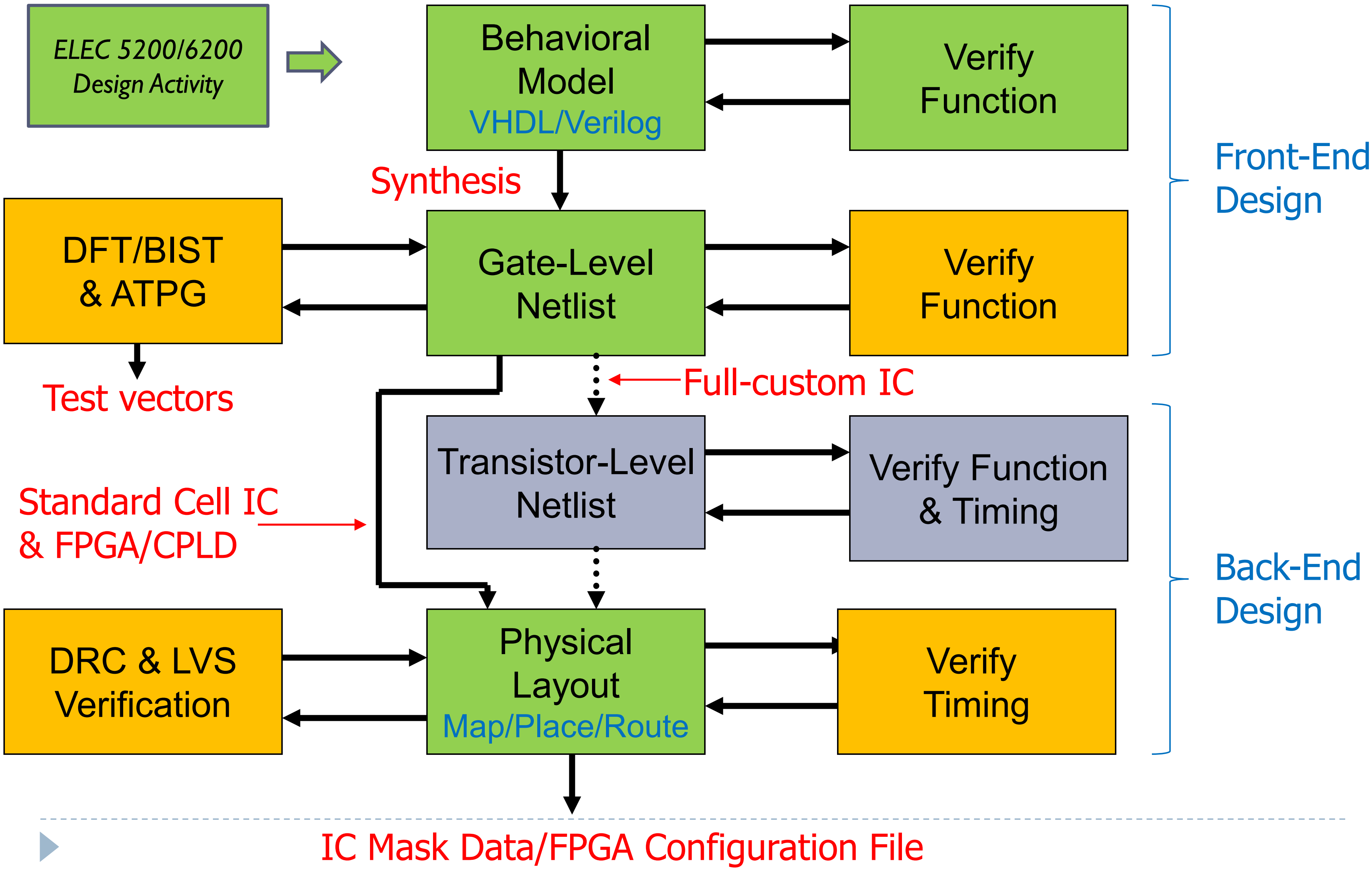
Modeling & Simulating ASIC Designs with VHDL

Reference: *Application Specific Integrated Circuits*
M. J. S. Smith, Addison-Wesley, 1997 Chapters 10 & 12

Online version: http://www10.edacafe.com/book/ASIC/ASICs.php

VHDL resources from other courses:
ELEC 4200: http://www.eng.auburn.edu/~strouce/elec4200.html
ELEC 5250/6250: http://www.eng.auburn.edu/~nelsovp/courses/elec5250_6250/

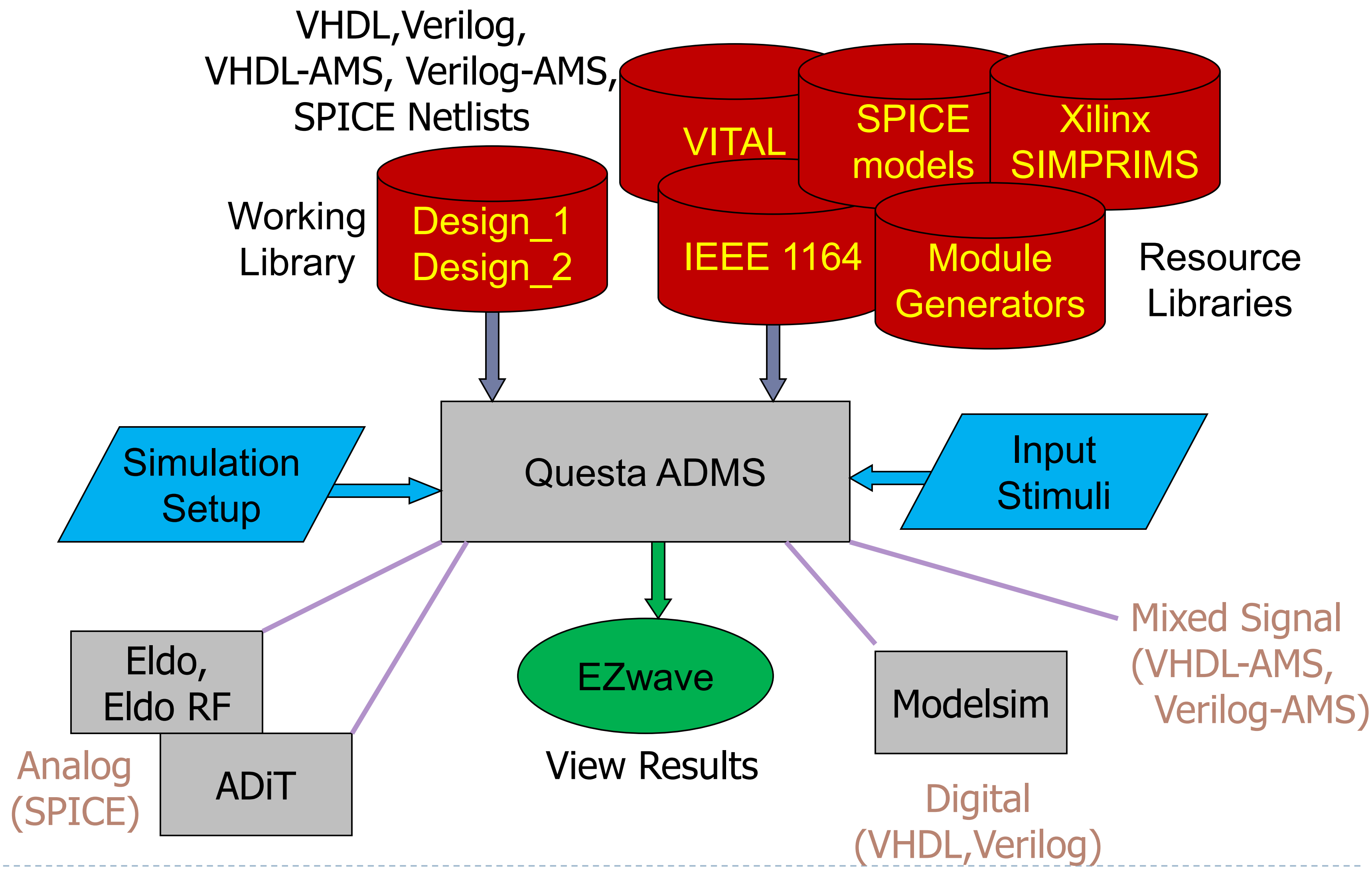
Digital ASIC Design Flow



ASIC CAD tools available in ECE

- Modeling and Simulation
  - Active-HDL (Aldec)
  - Questa ADMS = Questa+Modelsim+Eldo+ADiT (Mentor Graphics)
  - Verilog-XL, NC_Verilog, Spectre (Cadence)
- Design Synthesis (digital)
  - Leonardo Spectrum (Mentor Graphics)
  - Design Compiler (Synopsys), RTL Compiler (Cadence)
  - **FPGA: Xilinx ISE; CPLD: Altera Quartus II**
- Design for Test and Automatic Test Pattern Generation
  - Tessent DFT Advisor, Fastscan, SoCScan (Mentor Graphics)
- Schematic Capture & Design Integration
  - Design Architect-IC (Mentor Graphics)
  - Design Framework II (DFII) - Composer (Cadence)
- Physical Layout
  - IC Station (Mentor Graphics)
  - SOC Encounter, Virtuoso (Cadence)
  - **Xilinx ISE/Altera Quartus II – FPGA/CPLD Synthesis, Map, Place & Route**
- Design Verification
  - Calibre DRC, LVS, PEX (Mentor Graphics)
  - Diva, Assura (Cadence)

Questa ADMS
Analog, Digital, Mixed-Signal Simulation



Hardware Description Languages

- **VHDL** = VHSIC Hardware Description Language
  (VHSIC = Very High Speed Integrated Circuits)
  - Developed by DOD from 1983 – based on ADA
  - IEEE Standard 1076-1987/1993/2002/2008
  - Based on the ADA language
- **Verilog** – created in 1984 by Philip Moorby of Gateway Design Automation (merged with Cadence)
  - IEEE Standard 1364-1995/2001/2005
  - Based on the C language
  - IEEE P1800 "System Verilog" in voting stage & will be merged with 1364

Other VHDL Standards

- **1076.1**–1999: VHDL-AMS (Analog & Mixed-Signal Extensions)
- **1076.2**–1996: Std. VHDL Mathematics Packages
- **1076.3**-1997: Std. VHDL Synthesis Packages
- **1076.4**-1995: Std. VITAL Modeling Specification (VHDL Initiative Towards ASIC Libraries)
- **1076.6**-1999: Std. for VHDL Register Transfer Level (RTL) Synthesis
- **1164**-1993: Std. Multivalue Logic System for VHDL Model Interoperability

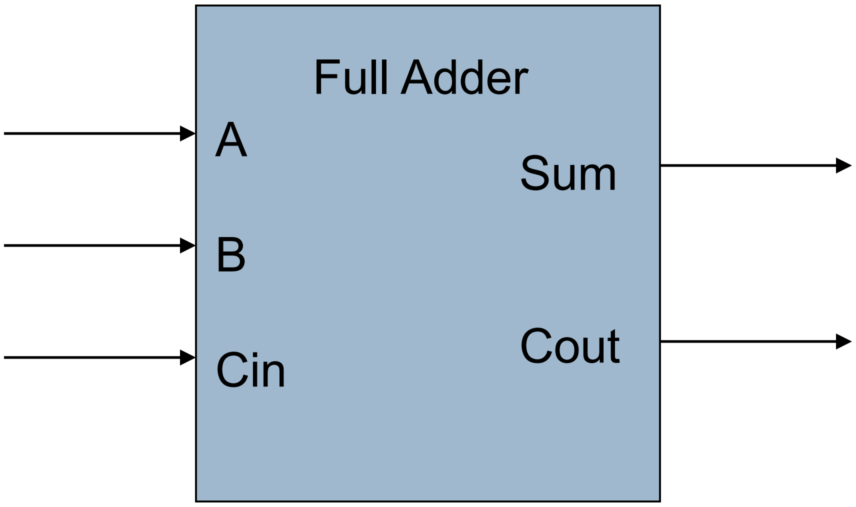
HDLs in Digital System Design

- Model and document digital systems
  - Hierarchical models
    - System, RTL (Register Transfer Level), Gates
  - Different levels of abstraction
    - Behavior, structure
- Verify circuit/system design via simulation
- Automated synthesis of circuits from HDL models
  - using a technology library
  - output is primitive cell-level netlist (gates, flip flops, etc.)

Anatomy of a VHDL model

- "Entity" describes the external view of a component
- "Architecture" describes the internal behavior and/or structure of the component
- Example:

1-bit full adder



Example: 1-Bit Full Adder

```
entity full_add1 is
   port (                        -- I/O ports
          a:     in   bit;       -- addend input
          b:     in   bit;       -- augend input
          cin:   in   bit;       -- carry input
          sum:   out bit;        -- sum output
          cout:  out bit);       -- carry output
end full_add1 ;
```

I/O Port Declarations

Comments follow double-dash

Type of signal

Signal name

Signal direction (mode)

Port Format: name: direction data_type;

- Direction
  - in - driven into the entity from an external source (can read, but not update within architecture)
  - out - driven from within the entity

    (can drive but not read within architecture)
  - inout – bidirectional; drivers both within the entity and external

    (can read or write within architecture)
  - buffer – like "out" but can read and write
- Data_type: any scalar or aggregate signal type

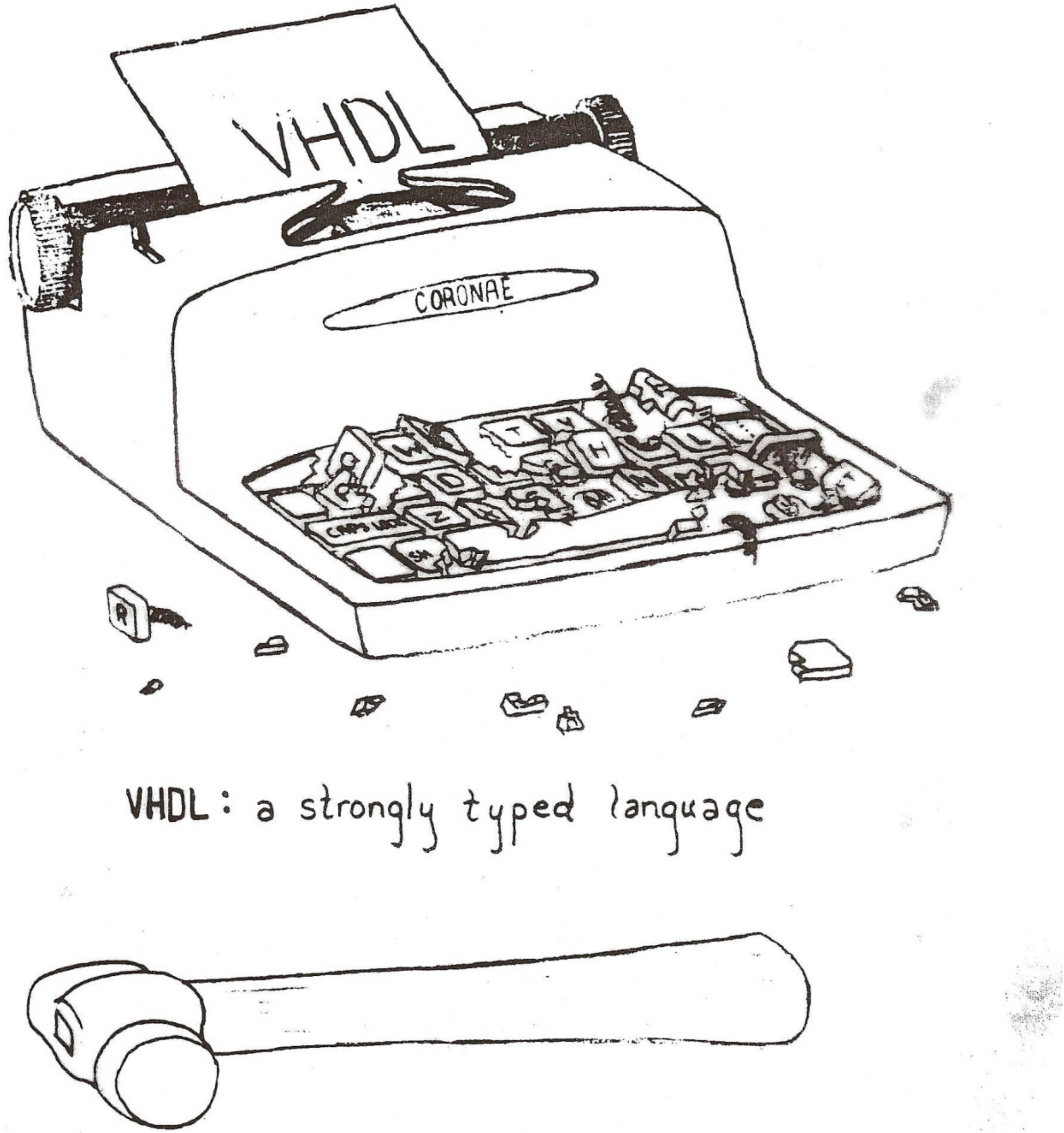
8-bit adder - entity

```
-- Interconnect 8 1-bit adders for 8-bit adder
entity Adder8 is
port (A, B: in BIT_VECTOR(7 downto 0);
        Cin: in BIT;
        Cout: out BIT;
        Sum: out BIT_VECTOR(7 downto 0));
end Adder8;
```

Built-in Data Types

- Scalar (single-value) signal types:
  - bit – values are '0' or '1'
  - boolean – values are TRUE and FALSE
  - integer - values $[-2^{31} \cdots +(2^{31}-1)]$ on 32-bit host
- Aggregate (multi-value) signal types:
  - bit_vector – array of bits

```
signal b: bit_vector(7 downto 0);
signal c: bit_vector(0 to 7);
b <= c after 1 ns;
c <= "01010011";
```


VHDL
CORONAE
CAPS LOCK
VHDL : a strongly typed language

Numeric literals

- String of scalar values:
  - '1' - scalar value
  - "11010101" - aggregate value (array of scalar values)
  - "1101_0101" - underline is ignored (improves readability)
- Based literals: designate number base between 2 and 16
  - 1st Format = base#digits#
    - 2#11010101# - base 2
    - 16#D5# - base 16
    - 8#325"
    - 16#2E4F_327F# - underline ignored
  - 2nd Format = base_specifier"digits"
    - B"11010101" = B"1101_0101" - binary
    - X"2E4F327F" = X"2E4F_327F" - hexadecimal
    - O"325" - letter "O" for octal

VHDL "Package"

- Package = file of type definitions, functions, procedures to be shared across VHDL models
  - User/vendor created
  - Standard lib's/3rd party – usually distributed in "libraries"
  - Example: IEEE libraries, Xilinx/Altera "primitives" libraries, etc.

package *name* ***is***

--type, function, procedure declarations

end package *name;*

package body *name* ***is*** *-- only if functions to be defined*

-- function implementations

end package body *name;*

IEEE std_logic_1164 package

```
-- Provides additional logic states as data values
package Part_STD_LOGIC_1164 is
 type STD_ULOGIC is ( 'U',   -- Uninitialized
                      'X',   -- Forcing Unknown
                      '0',   -- Forcing 0
                      '1',  -- Forcing 1
                      'Z',  -- High Impedance
                      'W',  -- Weak Unknown
                       'L',  -- Weak 0
                      'H',  -- Weak 1
                       '-' -- Don't Care);

type STD_ULOGIC_VECTOR is array (NATURAL range <>) of STD_ULOGIC;
```

Bus resolution

- subtype STD_LOGIC is resolved STD_ULOGIC;
  - Most common data type in system design
  - Bus resolution function specifies value when there are multiple drivers of this type

function resolved (s : STD_ULOGIC_VECTOR) return STD_ULOGIC;

- type STD_LOGIC_VECTOR is array (NATURAL range <>) of STD_LOGIC;
  - Use for multi-bit values in RTL designs

Bus resolution function

std_logic type includes a "bus resolution function" to determine the signal state where there are multiple drivers

Driver B value

Driver A value

	'0'	'1'	'Z'	'X'
'0'	'0'	'X'	'0'	'X'
'1'	'X'	'1'	'1'	'X'
'Z'	'0'	'1'	'Z'	'X'
'X'	'X'	'X'	'X'	'X'

Resolved Bus Value

Example: 1-Bit Full Adder

```
library ieee;                          --supplied library
use ieee.std_logic_1164.all;  --package of definitions
entity full_add1 is
   port (                                   -- I/O ports
         a:      in   std_logic;        -- addend input
         b:      in   std_logic;        -- augend input
         cin:    in   std_logic;        -- carry input
         sum:    out std_logic;        -- sum output
         cout:   out std_logic);       -- carry output
end full_add1 ;
```

Example: 8-bit full adder

```
-- 8-bit inputs/outputs
library ieee;                          --supplied library
use ieee.std_logic_1164.all;  --package of definitions
entity full_add8 is
   port ( a:     in std_logic_vector(7 downto 0);
          b:     in std_logic_vector(7 downto 0);
          cin:   in std_logic;
          sum: out std_logic _vector(7 downto 0);
          cout: out std_logic);
end full_add8 ;
```

User-Defined Data Types

- Any abstract data type can be created
- Examples:

```
type mnemonic is (add,sub,mov,jmp);
signal op: mnemonic;

type byte is array(0 to 7) of bit;
signal dbus: byte;
```

- Subtype of a previously-defined type:

```
subtype int4 is integer range 0 to 15;
subtype natural is integer range 0 to integer'high;
```

Miscellaneous – for RTL design

- "Alias" for existing elements

```
signal instruction: bit_vector(0 to 31);
alias opcode: bit_vector(0 to 5) is instruction(0 to 5);
alias rd: bit_vector(0 to 4) is instruction(6 to 10);
alias rs: bit_vector(0 to 4) is instruction(11 to 15);
```

- Fill a vector with a constant (right-most bits):

```
A <= ('0','1','1', others => '0');
A <= (others => '0');   -- set to all 0
B(15 downto 0) <= C(15 downto 0);
B(31 downto 16) <= (others => C(15)); -- sign extension!
```

- Concatenate bits and bit_vectors

```
A <= B & C(0 to 3) & "00"; -- A is 16 bits, B is 10 bits
```

“Architecture” defines function/structure

```
entity Half_Adder is
      port (X, Y : in std_logic:= '0';         -- formals
               Sum, Cout : out std_logic); -- formals
end;

architecture Behave of Half_Adder is
begin
      Sum <= X xor Y;    -- use formals from entity
      Cout <= X and Y;
end Behave;
```

Behavioral architecture example
(no circuit structure specified)

```
architecture dataflow of full_add1 is
begin
   sum <= a xor b xor cin;
   cout <= (a and b) or (a and cin) or (b and cin);
end;
```

Example using an internal signal

```
-- can both drive and reference an internal signal
architecture dataflow of full_add1 is
   signal x1: std_logic; -- internal signal
begin
   x1 <= a xor b;                --drive x1
   sum <= x1 xor cin;            --reference x1
   cout <= (a and b) or (a and cin) or (b and cin);
end;
```

Signal assignment statement

- Model signal driven by a value (signal value produced by "hardware")

  a <= b and c after 1 ns;

  - Data types must match (strongly typed)
  - Delay can be specified (as above)
  - Infinitesimally small delay "delta" used if no delay specified:

    a <= b and c;

  - Signals cannot change in zero time!
- Delay usually unknown in behavioral & RTL models and therefore omitted

VHDL Signals and Simulation

- Signal assignment creates a "driver" for the signal
  - An "event" is a time/value pair for a signal change

    Ex. ('1', 5 ns) – Signal assigned value '1' at current time + 5ns
  - Driver contains a queue of pending events
    - Only one driver per signal (except for special buses)
    - *can only drive signal at one point in the model*
- Statements appear to be evaluated *concurrently*
  - Time held constant during statement evaluation
  - Evaluate each statement affected by a signal event at time T
  - Resulting events "scheduled" in the affected signal driver
  - New values assigned when time advances to scheduled event time

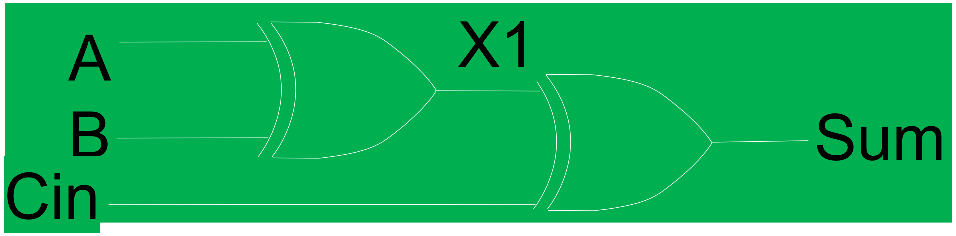
Event-Driven Simulation Example

```
a <= b after 1ns;
c <= a after 1ns;
```

Time	a	b	c	
T	‘0’	‘0’	‘0’	
T+1	‘0’	‘1’	‘0’	- external event on b
T+2	‘1’	‘1’	‘0’	- resulting event on a
T+3	‘1’	‘1’	‘1’	- resulting event on c

Structural architecture example (no behavior specified)

```
architecture structure of full_add1 is
    component xor             -- declare component to be used
            port (x,y: in bit;
                    z: out bit);
    end component;
    for all: xor use entity work.xor(eqns); -- if multiple arch's
    signal x1: bit;    -- signal internal to this component
begin
    G1: xor port map (a, b, x1);          -- instantiate 1st  xor gate
    G2: xor port map (x1, cin, sum);  -- instantiate 2nd xor gate
   …add circuit for carry output…
end;
```

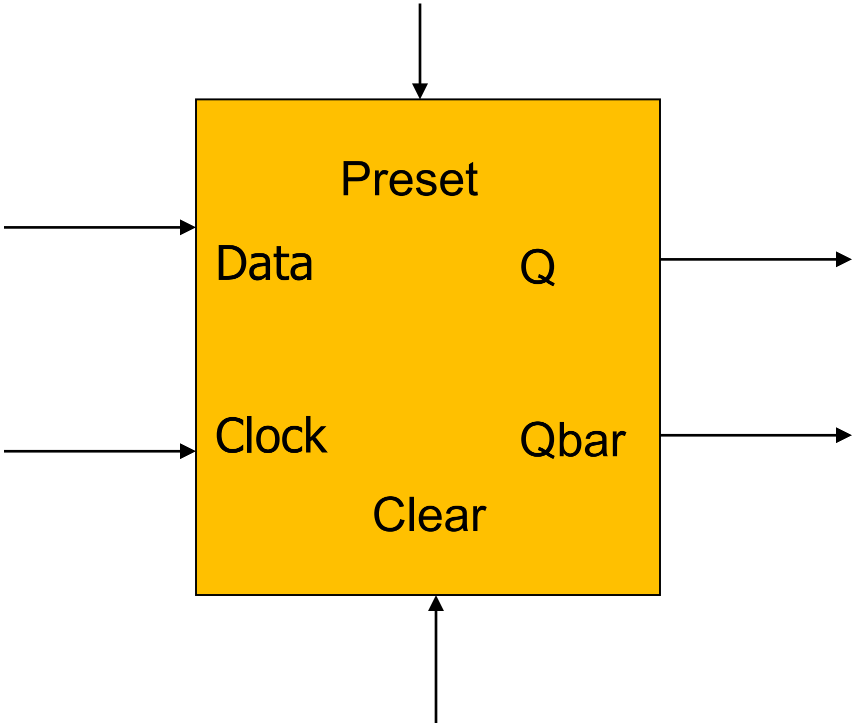


Example: adder behavioral model

```
library ieee;
use ieee.numeric_std.all; --defines arithmetic functions
                          --on types SIGNED/UNSIGNED
entity adder is
   port ( a:      in   signed(31 downto 0); -- "signed" type
          b:      in   signed(31 downto 0); --  related to
          sum:  out signed(31 downto 0); -- std_logic type
end adder ;
architecture behave of adder is
   begin
        sum <= a + b;    -- adder to be synthesized
   end;
```

Example: D flip-flop

```
entity DFF is
  port (Preset: in bit;
        Clear: in bit;
        Clock: in bit;
        Data: in bit;
        Q: out bit;
        Qbar: out bit);
end DFF;
```

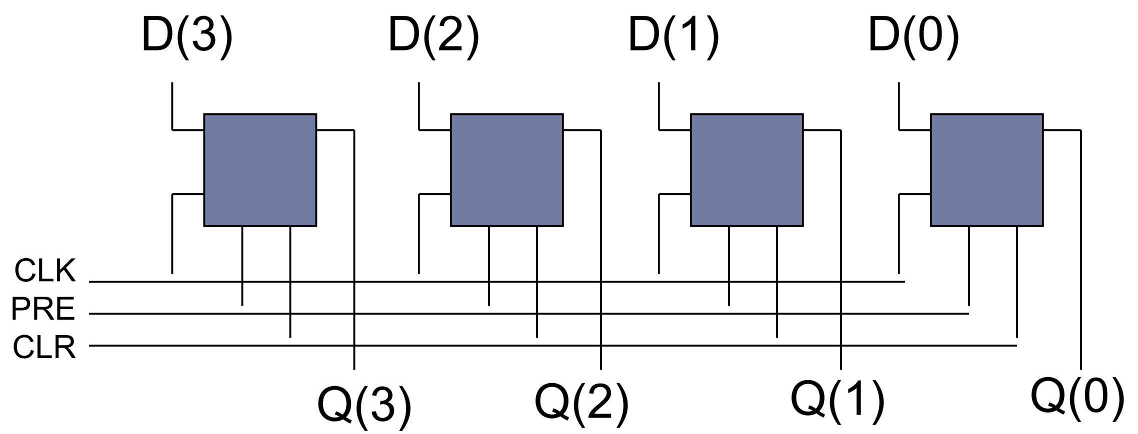


7474 D flip-flop equations

```
architecture eqns of DFF is
        signal A,B,C,D: bit;
        signal QInt, QBarInt: bit;
begin
        A <= not (Preset and D and B) after 1 ns;
        B <= not (A and Clear and Clock) after 1 ns;
        C <= not (B and Clock and D) after 1 ns;
        D <= not (C and Clear and Data) after 1 ns;
        Qint <= not (Preset and B and QbarInt) after 1 ns;
        QBarInt <= not (QInt and Clear and C) after 1 ns;
        Q <= QInt;
        QBar <= QBarInt;
end;
```

4-bit Register (Structural Model)

```
entity Register4 is
  port ( D: in bit_vector(0 to 3);
         Q: out bit_vector(0 to 3);
         Clk: in bit;
         Clr: in bit;
         Pre: in bit);
end Register4;
```



Register Structure

```
architecture structure of Register4 is
    component DFF        -- declare library component to be used
       port (Preset: in bit;
               Clear: in bit;
               Clock: in bit;
               Data: in bit;
               Q: out bit;
               Qbar: out bit);
     end component;
     signal Qbar: bit_vector(0 to 3); -- dummy for unused FF output
begin   -- Signals connected to ports in order listed above
    F3: DFF port map (Pre, Clr, Clk, D(3), Q(3), Qbar(3));
    F2: DFF port map (Pre, Clr, Clk, D(2), Q(2), Qbar(2));
    F1: DFF port map (Pre, Clr, Clk, D(1), Q(1), OPEN);
    F0: DFF port map (Pre, Clr, Clk, D(0), Q(0), OPEN);
end;      -- keyword OPEN may be connected to unused output
```

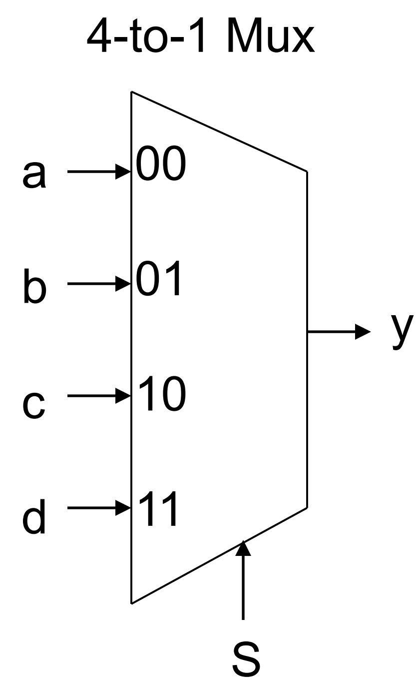
Register Structure
(short cut – “generate” statement)

```
begin
   for k in 0 to 3 generate
      F: DFF port map (Pre, Clr, Clk, D(k), Q(k), OPEN);
   end generate;
end;
```

- Generates multiple copies of the given statement(s)
- Value of k inserted where specified
- Iteration number k is appended to each label F
- Result is identical to previous example

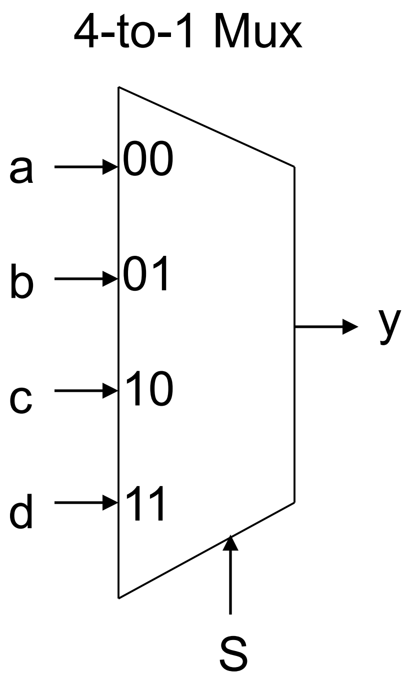
Conditional Signal Assignment

```
    signal a,b,c,d,y: bit;
    signal S: bit_vector(0 to 1);
begin
    with S select
        y <= a after 1 ns when "00",
             b after 1 ns when "01",
             c after 1 ns when "10",
             d after 1 ns when "11";
        (or:  d after 1 ns when others;)
```



32-bit-wide 4-to-1 multiplexer

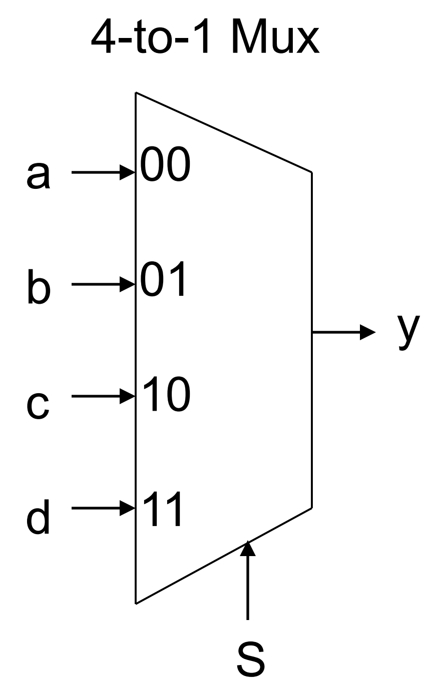
```
    signal a,b,c,d,y: bit_vector(0 to 31);
    signal S: bit_vector(0 to 1);
begin
    with S select
        y <=  a after 1 ns when "00",
              b after 1 ns when "01",
              c after 1 ns when "10",
              d after 1 ns when "11";
```



a,b,c,d,y can be any type, as long as they are the same

Conditional Signal Assignment – Alternate Format

```
y <=  a after 1 ns when (S="00") else
      b after 1 ns when (S="01") else
      c after 1 ns when (S="10") else
      d after 1 ns;
```



Any boolean expression can be used for each condition.

Ex. y <= a after 1 ns when (F='1') and (G='0') …

Unconstrained Bit Vectors

- Model a component with variable data widths

```
entity mux is
        port (a,b: in bit_vector;  -- unconstrained
                    c: out bit_vector;
                    s: in bit );
end mux;
architecture x of mux is
begin
        c <= a when (s='0') else b;
end;
```

Unconstrained Bit Vectors

- Vector constrained when instantiated:

```
signal s1,s2: bit;
signal a5,b5,c5: bit_vector (0 to 4);
signal a32,b32,c32: bit_vector (0 to 31);
component mux
 port (a,b: in bit_vector;  -- unconstrained
       c: out bit_vector;
       s: in bit );
end component;
begin
  M5:   mux port map (a5,b5,c5,s1);       -- 5-bit mux
  M32: mux port map (a32,b32,c32,s2); -- 32-bit mux
```

Parameterized models

- Allows a generic component with variable sizes:

```
entity mux is
    generic (N: integer := 32);
    port (a,b: in bit_vector(N-1 downto 0);
          c: out bit_vector(N-1 downto 0);
          s: in bit );
end mux;
architecture x of mux is
begin
    c <= a when (s='0') else b;
end;
```

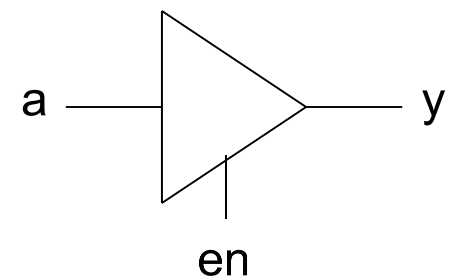
Parameterized Bit Vectors

- Vector constrained when instantiated:

```
signal s1,s2: bit;
signal a5,b5,c5: bit_vector (0 to 4);
signal a32,b32,c32: bit_vector (0 to 31);
component mux
    generic (N: integer := 32);
    port (a,b: in bit_vector(N-1 downto 0);
          c: out bit_vector(N-1 downto 0);
          s: in bit );
end component;
begin
  M5:  mux generic map (5) port map (a5,b5,c5,s1);      -- 5-bit mux
  M32: mux generic map (32) port map (a32,b32,c32,s2); -- 32-bit mux
```

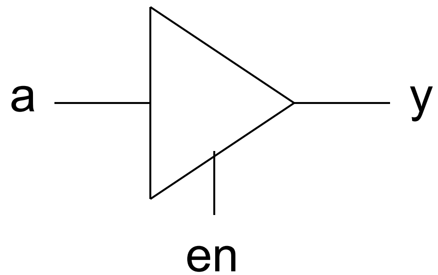
Tristate bus buffer example

```
library ieee;
use ieee.std_logic_1164.all;
entity tristate is
  port ( a: in std_logic_vector(0 to 7);
         y: out std_logic_vector(0 to 7);
         en: in bit);
end tristate;
architecture a1 of tristate is
begin
   y <= a               after 1 ns when (en='1') else
        "ZZZZZZZZ"      after 1 ns;
end;    -- signal types of y and a match
```



Tristate buffer example (incorrect)

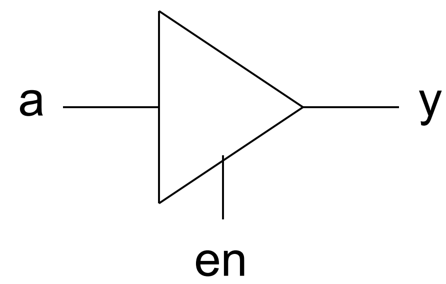
```
library ieee;
use ieee.std_logic_1164.all;
entity tristate is
  port ( a: in bit;
            y: out std_logic;
            en: in bit);
end tristate;
architecture a1 of tristate is
begin
   y <= a after 1 ns when (en='1') else
         'Z' after 1 ns;
end;
```



Type mismatch between y and a

Tristate buffer example (correct)

```
library ieee;
use ieee.std_logic_1164.all;
entity tristate is
  port ( a: in bit;
              y: out std_logic;
              en: in bit);
end tristate;
architecture a1 of tristate is
begin
   y <=  ‘0’ after 1 ns when (en=‘1’) and (a=‘0’) else
         ‘1’ after 1 ns when (en=‘1’) and (a=‘1’) else
         ‘Z’ after 1 ns;
end;
```



VHDL “Process” Construct

- Allows conventional programming language methods to describe circuit behavior
- Supported language constructs (“sequential statements”) – only allowed within a process:
  - variable assignment
  - if-then-else (elsif)
  - case statement
  - while (condition) loop
  - for (range) loop

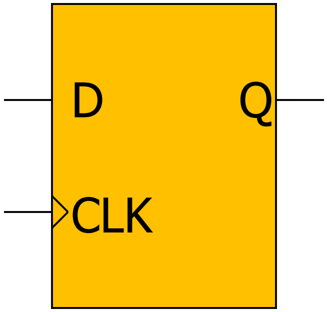
Process Format

[label:] process (*sensitivity list*)
declarations
begin
sequential statements
end process;

- Process statements executed once at start of simulation
- Process halts at "end" until an event occurs on a signal in the "sensitivity list"

Using a "process" to model sequential behavior

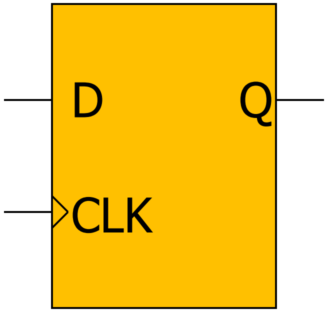
```
entity DFF is
  port (D,CLK: in bit;
                Q: out bit);
end DFF;
architecture behave of DFF is
begin
    process(clk)    -- "process sensitivity list"
    begin
          if (clk'event and clk='1') then
                    Q <= D after 1 ns;
          end if;
    end process;
end;
```



- Process statements executed sequentially (sequential statements)
- clk'event is an attribute of signal clk which is TRUE if an event has occurred on clk at the current simulation time

Alternative to sensitivity list

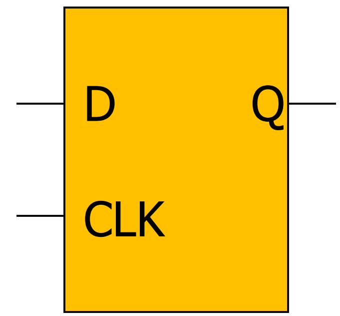
```
entity DFF is
  port (D,CLK: in bit;
              Q: out bit);
end DFF;
architecture behave of DFF is
begin
    process    -- no "sensitivity list"
    begin
          wait on clk; -- suspend process until event on clk
          if (clk='1') then
                    Q <= D after 1 ns;
          end if;
    end process;
end;
```



- Other "wait" formats: wait until (clk'event and clk='1'); wait for 20 ns;
- Process executes endlessly if no sensitivity list or wait statement!

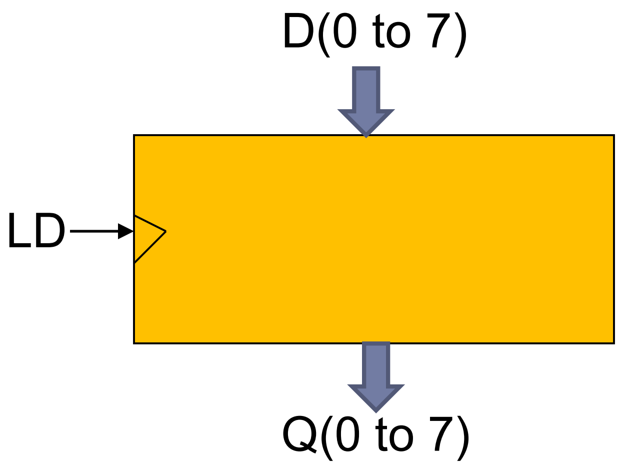
D latch vs. D flip-flop

```
entity Dlatch is
  port (D,CLK: in bit;
           Q: out bit);
end Dlatch;
architecture behave of Dlatch is
begin
   process(D, clk)
   begin
         if (clk=‘1’) then
                 Q <= D after 1 ns;
         end if;
   end process;
end;
-- For latch, Q changes whenever the latch is enabled by CLK=‘1’ rather than being
   edge-triggered)
```



Defining a "register" for an RTL model (not gate-level)

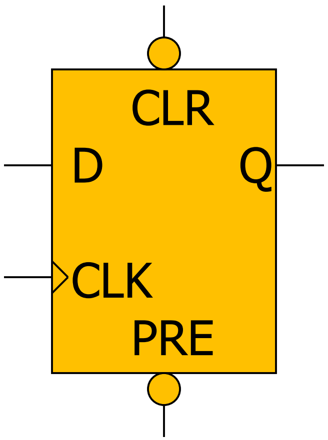
```
entity Reg8 is
  port (D: in bit_vector(0 to 7);
        Q: out bit_vector(0 to 7)
        LD: in bit);
end Reg8;
architecture behave of Reg8 is
begin
   process(LD)
   begin
        if (LD'event and LD='1') then
                Q <= D after 1 ns;
        end if;
   end process;
end;
```



D and Q can be any abstract data type

Synchronous vs. Asynchronous Flip-Flop Inputs

```
entity DFF is
  port (D,CLK: in bit;
        PRE,CLR: in bit;
        Q: out bit);
end DFF;
architecture behave of DFF is
begin
    process(clk,PRE,CLR)
    begin
        if (CLR='0') then  -- CLR has precedence
            Q <= '0' after 1 ns;
        elsif (PRE='0') then  -- Then PRE has precedence
            Q <= '1' after 1 ns;
        elsif (clk'event and clk='1') then
            Q <= D after 1 ns; -- Only if CLR=PRE='1'
        end if;
    end process;
end;
```



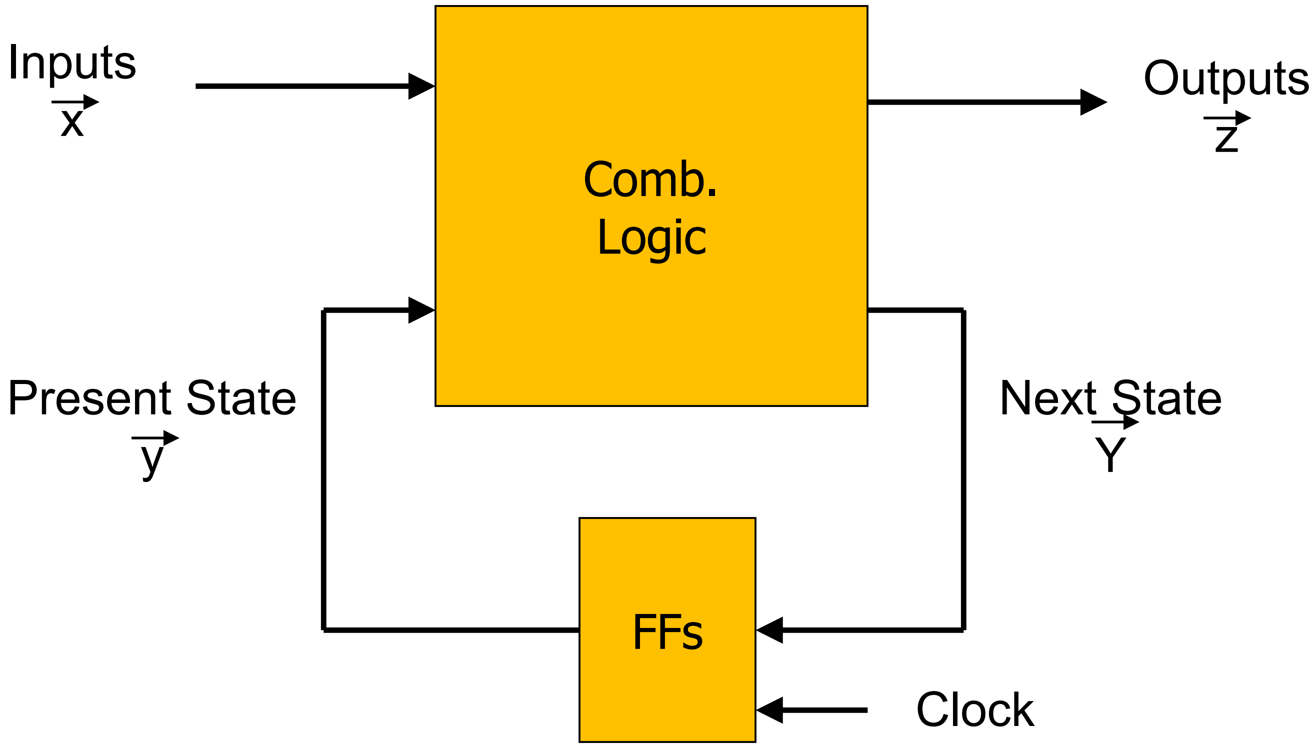
Using a "variable" to describe sequential behavior within a process

```
cnt: process(clk)
        variable count: integer;     -- internal counter state
    begin                            -- valid only in a process
        if clk='1' and clk'event then
          if ld='1' then             -- "to_integer" must be supplied
                count := to_integer(Din);
          elsif cnt='1' then
                count := count + 1;
          end if;
        end if;                 -- "to_bitvector" must be supplied
        Dout <= to_bitvector(count);
   end process;
```

Modeling Finite State Machines (Synchronous Sequential Circuits)

- FSM design & synthesis process:
  1. Design state diagram (behavior)
  2. Derive state table
  3. Reduce state table
  4. Choose a state assignment
  5. Derive output equations
  6. Derive flip-flop excitation equations
- Synthesis steps 2-6 can be automated, given the state diagram

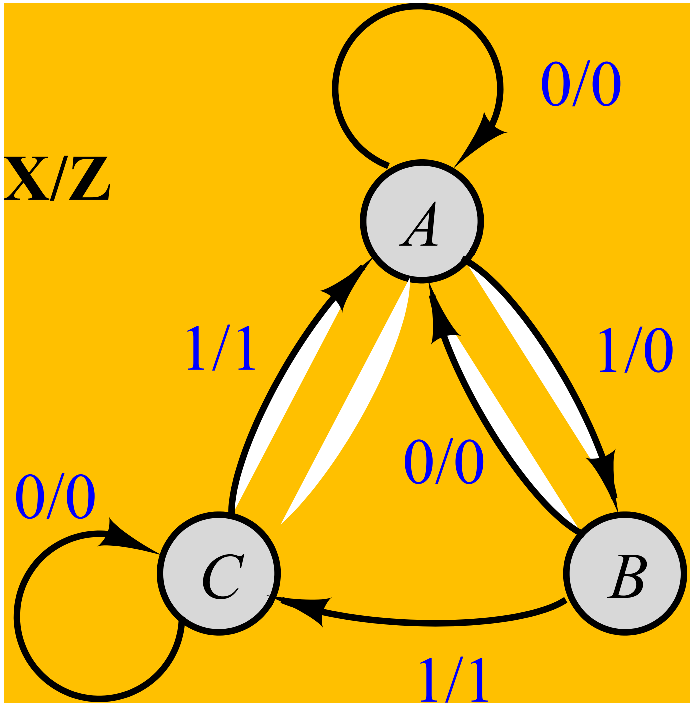
Synchronous Sequential Circuit Model



Mealy Outputs z = f(x,y), Moore Outputs z = f(y)

Next State Y = f(x,y)

Synchronous Sequential Circuit (FSM) Example



Present state	Input *x* 0	Input *x* 1
A	A/0	B/0
B	A/0	C/1
C	C/0	A/1

Next state/output

FSM Example – entity definition

```
entity seqckt is
    port (
            x: in bit;              -- FSM input
            z: out bit;             -- FSM output
            clk: in bit );    -- clock
end seqckt;
```

FSM Example - behavioral model

```
architecture behave of seqckt is
   type states is (A,B,C);  -- symbolic state names
   signal curr_state,next_state: states;
begin
   -- Model the memory elements of the FSM
   process (clk)
   begin
         if (clk'event and clk='1') then
                  pres_state <= next_state;
         end if;
   end process;
```

(continue on next slide)

FSM Example - continued

```
-- Model the next-state and output functions of the FSM
process (x, pres_state) -- function inputs
begin
      case pres_state is  -- describe each state
          when A => if (x = '0') then
                            z <= '0';
                            next_state <= A;
                    else    -- (x = '1')
                            z <= '0';
                            next_state <= B;
                    end if;
```

(continue next slide for pres_state = B and C)

FSM Example (continued)

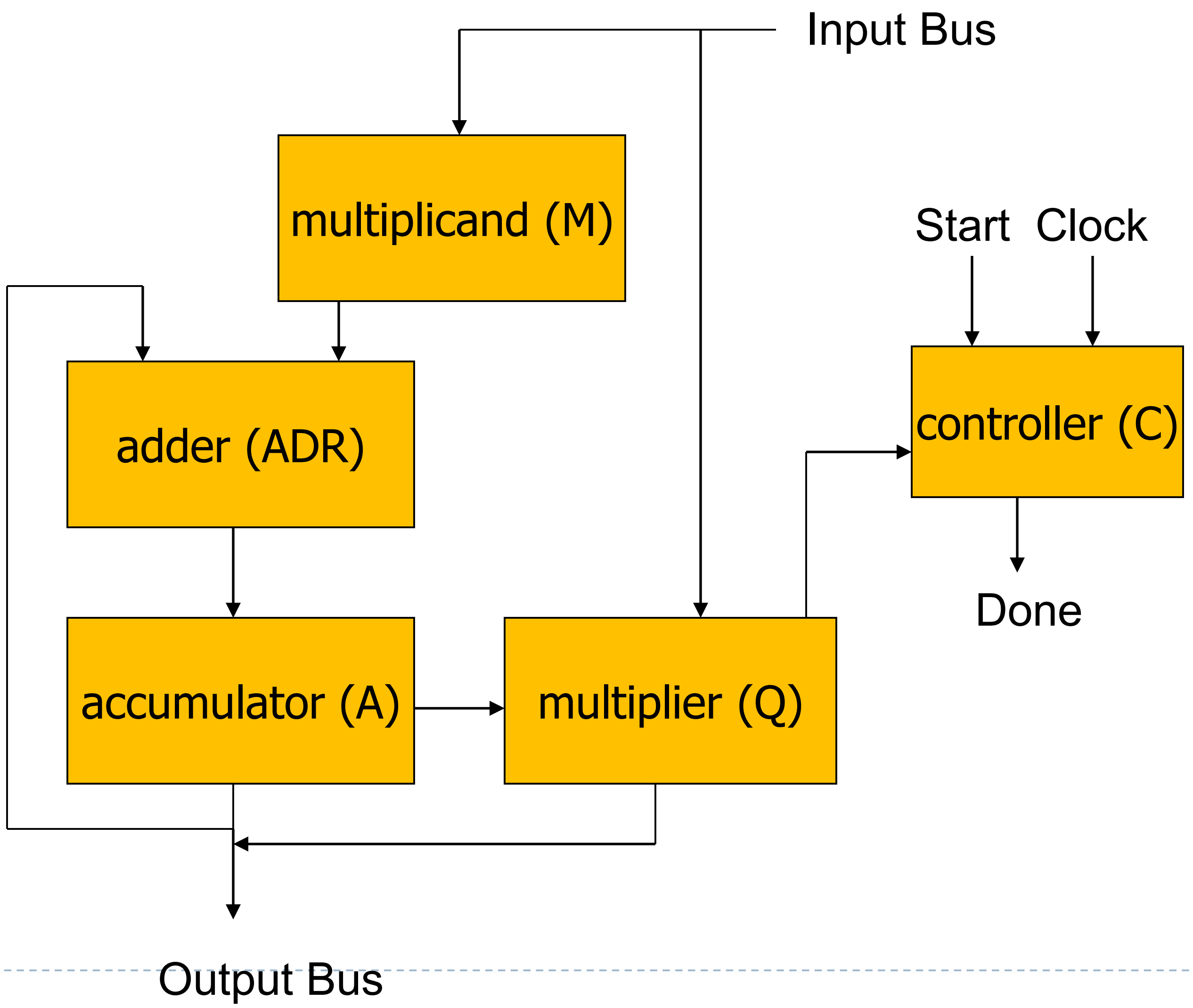
```
          when B => if (x='0') then
                              z <= '0';
                              next_state <= A;
                         else
                              z <= '1';
                              next_state <= C;
                         end if;
          when C => if (x='0') then
                              z <= '0';
                              next_state <= C;
                         else
                              z <= '1';
                              next_state <= A;
                         end if;
     end case;
  end process;
end;
```

Alternative Format for Output and Next State Functions

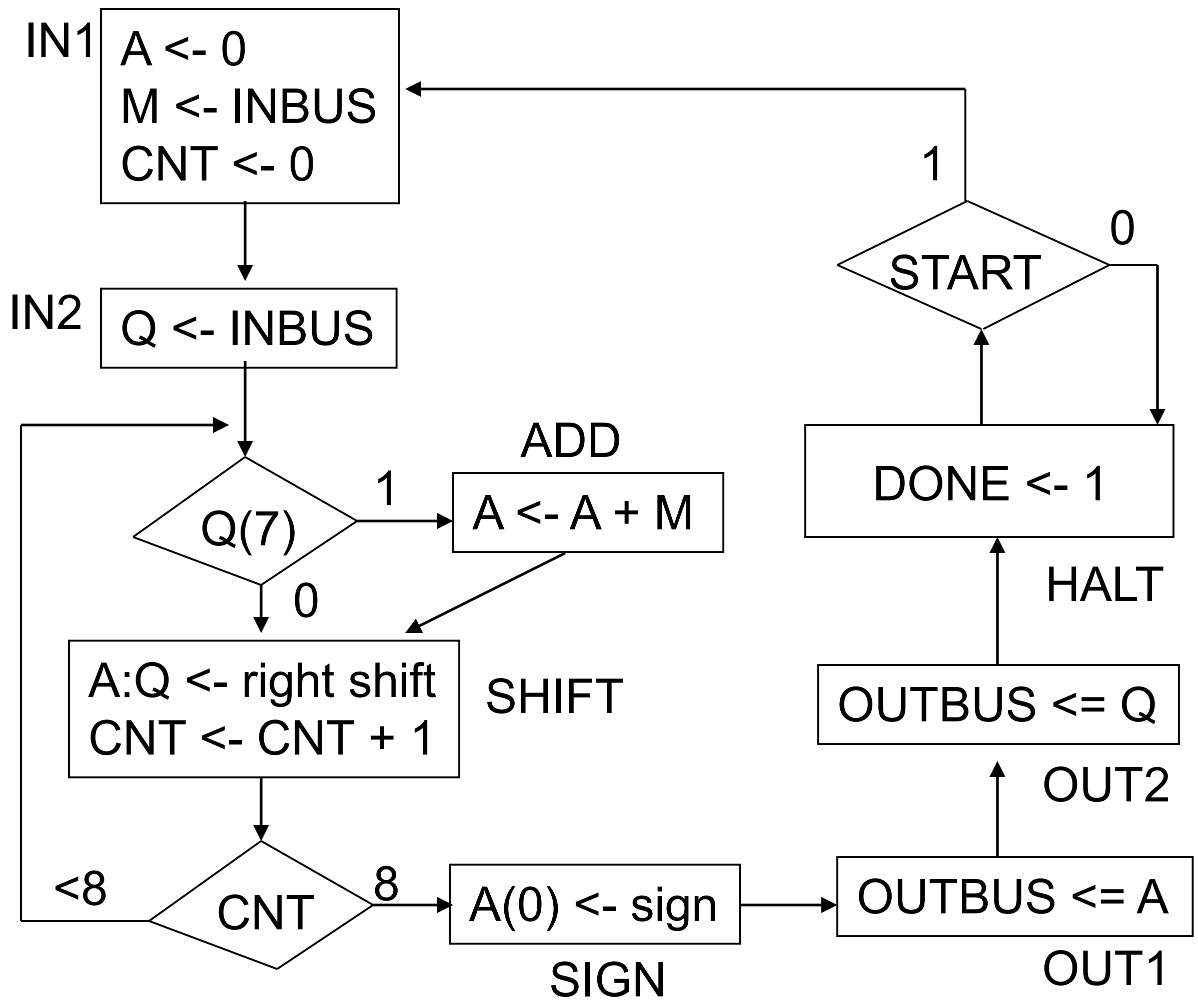
```
z <= ‘1’ when ((curr_state = B) and (x = ‘1’))
                or ((curr_state = C) and (x = ‘1’))
        else ‘0’;

next_state <= A when ((curr_state = A) and (x = ‘0’))
                          or ((curr_state = B) and (x = ‘0’))
                          or ((curr_state = C) and (x = ‘1’)) else
                     B when ((curr_state = 1) and (x = ‘1’)) else
                     C;
```

System Example: 8x8 multiplier



Multiply Algorithm



Multiplier – Top Level

```
entity multiplier is
 port  (INBUS:     in  bit_vector(0 to 7);
        OUTBUS: out bit_vector(0 to 7);
        CLOCK:    in  bit;
        START:     in  bit;
        DONE:       out bit);
end multiplier;

architecture struc of multiplier is

   [component declarations go here]

  -- internal signals to interconnect components
  signal AR, MR, QR, AD, Ain: bit_vector(0 to 7);
  signal AMload, AMadd, Qload, AQshift, AQoutEn, AQoutSel: bit;
  signal SignLd: bit;
```

Multiplier – Top Level (continued)

```
begin

OUTBUS <= AR when AQoutEn = '1' and AQoutSel = '0' else
          QR when AQoutEn = '1' and AQoutSel = '1' else
          "00000000";

Ain(0) <= AD(0) when AMadd = '1' else MR(0) xor QR(7);
Ain(1 to 7) <= AD(1 to 7);

M:   mreg   port map (INBUS, MR, AMload);
Q:          Qreg   port map (INBUS, QR, AR(7), Qload, SignLd, AQshift);
A:   areg   port map (Ain, AR, AMadd, SignLd, AQshift, AMload);
ADR: adder port map (AR, MR, AD);
C:    mctrl  port map (START, CLOCK, QR(7), AMload, AMadd, Qload, AQshift,
                       SignLd, AQoutEn, AQoutSel, DONE);
end;
```

Multiplicand Register (mreg)

```
-- simple parallel-load register
entity mreg is
 port  (Min:  in  bit_vector(0 to 7);
        Mout: out bit_vector(0 to 7);
        Load: in  bit);
end mreg;

architecture comp of mreg is
begin
   process (Load)              -- wait for change in Load
   begin
          if Load = '1' then
                   Mout <= Min;      -- parallel load
          end if;
   end process;
end;
```

Accumulator Register (areg)

```
-- shift register with clear and parallel load
entity Areg is
 port  (Ain:     in  bit_vector(0 to 7);
        Aout:   out bit_vector(0 to 7);
        Load:   in  bit;                    -- load entire register
        Load0: in  bit;                     -- load a0 only
        Shift:  in  bit;                    -- shift right
        Clear:  in  bit);      -- clear register
end Areg;

architecture comp of areg is
    signal A: bit_vector(0 to 7);           -- internal state
```

Accumulator Register (areg)

```
begin
  Aout <= A;                          -- internal value to outputs

  process (Clear, Load, Load0, Shift)  -- wait for event
  begin
      if Clear = '1' then
          A <= "00000000";            -- clear register
          elsif Load = '1' then
          A <= Ain;                   -- parallel load
          elsif Shift = '1' then
          A <= '0' & A(0 to 6);       -- right shift
          elsif Load0 = '1' then
          A(0) <= Ain(0);             -- load A(0) only
          end if;
  end process;
end;
```

Multiplier/Product Register (Qreg)

```
-- shift register with parallel load

entity Qreg is
 port  (Qin:    in  bit_vector(0 to 7);
        Qout:   out bit_vector(0 to 7);
        SerIn:  in  bit;                 -- serial input for shift
        Load:   in  bit;                 -- parallel load
        Clear7: in  bit;                 -- clear bit 7
        Shift:  in  bit);                -- right shift
end Qreg;

architecture comp of qreg is
    signal Q: bit_vector(0 to 7);        -- internal storage
```

Multiplier/Product Register (Qreg)

```
begin

  Qout <= Q;      -- drive output from internal storage

  process (Load, Shift, Clear7)   -- wait for event
  begin
       if Load = '1' then
             Q <= Qin;                            -- load Q
       elsif Shift = '1' then
             Q <= SerIn & Q(0 to 6);   -- shift Q right
       elsif Clear7 = '1' then
             Q(7) <= '0';                         -- clear bit Q(7)
       end if;
  end process;
end;
```

8-bit adder (behavioral)

```
use work.qsim_logic.all; -- contains bit_vector addition
entity adder is
    port( X,Y: in  bit_vector(0 to 7);
            Z:  out bit_vector(0 to 7));
end adder;

architecture comp of adder is
      signal  temp: bit_vector(0 to 8);
begin
      temp <= ("00" & X(1 to 7)) + ("00" & Y(1 to 7));
      Z <= temp (1 to 8);
end;
```

Multiplier Controller

```
entity mctrl is
    port (Start:        in  bit;        -- start pulse
          Clock:        in  bit;        -- clock input
          Q7:            in  bit;       -- LSB of multiplier
          AMload:       out bit;        -- load M & A registers
          AMadd:        out bit;        -- load adder result into A
          Qload:        out bit;        -- Load Q register
          AQshift:      out bit;        -- shift A & Q registers
          SignLd:       out bit;        -- load sign into A(0)
          AQoutEn:      out bit;        -- enable output
          AQoutSel:     out bit;        -- select A or Q for output
          DONE:         out bit);       -- external DONE signal
end mctrl;
```

Multiplier Controller - Architecture

```
architecture comp of mctrl is
    type states is (Halt,In1,In2,Add,Shift,Sign,Out1,Out2);
    signal State:    States := Halt;                      -- state of the controller
begin
    -- decode state variable for outputs
    AMload   <= '1' when State = In1 else '0';
    Qload     <= '1' when State = In2 else '0';
    AMadd    <= '1' when State = Add and Q7 = '1' else '0';
    AQshift   <= '1' when State = Shift else '0';
    AQoutSel <= '1' when State = Out2 else '0';
    SignLd    <= '1' when State = Sign else '0';
    AQoutEn  <= '1' when State = Out1 or State = Out2 else '0';
    DONE      <= '1' when State = Halt else '0';
```

Controller – State transition process

```
process (Clock) -- implement state machine state transitions
        variable Count: integer;
 begin
        if Clock = '1' then
        case State is
            when Halt =>  if Start = '1' then       -- wait for start pulse
                              State <= In1;
                              Count := 0;
                          end if;
            when In1   =>  State <= In2;      -- Read 1st operand
            when In2   =>  State <= Add;      -- Read 2nd operand
```

(Continued)

Controller – State transition process (continued)

```
            when Add   =>  State <= Shift;  -- Add multiplicand to accumulator
                           Count := Count + 1;
            when Shift =>  if Count = 7 then-- Shift accumulator/multiplier
                               State <= Sign;
                                   else
                               State <= Add;
                             end if;
            when Sign    => State <= Out1;    -- Set sign of result
            when Out1    => State <= Out2;    -- Output lower half of product
            when Out2    => State <= Halt;    -- Output upper half of product
        end case;
    end if;
end process;
```

64K x 8 Memory Example

```
library ieee;
use ieee.std_logic_1164.all;
use work.qsim_logic.all;         -- package with to_integer() func

entity memory8 is
 port (dbus: inout std_logic_vector(0 to 7);
       abus: in    std_logic_vector(0 to 15);
       ce:   in bit;              -- active low chip enable
       oe:   in bit;              -- active low output enable
       we:   in bit);             -- active low write enable
end memory8;
```

64K x 8 Memory Example

```
architecture reglevel of memory8 is
begin
 process (ce,oe,we,abus,dbus)
     type mem is array(natural range <>) of std_logic_vector(0 to 7);
     variable M: mem(0 to 65535);
 begin
     if (ce='0') and (oe='0') then                    -- read enabled
          dbus <= M(to_integer(abus)); -- drive the bus
     elsif (ce='0') and (we='0') then                 -- write enabled
          dbus <= "ZZZZZZZZ";                          -- disable drivers
          M(to_integer(abus)) := dbus;  -- write to M
   else
          dbus <= "ZZZZZZZZ";                          --disable drivers
   end if;
 end process;
end;
```