,
    e35,
    e50,
    e40,
    e55,
    e45
  );

  -- Declare current and next state signals
  SIGNAL current_state : STATE_TYPE ;
  SIGNAL next_state : STATE_TYPE ;
```



---

# "clocked" Process

---

```
-----  
clocked : PROCESS (  
    clk,  
    rst  
)  
-----  
BEGIN  
    IF (rst = '1') THEN  
        current_state <= standby;  
        -- Reset Values  
    ELSIF (clk'EVENT AND clk = '1') THEN  
        current_state <= next_state;  
        -- Default Assignment To Internals  
  
    END IF;  
  
END PROCESS clocked;
```





---

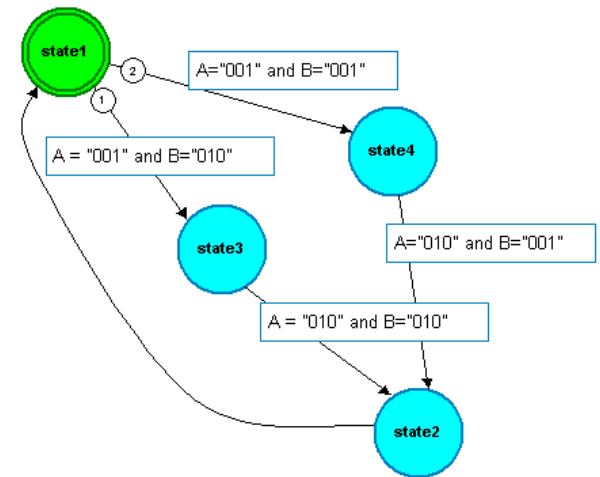
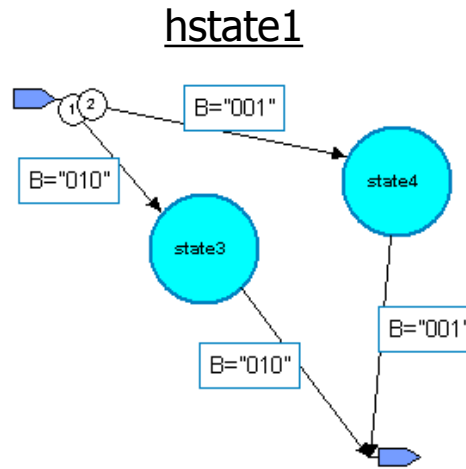
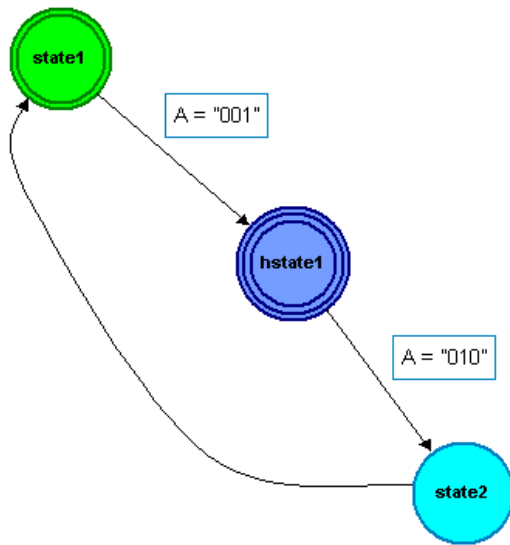
# "output" process

---

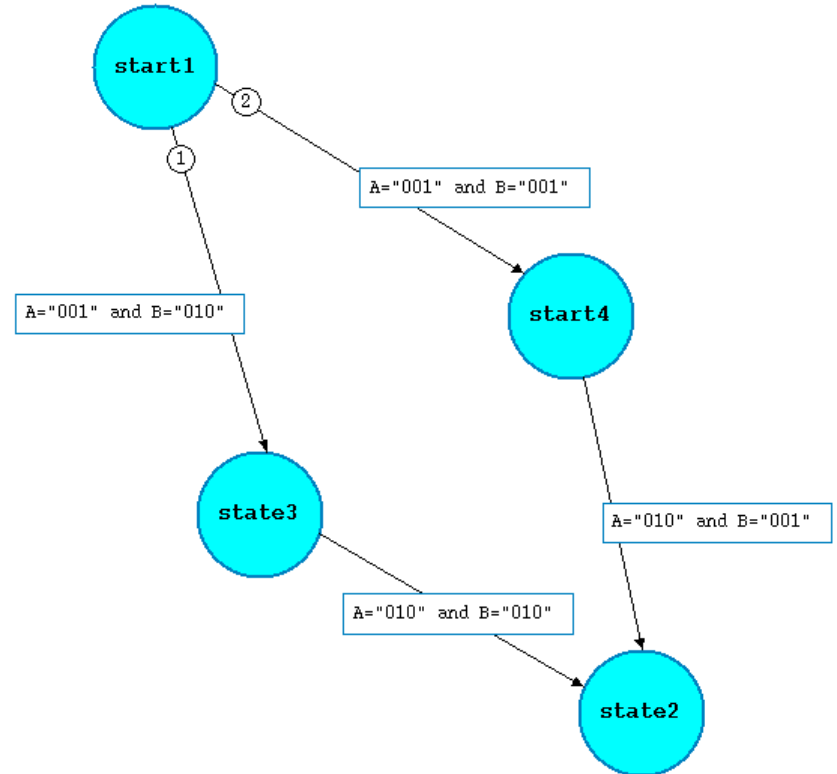
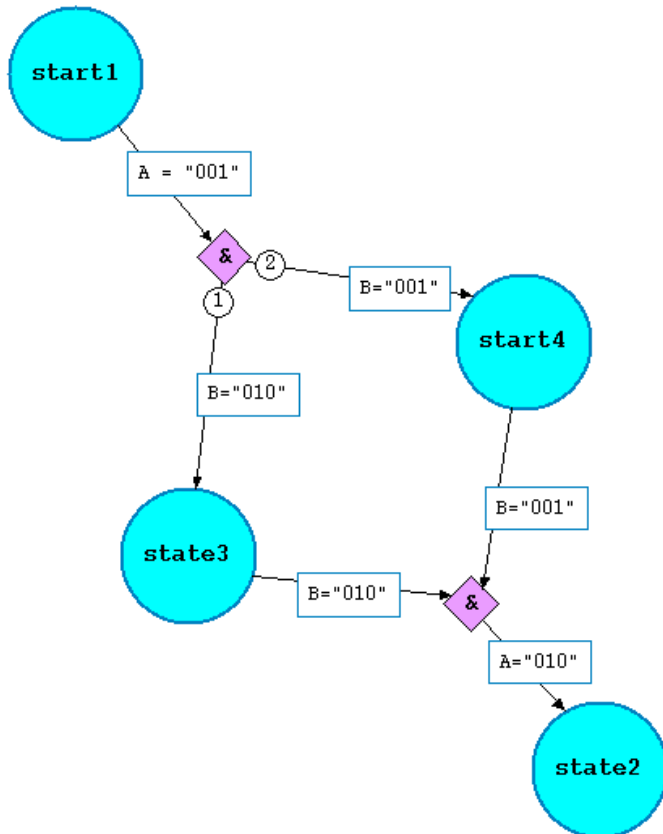
```
-----  
output : PROCESS (  
    current_state  
)  
-----  
BEGIN  
    -- Default Assignment  
    change <= "00";  
    release <= '0';  
    -- Default Assignment To Internals  
  
    -- Combined Actions  
    CASE current_state IS  
    WHEN standby =>  
        change <= "00" ;  
        release <= '0' ;  
    WHEN e5 =>  
        change <= "00" ;  
        release <= '0' ;  
    WHEN e10 =>  
        change <= "00" ;  
        release <= '0' ;  
    WHEN e25 =>  
        change <= "00" ;  
        release <= '0' ;  
    WHEN e30 =>  
        change <= "00" ;  
        release <= '0' ;  
    WHEN e15 =>  
        change <= "00" ;  
        release <= '0' ;  
  
    ...
```



# Hierarchical States



# Junctions



---

# Lab 4 Notes

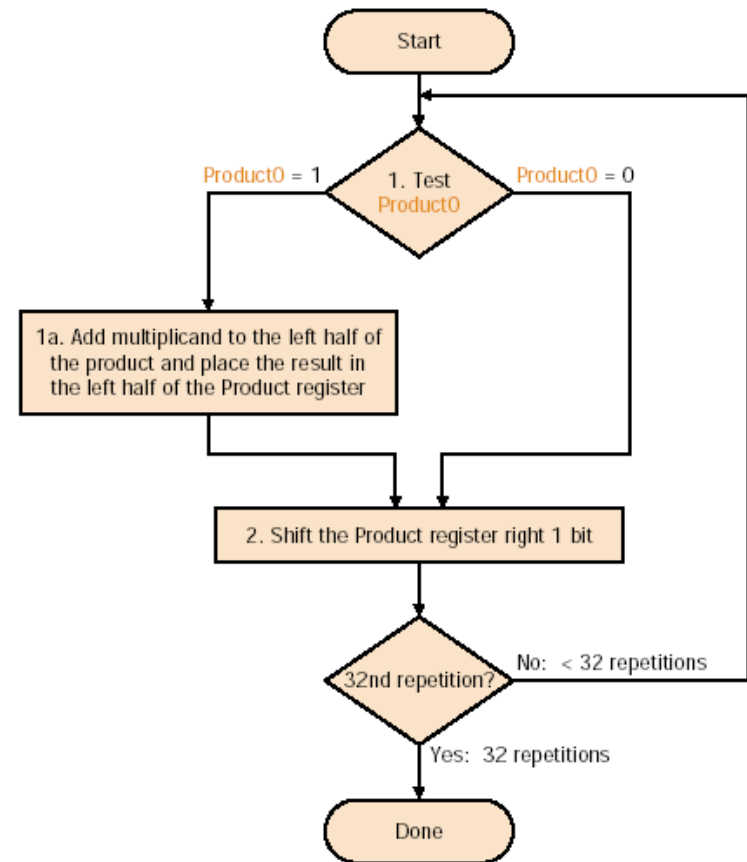
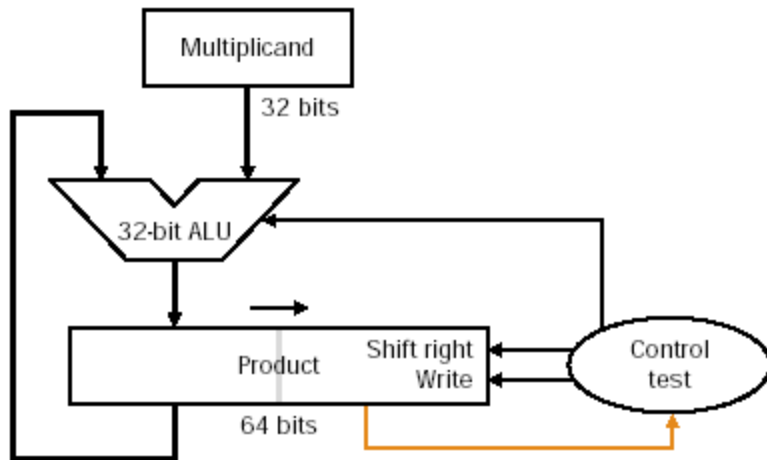
---

$$\begin{array}{r} 1000 \\ \times 1001 \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 1001000 \end{array}$$

← multiplicand  
← multiplier  
← product



# Binary Multiplication



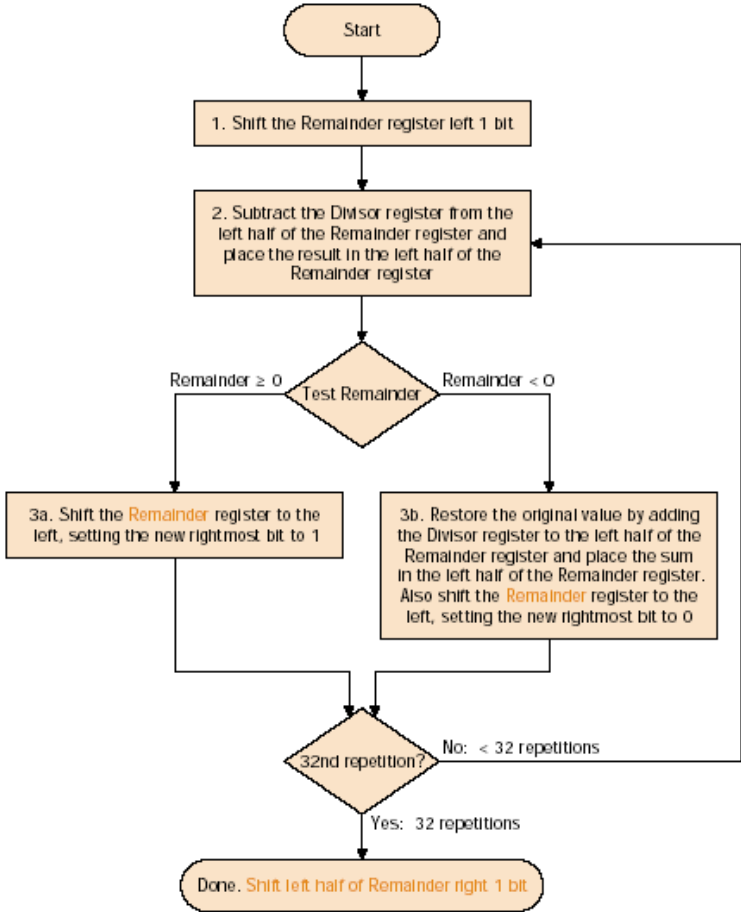
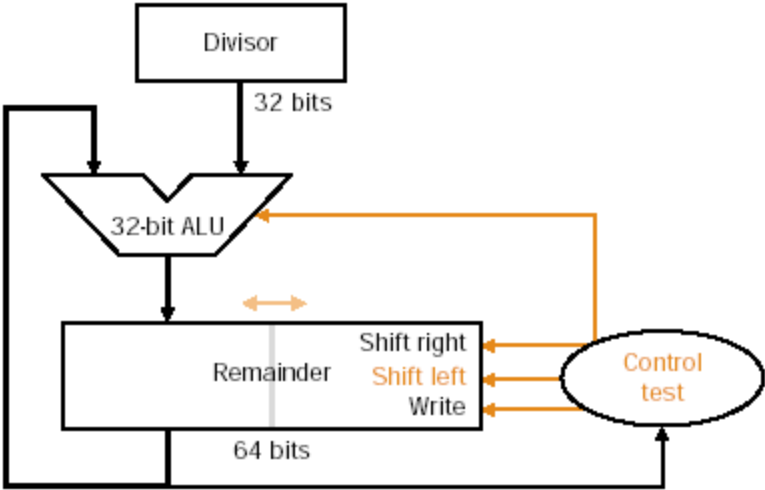
# Multiplication Example

<b>multiplicand register (MR)</b>	<b>product register (PR)</b>	<b>next action</b>	<b>it</b>
1001	0000 0101	LSB of PR is 1, so $PR[7:4]=PR[7:4]+MR$	1
1001	1001 0101	shift PR	1
1001	0100 1010	LSB of PR is 0, so shift	2
1001	0010 0101	LSB of PR is 1, so $PR[7:4]=PR[7:4]+MR$	3
1001	1011 0101	shift PR	3
1001	0101 1010	LSB of PR is 0, so shift	4
1001	0010 1101	PR is 45, done!	X





# Binary Division



# Division Example

divisor register (DR)	remainder register (RR)	next action	it
0100	0000 1011	shift RR left 1 bit	0
0100	0001 0110	$RR[7:4]=RR[7:4]-DR$	1
0100	1101 0110	$RR < 0$ , so $RR[7:4]=RR[7:4]+DR$	1
0100	0001 0110	shift RR to left, shift in 0	1
0100	0010 1100	$RR[7:4]=RR[7:4]-DR$	2
0100	1110 1100	$RR < 0$ , so $RR[7:4]=RR[7:4]+DR$	2
0100	0010 1100	shift RR to left, shift in 0	2
0100	0101 1000	$RR[7:4]=RR[7:4]-DR$	3
0100	0001 1000	$RR \geq 0$ , so shift RR to left, shift in 1	3
0100	0011 0001	$RR[7:4]=RR[7:4]-DR$	4
0100	1111 0001	$RR < 0$ , so $RR[7:4]=RR[7:4]+DR$	4
0100	0011 0001	shift RR to left, shift in 0	4
0100	0110 0010	shift $RR[7:4]$ to right	
0100	0011 0010	done, quotient=2, remainder=3	

