Progettazione e sintesi di circuiti digitali

Lecture 5

VHDL-based digital design flow
Digital design flow

1. **System specification**
2. **System architecture**
3. **Algorithmic description**
   - Behavioral synthesis
   - Logic synthesis
4. **RTL model**
5. **Gate netlist**
6. **Place & route**
   - **Circuit layout**
   - **ASIC or FPGA**
7. **Testbench**

Simulation steps:
- = ?
Digital design flow

- System specification
- System architecture
- Algorithmic description
- Behavioral synthesis
- RTL model
- Logic synthesis
- Gate netlist
- Simulation
- Testbench
- Place & route
- Circuit layout
- Simulation

SystemC, SystemVerilog, VHDL

ASIC or FPGA
Hardware description languages (HDL)

• Contain elements of programming languages ...
  – data and objects (constants, variables, expressions, ...)
  – logic and arithmetic operators
  – procedural instructions (if, while, for, ...)
  – functions, subprograms
• ... and specific elements for efficient hardware modelling
  – components with directional terminals, hierarchical structure
  – concurrent constructs
• Support simulation and synthesis
  – event-driven and cycle-based simulation
  – logic synthesis (mature), behavioral synthesis (in development)
VHDL

- VHDL = VHSIC Hardware Description Language
  - VHSIC = Very High Speed Integrated Circuits
- Conceived around 1980 by US DoD to document behavior of ASICs in military equipment
- Became IEEE standard in 1987 (IEEE 1076-1987)
- VHDL event-driven simulators developed right from the beginning
- Logic synthesis came later and only on a language subset
VHDL Design Units

• They are the basic building blocks of the language

• A design unit must be contained in a single file
  – a file may contain any number of design units
  – nonetheless, they are analyzed by simulators or synthesizers as if contained each in a separate file

• Six kind of design units in VHDL
  – entity
    • architecture
  – package
    • package body
  – configuration declaration
  – context declaration

  new in IEEE 1076-2008
Entities and architectures

- **Entity**: defines interface and name of a circuit
- **Architecture**: defines what circuit does by describing its content
  - it is a secondary unit ⇒ may exist only associated to an entity
  - an entity may have multiple architectures, each representing a different implementation of the circuit
Entities and architectures: example

- **entity, port clause**
  - terminal names: A, B, ...
  - direction: in, out
  - data type: Bit

- **architecture**
  - declarative part (before `begin`) where inner signals are declared
  - statement part (between `begin` and `end`) where circuit behavior (or structure) is described

```
entity FADD is
  port (A, B, CI: in Bit;
       CO, S: out Bit);
end FULL_ADDER;

architecture BHV of FADD is
  signal P, S1, G, C1: Bit;
begin
  P  <= A xor B;
  G  <= A and B;
  S1 <= P xor CI;
  C1 <= P and CI;
  S  <= S1;
  CO <= C1 or G;
end BHV;
```

A. Neviani - P.S.C.D. 8
entity <entity_name> is
  port(
    -- port assignments
    <signal_name>: in|out|inout type;
    ...
    -- all port assignments are followed by semi-colons
    -- except the last one
  );
generic(
    -- generic assignments
    <generic_name>: type [:= <initial_value>];
    ...
    -- all generic assignments are followed by
    -- semi-colons except the last one
);
end [entity | <entity_name>];
Architectures: statement part

Circuit description inside an architecture is done in the statement part through concurrent statements, whose execution is triggered by an event on signals the statement is sensitive to. There are three types of concurrent statements:

• **signal assignments** (simple, conditional and selected)

• **component instances**, (the component behavior being described by another entity – architecture couple)

• **process statements**, that define an environment for procedural statements
Architecture: syntax

architecture <architecture_name> of <entity_name> is

[component declarations
  function declarations
  signal declarations
  constant declarations
  variable declarations
  type declarations
  ...]

begin

[-- concurrent statements
  signal assignment
  component instance
  process statement
]

end architecture;
Architectures: example

architecture MIX of FADD is
  component XOR2
    port(X, Y: in Bit;
         Z: out Bit);
  end component;
signal P, G: Bit;
begin
  -- signal assignment
  CO <= (P and CI) or G;
  -- component instance
  S1: XOR2 port map(X => P,
                     Y => CI, Z => S);
  -- process statement
  PG: process(A, B, CI)
     begin
      -- signal assignment
      G <= A and B;
      -- procedural statements
      if (A = B) then
        P <= '0';
      else
        P <= '1';
      end if;
  end process;
end MIX;
Signals

- **VHDL signals** are the language objects used to represent actual signals inside a circuit
  - can be thought of as an ordered list of data couples \((t_k, v_k)\)
    - \(v_k\) = signal value
    - \(t_k\) = time when signal took that value
  - possible values depend on signal type
  - signals must be declared
    ```vhdl
    signal <signal_name> : type [:=initial value];
    ```
  - **ports** are signals with type and direction that provide an interface through the entity
    ```vhdl
    <signal_name>: in|out|inout type;
    ```
  - signal value is determined by a driver waveform through a signal assignment statement
    ```vhdl
    <signal_name> <= [delay_mechanism] <waveform> [delay];
    ```
Signal assignment: important notice

• Signal assignment in VHDL is a concurrent statement
  – it works very differently compared to variable assignment in a procedural programming language like C or Java
  – concurrency means that the statement is executed only when one or more signals in the driver undergo an event (i.e., a change of value)
  – assignement of the driver value to the target is not immediate, it rather occurs through an event queue
VHDL simulation model

• How VHDL allows to describe a circuit is strictly linked to the language simulation model
  – the definition of the simulation model is contained in the VHDL reference manual

• The model is based on event-driven simulation. Three essential concepts:
  – simulation time
  – delta time (or delta cycle)
  – event processing / process execution
VHDL event-driven simulation

• VHDL simulators keep track of the internal time that has been simulated (*simulation time*):
  – measured as integral multiple of basic time unit
  – time delays smaller that basic unit time not simulated

• Simulation cycle alternates between event processing and process execution:
  – signals are updated during event processing phase and trigger the execution of concurrent statements
  – triggered concurrent statements are executed during process execution phase and generate *signal transactions* (i.e., potential changes of signal values)
VHDL event-driven simulation

- Example: assume the simple assignement below is executed at simulation time $t_1=12$ ns
  
  ```vhdl
  A <= '1';
  ```

- A transaction with value ‘1’ is scheduled on the event (or transaction) queue at time $t_2=12$ ns + $\delta$

<table>
<thead>
<tr>
<th></th>
<th>12+$\delta$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A⟵‘1’</td>
<td></td>
</tr>
</tbody>
</table>

- During event processing, when the transaction becomes due, signal $A$ becomes active

- If the transaction changes the value of $A$, then an event is generated for $A$; otherwise, the transaction is discarded
  - the couple (time, new value) is added to the signal waveform data structure $\Rightarrow$ in this example (12, ‘1’)

A. Neviani - P.S.C.D. 17
VHDL event-driven simulation

Important notes:

- Zero-delay assignments (like A <= ‘1’;) cause scheduling a transaction with a delta delay with respect to current simulation time
- A signal becoming active does not necessarily end up in a signal event
  - transaction ≠ event
- A signal event (generated during the event processing phase) triggers the execution of processes sensitive to that signal (process execution phase)
- Process execution schedules new transactions in the event queue
- Simulation ends when the event queue is empty
VHDL simulation example

Consider the following input signal waveforms and logic circuit:
VHDL simulation example

VHDL code modeling the input signals and the logic circuit:

```vhdl
signal A,B,CI,P,G,S,CO: std_logic;

-- input signals waveforms
A  <= '0', '1' after 5 ns;
B  <= '0', '1' after 10 ns,
CI <= '0';
-- assignements modeling a full adder
P  <= A xor B;
G  <= A and B;
S  <= P xor CI;
CO <= (P and CI) or G;
--
...
```

VHDL simulation example

Process execution:
Simulation begins. Signal values are initialized. All processes are executed once. Transactions are scheduled in the event queue.
VHDL simulation example

Event processing:
Transactions at time 0 in the queue are processed. Three of them are events (A, B and Cl change from ‘U’ to ‘0’). New values added in A, B and Cl (not shown) data structures.
## VHDL simulation example

### Signals data structures

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>U 0</td>
<td>U 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

### Event queue

<table>
<thead>
<tr>
<th>...</th>
<th>0</th>
<th>0+δ</th>
<th>0+δ</th>
<th>0+δ</th>
<th>0+δ</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>CO→U</td>
<td>P→0</td>
<td>G→0</td>
<td>S→U</td>
<td>CO→U</td>
<td>A→1</td>
<td>B→1</td>
</tr>
</tbody>
</table>

### Process execution:

Processes sensitive to A, B and CI are executed:

- **P** <= A xor B;
  - expressions evaluate to ‘0’ since value of A and B is ‘0’
- **G** <= A and B;
- **S** <= P xor CI;
  - expressions evaluate to ‘U’ since value of P and G is ‘U’
- **CO** <= (P and CI) or G;

New transactions on P and G added to the event queue.
Event processing:
The timewheel moves to $t=0+\delta$ and the corresponding transactions in the queue are processed. Two of them are events (value of P and G changes from ‘U’ to ‘0’), the remaining two are not events and are discarded. New values added in P and G data structures.
Process execution:
Processes sensitive to P and G are executed:

\[ S \leftarrow P \text{ xor } CI; \]

\[ CO \leftarrow (P \text{ and } CI) \text{ or } G; \]

New transactions on S and CO added to the event queue.
**VHDL simulation example**

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>0</th>
<th>U</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>0</th>
<th>0</th>
<th>U</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>0</th>
<th>0</th>
<th>U</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>0</td>
<td>0</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

**Event queue**

<table>
<thead>
<tr>
<th>...</th>
<th>0</th>
<th>0+δ</th>
<th>0+δ</th>
<th>0+2δ</th>
<th>0+2δ</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>CO→U</td>
<td>P→0</td>
<td>G→0</td>
<td>S→0</td>
<td>CO→0</td>
<td>A→1</td>
<td>B→1</td>
</tr>
</tbody>
</table>

**Event processing:**

The timewheel move to t=0+2δ and the corresponding transactions in the queue are processed. They are both events (value of S and CO changes from ‘U’ to ‘0’). New values added in S and CO data structures.
VHDL simulation example

Process execution:
There are no processes sensitive to signals S and CO. Process execution ends with no operations.
VHDL simulation example

Event processing:
The timewheel move to $t=5$ and the corresponding transaction in the queue is processed. It is an event on signal A, whose new value is added in the corresponding data structure.
VHDL simulation example

Process execution:
Processes sensitive to A are executed:

\[ P \leftarrow A \text{xor} B; \]
\[ G \leftarrow A \text{and} B; \]

New transactions on P and G added to the event queue.
Event processing:
The timewheel move to $t=5+\delta$ and the corresponding transactions in the queue are processed. One is an event on signal $P$, the other is not an event.
VHDL simulation example

Process execution:
Processes sensitive to P are executed:

\[
\begin{align*}
S & \leftarrow P \text{ xor } CI; \\
CO & \leftarrow (P \text{ and } CI) \text{ or } G;
\end{align*}
\]

New transactions on S and CO added to the event queue.
### VHDL simulation example

**Event queue**

<table>
<thead>
<tr>
<th>...</th>
<th>0+2δ</th>
<th>5</th>
<th>5+δ</th>
<th>5+2δ</th>
<th>5+2δ</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>CO→0</td>
<td>A→1</td>
<td>P→1</td>
<td>S→1</td>
<td>CO→0</td>
<td>B→1</td>
</tr>
</tbody>
</table>

**Event processing:**

The timewheel move to $t=5+2\delta$ and the corresponding transactions in the queue are processed. One is an **event** on signal S, the other is not an event.
VHDL simulation example

| A | 0 | 0 | 5 |
| U | 0 | 1 |
| B | 0 | 0 |   |
|   | U | 0 |   |

| P | 0 | 0 | 5 |
| U | 0 | 1 |

| G | 0 | 0 |   |
|   | U | 0 |   |

| S | 0 | 0 | 5 |
| U | 0 | 1 |

| CO | 0 | 0 |   |
|    | U | 0 |   |

**event queue**

| ... | 0+2δ | 5   | 5+δ | 5+2δ | 5+2δ | 10  |
| ... | CO→0 | A→1 | P→1 | S→1  | CO→0 | B→1 |

**Process execution:**

There are no processes sensitive to signal S. Process execution ends with no operations.
VHDL simulation example

Event processing:
The timewheel move to $t=10$ and the corresponding transaction in the queue is processed. It is an event on signal B, whose value is updated in the corresponding data structure.
VHDL simulation example

Process execution:
Processes sensitive to B are executed:

\[ P \leftarrow A \text{ xor } B; \]
\[ G \leftarrow A \text{ and } B; \]

New transactions on \( P \) and \( G \) added to the event queue.
VHDL simulation example

Event processing:
The timewheel move to $t=10+\delta$ and the corresponding transactions in the queue are processed. They are both events, one on signal P, the other on signal G.
VHDL simulation example

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>0</th>
<th>5</th>
<th>U</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>U</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

| P | 0 | 0 | 5 | 10| U | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| G | 0 | 0 | 10| U | 0 | 1 |

<table>
<thead>
<tr>
<th>S</th>
<th>0</th>
<th>0</th>
<th>5</th>
<th>U</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>U</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**event queue**

<table>
<thead>
<tr>
<th>...</th>
<th>5+2Δ</th>
<th>10</th>
<th>10+Δ</th>
<th>10+Δ</th>
<th>10+2Δ</th>
<th>10+2Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>CO→0</td>
<td>B→1</td>
<td>P→0</td>
<td>G→1</td>
<td>S→0</td>
<td>CO→1</td>
</tr>
</tbody>
</table>

**Process execution:**

Processes sensitive to P and G are executed:

\[ S \leftarrow P \text{xor} \text{ CI}; \]
\[ \text{CO} \leftarrow (P \text{ and } \text{ CI}) \text{ or } G; \]

New transactions on S and CO added to the event queue.
VHDL simulation example

Event processing:
Processes sensitive to P and G are executed:
\[ S \leftarrow P \text{xor} \ CI; \]
\[ \text{CO} \leftarrow (P \text{ and } \ CI) \text{ or } G; \]
New transactions on S and CO added to the event queue.
**VHDL simulation example**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>P</th>
<th>S</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 5</td>
<td>0 0 10</td>
<td>0 0 5 10</td>
<td>0 0 1 0</td>
<td>0 0 10</td>
</tr>
<tr>
<td>U 0 1</td>
<td>U 0 1</td>
<td>U 0 1</td>
<td>U 0 1</td>
<td>U 0 1</td>
</tr>
</tbody>
</table>

**event queue**

<table>
<thead>
<tr>
<th>...</th>
<th>5+2δ</th>
<th>10</th>
<th>10+δ</th>
<th>10+δ</th>
<th>10+2δ</th>
<th>10+2δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO→0</td>
<td>B→1</td>
<td>P→0</td>
<td>G→1</td>
<td>S→0</td>
<td>CO→1</td>
<td></td>
</tr>
</tbody>
</table>

**Process execution:**
There are no processes sensitive either to signal S or to signal CO. Process execution ends with no operations.
VHDL simulation example

Event processing:
There are no further transactions in the queue. The simulation ends.
VHDL simulation example

Simulation result post-processing:
A graphical representation of signals waveforms can be derived from signals data structures.
Exercise: manually execute the following VHDL code:

```vhdl
...  
-- signal declaration and initialization
signal R, S, Qb: std_logic := '0';
signal Q: std_logic := '1';
-- input signal waveform
R  <= '1';
-- logic circuit model
Q  <= R nor Qb;
Qb <= S nor Q;
... 
```
Logic synthesis is a step that transforms a Register Transfer Level Model (RTL) in a gate netlist.
VHDL synthesis mechanism

- VHDL was designed for circuit documentation and simulation, not synthesis
  - synthesizers (a.k.a. compilers) must make an interpretation of the language
  - interpretation based on special VHDL constructs (templates) that map to hardware with equivalent behavior
  - VHDL models restricted to a semantic subset of the language in order to be synthesizable
Logic synthesis process

• Logic synthesis can be decomposed in three steps, the second and the third being intertwined:
  – **translation**: the model is translated into a network of logic (boolean functions, generic memory elements, MUX, DEMUX) and arithmetic (adders, multipliers, comparators) templates
  – **optimization**: the intermediate logic network is turned into a set of boolean expressions that are manipulated to meet given optimization goals (delay/area/power)
  – **technology mapping**: the boolean expressions are restructured in order to match the actual gates available in the technology library
Logic synthesis process: example

Consider a full-adder described by the following VHDL model:

```vhdl
entity FULL_ADDER is
    port (A, B, CI: in Bit;
          CO, S: out Bit);
end FULL_ADDER;

architecture BHV of FULL_ADDER is
signal P, S1, G, C1: Bit;
begin
    P  <= A xor B;
    G  <= A and B;
    S1 <= P xor CI;
    C1 <= P and CI;
    S  <= S1;
    CO <= C1 or G;
end BHV;
```
Logic synthesis process: example

- The model is translated into an intermediate format using templates of generic logic gates.
- Every signal assignment maps directly into a block of combinational logic.
- More complex constructs (e.g., process statements) may result into abstract templates (adders, generic memory elements) with no evident mapping to logic gates.
Logic synthesis process: example

• A set of boolean equations is derived from the intermediate representation

• Equations are manipulated to meet a set of optimization goals (delay/area/power below a given value)

• In this example, the logic depth has been reduced to decrease propagation delays
Logic synthesis process: example

- In the **technology mapping** phase, the logic expressions are restructured to match the actual gates available in the technology library.
- Technology mapping and optimization are performed together and iteratively, since optimization requires data on gate delay/area/power.
Logic synthesis process: example

Logic synthesis output: VHDL gate netlist using components NAND2, NAND3, XOR2 from a technology library

```
architecture STR of FULL_ADDER is
  -- component declaration
  component NAND2
    port(A, B: in bit;
       Z: out bit);
  end component;
  component NAND3
    port(A, B, C: in bit;
       Z: out bit);
  end component;
  component XOR2
    port(A, B: in bit;
       Z: out bit);
  end component;
  -- internal connections
  signal X1, X2, X3, X4: bit;
begin
  -- gate netlist (rete di celle)
  G1: NAND2 port map(A => A,
                      B => B, Z => X1);
  G2: NAND2 port map(A => B,
                      B => CI, Z => X2);
  G3: NAND2 port map(A => CI,
                      B => A, Z => X3);
  G4: NAND3 port map(A => X1,
                      B => X2, C => X3, Z => CO);
  G5: XOR2 port map(A => A,
                      B => B, Z => X4);
  G6: XOR2 port map(A => CI,
                      B => X4, Z => S);
end STR;
```
VHDL package and package body

- A **package** is a primary design unit containing the declarations of data types, constants, functions and procedures that may be shared by different VHDL models.
- A **package body** is a secondary design unit, linked to a package, containing the definition of functions and procedures declared in the package.

```vhdl
package PKG1 is
    type LOGIC3 is ('0', '1', 'z');
    constant LO: LOGIC3 := '0';
    constant HI: LOGIC3 := '1';
    function INVERT (i: LOGIC3) return LOGIC3;
-- other function and data type declarations
... end PKG1;
```

```vhdl
package body PKG1 is
    function INVERT (i: LOGIC3) return LOGIC3 is
    begin
        case i is
            when '0' => return '1';
            when '1' => return '0';
            when 'z' => return 'z';
        end case;
    end INVERT;
-- other function definitions
... end BHV;
```
Libraries

• In VHDL, a library is a collection of analyzed design units
  – analysis translates a VHDL model into an intermediate form before simulation
  – reference to a library occurs through a logic name
  – logic name WORK identifies the default library where models analyzed by the user are stored
  – STD predefined library contains packages STANDARD and TEXTIO
  – library IEEE contains several useful packages, among which STD_LOGIC_1164 and NUMERIC_STD

• Except for STD, which is predefined, library access requires a library declaration before each primary unit, and a use clause to declare package access

  library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
Overall view of VHDL design units

- design entity
- entity declaration
- package declaration
- package body
- architecture body 1
- architecture body 2
- architecture body N
- configuration