Modeling & Simulating
ASIC Designs with VHDL

Reference: Smith text: Chapters 10 & 12
Hardware Description Languages

- **VHDL** = VHSIC Hardware Description Language (VHSIC = Very High Speed Integrated Circuits)
  - Developed by DOD from 1983 – based on ADA
  - Based on the ADA language

- **Verilog** – created in 1984 by Philip Moorby of Gateway Design Automation (merged with Cadence)
  - 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 ;
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
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 \([-2^{31} \ldots + (2^{31} - 1)]\) on 32-bit host

- **Aggregate (multi-value) signal types:**
  - bit_vector — array of bits

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

--others subtypes: X01,X01Z,UX01,UX01Z – subsets of std_logic/ulogic
VHDL “Package”

- Package = file of type definitions, functions, procedures to be shared across VHDL models
  - User created
  - Standards/3rd party – usually distributed in “libraries”

```vhdl
package name is
  -- type, function, procedure declarations
end package name;
```

```vhdl
package body name is  -- only if functions used
  -- function implementations
end package body name;
```
Bus resolution

- Most commonly used data type in system design for synthesis is `STD_LOGIC`

- function `resolved (s : STD_ULOGIC_VECTOR)`
  return `STD_ULOGIC`;

- subtype `STD_LOGIC` is `resolved` `STD_ULOGIC`;

- type `STD_LOGIC_VECTOR` is array (NATURAL range `<>`) of `STD_LOGIC`;
std_logic includes a “bus resolution function” to determine the signal state where there are multiple drivers.

<table>
<thead>
<tr>
<th>Driver A value</th>
<th>Driver B value</th>
<th>Resolved Bus Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'0'</td>
<td>'0'</td>
</tr>
<tr>
<td>'0'</td>
<td>'X'</td>
<td>'X'</td>
</tr>
<tr>
<td>'1'</td>
<td>'X'</td>
<td>'1'</td>
</tr>
<tr>
<td>'Z'</td>
<td>'0'</td>
<td>'Z'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
</tr>
</tbody>
</table>

Driver B value

<table>
<thead>
<tr>
<th>Driver A value</th>
<th>Driver B value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'0'</td>
</tr>
<tr>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>'Z'</td>
<td>'Z'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
</tr>
</tbody>
</table>

Diagram:
- Driver A
- Driver B

Bus resolution function
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

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;
  ```
Formats for identifiers and literal values
(Smith Chap. 10)

signal I1 : integer;
signal RI : real;
signal C1 : CHARACTER;
signal S16 : STRING(1 to 16);
signal BV4 : BIT_VECTOR(0 to 3);
signal BV12 : BIT_VECTOR(0 to 11);
signal BV16 : BIT_VECTOR(0 to 15);
Abstract literals are decimal or based literals.

Integer literal examples (each of these is the same):

\[
\begin{align*}
I1 & := 120000; \\
\text{Int} & := 12e4; \\
\text{Int} & := 120_000;
\end{align*}
\]

Based literal examples (base between 2 and 16):

\[
\begin{align*}
I1 & := 2#1111_1111_1111_1111#; \\
I1 & := 16#FFFF#; \\
I1 & := 16:FFFF:;  \quad \text{-- use : if # not available}
\end{align*}
\]

Real literal examples (each of these is the same):

\[
\begin{align*}
\text{RI} & := 120000.0; \\
\text{RI} & := 1.2e5; \\
\text{RI} & := 12.0E4;
\end{align*}
\]
-- Character literal must be one of the 191 graphic characters.
-- 65 of the 256 ISO Latin-1 set are non-printing control characters

C1 := 'A';
C1 := 'a'; -- different from each other

-- String literal examples:
S16 := " string" & " literal";       -- concatenate long strings
S16 := """Hello,"" I said!"";         -- doubled quotes
S16 := % string literal%;          -- can use % instead of "
S16 := %Sale: 50%% off!!!%;        -- doubled %

-- Bit-string literal examples:
BV4   := B"1100"; -- binary bit-string literal
BV12  := O"7777"; -- octal bit-string literal
BV16  := X"FFFF"; -- hex bit-string literal
“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
entity Half_Adder is
  port (X, Y : in BIT := '0';
        Sum, Cout : out BIT); -- 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 after 1 ns;
    cout <= (a and b) or (a and cin) or
           (b and cin) after 1 ns;
end;
Example using an internal signal

architecture dataflow of full_add1 is

    signal x1: std_logic; -- internal signal

begin

    x1 <= a xor b after 1 ns;
    sum <= x1 xor cin after 1 ns;
    cout <= (a and b) or (a and cin) or
           (b and cin) after 1 ns;

end;
VHDL Signals and Simulation

- Signal assignment statement 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
Signal assignment statement

- Model signal driven by a value (signal value produced by “hardware”)
  
  \[ a <= b \text{ and } c \text{ after 1 ns; } \]

- Data types must match (strongly typed)

- Delay can be specified (as above)

- Infinitesimally small delay “delta” inserted if no delay specified:
  
  \[ a <= b \text{ and } c; \]

- Signals cannot change in zero time!

- Delay usually unknown in behavioral models and therefore omitted
VHDL: a strongly typed language
Concurrent Statements and Event-Driven Simulation

- Statements appear to be evaluated concurrently
- Time held constant during statement evaluation
- Each statement affected by a signal event at time $T$ is evaluated
- Any resulting events are “scheduled” in the affected signal driver
- New values take effect when time advances to the scheduled event time
### Event-Driven Simulation Example

\[
\begin{align*}
\text{a} & \leq \text{b} \text{ after 1ns;} \\
\text{c} & \leq \text{a} \text{ after 1ns;}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Time</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T+1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T+2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T+3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>