Circuit Modeling with Hardware Description Languages

Prof. Hubert Kaeslin
Microelectronics Design Center
ETH Zürich

Morgan Kaufmann “Top-Down Digital VLSI Design” Chapter 4

last update: July 18, 2014
Content

You will learn
to write high-quality HDL models for circuit synthesis and simulation.

- Why hardware synthesis?
- Key concepts behind hardware description languages
  - What sets HDLs apart from a programming language
  - Essential VHDL and/or SystemVerilog language constructs
- Putting HDLs to service for hardware synthesis
  - Synthesis subset
  - Patterns for registers and finite state machines
  - Timing constraints
  - How to establish a register transfer level model
- VHDL versus SystemVerilog

Simulation and testbench coding are postponed to chapter 5 “Functional Verification”.
Motivation and background
Why hardware synthesis?

Current situation for VLSI designers:

- Buyers ask for more and more functions on a single chip.
- Technology supports ever higher integration densities (Moore’s law).
- Market pressure vetoes dilation of development times.
Why hardware synthesis?

Current situation for VLSI designers:

- Buyers ask for more and more functions on a single chip.
- Technology supports ever higher integration densities (Moore’s law).
- Market pressure vetoes dilation of development times.

Hardware description languages (HDL) and design automation come to the rescue in four ways:

- Move design entry to higher levels of abstraction.
- Allow designers to focus on functionality as synthesis tools automate the construction of structural and physical views.
- Facilitate reuse by capturing a circuit description in a parametrized, technology- and platform-independent form.
- Make functional verification more efficient by supporting stimuli generation, automatic response checking, assertion-based verification, etc.
## Languages for modeling digital hardware I

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VHDL</strong></td>
<td>An HDL that not only supports structural and behavioral circuit models but testbench models too. A subset is synthesizable. Syntactically similar to Ada.</td>
</tr>
<tr>
<td><strong>Verilog</strong></td>
<td>Conceptually similar to VHDL, no type checking and more limited capabilities for design abstraction. A subset is synthesizable. Syntactically similar to C. Superseded by ...</td>
</tr>
<tr>
<td><strong>System-Verilog</strong></td>
<td>A superset of Verilog that includes many advanced features from VHDL and from specialized verification languages (OpenVera, PLS). A subset is synthesizable. Supports object-oriented programming.</td>
</tr>
<tr>
<td><strong>SystemC</strong></td>
<td>Extends C++ with class libraries and a simulation kernel. Adds <strong>clocking</strong> information to C++ functions. Intended for software/hardware co-design and co-simulation. Does not support any <strong>timing</strong> finer than one clock cycle. Synthesis path is via translation to RTL VHDL or Verilog.</td>
</tr>
</tbody>
</table>
## Languages for modeling digital hardware II

<table>
<thead>
<tr>
<th>Criterion</th>
<th>VHDL</th>
<th>Verilog</th>
<th>SystemVerilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis support</td>
<td>yes</td>
<td>yes</td>
<td>growing</td>
</tr>
<tr>
<td>Parametrization &amp; abstract modeling</td>
<td>good</td>
<td>poor</td>
<td>good</td>
</tr>
<tr>
<td>Type checking &amp; scoping rules</td>
<td>strong</td>
<td>none</td>
<td>loose</td>
</tr>
<tr>
<td>Deterministic event queue mechanism</td>
<td>yes</td>
<td>not really</td>
<td>not really</td>
</tr>
<tr>
<td>Modeling of electric phenomena</td>
<td>9-valued</td>
<td>4-valued</td>
<td>4-valued</td>
</tr>
<tr>
<td>High-level verification support</td>
<td>limited</td>
<td>poor</td>
<td>excellent</td>
</tr>
</tbody>
</table>

For me, I find VHDL is like swimming with a lifeguard on duty, whereas Verilog is like swimming with a lifebuoy hanging by the poolside. (Blogger on EETimes 2011)

Many companies currently use VHDL for synthesis and SystemVerilog for system-level verification. Will SystemVerilog one day supersede Verilog and VHDL, and reconcile their user communities?
Languages for modeling digital hardware II

<table>
<thead>
<tr>
<th>Criterion</th>
<th>VHDL</th>
<th>Verilog</th>
<th>SystemVerilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis support</td>
<td>yes</td>
<td>yes</td>
<td>growing</td>
</tr>
<tr>
<td>Parametrization &amp; abstract modeling</td>
<td>good</td>
<td>poor</td>
<td>good</td>
</tr>
<tr>
<td>Type checking &amp; scoping rules</td>
<td>strong</td>
<td>none</td>
<td>loose</td>
</tr>
<tr>
<td>Deterministic event queue mechanism</td>
<td>yes</td>
<td>not really</td>
<td>not really</td>
</tr>
<tr>
<td>Modeling of electric phenomena</td>
<td>9-valued</td>
<td>4-valued</td>
<td>4-valued</td>
</tr>
<tr>
<td>High-level verification support</td>
<td>limited</td>
<td>poor</td>
<td>excellent</td>
</tr>
</tbody>
</table>

For me, I find VHDL is like swimming with a lifeguard on duty, whereas Verilog is like swimming with a lifebuoy hanging by the poolside. (Blogger on EETimes 2011)

- Many companies currently use VHDL for synthesis and SystemVerilog for system-level verification.
- Will SystemVerilog one day supersede Verilog and VHDL, and reconcile their user communities?
Motivation and background

Key concepts and constructs of VHDL

Key concepts and constructs of SystemVerilog

Automatic circuit synthesis from HDL models

Conclusions

Why hardware synthesis?

Alternatives for modeling digital hardware

Why bother learning hardware description languages?

A first look at VHDL and SystemVerilog

HDLs shown in the Y-chart

© Hubert Kaeslin Microelectronics Design Center ETH Zürich

Circuit Modeling with Hardware Description Languages 7 / 227
The genesis of VHDL

1983  US DoD commissions IBM, Intermetrics and Texas Instruments to define a HDL for documentation purposes. Ada is taken as a starting point. There are no plans for automatic synthesis.

1986  Military restrictions lifted, rights transferred to IEEE.


1993  Language standard significantly revised to become IEEE 1076-93. Nine-valued logic system accepted as IEEE 1164 standard. Though confined to a language subset, synthesis begins to catch on.

1999  A major extension towards modeling of analog and mixed-signal circuits is accepted as separate a standard IEEE 1076.1.


The genesis of SystemVerilog

1984  Gateway Design Autom. develops Verilog for a proprietary logic simulator.
1989  Gateway acquired by Cadence.
1990  Verilog made an open standard.
1995  Verilog accepted as IEEE 1364 standard. (questionable politics involved)
2001  IEEE 1364-2001 brings major extensions for circuit modeling.
2005  IEEE 1364-2005 is a minor revision.

SystemVerilog, created by the Accellera consortium, is accepted as a separate standard named IEEE 1800. (more politics involved)

Quiz: “What do Sausage and EDA Standards have in common?”
The genesis of SystemVerilog

1984  Gateway Design Autom. develops Verilog for a proprietary logic simulator.
1989  Gateway acquired by Cadence.
1990  Verilog made an open standard.
1995  Verilog accepted as IEEE 1364 standard. (questionable politics involved)
2001  IEEE 1364-2001 brings major extensions for circuit modeling.
2005  IEEE 1364-2005 is a minor revision.
       SystemVerilog, created by the Accellera consortium, is accepted as a separate standard named IEEE 1800. (more politics involved)

Quiz: “What do Sausage and EDA Standards have in common?”
Answer: “Those who like sausage or EDA standards should never watch either one be made!” (Stuart Sutherland).

Why bother learning hardware description languages? I

Idea: View HDLs as nothing more than intermediate formats for exchanging data between system design tools and VLSI CAE/CAD suites. Have electronic system-level tools generate code from specs automatically.
Why bother learning hardware description languages? I

Idea: View HDLs as nothing more than intermediate formats for exchanging data between system design tools and VLSI CAE/CAD suites. Have electronic system-level tools generate code from specs automatically.

△ Software for system design has a focus, there is no universal tool.
  ○ Transformatorial systems as found in signal processing and telecommunications.
  ○ Reactive system as found in controllers and interface protocols.
  ○ Specific applications such as data networks, image processing, instruction set computers, etc.
  ○ HDL generators typically restricted to few predefined architectural patterns.

△ HDL code generated by most ESL tools is nothing else than a translation of software code and inadequate for circuit synthesis.

△ HDLs are indispensable for modeling library cells and virtual components.
Why bother learning hardware description languages? II

△ HDLs are being used all along digital VLSI design flows.

<table>
<thead>
<tr>
<th>Design stage &amp; model</th>
<th>Main purpose</th>
<th>Level of abstraction</th>
<th>Timing</th>
<th>Predominant languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Algorithmic model</td>
<td>system-level simulation</td>
<td>behavioral</td>
<td>none</td>
<td>C, MATLAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>tentative</td>
<td>VHDL, SysVer</td>
</tr>
<tr>
<td>2. RTL model</td>
<td>simulation</td>
<td>register transfer</td>
<td>optional fake delays</td>
<td>VHDL, SysVer</td>
</tr>
<tr>
<td></td>
<td>synthesis</td>
<td></td>
<td>constraints in Tcl</td>
<td></td>
</tr>
<tr>
<td>3. Post-synthesis netlist</td>
<td>simulation, timing analysis, place &amp; route</td>
<td>gate level</td>
<td>estimated with wire load models</td>
<td>Verilog, (VHDL&amp;VITAL)</td>
</tr>
<tr>
<td>4. Post-layout netlist</td>
<td>simulation, timing analysis, sign-off</td>
<td>gate level</td>
<td>extracted from layout and back-annotated</td>
<td>Verilog, (VHDL&amp;VITAL)</td>
</tr>
</tbody>
</table>
Why bother learning hardware description languages? II

△ HDLs are being used all along digital VLSI design flows.

<table>
<thead>
<tr>
<th>Design stage &amp; model</th>
<th>Main purpose</th>
<th>Level of abstraction</th>
<th>Timing</th>
<th>Predominant languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Algorithmic model</td>
<td>system-level simulation</td>
<td>behavioral</td>
<td>none</td>
<td>C, MATLAB</td>
</tr>
<tr>
<td>2. RTL model</td>
<td>simulation synthesis</td>
<td>register transfer</td>
<td>optional fake delays</td>
<td>VHDL, SysVer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>constraints in Tcl</td>
<td></td>
</tr>
<tr>
<td>3. Post-synthesis netlist</td>
<td>simulation, timing analysis, place &amp; route</td>
<td>gate level</td>
<td>estimated with wire load models</td>
<td>Verilog, (VHDL &amp; VITAL)</td>
</tr>
<tr>
<td>4. Post-layout netlist</td>
<td>simulation, timing analysis, sign-off</td>
<td>gate level</td>
<td>extracted from layout and back-annotated</td>
<td>Verilog, (VHDL &amp; VITAL)</td>
</tr>
</tbody>
</table>

Conclusion

For the foreseeable future, **VHDL** and **SystemVerilog** are bound to remain prominent hubs for all VLSI design activities.
Requirements for HDLs

Show around a motherboard or some other mounted PCB.

What features must a formal language have to capture the essence of electronic circuitry?
A first look at VHDL and SystemVerilog

In a nutshell, HDLs can be characterized as follows:

\[
\begin{align*}
\text{HDL} &= \text{Structured programming language} \\
&+ \text{circuit hierarchy and connectivity} \\
&+ \text{interacting concurrent processes} \\
&+ \text{a discrete replacement for electrical signals} \\
&+ \text{an event-driven scheme of execution} \\
&+ \text{model parametrization facilities} \\
&+ \text{verification aids}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Feature</th>
<th>VHDL</th>
<th>SysVer</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit hierarchy and connectivity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>interacting concurrent processes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>a discrete replacement for electrical signals</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>an event-driven scheme of execution</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>model parametrization facilities</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>verification aids</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 5
A first look at VHDL and SystemVerilog

In a nutshell, HDLs can be characterized as follows:

<table>
<thead>
<tr>
<th>HDL</th>
<th>Structured programming language</th>
<th>VHDL</th>
<th>SysVer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>circuit hierarchy and connectivity</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>interacting concurrent processes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>a discrete replacement for electrical signals</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>an event-driven scheme of execution</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>model parametrization facilities</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>verification aids</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Limitation:
- No way to express timing constraints
Linguistic ambiguity in the context of hardware modeling:

<table>
<thead>
<tr>
<th>Meaning of “sequential” with reference to</th>
<th>Synonym</th>
<th>Antonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>○ program execution during simulation</td>
<td>step-by-step</td>
<td>concurrent, parallel</td>
</tr>
<tr>
<td>○ nature of circuit being modeled</td>
<td>memorizing</td>
<td>combinational, memoryless</td>
</tr>
</tbody>
</table>
... before we go into the details

Teaching follows two threads:

**Lab hours** Become acquainted with software tools and acquire coding skills.

**Lectures** Understand the underlying concepts and mechanisms.

- modeling of electrical phenomena
- simulation cycle
- testbench design
- synthesis procedure
- handling of macrocells (RAM)
- delay modeling, timing checks, timing constraints
- code portability
- ...

Both are needed! Circuit design is neither pure theory nor ignorant hacking.

A fool with a tool is still a fool.
... before we go into the details

Teaching follows two threads:

**Lab hours**  Become acquainted with software tools and acquire coding skills.

**Lectures**  Understand the underlying concepts and mechanisms.
- modeling of electrical phenomena
- simulation cycle
- testbench design
- synthesis procedure
- handling of macrocells (RAM)
- delay modeling, timing checks, timing constraints
- code portability
- ...

Both are needed!  Circuit design is neither pure theory nor ignorant hacking.

A fool with a tool is still a fool.
Key concepts and constructs of VHDL

For a SystemVerilog course, skip the next 95 or so slides.
Hardware description language requirements

HDL requirement no.1

Means for expressing how circuits are being composed from subcircuits and how those subcircuits connect to each other.
1st HDL capability: Circuit hierarchy and connectivity

Figure: Hierarchical composition ...
Entity declaration

Specifies the external interface of a design entity (small or large).

```vhdl
-- entity declaration
entity lfsr4 is
  port (
    Clk_CI : in std_logic;
    Rst_RBI : in std_logic; -- reset is active low
    Ena_SI : in std_logic;
    Oup_DO : out std_logic);
end lfsr4;
```

- A port list declares all signals of an entity that are accessible from outside (i.e. the terminals of a circuit as opposed to its inner nodes).
Architecture body I: a structural circuit model

Figure: Linear-feedback shift register circuit to be described.
Architecture body I: a structural circuit model

Refer to transparency lfsr4struc.vhd for code!

Describes a circuit or netlist assembled from components and wires.

1. Declare all components to be used.
2. Declare all signals that run back and forth unless they are already known from the port clause.
3. Instantiate components specifying all terminal-to-signal connections.
How to compose a circuit from components

How do you proceed when asked to fit a circuit board with components?

1. Think of a part’s exact name, e.g. GTECH_FD2
2. Fetch a copy and assign it some unique identifier it, e.g. u10
3. Solder its terminals to existing metal pads on the board

The component instantiation statement does exactly that. Example:

```vhdl
u10 : GTECH_FD2
    port map( D => n11, CP => Clk_CI, CD => Rst_RBI, Q => State_DP(1) );
```

**Note**

The association operator `=>` does not indicate an assignment but an association of two signals that stands for an electrical connection.
The essence of structural circuit modeling

- VHDL can describe the hierarchical composition of a circuit by
  - instantiating components or entities and by
  - interconnecting them with the aid of signals.
- Structural HDL models hold the same information as circuit netlists do.
- Manually writing structural HDL models is not particularly attractive.
- Most structural models are in fact obtained from RTL models by automatic synthesis.
The essence of structural circuit modeling

- VHDL can describe the hierarchical composition of a circuit by
  - instantiating components or entities and by
  - interconnecting them with the aid of signals.

- Structural HDL models hold the same information as circuit netlists do.
- Manually writing structural HDL models is not particularly attractive.
- Most structural models are in fact obtained from RTL models by automatic synthesis.

**HDL requirement no.2**

Means for expressing circuit behavior including the combined effects of multiple subcircuits that operate jointly and concurrently.
2nd HDL capability: Interacting concurrent processes

Figure: ... plus behavior modeled with the aid of concurrent processes ...
Constants, variables, and signals

What everyone knows from software languages:

- **Constant declaration**
  - Example
  
    ```vhdl
class constant FERMAT_PRIME_4 : integer := 65537;
    ```

- **Variable declaration**
  - Examples
  
    ```vhdl
    class variable Brd : real := 2.48678E5;
    variable Ddr : real := 1.08179E5;
    ```

- **Variable assignment**
  - Example
  
    ```vhdl
    Brd := Brd + Ddr;
    ```
Constants, variables, and signals

What everyone knows from software languages:

- **Constant declaration**
  
  Example

  ```
  constant FERMAT_PRIME_4 : integer := 65537;
  ```

- **Variable declaration**
  
  Examples

  ```
  variable Brd : real := 2.48678E5;
  variable Ddr : real := 1.08179E5;
  ```

- **Variable assignment**
  
  Example

  ```
  Brd := Brd + Ddr;
  ```

... plus a special vehicle for exchanging information between processes:

- **Signal declaration**
  
  Example

  ```
  signal Error_D, Actual_D, Wanted_D : integer := 0;
  ```

- **Signal assignment**
  
  Example

  ```
  Error_D <= Actual_D - Wanted_D;
  ```
Practical advice

Hints

- VHDL is case-insensitive, e.g. `clk_ci = CLK_CI` (except for extended identifiers written between backslashes, e.g. \clk\_ci\ \neq \CLK\_CI\ ).

- Naming a signal or a port `In` or `Out` is all too tempting, yet these are reserved words in VHDL. We recommend `Inp` and `Oup` instead.

- Two distinct symbols are being used for variable assignment `:=` and for signal assignment `<=`.

- Code is easier to read when `signals` can be told from variables by their visual appearance. We append an underscore followed by a suffix of upper-case letters to `signals`, e.g. `Carry_DB`, `AddrCnt_SN`, `Irq_AMI`.

Details of our naming convention are to follow in chapter 6 “The Case for Synchronous Design”.

How to describe combinational logic behaviorally I

1. Concurrent signal assignment:
   - Syntactically simplest form of a process.
   - Drives one signal.

Example:

```vhdl
signal Aa_D, Bb_D, Cc_D, Oup_D : std_logic;
.....

Oup_D <= Aa_D xor (Bb_D and not Cc_D)
```

- Typically used to model some combinational behavior (such as an arithmetic or logic operation) when there is no need for branching.

Glimpse ahead: A concurrent/selected/conditional signal assignment gets activated by any change of any signal on the right-hand side.
How to describe combinational logic behaviorally II

2. Selected signal assignment. Example:

type month is (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER);
signal ThisMonth_D : month;
type quarter is (Q1ST, Q2ND, Q3RD, Q4TH);
signal ThisQuarter_D : quarter;
.....

with ThisMonth_D select
    ThisQuarter_D <= Q1ST when JANUARY | FEBRUARY | MARCH,
                    Q2ND when APRIL | MAY | JUNE,
                    Q3RD when JULY | AUGUST | SEPTEMBER,
                    Q4TH when others;

▶ This is a form of conditional execution reminiscent of a multiplexer.

Note: The | symbol separates choices, it does not express a logic or operation.
How to describe combinational logic behaviorally III

3. Conditional signal assignment. Example:

```vhdl
subtype day is integer range 1 to 31;
signal ThisDay_D is day;
signal Spring_D is boolean;
.....

Spring_D <= true when (ThisMonth_D=MARCH and ThisDay_D>=21) or
ThisMonth_D=APRIL or ThisMonth_D=MAY or
(ThisMonth_D=JUNE and ThisDay_D<=20)
else false;
```

- This is a syntactically different form of conditional execution.

Note: There are two <= symbols here. One stands for a signal assignment, the other for a comparison operator.
How to describe combinational logic behaviorally IV


Example:

```vhdl
memless1: process (all)
begin
  Spring_D <= false;  -- execution begins here
  if ThisMonth_D=MARCH and ThisDay_D>=21 then Spring_D <= true; end if;
  if ThisMonth_D=APRIL then Spring_D <= true; end if;
  if ThisMonth_D=MAY then Spring_D <= true; end if;
  if ThisMonth_D=JUNE and ThisDay_D<=20 then Spring_D <= true; end if;
end process memless1;  -- process suspends here
```
Process statement versus signal assignments

When compared to a concurrent/selected/conditional signal assignment, a process statement

▶ is capable of updating two or more signals at a time,
▶ captures the instructions for doing so in a sequence of statements that may not only include branching but also loops,
▶ gives the liberty to make use of variables for temporary storage,
▶ provides more detailed control over the conditions for activation.
Process statement versus signal assignments

When compared to a concurrent/selected/conditional signal assignment, a process statement

- is capable of updating two or more signals at a time,
- captures the instructions for doing so in a sequence of statements that may not only include branching but also loops,
- gives the liberty to make use of variables for temporary storage,
- provides more detailed control over the conditions for activation.

Observation

The process statement is best summed up as being concurrent outside and sequential inside.
How to describe a register behaviorally

- Code example of an edge-triggered register that features
  1. an asynchronous reset,
  2. a synchronous load, and
  3. an enable.

```
p_memzing : process (Clk_C,Rst_RB)
begin
    -- activities triggered by asynchronous reset
    if Rst_RB='0' then
        State_DP <= (others => '0'); -- shorthand for all bits zero
    -- activities triggered by rising edge of clock
    elsif Clk_C'event and Clk_C='1' then
        -- when synchronous load is asserted
        if Lod_S='1' then
            State_DP <= (others => '1'); -- shorthand for all bits one
        -- else assume new value iff enable is asserted
        elsif Ena_S='1' then
            State_DP <= State_DN; -- admit next state into state register
        end if;
    end if;
end process p_memzing;
```
Architecture body II: a behavioral circuit model

Describes how concurrent processes interact via signals and how they alter them.

```vhdl
-- architecture body
architecture behavioral of lfsr4 is
  signal State_DP, State_DN : std_logic_vector(1 to 4);
  -- for present and next state respectively
begin
  -- computation of next state using concatenation of bits
  State_DN <= (State_DP(3) xor State_DP(4)) & State_DP(1 to 3);
  -- updating of state
  process (Clk_CI,Rst_RBI)
  begin
    -- activities triggered by asynchronous reset
    if Rst_RBI='0' then
      State_DP <= "0001";
    -- activities triggered by rising edge of clock
    elsif Clk_CI'event and Clk_CI='1' then
      if Ena_SI='1' then
        State_DP <= State_DN; -- admit next state into state register
      end if;
    end if;
  end process;
  -- updating of output
  Oup_DO <= State_DP(4);
end behavioral;
```

```
D C B A
(1) (2) (3) (4)
State_DP(1) (2) (3) (4)
D C B A
(1) (2) (3) (4)
State_DN(1) (2) (3) (4)

xor
memorizing
process
models
combinational
operations

Ena_SI
Rst_RBI
Clk_CI

memorizing process
models a register

models a wire
Oup_DO
```
The essence of **behavioral** circuit modeling

In VHDL, the behavior of a digital circuit typically gets described by a collection of concurrent processes that

- execute simultaneously, that
- communicate via signals, and where
- each such process represents some subfunction.

**Hint for RTL synthesis**

- Model each register with a `process` statement.
- Prefer concurrent, selected, and conditional signal assignments for describing the combinational logic in between.
Hardware modeling styles

Observation

VHDL allows for procedural, dataflow, and structural modeling styles to be freely combined in a single model.
Procedural, dataflow, and structural models compared I

Refer to transparency fulladd.vhd for code!

Compare in terms of

1. number of processes
2. number of internal signals
3. number of variables
4. impact of ordering of statements
5. interaction with event queue
6. portability of source code

Note: Adders are normally synthesized from algebraic expressions, a full-adder has been chosen here for its simplicity and obviousness.
Procedural, dataflow, and structural models compared II

Figure: Modeling styles illustrated with a full adder as example.
Example: The ones counter

Refer to transparency onescnt.vhd for code!

Observe

1. In spite of its name, this is a memoryless subfunction that finds applications in large adder circuits.

2. The output is a 3 bit number that indicates how many of the four input bits are 1 (logic high).

3. The great diversity of modeling styles to express exactly the same functionality.
Example: The ones counter

Refer to transparency onescnt.vhd for code!

Observe

1. In spite of its name, this is a memoryless subfunction that finds applications in large adder circuits.
2. The output is a 3 bit number that indicates how many of the four input bits are 1 (logic high).
3. The great diversity of modeling styles to express exactly the same functionality.

Observation

Some code examples are compact and easy to understand, others are more cryptic or tend to grow exponentially.
3rd capability: A discrete replacement for electrical signals

Figure: ... plus data types for modeling electrical phenomena ...
What you ought to know about bidirectional busses I

Figure: Memory read and write transfers in a computer.
What you ought to know about bidirectional busses II

Requirements:

- Each bidirectional line is to be driven from multiple places, so one needs a multi-driver signal (as opposed to a single-driver signal).
- Driving alternates.
- Buffers must be able to electrically release the line hence the name “three-state” output (0, 1, disabled output = high-impedance state).
- Requires some kind of access control mechanism (centralized or distributed).
What you ought to know about bidirectional busses II

Requirements:

- Each bidirectional line is to be driven from multiple places, so one needs a multi-driver signal (as opposed to a single-driver signal).
- Driving alternates.
- Buffers must be able to electrically release the line hence the name “three-state” output (0, 1, disabled output = high-impedance state).
- Requires some kind of access control mechanism (centralized or distributed).

Failure modes:

- Stationary drive conflict ↦ functional failure or damage.
- Floating voltage ↦ electrically undesirable condition.

*Presentation focusses on HDL modeling, remedies to be discussed in chapter 10 "Gate- and Transistor-Level Design".*
Why do binary types not suffice to model digital circuits?

Digital circuits exhibit characteristics and phenomena such as

- transients,
- three-state outputs,
- drive conflicts, and
- power-up

that can not be modeled with 0 and 1 alone.
Why do binary types not suffice to model digital circuits?

Digital circuits exhibit characteristics and phenomena such as

- transients,
- three-state outputs,
- drive conflicts, and
- power-up

that cannot be modeled with 0 and 1 alone.

**HDL requirement no.3**

A multi-valued logic system capable of capturing the effects of both node voltage and source impedance.
The IEEE 1164 logic system I

Voltage is quantized into three logic states

- **low** logic low, that is below $U_l$.
- **high** logic high, that is above $U_h$.
- **unknown** either “low”, “high” or anything in between
e.g. as a result from a short between two drivers.
The IEEE 1164 logic system I

Voltage is quantized into three logic states

- low: logic low, that is below $U_l$.
- high: logic high, that is above $U_h$.
- unknown: either “low”, “high” or anything in between
  e.g. as a result from a short between two drivers.

Source impedance gets mapped onto three drive strengths

- strong: as exhibited by a driving output
- high-impedance: as exhibited by a disabled three-state output
- weak: somewhere between “strong” and “high-impedance”
  e.g. as exhibited by a passive pull-up/down resistor.
The IEEE 1164 logic system II

No charge retention in high-impedance state $\leadsto$

- charged low
- charged high
- charged unknown

are all merged into a single value of undetermined state (voltage).
The IEEE 1164 logic system II

No charge retention in high-impedance state ⇐⇒
- charged low
- charged high
- charged unknown

are all merged into a single value of undetermined state (voltage).

Two extra logic values are added, namely:
- uninitialized signal has never been assigned any value since power-up (applicable to simulation only).
- don’t care don’t care condition for logic minimization, distinction between “low” or “high” immaterial (applicable to synthesis only).
The IEEE 1164 logic system III

Uses a total of nine logic values to model electrical signals.

<table>
<thead>
<tr>
<th>logic value</th>
<th>logic state</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>low</td>
</tr>
<tr>
<td>uninitialized</td>
<td>U</td>
</tr>
<tr>
<td>strength</td>
<td>strong</td>
</tr>
<tr>
<td></td>
<td>weak</td>
</tr>
<tr>
<td></td>
<td>high-impedance</td>
</tr>
<tr>
<td>don’t care</td>
<td>–</td>
</tr>
</tbody>
</table>

Defines two data types that share the above set of values:

- `std_ulogic` type
- `std_logic` subtype

*Difference to be explained soon*
Illustrations

Figure: The IEEE 1164 standard MVL-9 illustrated.
Collapsing of logic values during synthesis

L and H are not normally honored by synthesis software. Most synthesis tools collapse “meaningless” (to them) values to more sensible ones, e.g.

- L $\rightarrow$ 0
- H $\rightarrow$ 1
- X or W $\rightarrow$ -
Collapsing of logic values during synthesis

L and H are not normally honored by synthesis software. Most synthesis tools collapse “meaningless” (to them) values to more sensible ones, e.g.

- L \rightarrow 0
- H \rightarrow 1
- X or W \rightarrow -

Hint for RTL synthesis

For the sake of clarity and portability, do not use logic values other than 0, 1, Z and - in VHDL source code that is intended for synthesis.
How to model a bidirectional line in VHDL I

Want to model a circuit node that can be driven from multiple subcircuits?
⇝ Use two or more conditional signal assignments.

Example:

```vhdl
signal Com_DZ, Aa_D, Bb_D, SelA_S, SelB_S : std_logic;

.....
Com_DZ <= not Aa_D when SelA_S='1' else 'Z';
.....
Com_DZ <= not Bb_D when SelB_S='1' else 'Z';
.....
```

**Note**

Node Com_DZ is left floating when neither of the two drivers is enabled.
How to model a bidirectional line in VHDL II

b) single-driver signals $Pp_{DZ}$ and $Qq_{DZ}$ may assume distinct logic values, no difference between `std_ulogic` and `std_logic`

if multi-driver signal $Com_{DZ}$ is of type `std_ulogic` then an error message gets issued
if `std_logic` then the conflict is resolved to $Com_{DZ} = 1$
How to model a bidirectional line in VHDL II

b) single-driver signals \(Pp_{DZ}\) and \(Qq_{DZ}\) may assume distinct logic values, no difference between \(\text{std}\_\text{ulogic}\) and \(\text{std}\_\text{logic}\).

if multi-driver signal \(\text{Com}_{DZ}\) is of type
\(\text{std}\_\text{ulogic}\) then an error message gets issued
\(\text{std}\_\text{logic}\) then the conflict is resolved to \(\text{Com}_{DZ} = 1\)

Observation

The distinction between types \(\text{std}\_\text{ulogic}\) and \(\text{std}\_\text{logic}\) matters only when simulating a multi-driver node:

\(\text{std}\_\text{logic}\) tacitely resolves all conflicts that might occur
\(\text{std}\_\text{ulogic}\) generates a message in case of conflict
The IEEE 1164 standard resolution function

```vhdl
-- resolution function "resolved"

constant resolution_table : stdlogic_table := (
  |
  | U   | X   | 0   | 1   | Z   | W   | L   | H   | -   |
  |
  -- |-----------------------------------------------|
  ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |  |
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |  |
  ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |  |
  ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |  |
  ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |  |
  ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |  |
  ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |  |
  ( 'U', 'X', '0', '1', 'H', 'W', 'H', 'W', 'X' ), -- | H |  |
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |  |
);

▶ This is the default resolution function, others can be added.
```
Data type std_logic versus std_ulogic

- Signals of type std_logic can accommodate multiple drivers whereas those of type std_ulogic can not.
- An error message will tell should a std_ulogic-type signal accidentally get involved in a naming conflict, so this is the more conservative choice.
- A signal is allowed to be driven from multiple processes iff a resolution function is defined that determines the outcome.
- There can be no such thing as a resolution function for variables, neither for bit, bit_vector, integer, real, and similar data types.
### Data types for modeling single-bit signals

<table>
<thead>
<tr>
<th>data type defined in</th>
<th>bit</th>
<th>std_ulogic</th>
<th>std_logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>value set per binary digit</td>
<td>VHDL 2</td>
<td>ieee.std_logic_1164 9</td>
<td>std_logic</td>
</tr>
<tr>
<td>for simulation purposes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>modeling of power-up phase</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>modeling of weakly driven nodes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>modeling of multi-driver nodes</td>
<td>n.a.</td>
<td>reported</td>
<td>resolved</td>
</tr>
<tr>
<td>handling of drive conflicts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for synthesis purposes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>three-state drivers</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>don’t care conditions</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
## Data types for modeling multi-bit signals

<table>
<thead>
<tr>
<th>data type(s)</th>
<th>integer, natural, positive</th>
<th>bit_vector</th>
<th>std_logic_vector</th>
<th>signed, unsigned</th>
<th>signed, unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>defined in</td>
<td>VHDL</td>
<td>VHDL</td>
<td>ieee. std_logic_1164</td>
<td>ieee. numeric_bit</td>
<td>ieee. numeric_std</td>
</tr>
<tr>
<td>value set per binary digit word width</td>
<td>2 bit</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>arithmetic operations</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>logic operations</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>access to subwords or bits</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>modeling of electrical effects</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Data types for modeling multi-bit signals

<table>
<thead>
<tr>
<th>data type(s)</th>
<th>integer, natural, positive</th>
<th>bit_vector</th>
<th>std_logic_vector</th>
<th>signed, unsigned</th>
<th>signed, unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>defined in</td>
<td>VHDL</td>
<td>VHDL</td>
<td>ieee. std_logic_1164</td>
<td>ieee. numeric_bit</td>
<td>ieee. numeric_std</td>
</tr>
<tr>
<td>value set per binary digit word width</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>arithmetic operations</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>logic operations</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>access to subwords or bits</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>modeling of electrical effects</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

- VHDL is strongly typed = extensive type checking is performed
  ⇝ must convert before assignment or comparison across types.
Converting between data types

Figure: VHDL type conversion paths (chart courtesy of Dr. Jürgen Wassner).
Orientation of binary vectors

Hint

Any vector that contains a data item coded in some positional number system should consistently be declared as \((i_{MSB} \text{ downto } i_{LSB})\) where \(2^i\) is the weight of the binary digit with index \(i\).

The MSB so has the highest index assigned to it and appears in the customary leftmost position because \(i_{MSB} \geq i_{LSB}\).

Example

```vhdl
signal Hour_D : unsigned(4 downto 0) := "10111";
```
Orientation of binary vectors

Hint

Any vector that contains a data item coded in some positional number system should consistently be declared as \((i_{\text{MSB}} \text{ downto } i_{\text{LSB}})\) where \(2^i\) is the weight of the binary digit with index \(i\).

The MSB so has the highest index assigned to it and appears in the customary leftmost position because \(i_{\text{MSB}} \geq i_{\text{LSB}}\).

Example

\[
\text{signal Hour_D : unsigned(4 downto 0) := "10111";}
\]

Types unsigned and signed are for integer numbers:

- unsigned \(\text{ii...i}\)
- signed \(\text{si...i}\) in 2’s complement format
Orientation of binary vectors

Hint

Any vector that contains a data item coded in some positional number system should consistently be declared as \((i_{MSB} \text{ downto } i_{LSB})\) where \(2^i\) is the weight of the binary digit with index \(i\).

The MSB so has the highest index assigned to it and appears in the customary leftmost position because \(i_{MSB} \geq i_{LSB}\).

Example

```vhdl
signal Hour_D : unsigned(4 downto 0) := "10111";
```

Types `unsigned` and `signed` are for integer numbers:

- `unsigned ` \(\text{ii...i}\.\)
- `signed ` \(\text{si...i}\.\) in 2’s complement format

What about fractional parts \(\text{.ff...f}\)?
## Data types for fractional and floating point numbers

Introduced with the IEEE 1076-2008 revision.

<table>
<thead>
<tr>
<th>type prefix</th>
<th>unresolved_</th>
<th>unresolved</th>
<th>unresolved</th>
</tr>
</thead>
<tbody>
<tr>
<td>data type</td>
<td>ufixed</td>
<td>sfixed</td>
<td>float</td>
</tr>
<tr>
<td>defined in</td>
<td>fixed_</td>
<td>float_</td>
<td></td>
</tr>
<tr>
<td></td>
<td>generic_</td>
<td>generic_</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pkg</td>
<td>pkg</td>
<td></td>
</tr>
<tr>
<td>arithmetics</td>
<td>fixed point</td>
<td>floating point</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
<td>signed</td>
<td></td>
</tr>
<tr>
<td>word width</td>
<td></td>
<td></td>
<td>at the programmer’s discretion</td>
</tr>
<tr>
<td>arithmetic operations</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>logic operations</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>access to subwords or bits</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>modeling of electrical effects</td>
<td></td>
<td></td>
<td>yes (resolved)</td>
</tr>
</tbody>
</table>
Data types for fractional numbers

Both signed and unsigned formats exist; 2’C format used for signed numbers.

Unsigned example

```vhdl
signal HourWithQuarter_D : ufixed(4 downto -2) := "101111";
```

(→ range 0 to 1111.11₂ = 31.75₁₀
in steps of 1/4, initial value = 10111.11₂ = 23.75₁₀)
Data types for fractional numbers

Both signed and unsigned formats exist; 2’C format used for signed numbers.

Unsigned example

```vhdl
signal HourWithQuarter_D : ufixed(4 downto -2) := "1011111";
```

(iviiii.ff) (→ range 0 to 1111.11₂ = 31.75₁₀
in steps of \(\frac{1}{4}\), initial value = 1011.11₂ = 23.75₁₀)

Signed example

```vhdl
signal HourWithQuarter_D : sfixed(4 downto -2) := "1011111";
```

(ssiiii.ff) (→ range 10000.00₂ = −16.00₁₀ to 01111.11₂ = 15.75₁₀
in steps of \(\frac{1}{4}\), initial value = 1011.11₂ = −9.75₁₀)
Data types for fractional numbers

Both signed and unsigned formats exist; 2’C format used for signed numbers.

Unsigned example

```vhdl
signal HourWithQuarter_D : ufixed(4 downto -2) := "1011111";
```

(→ range 0 to $1111.11_2 = 31.75_{10}$
in steps of $\frac{1}{4}$, initial value = $10111.11_2 = 23.75_{10}$)

Signed example

```vhdl
signal HourWithQuarter_D : sfixed(4 downto -2) := "1011111";
```

(→ range $10000.00_2 = -16.00_{10}$ to $01111.11_2 = 15.75_{10}$
in steps of $\frac{1}{4}$, initial value = $10111.11_2 = -9.75_{10}$)

- For maximum versatility, some arithmetic aspects are kept user-adjustable via generics:
  - Rounding behavior. (round ≈ vs. truncate ↓).
  - Overflow behavior (saturate ∈ \[\] vs. wrap around \[\[\].
  - Number of guard bits for division operation
    (extra digit positions used to reduce the roundoff error)
Data types for floating point numbers

Floating point numbers include a sign bit and an exponent by definition.

- Formats adhere to the principles of the IEEE 754 standard, except
  # of bits for exponent and mantissa are defined in type declaration.
- Mantissa is coded as a fractional number in 2’C format.
- Exponent is coded in O-B (offset-binary) format.
Data types for floating point numbers

Floating point numbers include a sign bit and an exponent by definition.

- Formats adhere to the principles of the IEEE 754 standard, except
  # of bits for exponent and mantissa are defined in type declaration.
- Mantissa is coded as a fractional number in 2'C format.
- Exponent is coded in O-B (offset-binary) format.

Example

```vhdl
signal ToyFloat_D : float(5 downto -8);
```

The number format so specified is

```vhdl
seeeee . fffffff
```

where

- #e = 5 and #f = 8
- s stands for the sign bit (of the mantissa)
- each e stands for one bit of the exponent (with an offset 2^e-1−1 = 15)
- each f stands for one bit of the mantissa (normalized to the interval [1...2) and with binary weights from \( \frac{1}{2} \) all the way down to \( \frac{1}{256} \))

Online translators available on the Internet!
4th HDL capability: An event-based model of time

Figure: ... plus an event queue mechanism that governs process activation ...
How does VHDL simulation work? I

Please recall:

A signal’s value can be altered by any of …

- Concurrent signal assignment (simplest)
- Selected signal assignment
- Conditional signal assignment
- process statement (most powerful).

Make sure to understand

- All the above constructs are concurrent processes aka threads of execution (in the sense of the German “nebenläufiger Prozess”).
- “process statement”, in contrast, refers to a specific VHDL language construct (identified by the presence of the reserved word process).
How does VHDL simulation work? II

- A typical circuit model comprises many many processes.
- No more than a few processor cores are normally available for running the simulation code.
- Yet, simulation is to yield the same result as if all processes were operating simultaneously.
How does VHDL simulation work? II

- A typical circuit model comprises many many processes.
- No more than a few processor cores are normally available for running the simulation code.
- Yet, simulation is to yield the same result as if all processes were operating simultaneously.

**HDL requirement no.4**

A mechanism that schedules processes for sequential execution and that combines their effects such as to perfectly mimic concurrency.
Notions of time

Simulation time is to an HDL what physical time is to the hardware being modeled. The simulator can be thought to maintain some kind of stop watch that registers the progress of simulation time.

Execution time (aka wall clock) refers to the time a computer takes to execute statements from the program code during simulation.
Notions of time

**Simulation time** is to an HDL what physical time is to the hardware being modeled. The simulator can be thought to maintain some kind of stop watch that registers the progress of simulation time.

**Execution time** (aka wall clock) refers to the time a computer takes to execute statements from the program code during simulation.

- In VHDL simulation, the continuum of time gets subdivided by events each of which occurs at a precise moment of simulation time.
- An event is said to happen whenever the value of a signal changes.
Event-driven simulation I

Event-driven simulation works in cycles where three stages alternate:

1. **Advance simulation time to the next transaction**
   thereby making it the current one.

2. **Set all signals** that are to be updated at the present moment of time to the target value associated with the current transaction.

3. **Invoke all processes** that need to respond to the new situation.
   Every signal assignment there supposed to modify a signal’s value causes a transaction to be entered into the event queue at that point in the future when the signal is anticipated to take on its new value.

Go to 1. and start a new simulation cycle.
The event-queue mechanism

Figure: Interactions between the event queue and processes in VHDL.
Event-driven simulation II

- Simulation stops when the event queue becomes empty or when simulation time reaches some predefined final value.
- As nothing happens between transactions, an event-driven simulator essentially skips from one transaction to the next.
  ~> No computational resources are wasted while models sit idle.
Event-driven simulation II

- Simulation stops when the event queue becomes empty or when simulation time reaches some predefined final value.
- As nothing happens between transactions, an event-driven simulator essentially skips from one transaction to the next. No computational resources are wasted while models sit idle.

**Note the analogy between event queue and agenda**

- Events are observable from the past evolution of a signal’s value.
- Transactions reflect future plans that may or may not materialize.
Delay modeling for simulation I

Delays are captured in an optional after clause in a signal assignment.
Example
\[
\text{Oup}_D \leq \text{InpA}_D + \text{InpB}_D \, \text{after} \, \text{TPD};
\]

Contamination delay can be modeled using two after clauses.
Example
\[
\text{Oup}_D \leq 'X' \, \text{after} \, \text{TCD}, \text{InpA}_D + \text{InpB}_D \, \text{after} \, \text{TPD};
\]

- Ramps can not be modeled.
Delay modeling for simulation II

In the absence of an \texttt{after} clause, delay is assumed to be zero and the transaction is scheduled for the next simulation cycle ($\delta$ delay).

Example

\begin{verbatim}
Oup_D <= InpA_D + InpB_D;
\end{verbatim}

Example

\begin{verbatim}
Oup_D <= InpA_D + InpB_D after 0 ns;
\end{verbatim}

Observation

The $\delta$ delay serves to maintain a consistent order of transactions in models that include zero delays.
Delay modeling for simulation II

In the absence of an `after` clause, delay is assumed to be zero and the transaction is scheduled for the next simulation cycle \((\delta \text{ delay})\).

Example

\[
\text{Oup}_D \leftarrow \text{InpA}_D + \text{InpB}_D;
\]

Example

\[
\text{Oup}_D \leftarrow \text{InpA}_D + \text{InpB}_D \text{ after } 0 \text{ ns};
\]

Observation

The \(\delta\) delay serves to maintain a consistent order of transactions in models that include zero delays.

When simulating models with no delays (other than \(\delta\)), it becomes difficult to distinguish between cause and effect from waveform output as the respective events appear to coincide.

Hint

Fake delays help to visually tell apart cause and effect.

Example

\[
\text{Oup}_D \leftarrow \text{InpA}_D + \text{InpB}_D \text{ after FAKEDELAY};
\]
Signal versus variable I

Figure: The past, present and future of VHDL variables and signals.
Signal versus variable II

**VHDL property**

- VHDL signals convey time-varying information between processes via the event queue. They are instrumental in process invocation which is directed by the same mechanism.

- As opposed to this, variables are confined to within a process statement or a subprogram and do not interact with the event queue in any way.
Watch out, frequent misconception!

Effects of variable and signal assignments.

**Variable assignment** ($\text{:=}$) Effect felt immediately, that is, in the next statement exactly as in any programming language.
Watch out, frequent misconception!

Effects of variable and signal assignments.

**Variable assignment (:=)** Effect felt immediately, that is, in the next statement exactly as in any programming language.

**Signal assignment (<=)** Does not become effective before the delay specified in the `after` clause has expired.

In the absence of an explicit indication, there is a delay of one simulation cycle, so the effect can never be felt in the next statement.
Event-driven simulation III

A process is either active or suspended at any time. Simulation time is stopped while the code of the processes presently active is being carried out.

This implies:

- All active processes are executed concurrently with respect to simulation time.
- All sequential statements inside a process statement are executed in zero simulation time.

Note

The order of process invocation with respect to execution time is undetermined.
Insight gained

In software languages:

- Execution strictly follows the order of statements in the source code.

During VHDL simulation:

- No fixed ordering for carrying out processes
  (including concurrent signal assignments and assertion statements).

Important observation

When to invoke a process gets determined solely by events on the signals that run back and forth between processes.
Further details on process activation

VHDL property

Concurrent, selected or conditional signal assignments have no sensitivity list. Any signal on the right-hand side of the assignment activates the process.

Spring_D <= true when (ThisMonth_D=MARCH and ThisDay_D>=21) or ThisMonth_D=APRIL or ThisMonth_D=may or (ThisMonth_D=JUNE and ThisDay_D<=20) else false;

- ThisMonth_D and ThisDay_D act as wake-up signals here.

On to the tricky process statement ...
Process statement with sensitivity list

**VHDL property**

The process statement provides a special clause, termed sensitivity list, where all wake-up signals must be declared.

```vhdl
memless2: process (ThisMonth_D, ThisDay_D) <-- sensitivity list
begin
  Spring_D <= false;  -- execution begins here
  if ThisMonth_D=MARCH and ThisDay_D>=21 then Spring_D <= true; end if;
  if ThisMonth_D=APRIL then Spring_D <= true; end if;
  if ThisMonth_D=may then Spring_D <= true; end if;
  if ThisMonth_D=JUNE and ThisDay_D<=20 then Spring_D <= true; end if;
end process memless2;  -- process suspends here
```

- Upon activation by a wake-up signal, instructions are executed one after the other until the end process statement is reached.
- The process then reverts to its suspended state.
A process statement may or may not exhibit memory

What happens if signal ThisDay_D is omitted from the sensitivity list?

```vhdl
memwhat: process (ThisMonth_D) <-- this sensitivity list is incomplete
begin
    Spring_D <= false; -- execution begins here
    if ThisMonth_D=MARCH and ThisDay_D>=21 then Spring_D <= true; end if;
    if ThisMonth_D=APRIL then Spring_D <= true; end if;
    if ThisMonth_D=MAY then Spring_D <= true; end if;
    if ThisMonth_D=JUNE and ThisDay_D<=20 then Spring_D <= true; end if;
end process memwhat; -- process suspends here
```
A process statement may or may not exhibit memory

What happens if signal ThisDay_D is omitted from the sensitivity list?

```vhdl
memwhat: process (ThisMonth_D) <-- this sensitivity list is incomplete
begin
    Spring_D <= false; -- execution begins here
    if ThisMonth_D=MARCH and ThisDay_D>=21 then Spring_D <= true; end if;
    if ThisMonth_D=APRIL then Spring_D <= true; end if;
    if ThisMonth_D=MAY then Spring_D <= true; end if;
    if ThisMonth_D=JUNE and ThisDay_D<=20 then Spring_D <= true; end if;
end process memwhat; -- process suspends here
```

- Events on ThisDay_D are unable to activate the process and, hence, no longer update signal Spring_D. Its current state then depends on past values of ThisDay_D.

The above code implies sequential circuit behavior!
A process statement may include wait statements provided it features no sensitivity list.

- Process execution suspends when a wait statement is reached.

- The wait statement comes in four flavors that differ in the nature of the condition for process reactivation.
  
<table>
<thead>
<tr>
<th>Statement</th>
<th>Wake-up Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait on ...</td>
<td>An event (value change) on any of the signals listed</td>
</tr>
<tr>
<td>wait until ...</td>
<td><em>idem</em> plus the logic conditions specified here</td>
</tr>
<tr>
<td>wait for ...</td>
<td>A predetermined lapse of time as specified here</td>
</tr>
<tr>
<td>wait</td>
<td>None, sleep forever as no wake-up condition is given</td>
</tr>
</tbody>
</table>
Process statement with a \texttt{wait}

as an alternative syntax for \texttt{memless2}:

\begin{verbatim}
memless3: process <-- no sensitivity list here
begin
  Spring_D <= false; -- execution begins here
  if ThisMonth_D=MARCH and ThisDay_D>=21 then Spring_D <= true end if;
  if ThisMonth_D=APRIL then Spring_D <= true end if;
  if ThisMonth_D=MAY then Spring_D <= true end if;
  if ThisMonth_D=JUNE and ThisDay_D<=20 then Spring_D <= true end if;
  wait on ThisMonth_D, ThisDay_D; -- process suspends here until reactivated
                              -- by an event on any of these signals
end process memless3; -- execution continues with first statement
\end{verbatim}

\begin{itemize}
  \item Functionally interchangeable with process \texttt{memless2} shown before.
  \item Process execution does not terminate with the \texttt{end process} statement but resumes at the top of the process body.
  \item \texttt{wait} placed at the end because all processes get activated once until they suspend during initialization at simulation time zero.
\end{itemize}
Not all process statements are amenable to synthesis

- Accepted coding styles for synthesis:

  process statement with sensitivity list
  with 1 wait
  with $\geq 2$ waits

  universally supported
  idem
  not normally supported
Not all process statements are amenable to synthesis

▶ Accepted coding styles for synthesis:

- *process statement with sensitivity list* with 1 wait: universally supported
- *process statement with sensitivity list* with ≥ 2 waits: *idem*
- *process statement with sensitivity list* with ≥ 2 waits: not normally supported

Reason:
Each *wait* statement is allowed to carry its own condition as to when process execution is to resume. Depending on the details, this may imply synchronous or asynchronous behavior.

*More detailed reasons follow in the synthesis section.*
What makes a VHDL process statement exhibit sequential behavior?

A process statement implies memory iff one or more of the conditions below apply.

- The process includes multiple `wait on` or `wait until` statements.
- The process evaluates input signals that have no wake-up capability.
- The process includes variables that get assigned no value before being used.
- The process fails to assign a value to its output signals for every possible combination of values of its inputs.

⇒ Circuit synthesized will or will not include flip-flops and/or latches.
Beware of frequent oversights!

If a process statement is to model combinational logic

- Assign to each output for all possible combinations of input values.
  - If a signal’s value does not matter, assign a don’t care.
  - Not assigning anything implies memory!

- Enumerate all inputs in the sensitivity list.

A syntax option introduced with the IEEE 1076-2008 revision helps:

```vhdl
memless1: process (all) <-- This is shorthand for a complete sensitivity list
begin
  Spring_D <= false;       -- execution begins here
  if ThisMonth_D=MARCH and ThisDay_D>=21 then Spring_D <= true; end if;
  if ThisMonth_D=APRIL then Spring_D <= true; end if;
  if ThisMonth_D=MAY then Spring_D <= true; end if;
  if ThisMonth_D=JUNE and ThisDay_D<=20 then Spring_D <= true; end if;
end process memless1;  -- process suspends here
```
Insight gained

VHDL knows of no specific language constructs and of no reserved words that could tell

- a sequential model from a combinational one,
- a synchronous from an asynchronous circuit,
- one type of finite state machine from a different one (Mealy, Moore and Medvedev).

**VHDL property**

What makes the difference is the detailed construction of the source code!

**Hint**

Make your intentions explicit in the source code (comments and process labels) to facilitate code understanding and the interpretation of EDA tool reports (presence of latches, number of flip-flops, through paths).
How to safely code sequential circuits

Recommendation

To be safe and universally accepted for synthesis, any process statement that models memorizing behavior must be organized as follows:

```
process (Clk_C, Rst_R) <- sensitivity list, no more signals accepted!
begin
    <-- no other statement allowed here!
    -- activities triggered by asynchronous active-high reset
    if Rst_R='1' then
        PresentState_DP <= STARTSTATE;
    ....
    -- activities triggered by rising edge of clock
    elsif Clk_C'event and Clk_C='1' then <- no more term allowed here!
        <-- extra subconditions, if any, accepted here.
        PresentState_DP <= NextState_DN; -- admit next state into state register
    ....
    <-- no further elsif or else clause allowed here!
    end if;
    <-- no statement allowed here!
end process;
```
Granularity of VHDL processes

Whether the subfunction being modeled by a concurrent process is simple or complex is entirely open. A single process statement can be made to capture almost anything between

- a humble piece of wire or
- an entire image compression circuit, for instance.

Hint

For the sake of modularity and legibility, do not cram too much functionality into a concurrent process.
Signal/variable initialization vs. hardware reset facility

VHDL supports assigning an initial value in a declaration statement.
Example

```vhdl
signal Acceleration_D : integer := 0;
```

Example

```vhdl
variable Speed : real := 1.25E2;
```

- The initial value defines the objects’s state at \( t = 0 \), just before the simulator enters the first simulation cycle.
Signal/variable initialization vs. hardware reset facility

VHDL supports assigning an initial value in a declaration statement.

Example

```vhdl
signal Acceleration_D : integer := 0;
```

Example

```vhdl
variable Speed : real := 1.25E2;
```

- The initial value defines the objects's state at \( t = 0 \), just before the simulator enters the first simulation cycle.
- A hardware reset mechanism remains ready to reconduct the circuit into a predetermined start state at any time \( t \geq 0 \) using a dedicated reset signal distributed to all bistables concerned.

Observation

These are two totally different things. An initialized signal or variable will neither model a reset facility nor synthesize into one.

A code example for how to model a reset has been given just a few slides back.
Detecting clock edges and other signal events

VHDL provides a signal attribute to detect signal transitions. Example:

```vhdl
if Clk_C'event and Clk_C='1' then ... endif;
```

will synthesize into (edge-triggered) flip-flops.

Alternative syntax:

```vhdl
if rising_edge(Clk_C) then ... endif;
```

(defined in IEEE 1164 std for `std_logic` and `std_ulegic`)
Detecting clock edges and other signal events

VHDL provides a signal attribute to detect signal transitions. Example:

\[
\text{if Clk\_C'event and Clk\_C='1' then ... endif;}
\]

→ will synthesize into (edge-triggered) flip-flops.

Alternative syntax:

\[
\text{if rising\_edge(Clk\_C) then ... endif;}
\]

(defined in IEEE 1164 std for std\_logic and std\_ulogic)

Signal attribute: a named characteristic of a signal, e.g.

- ’event → typically the only one supported for synthesis
- ’transaction
- ’driving
- ’last\_value
- ’stable → most useful in simulation models
- ...
- user-defined
How to check timing conditions

Please recall:

- Latches, flip-flops, RAMs, etc. impose timing requirements that must not be violated, otherwise circuit behavior becomes unpredictable.
How to check timing conditions

Please recall:

- Latches, flip-flops, RAMs, etc. impose timing requirements that must not be violated, otherwise circuit behavior becomes unpredictable.

A simulation model is in charge of two things:

1. Check whether input waveforms indeed conform with timing requirements (if any).
2. Evaluate input data to update outputs and/or state.

VHDL supports this plan with

- signal attribute ‘stable’ and
- assertion statements.
How to check timing conditions

Please recall:

- Latches, flip-flops, RAMs, etc. impose timing requirements that must not be violated, otherwise circuit behavior becomes unpredictable.

A simulation model is in charge of two things:

1. Check whether input waveforms indeed conform with timing requirements (if any).
2. Evaluate input data to update outputs and/or state.

VHDL supports this plan with

- signal attribute 'stable and
- assertion statements.

Concurrent assertion statement A passive process capable of checking user-defined properties and of generating a message but not of updating signals.
Example: Setup and hold time checks

```vhdl
architecture behavioral of setff is
    signal State_DP : std_logic; -- state signal
begin

    assert (not (Clk_CI'event and Clk_CI='1' and not Dd_DI'stable(1.09 ns)))
        report "setup time violation" severity warning;
    assert (not (Dd_DI'event and Clk_CI='1' and not Clk_CI'stable(0.60 ns)))
        report "hold time violation" severity warning;

    memzing: process (Clk_CI, Rst_RBI)
    begin
        if Rst_RBI='0' then
            State_DP <= '0';
        elsif Clk_CI'event and Clk_CI='1' then
            State_DP <= Dd_DI;
        end if;
    end process memzing;

    Qq_DO <= State_DP after 0.92 ns;
end behavioral;
```
Inspecting the event queue to check for timing violations

Figure: Done by searching the event queue for past events.

Observation

Any inspection of the event queue for compliance with timing requirements must necessarily look backward in time.
5th HDL capability: Facilities for model parametrization

Figure: ... plus parametrization with adjustable quantities and conditional items.
Why it pays to keep HDL models parametrized

1. Imagine you have devised a synthesis model for a datapath unit
   - 16 data registers
   - 17 arithmetic and logic operations
   - 32 bit word width
Why it pays to keep HDL models parametrized

1. Imagine you have devised a synthesis model for a datapath unit
   - 16 data registers
   - 17 arithmetic and logic operations
   - 32 bit word width

2. In addition, you need a similar unit for address computations
   - 5 data registers
   - 8 arithmetic and logic operations
   - 24 bit word width
Why it pays to keep HDL models parametrized

1. Imagine you have devised a synthesis model for a datapath unit
   - 16 data registers
   - 17 arithmetic and logic operations
   - 32 bit word width

2. In addition, you need a similar unit for address computations
   - 5 data registers
   - 8 arithmetic and logic operations
   - 24 bit word width

Easy to derive model 2. by modifying the existing HDL code, but
   - maintenance effort doubled
   - what if you later needed a third and a fourth model?

HDL requirement no.5

Means for accommodating distinct architecture choices and parameter settings within a single piece of code.
Generics

```vhdl
component parityoddw -- w-input odd parity gate
generic ( 
    WIDTH : natural range 2 to 32; -- number of inputs with supported range 
    TCD : time := 0 ns, -- contamination delay with default value 
    TPD : time := 1.0 ns ); -- propagation delay with default value

port ( 
    Inp_DI : in std_logic_vector(WIDTH-1 downto 0); 
    Oup_DO : out std_logic );
end component;
```

.....

```
constant NUMBITS : natural = 12;
.....
```

```vhdl
-- component instantiation statement
u173: parityoddw
generic map ( WIDTH => NUMBITS, TCD => 0.05 ns, TPD => (NUMBITS * 0.1 ns) )
port map ( Inp_DI => DataVec_D, Oup_DO => Parbit_D );
```

**Signals** carry time-varying info between processes and design entities. **Generics** serve to disseminate time-invariant details to design entities, they do not have any direct hardware counterpart.
Conditional spawning of processes I

Consider a cellular automaton: Game of Life by John H. Conway (1970)

Set of rules

- alive = 3
- alive ≤ 1
- alive ≥ 4

 birth
 isolation
 overcrowding

2D array of identical cells
Conditional spawning of processes I

Consider a cellular automaton: Game of Life by John H. Conway (1970)

Show http://www.bitstorm.org/gameoflife/

- Set of rules:
  - alive = 3
  - alive ≤ 1
  - alive ≥ 4

- Birth:

- Isolation:

- Overcrowding:

2D array of identical cells

HDL requirement no. 5’

Means for varying the number of processes (and of components too) as a function of parameter settings made after the source code is frozen.
Conditional spawning of processes II

The `generate` statement

- allows to decide on the number of concurrent processes immediately before simulation or synthesis begins with no changes to the basic code
- produces processes under control of constants and generics
- comes in two flavors
  
  **if ... generate**
  
  to capture the conditional presence or absence of a process

  **for ... generate**
  
  to capture a number of replications of a process where the number is subject to change
Example: Game of Life

.....

-- spawn a process for each cell in the array
row : for ih in HEIGHT-1 downto 0 generate       -- repetitive generation
cell : for iw in WIDTH-1 downto 0 generate      -- repetitive generation

memzing: process(Clk_C)

subtype live_neighbors_type is integer range 0 to 8;
variable live_neighbors : live_neighbors_type;
begin
if Clk_C’event and Clk_C='1' then
live_neighbors := live_neighbors_at(ih,iw);
if State_DP(ih,iw)='0' and live_neighbors=3 then
State_DP(ih,iw) <= '1';     -- birth
elsif State_DP(ih,iw)='1' and live_neighbors<=1 then
State_DP(ih,iw) <= '0';     -- death from isolation
elsif State_DP(ih,iw)='1' and live_neighbors>=4 then
State_DP(ih,iw) <= '0';     -- death from overcrowding
end if;
end if;
end process memzing;
end generate cell;
end generate row;
.....
Conditional instantiation of components

The `generate` mechanism also works for component instantiation.

Refer to transparency `binary2gray.vhd(structural)` for code!

As usual:

1. Declare all components to be used.
2. Declare all signals that run back and forth unless they are already known from the `port` clause.
3. Instantiate components specifying all terminal-to-signal connections.

New:

4. Use `if ... generate` and `for ... generate` statements to make instantiation conditional.
Multiple models for one circuit block

VHDL accepts multiple architecture bodies for the same entity declaration. Why would you want that?
Multiple models for one circuit block

VHDL accepts multiple architecture bodies for the same entity declaration. Why would you want that?

- Because over a design cycle the same functionality needs to be modeled at distinct levels of detail.
  1. Algorithmic model (purely behavioral)
  2. RTL model (for simulation and synthesis)
  3. Post synthesis gate-level netlist (timing estimated)
  4. Post layout gate-level netlist (timing back-annotated)

- To evaluate different circuit implementations for one block (in terms of $A$, $t_{lp}$, $E$, etc.).
Entity declaration versus architecture body

**bold** identifier, i.e. name of entity or architecture

**italic** type of information conveyed

**internal view**

one or more architecture bodies

**external view**

one entity declaration

**behavioral**

architecture behavioral of binary2gray is

```vhdl
definition
function bintogray (arg : bit_vector) return bit_vector is
variable scrapa : bit_vector(arg'length-1 downto 0);
begin
scrapa := arg;
for i in 0 to arg'length-2 loop
if scrapa(i+1)='1' then
scrapa(i) := not scrapa(i);
end if;
end loop;
return scrapa;
end bintogray;
end definition
```

```vhdl
begin
Oup_DO <= bintogray(Inp_DI) after TPD;
end behavioral;
```

**structural**

as the design progresses

```vhdl
Inp_DI(WIDTH-1) Inp_DI(WIDTH-2) ..... Inp_DI(1) Inp_DI(0)
```

```vhdl
Oup_DO(WIDTH-2) ..... Oup_DO(1) Oup_DO(0)
```

```vhdl
Oup_DO(WIDTH-1)
```

**typically used for synthesis**

behavioral

**typically used for place & route**

structural

Circuit hierarchy and connectivity

Interacting concurrent processes

A discrete replacement for electrical signals

An event-driven scheme of execution

Facilities for model parametrization

Concepts borrowed from programming languages
Configuration specification and binding

With multiple architecture bodies, there must be a way to indicate which one to use for simulation and synthesis

\[ \text{for u113: binary2gray use entity binary2gray(behavioral);} \]
\[ \text{for u188: binary2gray use entity binary2gray(structural);} \]

The mechanism is more general. A component instantiated under one name can be bound to an entity with a different name, and this binding does not need to be the same for all instances of that component.

\[ \text{for all: xnor2_gate use entity GTECH_XNOR2(behavioral);} \]
\[ \text{for all: inverter_gate use entity GTECH_NOT(behavioral);} \]
Configuration specification and binding

With multiple architecture bodies, there must be a way to indicate which one to use for simulation and synthesis.  

```vhdl
for u113: binary2gray use entity binary2gray(behavioral);
for u188: binary2gray use entity binary2gray(structural);
```

The mechanism is more general. A component instantiated under one name can be bound to an entity with a different name, and this binding does not need to be the same for all instances of that component.

```vhdl
for all: xnor2_gate use entity GTECH_XNOR2(behavioral);
for all: inverter_gate use entity GTECH_NOT(behavioral);
```

**Warning**

Do not pack two functionally distinct behaviors into two architecture bodies that belong to the same entity declaration as this is extremely confusing!
Insight gained

VHDL provides a range of constructs for writing parametrized circuit models:

- generic quantities
- for...generate and if...generate statements
- multiple architecture bodies for a single entity
- configurations along with the pertaining declaration and specification statements

**VHDL property**

It is possible to establish a model without committing the code to any specific number of processes and/or instantiated components.
Insight gained

VHDL provides a range of constructs for writing parametrized circuit models:

- generic quantities
- for...generate and if...generate statements
- multiple architecture bodies for a single entity
- configurations along with the pertaining declaration and specification statements

**VHDL property**

It is possible to establish a model without committing the code to any specific number of processes and/or instantiated components.

⇝ A preparatory step must take place before simulation or synthesis can begin ⟷ elaboration and binding.
Delay calculation from layout

- outside the scope of VHDL or SystemVerilog

HDL simulation flow

- circuit models plus testbench
- full language supported

RTL synthesis flow

- circuit models exclusively
- subset of language only

SynopsysDC commands

© Hubert Kaeslin Microelectronics Design Center ETH Zürich
What you ought to know about programming

Proven concepts from safe and modular programming include

▶ Structured flow control statements (no goto)
▶ Typing and type checking
▶ Data structures (enumerated types, arrays, records)
▶ Subprograms
▶ Packages (collections of type declarations and subprograms)
▶ Information hiding (declaration module vs. implementation module)
What you ought to know about programming

Proven concepts from safe and modular programming include

- Structured flow control statements (no goto)
- Typing and type checking
- Data structures (enumerated types, arrays, records)
- Subprograms
- Packages (collections of type declarations and subprograms)
- Information hiding (declaration module vs. implementation module)

HDL requirement no.6

Make those ideas available to HDL model developers too.

No graphic illustration at this point.
Concepts borrowed from programming languages

- Structured flow control statements
  - if...then...elsif...else, case
  - loop, exit, next
- Strong typing (type, subtype, type checking at compile time)
- Enumerated types
- Composite data types (array, record)
- Subprograms (function, procedure)
- Packages (package)
- Information hiding
  - declaration module (entity declaration, package declaration)
  - implementation module (architecture body, package body)
Data types and subtypes

A data type defines a set of values and a set of operations. Users may declare their own data types or use predefined ones. Type declaration.

Example

```vhdl
type month is (JANUARY, FEBRUARY, ..., DECEMBER);
```

**VHDL property**

VHDL is strongly typed. Extensive type checking is performed. Types must be made to match prior to assignment or comparison.
Data types and subtypes

A data type defines a set of values and a set of operations. Users may declare their own data types or use predefined ones. Type declaration.

Example

```vhdl
type month is (JANUARY, FEBRUARY, ..., DECEMBER);
```

**VHDL property**

VHDL is strongly typed. Extensive type checking is performed. Types must be made to match prior to assignment or comparison.

A subtype shares the operations with its parent type, but differs in that it takes on a subset of data values only. Subtype declaration.

Example

```vhdl
subtype day is integer range 1 to 31;
```

**Hint**

It is good practice to indicate an upper and a lower bound when using integers for hardware modeling. On-line range checking (simulation) and economic sizing of circuits (synthesis) otherwise remain elusive.
Package declaration and package body

A package is a named collection of types and/or subprograms that is made visible by referring to it in a use clause. Example:

```vhdl
-- package declaration
package calendar is
    type month is (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
        AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER);
    subtype day is integer range 1 to 31;
    function nextmonth (given_month : month) return month;
    function nextday (given_day : day) return day;
end calendar;

-- package body
package body calendar is

    function nextmonth (given_month : month) return month is
        begin
            if given_month=month'right then return month'left;
            else return month'rightof(given_month);
        end if;
    end nextmonth;

    function nextday (given_day : day) return day is
        ......
    end nextday;
end calendar;
```
Predefined packages

standard package defines data types and subtypes of VHDL along with the pertaining logic and arithmetic operations and a few more features. Always gets precompiled into design library std. Users do not normally need to care much about it.

textio package defines subprograms related to the reading and writing of ASCII files (obviously not intended for synthesis). Always gets precompiled into design library std. Source code must include the line use std.textio.all; to make definitions immediately available.
Design unit, design file and design library

**VHDL property**

VHDL supports information hiding and incremental compilation.

**Design unit**

a language construct amenable to compilation on its own.
- package declaration,
- package body,
- entity declaration,
- architecture body, and
- configuration declaration.

**Design file**

a file that holds one or more design units.

**Design library**

a named repository for a collection of design units after compilation on a host computer.

Specific for a platform (host computer and software product).
Can accommodate many design files and design units.
Motivation and background
Key concepts and constructs of VHDL
Key concepts and constructs of SystemVerilog
Automatic circuit synthesis from HDL models
Conclusions

Circuit hierarchy and connectivity
Interacting concurrent processes
A discrete replacement for electrical signals
An event-driven scheme of execution
Facilities for model parametrization
Concepts borrowed from programming languages

VHDL source code and intermediate data
Key concepts and constructs of SystemVerilog

For a VHDL course, skip the next 75 or so slides.
Hardware description language requirements

**HDL requirement no.1**

Means for expressing how circuits are being composed from subcircuits and how those subcircuits connect to each other.
1st HDL capability: Circuit hierarchy and connectivity

![Hierarchical composition diagram](image)

**Figure:** Hierarchical composition ...

© Hubert Kaeslin Microelectronics Design Center ETH Zürich
Module header

Specifies the external interface of a (sub)circuit (small or large).

// module header with external interface
module lfsr4
    ( output logic Oup_DO,
      input logic Clk_CI, Rst_RBI, Ena_SI );

    ...

endmodule

- The port list declares all signals of a module that are accessible from outside (i.e. the terminals of a circuit as opposed to its inner nodes).
Module body I: a structural circuit model

Figure: Linear-feedback shift register circuit to be described.
Module body I: a structural circuit model

Refer to transparency ifsr4struc.sv for code!

Describes a circuit or netlist assembled from components and wires.

1. Declare all modules to be used.
2. Declare all variables that run back and forth unless they are already known from the port list.
3. Instantiate modules specifying all terminal-to-signal connections.
Practical advice

**Hints**

- SystemVerilog is case-sensitive, e.g. `clk.ci ≠ CLK_CI`.
- Naming a variable `input` or `output` is all too tempting, yet these are reserved words in SystemVerilog. We recommend `Inp` and `Oup` instead.
Practical advice

Hints

- SystemVerilog is case-sensitive, e.g. clk_ci \neq \text{CLK_CI}.
- Naming a variable input or output is all too tempting, yet these are reserved words in SystemVerilog. We recommend \text{Inp} and \text{Oup} instead.

No rule without exceptions. Case-insensitive are

- the letters d, h, o and b that indicate the base in decimal, hexadecimal, octal and binary numbers,
- the hex digits A through F, and
- the logic values X and Z.

Example:

16'hFE39 // casing of a 16-bit hexadecimal number for max. legibility
How to compose a circuit from components

How do you proceed when asked to fit a circuit board with components?

1. Think of a part’s exact name, e.g. GTECH_FD2
2. Fetch a copy and assign it some unique identifier it, e.g. u10
3. Solder its terminals to existing metal pads on the board

The module instantiation statement does exactly that. Example:

```vhdl
GTECH_FD2 u10
  ( .D(n11), // port map begins here
    .CP(Clk_C),
    .CD(Rst_RB),
    .Q(State_DP[1]) ); // port map ends here
```

Note

Each `.instance_terminal(circuit_node)` item stands for an electrical connection.
The essence of **structural** circuit modeling

- SystemVerilog can describe the hierarchical composition of a circuit by
  - instantiating modules and by
  - interconnecting them with the aid of wires normally modeled as variables.
- Structural HDL models hold the same information as circuit netlists do.
- Manually writing structural HDL models is not particularly attractive.
- Most structural models are in fact obtained from RTL models by automatic synthesis.
The essence of structural circuit modeling

- SystemVerilog can describe the hierarchical composition of a circuit by
  - instantiating modules and by
  - interconnecting them with the aid of wires normally modeled as variables.
- Structural HDL models hold the same information as circuit netlists do.
- Manually writing structural HDL models is not particularly attractive.
- Most structural models are in fact obtained from RTL models by automatic synthesis.

HDL requirement no.2

Means for expressing circuit behavior including the combined effects of multiple subcircuits that operate jointly and concurrently.
2nd HDL capability: Interacting concurrent processes

Figure: ... plus behavior modeled with the aid of concurrent processes ...
Constants and variables

What everyone knows from software languages:

- **Constant declaration**
  - Example
    ```
    const integer FERMAT_PRIME_4 = 65537;
    ```

- **Variable declaration**
  - Examples
    ```
    var real Brd = 2.48678E5;
    real Ddr := 1.08179E5;
    ```
  - (keyword `var` is optional)

- **Variable assignment (procedural assignment)**
  - Example
    ```
    Brd = Brd + Ddr;
    ```
Constants and variables

What everyone knows from software languages:

- **Constant declaration**
  Example
  
  ```
  const integer FERMAT_PRIME_4 = 65537;
  ```

- **Variable declaration**
  Examples
  ```
  var real Brd = 2.48678E5;
  real Ddr := 1.08179E5;
  ```

- **Variable assignment** (procedural assignment)
  Example
  ```
  Brd = Brd + Ddr;
  ```

... plus an HDL particularity:

- **Variable assignment** (continuous assignment)
  Example
  ```
  assign Brd = Brd + Ddr;
  ```
How to describe combinational logic behaviorally I

Continuous assignment

- Syntactically simplest form of a process.
- Drives one variable.

Example:

```vhdl
logic Aa_D, Bb_D, Cc_D, Oup_D;
......
assign Oup_D = Aa_D ^ (Bb_D & ~Cc_D);
```

- Typically used to model some combinational behavior (such as an arithmetic or logic operation) when there is no need for branching.

---

0Logic operators: \( ^ = \text{XOR}, \& = \text{AND}, | = \text{OR}, \sim = \text{NOT} \)
How to describe combinational logic behaviorally II

Continuous assignment with a condition operator included

Example:

```vhdl
logic Add_S, Aa_D, Bb_D, Oup_D;
.....
assign Oup_D = Add_S ? (Aa_D + Bb_D) : (Aa_D - Bb_D);
```

Syntax: `conditional_expression ? then_expression : else_expression`

Glimpse ahead: A continuous assignment gets activated by any change of any signal on the right-hand side.
How to describe combinational logic behaviorally III

Procedural blocks

- `always_comb, always_ff, always_latch` → for circuit modeling.
- `always, initial, final` → for testbench design.
How to describe combinational logic behaviorally III

Procedural blocks

- always_comb, always_ff, always_latch → for circuit modeling.
- always, initial, final → for testbench design.

- There is a special construct for combinational circuitry.

Example:

```verilog
always_comb
begin
    Spring_D = 0; // execution begins here
    if (ThisMonth_D==MARCH & ThisDay>=21) Spring_D = 1;
    if (ThisMonth_D==APRIL) Spring_D = 1;
    if (ThisMonth_D==MAY) Spring_D = 1;
    if (ThisMonth_D==JUNE & ThisDay<=20) Spring_D = 1;
end // process suspends here
```
Procedural block versus continuous assignment

When compared to a continuous assignment, a procedural block

▶ is capable of updating two or more variables at a time,
▶ captures the instructions for doing so in a sequence of statements that are going to be executed one after the other,
▶ gives the liberty to make use of variables for temporary storage,
▶ provides more detailed control over the conditions for activation.
Procedural block versus continuous assignment

When compared to a continuous assignment, a procedural block
- is capable of updating **two or more variables at a time**, 
- captures the instructions for doing so in a **sequence of statements** that are going to be executed one after the other, 
- gives the liberty to make use of variables **for temporary storage**, 
- provides more detailed control over the **conditions for activation**.

**Observation**

Procedural blocks are best summed up as being concurrent outside and sequential inside.

- They are indispensable to model memorizing circuit behavior.
How to describe a register behaviorally

**Procedural blocks (revisited)**

- `always_comb, always_ff, always_latch` $\mapsto$ for circuit modeling.
- `always, initial, final` $\mapsto$ for testbench design.

- There is a special construct for edge-triggered circuitry and a separate one for level-sensitive circuitry.

$\mapsto$ A net progress over Verilog and VHDL.
How to describe a register behaviorally II

Example of an edge-triggered register that features

1. an asynchronous reset,
2. a synchronous load, and
3. an enable.

```vhdl
always_ff @(posedge Clk_C, negedge Rst_RB) // sensitivity list
  // activities triggered by asynchronous reset
  if (~Rst_RB)
    State_DP <= '0; // shorthand for all bits zero
  // activities triggered by rising edge of clock
  else
    // when synchronous load is asserted
    if (Lod_S)
      State_DP <= '1; // shorthand for all bits one
    // otherwise assume new value iff enable is asserted
    else if (Ena_S)
      State_DP <= State_DN; // admit next state into state register
```
More than just VHDL nostalgia

Suggestion

Code is easier to read when when “signals” can be told from local variables. We append an underscore followed by a suffix of upper-case letters to those variables that convey information between concurrent processes, e.g. Carry_DB, AddrCnt_SN, Irq_AMI.

Details of our naming convention are to follow in chapter 6 “The Case for Synchronous Design”.

© Hubert Kaeslin Microelectronics Design Center ETH Zürich
Module body II: a behavioral circuit model

Describes how concurrent processes interact via signals and how they alter them.

```vhdl
// external interface of module
module lfsr4 (  
    input logic Clk_CI, Rst_RBI, Ena_SI, // reset is active low  
    output logic Oup_DO );

// behavioral model for module

// declare internal variables
logic [1:4] State_DP, State_DN; // for present and next state

// computation of next state using concatenation of bits

// updating of state
always_ff @(posedge Clk_CI, negedge Rst_RBI)
    // activities triggered by asynchronous reset
    if (~Rst_RBI)
        State_DP <= 4'b0001;
    // activities triggered by rising edge of clock
    else
        if (Ena_SI)
            State_DP <= State_DN; // admit next state into state register

// updating of output
assign Oup_DO = State_DP[4];
endmodule
```

D C B A
xor
D C B A
(1) (2) (3) (4)
State_DN
(1) (2) (3) (4)
State_DP
(4)State_DP
models combinational operations
models a register
models a wire
memorizing process
memoryless process
memorizing process
memoryless process

© Hubert Kaeslin Microelectronics Design Center ETH Zürich
The essence of **behavioral** circuit modeling

In SystemVerilog, the behavior of a digital circuit typically gets described by a collection of concurrent processes that

- execute simultaneously, that
- communicate via variables, and where
- each such process represents some subfunction.

**Hint for RTL synthesis**

- Model each register with an `always_ff` statement (or an `always_latch` if level-sensitive rather than edge-triggered).
- Prefer continuous assignments for describing the combinational logic in between.
Hardware modeling styles

Observation

SystemVerilog allows for procedural, dataflow, and structural modeling styles to be freely combined in a single model.
Procedural, dataflow, and structural models compared I

Refer to transparency fulladd.sv for code!

Compare in terms of

1. number of processes
2. number of variables that communicate between processes
3. number of variables confined to within one process
4. impact of ordering of statements
5. interaction with event queue
6. portability of source code

Note: Adders are normally synthesized from algebraic expressions, a full-adder has been chosen here for its simplicity and conciseness.
Procedural, dataflow, and structural models compared II

**Figure:** Modeling styles illustrated with a full adder as example.
Example: The ones counter

Refer to transparency onescnt.sv for code!

Observe

1. In spite of its name, this is a memoryless subfunction that finds applications in large adder circuits.
2. The output is a 3 bit number that indicates how many of the four input bits are 1 (logic high).
3. The great diversity of modeling styles to express exactly the same functionality.
Example: The ones counter

Refer to transparency onescnt.sv for code!

Observe

1. In spite of its name, this is a memoryless subfunction that finds applications in large adder circuits.

2. The output is a 3 bit number that indicates how many of the four input bits are 1 (logic high).

3. The great diversity of modeling styles to express exactly the same functionality.

Observation

Some code examples are compact and easy to understand, others are more cryptic or tend to grow exponentially.
3rd capability: A discrete replacement for electrical signals

Figure: ... plus data types for modeling electrical phenomena ...
What you ought to know about bidirectional busses I

Figure: Memory read and write transfers in a computer.
What you ought to know about bidirectional busses II

Requirements:

- Each bidirectional line is to be driven from multiple places, so one needs a multi-driver signal (as opposed to a single-driver signal).
- Driving alternates.
- Buffers must be able to electrically release the line hence the name “three-state” output (0, 1, disabled output = high-impedance state).
- Requires some kind of access control mechanism (centralized or distributed).
What you ought to know about bidirectional busses II

Requirements:

- Each bidirectional line is to be driven from multiple places, so one needs a multi-driver signal (as opposed to a single-driver signal).
- Driving alternates.
- Buffers must be able to electrically release the line hence the name “three-state” output (0, 1, disabled output = high-impedance state).
- Requires some kind of access control mechanism (centralized or distributed).

Failure modes:

- Stationary drive conflict \(\leftrightarrow\) functional failure or damage.
- Floating voltage \(\leftrightarrow\) electrically undesirable condition.

*Presentation focusses on HDL modeling, remedies to be discussed in chapter 10 "Gate- and Transistor-Level Design".*
Why do binary types not suffice to model digital circuits?

Digital circuits exhibit characteristics and phenomena such as

- transients,
- three-state outputs,
- drive conflicts, and
- power-up

that can not be modeled with 0 and 1 alone.
Why do binary types not suffice to model digital circuits?

Digital circuits exhibit characteristics and phenomena such as

- transients,
- three-state outputs,
- drive conflicts, and
- power-up

that cannot be modeled with 0 and 1 alone.

HDL requirement no.3

A multi-valued logic system capable of capturing the effects of both node voltage and source impedance.
The SystemVerilog logic system I

**Voltage** is quantized into three logic states

- **low** logic low, that is below $U_l$.
- **high** logic high, that is above $U_h$.
- **unknown** either “low”, “high” or anything in between
e.g. as a result from a short between two drivers.
The SystemVerilog logic system I

Voltage is quantized into three logic states
- low logic low, that is below $U_l$.
- high logic high, that is above $U_h$.
- unknown either “low”, “high” or anything in between
  e.g. as a result from a short between two drivers.

Source impedance gets mapped onto two drive strengths
- driven as exhibited by a driving output
- high-impedance as exhibited by a disabled three-state output

▶ Note the absence of drive strength “weak”
  as exhibited by a passive pull-up/-down resistor or a snapper.
The SystemVerilog logic system II

No charge retention in high-impedance state $\Rightarrow$
- charged low
- charged high
- charged unknown

are all merged into a single value of undetermined state (voltage).
The SystemVerilog logic system II

No charge retention in high-impedance state $\leadsto$
- charged low
- charged high
- charged unknown

are all merged into a single value of undetermined state (voltage).

Two extra conditions ought to be distinguished, namely:
- uninitialized signal has never been assigned any value since power-up (applicable to simulation only).
- don’t care don’t care condition for logic minimization, distinction between “low” or “high” immaterial (applicable to synthesis only).
The SystemVerilog logic system II

No charge retention in high-impedance state $\sim$

- charged low
- charged high
- charged unknown

are all merged into a single value of undetermined state (voltage).

Two extra conditions ought to be distinguished, namely:

- uninitialized signal has never been assigned any value since power-up (applicable to simulation only).
- don’t care don’t care condition for logic minimization, distinction between “low” or “high” immaterial (applicable to synthesis only).

Oddities:

- No attempt to distinguish between “unknown” and “uninitialized”.
- Same symbol $X$ is used as for “unknown” and “don’t care”.

• charged low
• charged high
• charged unknown
The SystemVerilog logic system III

Uses a total of four logic values to model electrical signals.

<table>
<thead>
<tr>
<th>logic value</th>
<th>logic state</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>low</td>
</tr>
<tr>
<td>uninitialized</td>
<td>known</td>
</tr>
<tr>
<td>strength driven</td>
<td>high</td>
</tr>
<tr>
<td>high-impedance</td>
<td>low</td>
</tr>
<tr>
<td>don’t care</td>
<td>known</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>low</th>
<th>unknown</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>uninitialized</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>strength driven</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>high-impedance</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>don’t care</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Defines two data types that share the above set of values:

- **logic**
  
  Difference to be explained soon

- **wire**

© Hubert Kaeslin Microelectronics Design Center ETH Zürich
Illustrations

Figure: The SystemVerilog MVL-4 illustrated.
How to model a bidirectional line in SystemVerilog I

Want to model a circuit node that can be driven from multiple subcircuits?  
⇝ Use a wire and two or more continuous assignments with a condition.

Example:

```verilog
wire Com_DZ;
logic Aa_D, Bb_D, SelA_S, SelB_S;

....
assign Com_DZ = SelA_S ? ~Aa_D : 1'bZ;
....
assign Com_DZ = SelB_S ? ~Bb_D : 1'bZ;
....
```

Note

Node Com_DZ is left floating when neither of the two drivers is enabled.
How to model a bidirectional line in SystemVerilog II

- Single-driver signals `Pp_DZ` and `Qq_DZ` may assume distinct logic values.
- No difference between `logic` and `wire`.

b) If a multi-driver signal `Com_DZ` is of type `logic`, then an error message gets issued. If of type `wire`, then the conflict is resolved to `Com_DZ = 1`.
How to model a bidirectional line in SystemVerilog II

single-driver signals Pp_DZ and Qq_DZ may assume distinct logic values,
no difference between logic and wire

if multi-driver signal Com_DZ is of type logic then an error message gets issued
wire then the conflict is resolved to Com_DZ = 1

Observation

Use wire for multi-driver nodes exclusively. When simulating wire tacitly resolves all conflicts that might occur
logic supports no multiple drivers, generates an error message
The SystemVerilog built-in resolution function

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>0</th>
<th>1</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
</tbody>
</table>

- This is the default resolution function for type `wire`, others can be added.
- There can be no resolution function for type `logic` nor for `integer` or any other type of variable.
Data types for modeling single-bit signals

<table>
<thead>
<tr>
<th>data type</th>
<th>bit</th>
<th>logic</th>
<th>wire</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>for simulation purposes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>modeling of power-up phase</td>
<td>no</td>
<td>passable</td>
<td>passable</td>
</tr>
<tr>
<td>modeling of weakly driven nodes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>modeling of multi-driver nodes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>handling of drive conflicts</td>
<td>n.a.</td>
<td>n.a.</td>
<td>resolved</td>
</tr>
<tr>
<td><strong>for synthesis purposes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>three-state drivers</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>don’t care conditions</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
## Data types for modeling multi-bit signals

<table>
<thead>
<tr>
<th>data type(s)</th>
<th>byte, shortint, int, longint</th>
<th>integer</th>
<th>bit vector</th>
<th>logic vector</th>
<th>wire vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>value set per binary digit word width</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8/16/32/64</td>
<td>32</td>
<td>at the programmer’s discretion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arithmetic operations</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>default signed/unsigned</td>
<td>signed</td>
<td>signed</td>
<td>unsigned.</td>
<td>unsigned.</td>
<td>unsigned.</td>
</tr>
<tr>
<td>logic operations</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>access to subwords or bits</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>modeling of electrical effects</td>
<td>no</td>
<td>passable*</td>
<td>no</td>
<td>passable*</td>
<td>passable</td>
</tr>
</tbody>
</table>

* Multiple drivers not allowed with this data type.
Orientation of binary vectors

Hint

Any vector that contains a data item coded in some positional number system should consistently be declared as \((i_{\text{MSB}} \downarrow \text{downto} \ i_{\text{LSB}})\) where \(2^i\) is the weight of the binary digit with index \(i\).

The MSB so has the highest index assigned to it and appears in the customary leftmost position because \(i_{\text{MSB}} \geq i_{\text{LSB}}\).

Example

\[
\text{logic [4:0] Hour_D = 5'b10111;}
\]
Orientation of binary vectors

Hint

Any vector that contains a data item coded in some positional number system should consistently be declared as \( (i_{MSB} \text{ downto } i_{LSB}) \) where \( 2^i \) is the weight of the binary digit with index \( i \).

The MSB so has the highest index assigned to it and appears in the customary leftmost position because \( i_{MSB} \geq i_{LSB} \).

Example

```
logic [4:0] Hour_D = 5'b10111;
```

Numerical data types can be declared as unsigned or signed with a modifier. Example:

```
int unsigned ii...i.
int signed si...i. in 2's complement format
```
A word of advice

Hint

Rather than silently relying on defaults to keep the code as terse as possible, make your intentions reasonably explicit in the code.

- Not only improves code quality but also accelerates debugging and code maintenance.
- Defaults are rather unsystematic in SystemVerilog.
- SystemVerilog is weakly typed = little type checking is performed.
  - Δ data words get tacitely extended or slashed in width to make things fit
  - ± almost no obligation to include type conversions
  - – impossible for tools to find suspect code fragments during compilation
4th HDL capability: An event-based model of time

Figure: ... plus an event queue mechanism that governs process activation ...
How does SystemVerilog simulation work? I

Please recall:

A variable’s (or a wire’s) value can be altered by any of ...

- continuous assignment.
- `always_comb`, `always_ff`, and `always_latch` block (for circuit modeling).
- `always`, `initial`, and `final` blocks (for testbenches).

Make sure to understand

- All the above constructs are concurrent processes aka threads of execution (in the sense of the German “nebenläufiger Prozess”).
- “Procedural block”, in contrast, refers only to the `always`, `always`, initial, and final blocks.
How does SystemVerilog simulation work? II

- A typical circuit model comprises many many processes.
- No more than a few processor cores are normally available for running the simulation code.
- Yet, simulation is to yield the same result as if all processes were operating simultaneously.
How does SystemVerilog simulation work? II

- A typical circuit model comprises many many processes.
- No more than a few processor cores are normally available for running the simulation code.
- Yet, simulation is to yield the same result as if all processes were operating simultaneously.

**HDL requirement no.4**

A mechanism that schedules processes for sequential execution and that combines their effects such as to perfectly mimic concurrency.
Notions of time

Simulation time is to an HDL what physical time is to the hardware being modeled. The simulator can be thought to maintain some kind of stop watch that registers the progress of simulation time.

Execution time (aka wall clock) refers to the time a computer takes to execute statements from the program code during simulation.
Notions of time

Simulation time is to an HDL what physical time is to the hardware being modeled. The simulator can be thought to maintain some kind of stop watch that registers the progress of simulation time.

Execution time (aka wall clock) refers to the time a computer takes to execute statements from the program code during simulation.

- In SystemVerilog simulation, the continuum of time gets subdivided by events each of which occurs at a precise moment of simulation time.
- An update event is said to happen whenever the value of a variable (or wire) changes.
Event-driven simulation I

Event-driven simulation works in cycles where three stages alternate:

1. **Advance simulation time to the next scheduled event** thereby making it the current one.

2. **Set all variables** that are to be updated at the present moment of time to the target value associated with the current event.

3. **Invoke all processes** that need to respond to the new situation.
   Every assignment there supposed to modify a variable’s value causes an event to be entered into the event queue at that point in the future when the variable is anticipated to take on its new value.

Go to 1. and start a new simulation cycle.
Basic event-queue mechanism

Figure: Interactions between the event queue and processes in SystemVerilog.
Event-driven simulation II

- Simulation stops when the event queue becomes empty or when simulation reaches a `$stop` or `$finish` instruction.

- As nothing happens between events, an event-driven simulator essentially skips from one scheduled event to the next.

  $\Rightarrow$ No computational resources are wasted while models sit idle.
Event-driven simulation II

- Simulation stops when the event queue becomes empty or when simulation reaches a `$stop` or `$finish` instruction.

- As nothing happens between events, an event-driven simulator essentially skips from one scheduled event to the next.

  $\sim\sim$ No computational resources are wasted while models sit idle.

Note the analogy between event queue and agenda

- Update events are observable from the past evolution of a variable’s value.

- Scheduled events reflect future plans that may or may not materialize.
Event-driven simulation II

- Simulation stops when the event queue becomes empty or when simulation reaches a $\text{stop}$ or $\text{finish}$ instruction.
- As nothing happens between events, an event-driven simulator essentially skips from one scheduled event to the next.
  \[\rightsquigarrow\] No computational resources are wasted while models sit idle.

Note the analogy between event queue and agenda

- Update events are observable from the past evolution of a variable’s value.
- Scheduled events reflect future plans that may or may not materialize.

This was just a first order approximation

The exact operation of the SystemVerilog “stratified event queue” is much more complicated as each cycle is organized into 17 ordered “regions”.

© Hubert Kaeslin Microelectronics Design Center ETH Zürich

Circuit Modeling with Hardware Description Languages 157 / 227
Event-driven simulation III

A process is either active or suspended at any time. Simulation time is stopped while the code of the processes presently active is being carried out.

This implies:

- All active processes are executed concurrently with respect to simulation time.
- All sequential statements inside a procedural block (*always, initial, final*) are executed in zero simulation time.

**Note**

The order of process invocation with respect to execution time is undetermined.
Insight gained

In software languages:

- Execution strictly follows the order of statements in the source code.

During SystemVerilog simulation:

- No fixed ordering for carrying out processes
  (including continuous assignments and assertion statements).

Important observation

When to invoke a process gets determined solely by events on the variables (and wires) that run back and forth between processes.
Sensitivity list, process suspension and activation I

▶ A matching event (value change) on any signal in the sensitivity list (re-)activates the process. Example:

```vhdl
// vvvvvv sensitivity list vvvvvv
always_ff @(posedge Clk_C, negedge Rst_RB)
....
```

▶ Constructs for temporarily suspending a process and for stating when it is to resume include:

- **Statement**  **Wake-up condition**
  @(...)  an update event on any of the signals listed here
  wait (...)  *idem* plus the logic conditions specified here
  #...  a predetermined lapse of time as specified here
Sensitivity list, process suspension and activation II

SystemVerilog property

Continuous assignments have no sensitivity list. Any variable on the right-hand side of the assignment activates the process.

Example:

```
assign Oup_D = InpA_D + InpB_D;
```

- InpA_D and InpB_D act as wake-up signals here.
Delay modeling for simulation

Delays are captured with an optional # term in a continuous assignment.

Example

assign #TPD Oup_D = InpA_D + InpB_D;
Delay modeling for simulation

Delays are captured with an optional # term in a continuous assignment.
Example

\[
\text{assign \ #TPD \ Oup\_D = InpA\_D + InpB\_D;}
\]

Contamination delay must be modeled using a procedural block.
Example:

```
always_comb
begin
  Oup\_D <= #TCD '{default:1'bX}; // revert all bits to unknown after tcd
  Oup\_D <= #TPD InpA\_D + InpB\_D; // propagate result to output after tpd
end
```

- Ramps can not be modeled.
SystemVerilog variables can convey time-varying information between processes via the event queue. They are instrumental in process invocation which is directed by the same mechanism.

**Figure:** The past, present and future of SystemVerilog variables.
## Blocking versus nonblocking assignments

### SystemVerilog property

How variables interact with the event queue depends on how exactly the assignment is coded.

<table>
<thead>
<tr>
<th>Assignment Operator</th>
<th>Continuous Operator</th>
<th>Procedural Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay term</td>
<td>assign ... =</td>
<td>= (blocking)</td>
</tr>
<tr>
<td>Operator</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Continuous</td>
<td>non-zero</td>
<td>non-zero</td>
</tr>
<tr>
<td>Operator</td>
<td></td>
<td>&lt;= (nonblocking)</td>
</tr>
<tr>
<td>Delay term</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>Execution of process</td>
<td>suspends until next update event on a right-hand operand</td>
<td>continues</td>
</tr>
<tr>
<td>Effect on variable</td>
<td>immediate</td>
<td>suspended for delay specified</td>
</tr>
<tr>
<td></td>
<td>deferred by delay specified</td>
<td>continues</td>
</tr>
<tr>
<td></td>
<td>immediate</td>
<td>deferred by delay specified</td>
</tr>
<tr>
<td></td>
<td>deferred by delay specified</td>
<td>following simultan. blocking assignm.</td>
</tr>
<tr>
<td></td>
<td>deferred by delay specified</td>
<td>deferred by delay specified</td>
</tr>
</tbody>
</table>
No binding order of execution for simultaneous events

SystemVerilog aberration

A simulator is free to execute processes scheduled for the same simulation time in arbitrary order ⇒ nondeterminism and race conditions loom.

∅ As opposed to VHDL, SystemVerilog knows of no δ delay.
No binding order of execution for simultaneous events

SystemVerilog aberration

A simulator is free to execute processes scheduled for the same simulation time in arbitrary order $\leadsto$ nondeterminism and race conditions loom.

∅ As opposed to VHDL, SystemVerilog knows of no $\delta$ delay.

Rules for writing safe RTL synthesis models

1. Prefer continuous assigns for uncomplicated combinational functions.
2. Do not use procedural blocks other than always_comb, always_ff and always_latch. (Use always block only where not for synthesis.)
3. In an always_comb block, always use blocking assignments ($=$).
4. In always_ff and always_latch blocks, use nonblocking assignments ($<=$) only.
5. Do not make #0 (zero delay expression) procedural assignments.
Signal/variable initialization vs. hardware reset facility

SystemVerilog supports assigning an initial value in a declaration statement.

Example

```
integer Acceleration_D = 0;
```

- The initial value defines the objects’s state at $t = 0$, just before the simulator enters the first simulation cycle.
Signal/variable initialization vs. hardware reset facility

SystemVerilog supports assigning an initial value in a declaration statement.

Example

```vhdl
integer Acceleration_D = 0;
```

- The initial value defines the objects's state at $t = 0$, just before the simulator enters the first simulation cycle.
- A hardware reset mechanism remains ready to reconduct the circuit into a predetermined start state at any time $t \geq 0$ using a dedicated reset signal distributed to all bistables concerned.

Observation

These are two totally different things. An initialized variable will neither model a reset facility nor synthesize into one.

A code example for how to model a reset has been given earlier, another one is to follow shortly.
How to check timing conditions

Please recall:

- Latches, flip-flops, RAMs, etc. impose timing requirements that must not be violated, otherwise circuit behavior becomes unpredictable.
How to check timing conditions

Please recall:

- Latches, flip-flops, RAMs, etc. impose timing requirements that must not be violated, otherwise circuit behavior becomes unpredictable.

Therefore, a simulation model is in charge of two things:

1. Check whether input waveforms indeed conform with timing requirements (if any).
2. Evaluate input data to update outputs and/or state.

SystemVerilog supports this plan with

- the `specify` block and
- twelve specialized constructs among which `$setup`, `$hold`, `$width` and `$period`. 
Example: Setup and hold time checks

```vhdl
// simulation model of a single-edge-triggered flip-flop with hardcoded timing
module setff
    ( input logic Clk_CI, logic Rst_RBI, logic Dd_DI,
      output logic Qq_DO );

logic State_DP; // state variable

specify
    $setup ( Dd_DI, posedge Clk_CI, 1.09ns ); // data evt, clock evt, min. sep.
    $hold ( posedge Clk_CI, Dd_DI, 0.60ns ); // clock evt, data evt, min. sep.
endspecify

always_ff @(posedge Clk_CI, negedge Rst_RBI)
    if (~Rst_RBI)
        State_DP <= 1'b0;
    else
        State_DP <= Dd_DI;

assign #0.92ns Qq_DO = State_DP;
endmodule
```
5th HDL capability: Facilities for model parametrization

Figure: ... plus parametrization with adjustable quantities and conditional items.
Why it pays to keep HDL models parametrized

1. Imagine you have devised a synthesis model for a datapath unit
   ▶ 16 data registers
   ▶ 17 arithmetic and logic operations
   ▶ 32 bit word width
Why it pays to keep HDL models parametrized

1. Imagine you have devised a synthesis model for a datapath unit
   - 16 data registers
   - 17 arithmetic and logic operations
   - 32 bit word width

2. In addition, you need a similar unit for address computations
   - 5 data registers
   - 8 arithmetic and logic operations
   - 24 bit word width
Why it pays to keep HDL models parametrized

1. Imagine you have devised a synthesis model for a datapath unit
   ▶ 16 data registers
   ▶ 17 arithmetic and logic operations
   ▶ 32 bit word width

2. In addition, you need a similar unit for address computations
   ▶ 5 data registers
   ▶ 8 arithmetic and logic operations
   ▶ 24 bit word width

Easy to derive model 2. by modifying the existing HDL code, but
   ▶ maintenance effort doubled
   ▶ what if you later needed a third and a fourth model?

HDL requirement no.5

Means for accommodating distinct architecture choices and parameter settings within a single piece of code.
Parameters

// w-input odd parity gate
module parityoddw
    #( parameter WIDTH, // number of inputs
        parameter TCD = 0ns, // contamination delay with default value
        parameter TPD = 1.0ns ) // propagation delay with default value
        ( input logic [WIDTH-1:0] Inp_DI,
          output logic Oup_DO );

endmodule

.....

// module instantiation statement
parityoddw #( .WIDTH(NUMBITS), .TCD(0.05ns), .TPD(NUMBITS * 0.1ns) )
        u173 ( .Inp_DI(DataVec_D) , .Oup_DO(Parbit_D) );

.....

Ports carry time-varying information between modules.
Parameters serve to disseminate time-invariant details to modules
(e.g. word widths, active-low/high signaling, timing quantities),
they do not have any direct hardware counterpart.
Conditional spawning of processes I

Consider a cellular automaton: Game of Life by John H. Conway (1970)

Show http://www.bitstorm.org/gameoflife/

2D array of identical cells

<table>
<thead>
<tr>
<th>alive = 3</th>
<th>birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>alive (\leq 1)</td>
<td>isolation</td>
</tr>
<tr>
<td>alive (\geq 4)</td>
<td>overcrowding</td>
</tr>
</tbody>
</table>
Consider a cellular automaton: Game of Life by John H. Conway (1970)

*Show* http://www.bitstorm.org/gameoflife/

2D array of identical cells

---

HDL requirement no.5’

Means for varying the number of processes (and of components too) as a function of parameter settings made after the source code is frozen.
Conditional spawning of processes II

The `generate` statement

- allows to decide on the number of concurrent processes immediately before simulation or synthesis begins with no changes to the basic code
- produces processes under control of constants and parameters
- comes in two flavors
  - `generate if`
    - to capture the conditional presence or absence of a process
  - `generate for`
    - to capture a number of replications of a process where the number is subject to change
Example: Game of Life

.....

```vhdl
// spawn a process for each cell in the array
generate
for (genvar ih = 0; ih<HEIGHT; ih++)
  for (genvar iw = 0; iw<WIDTH; iw++)
    always_ff @(posedge Clk_C) begin // sensitivity list
      integer live_neighbors;
      live_neighbors = live_neighbors_at(ih,iw);
      if (State_DP[ih][iw]=='b0 && live_neighbors==3)
        State_DP[ih][iw] <= 'b1; // birth
      else if (State_DP[ih][iw]=='b1 && live_neighbors<=1)
        State_DP[ih][iw] <= 'b0; // death from isolation
      else if (State_DP[ih][iw]=='b1 && live_neighbors>=4)
        State_DP[ih][iw] <= 'b0; // death from overcrowding
    end // always_ff
endgenerate
.....
```
Multiple models for one circuit block

SystemVerilog allows multiple modules for the same circuit
Why would you want that?
Multiple models for one circuit block

SystemVerilog allows multiple modules for the same circuit
Why would you want that?

- Because over a design cycle the same functionality needs to be modeled at distinct levels of detail.
  1. Algorithmic model (purely behavioral)
  2. RTL model (for simulation and synthesis)
  3. Post synthesis gate-level netlist (timing estimated)
  4. Post layout gate-level netlist (timing back-annotated)

- To evaluate different circuit implementations for one block (in terms of $A$, $t_{lp}$, $E$, etc.).
Conditional compilation of source code

With multiple modules for one subcircuit, there must be a way to indicate which one to use for simulation and synthesis \( \leadsto \) ‘ifdef statement.

```vhdl
// parametrized binary to Gray code converter
module binary2gray
    #( parameter ... ) // parameters
    (.....); // inputs and outputs

    'ifdef usebehavioral
        // module body with behavioral model follows here
        .....,
    'else
        // module body with structural model follows here
        .....,
    'endif
endmodule
```
Conditional compilation of source code

With multiple modules for one subcircuit, there must be a way to indicate which one to use for simulation and synthesis \(\leadsto\) ‘ifdef statement.

```
// parametrized binary to Gray code converter
module binary2gray
    #( parameter ...) // parameters
    (.....); // inputs and outputs

    ‘ifdef usebehavioral
        // module body with behavioral model follows here
        ...
    ‘else
        // module body with structural model follows here
        ...
    ‘endif
endmodule
```

**Warning**

Warning: Do not give two modules with functionally distinct behaviors identical names as this is extremely confusing!
Insight gained

SystemVerilog provides a range of constructs for writing parametrized circuit models:

▶ parameter quantities
▶ generate...for and generate...if statements
▶ ‘define, ‘undef, ‘ifdef and other compiler directives.

SystemVerilog property

It is possible to establish a model without committing the code to any specific number of processes and/or instantiated components.
Insight gained

SystemVerilog provides a range of constructs for writing parametrized circuit models:

- parameter quantities
- generate...for and generate...if statements
- ‘define, ‘undef, ‘ifdef and other compiler directives.

SystemVerilog property

It is possible to establish a model without committing the code to any specific number of processes and/or instantiated components.

⇝ A preparatory step must take place before simulation or synthesis can begin → elaboration and binding.
Delay calculation from layout
- outside the scope of VHDL or SystemVerilog

HDL simulation flow
- circuit models plus testbench
- full language supported

RTL synthesis flow
- circuit models exclusively
- subset of language only

SynopsysDC commands
- analyze elaborate compile / design optimization
- HDL simulation flow
  - full language supported
  - circuit models plus testbench
- RTL synthesis flow
  - circuit models exclusively
  - subset of language only

Circuit hierarchy and connectivity
- Interacting concurrent processes
- A discrete replacement for electrical signals
- An event-driven scheme of execution

Facilities for model parametrization
- Concepts borrowed from programming languages
What you ought to know about programming

Proven concepts from safe and modular programming include

- Structured flow control statements (no goto)
- Typing and type checking
- Data structures (enumerated types, arrays, records)
- Subprograms
- Packages (collections of type declarations and subprograms)
- Information hiding (declaration module vs. implementation module)
What you ought to know about programming

Proven concepts from safe and modular programming include

- Structured flow control statements (no goto)
- Typing and type checking
- Data structures (enumerated types, arrays, records)
- Subprograms
- Packages (collections of type declarations and subprograms)
- Information hiding (declaration module vs. implementation module)

HDL requirement no.6

Make those ideas available to HDL model developers too.

No graphic illustration at this point.
Concepts borrowed from programming languages

- Structured flow control statements
  - if ... else if ... else, case...endcase
  - for, while, repeat

△ Little type checking $\iff$ almost anything compiles!
- Enumerated types (enum)
-Composite data types (struct)
- Subprograms (function, task)
- Packages (package)

△ Limited information hiding, no separation into declaration and implementation module.
Package

A package is a named collection of types and/or subprograms that is made visible by referring to it in an import clause. Example:

```vhdl
package calendar;

    typedef enum {JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
        AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER} month;
    typedef logic unsigned [4:0] day;

    function month nextmonth (month given_month);
        return given_month.next;  // wraps around at the end
    endfunction

    function day nextday (day given_day);
        ....
    endfunction

endpackage: calendar
```
System tasks I

System tasks are commands to the simulator and not for synthesis. Examples:

```vhdl
$display("Simulation ended after %4d checks and with %4d error(s).", checkcnt, errorcnt);

int simvectorfile = $fopen("./simvectors/moore6st_simvector.asc", "r");

$fclose(simvectorfile);

while( !$feof(simvectorfile)) begin
  void'($fgets(readstr, simvectorfile));
  fmatch = $sscanf(readstr, "%b %b %b", StimuliRec.Clr_S, StimuliRec.Inp_D, ExpRespRec.Oup_D);
  ...
end

$readmemh("./sim/vectors/stim.txt", stimuli);

$error("Expected 'b%b does not match actual 'b%b", expresp, ActResp_D);

assert (simvectorfile) else $fatal("Could not open simvector file.");
```
### System tasks II, incomplete overview

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Formatted text output</strong></td>
<td></td>
</tr>
<tr>
<td>$\texttt{write}$</td>
<td>write line to standard output immediately with no newline character</td>
</tr>
<tr>
<td>$\texttt{display}$</td>
<td>write line to standard output immediately preceded by a newline character</td>
</tr>
<tr>
<td>$\texttt{strobe}$</td>
<td>idem at the end of current time slot, i.e. before advancing simulation time</td>
</tr>
<tr>
<td>$\texttt{monitor}$</td>
<td>idem when specified events occur</td>
</tr>
<tr>
<td><strong>File operations</strong></td>
<td></td>
</tr>
<tr>
<td>$\texttt{fopen}$</td>
<td>open a file</td>
</tr>
<tr>
<td>$\texttt{fclose}$</td>
<td>close a file</td>
</tr>
<tr>
<td>$\texttt{fread}$</td>
<td>read from a file</td>
</tr>
<tr>
<td>$\texttt{fgets}$</td>
<td>read characters from a file and assembles them into a string</td>
</tr>
<tr>
<td>$\texttt{sscanf}$</td>
<td>parse formatted text from a string</td>
</tr>
<tr>
<td>$\texttt{feof}$</td>
<td>return a non-zero value when end of file found and 0 if not so</td>
</tr>
<tr>
<td>$\texttt{fwrite}$</td>
<td>same as $\texttt{write}$ for writing to a file</td>
</tr>
<tr>
<td><strong>Memory load and dump</strong></td>
<td></td>
</tr>
<tr>
<td>$\texttt{readmemb/h}$</td>
<td>load memory from a text file in binary/hex format</td>
</tr>
<tr>
<td>$\texttt{writememb/h}$</td>
<td>dump memory to a text file in binary/hex format</td>
</tr>
<tr>
<td><strong>Simulation control</strong></td>
<td></td>
</tr>
<tr>
<td>$\texttt{stop}$</td>
<td>suspend simulation</td>
</tr>
<tr>
<td>$\texttt{finish}$</td>
<td>terminate simulation</td>
</tr>
<tr>
<td><strong>Run time information with severity levels (standalone and for use in assertions)</strong></td>
<td></td>
</tr>
<tr>
<td>$\texttt{info}$</td>
<td>print argument to simulator window and continue</td>
</tr>
<tr>
<td>$\texttt{warning}$</td>
<td>print argument to simulator window, count as warning, and continue</td>
</tr>
<tr>
<td>$\texttt{error}$</td>
<td>print argument to simulator window, count as error, and continue</td>
</tr>
<tr>
<td>$\texttt{fatal}$</td>
<td>print argument to simulator window and terminate simulation</td>
</tr>
<tr>
<td><strong>Random number generation (for use in stimuli preparation)</strong></td>
<td></td>
</tr>
<tr>
<td>$\texttt{random}$</td>
<td>return a random signed integer</td>
</tr>
<tr>
<td>$\texttt{urandom}$</td>
<td>return a random unsigned integer</td>
</tr>
<tr>
<td><strong>Enquiries about the event queue (for use in properties and assertions)</strong></td>
<td></td>
</tr>
<tr>
<td>$\texttt{rose}$</td>
<td>return 1 iff argument has changed to 1</td>
</tr>
<tr>
<td>$\texttt{fell}$</td>
<td>return 1 iff argument has changed to 0</td>
</tr>
<tr>
<td>$\texttt{stable}$</td>
<td>return 1 iff argument had not changed value</td>
</tr>
<tr>
<td>$\texttt{past}$</td>
<td>return argument’s value a specified number of clock cycles earlier</td>
</tr>
</tbody>
</table>
Subject

Circuit synthesis from HDL models
HDL synthesis overview

Figure: Major steps of automated RTL synthesis.

- Syntax analysis is different for VHDL and for SystemVerilog models. After that, the synthesis process becomes essentially the same.
Synthesis subset

The predominant HDLs have not originally been intended for synthesis.

- While almost all VHDL simulators support the full IEEE 1076 standard, only a subset of the legal language constructs is amenable to synthesis.
- The same holds for SystemVerilog and the IEEE 1800 standard.

⇒ Good HDL code is written such as to be portable across platforms and synthesis tools.

Guiding principle

Limit yourself to safe, unambiguous, and universally accepted constructs!
Synthesis subset

The predominant HDLs have not originally been intended for synthesis.

- While almost all VHDL simulators support the full IEEE 1076 standard, only a subset of the legal language constructs is amenable to synthesis.
- The same holds for SystemVerilog and the IEEE 1800 standard.

Good HDL code is written such as to be portable across platforms and synthesis tools.

Guiding principle

Limit yourself to safe, unambiguous, and universally accepted constructs!

- Apart from that, the impact of coding style on combinational random logic is fairly small (surprisingly perhaps as it is often overstated).
Not all data types are amenable to synthesis

**VHDL**

- integer
- boolean and bit
- std_logic and std_ulogic
- unsigned and signed
- enumerated type
- ufixed, sfixed and float
- array and record of fixed size

**SystemVerilog**

- integer, shortint, int, longint
- bit
- logic and wire
- byte
- enumerated type
- array and struct of fixed size
Not all data types are amenable to synthesis

**VHDL**

**Supported:**
- integer
- boolean and bit
- std_logic and std_ulogic
- unsigned and signed
- enumerated type
- ufixed, sfixed and float
- array and record of fixed size

**Not supported:**
- real
- time
- character
- file

**SystemVerilog**

**Supported:**
- integer, shortint, int, longint
- bit
- logic and wire
- byte
- enumerated type
- array and struct of fixed size

**Not supported:**
- real
- time-related data types
- string, queue, and other dynamic data types
- file-related data types
Hardware-compatible wake-up conditions

- While HDLs allow the modeling of arbitrary behavior (as long as it is causal, discrete in value, and discrete in time), automatic synthesis only supports synchronous clock-driven subcircuits and — at a higher level — conglomerates of such subcircuits.

Observation

Any process that is supposed to model a piece of hardware must execute upon activation, return to the same instruction, and suspend there.
Hardware-compatible wake-up conditions

- While HDLs allow the modeling of arbitrary behavior (as long as it is causal, discrete in value, and discrete in time), automatic synthesis only supports synchronous clock-driven subcircuits and — at a higher level — conglomerates of such subcircuits.

Observation

Any process that is supposed to model a piece of hardware must execute upon activation, return to the same instruction, and suspend there.

Why?

- Each *wait* or similar statement is allowed to carry its own condition as to when process execution is to resume.
  - Depending on the details, this may imply asynchronous behavior.
  - Extremely difficult, if not impossible, to implement in a physical circuit.
Formalisms for describing finite state machines

- **Output function**
  \[ o(k) = g(i(k), s(k)) \]
- **Next state function**
  \[ s(k+1) = f(i(k), s(k)) \]

**Figure:** Data dependency graph (a), state chart (b), Nassi-Shneiderman diagram (c).
### Explicit versus implicit state models

<table>
<thead>
<tr>
<th>Modeling Style</th>
<th>Explicit State</th>
<th>Implicit State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inspired from</strong></td>
<td>Data dependency graph or schematic diagram</td>
<td>Nassi-Shneiderman diagram</td>
</tr>
<tr>
<td><strong>Synchronization mechanism</strong></td>
<td>Sensitivity list or single wait statement (semantically equivalent)</td>
<td>Multiple wait statements</td>
</tr>
<tr>
<td><strong>State variable</strong></td>
<td>Declared explicitly as signal or variable and thus of user-defined type</td>
<td>Hidden in pointer to current statement</td>
</tr>
<tr>
<td><strong>States captured by</strong></td>
<td>(Subrange of) integer or vector of bits</td>
<td>Multiple wait statements</td>
</tr>
<tr>
<td><strong>State transitions captured by</strong></td>
<td>Arithmetic and/or logic operations</td>
<td>Control flow</td>
</tr>
<tr>
<td><strong>Output function captured by</strong></td>
<td>Arithmetic and/or logic operations</td>
<td>Assignment statements</td>
</tr>
<tr>
<td><strong>Immediate hardware equivalent</strong></td>
<td>Yes</td>
<td>Depending on wait conditions</td>
</tr>
<tr>
<td><strong>Synchronous</strong></td>
<td>Yes</td>
<td>Idem</td>
</tr>
<tr>
<td><strong>Synthesizable</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Explicit State Concepts:**
- Computed state
- Enumerated state

**Implicit State Concepts:**
- Hidden in pointer to current statement

**Key Concepts and Constructs:**
- Data types
- Finite state machines and sequential subcircuits in general
- RAM and ROM macrocells
- Timing constraints
- Limitations and caveats
- How to establish a register transfer level model step by step
What you ought to know from automata theory

**Mealy machine** output is a function of both state and input

\[ o(k) = g(i(k), s(k)) \]
\[ s(k + 1) = f(i(k), s(k)) \]

\( \Rightarrow \) latency 0, through path, hazards likely.

**Moore machine** output is a function of state exclusively

\[ o(k) = g(s(k)) \]
\[ s(k + 1) = f(i(k), s(k)) \]

\( \Rightarrow \) latency 1, no through path, hazards likely.

**Medvedev machine** subclass of Moore with identity as output function

\[ o(k) = s(k) \]
\[ s(k + 1) = f(i(k), s(k)) \]

\( \Rightarrow \) latency 1, no through path, hazard-free.
How to capture a finite state machine in HDL

Mealy machine

- 

Moore machine

- coded in one process

Medvedev machine

- coded in one process

- some synthesizers unnecessarily produce duplicate flip-flops

- widely supported by synthesizers d) e) f)

- coded in two (or more) processes

- widely supported by synthesizers
How to write portable synthesis code (VHDL)

For the sake of code portability and trouble-free synthesis ...

Good VHDL synthesis code shall

- model circuits at the RTL (register transfer level) throughout,
- collect combinational and sequential logic in separate processes,
- have all memorizing process statements conform with our skeleton,
- prefer concurrent, conditional and selected signal assignments for combinational logic,
- have all memoryless process statements coded according to the rules.
How to write portable synthesis code (VHDL)

For the sake of code portability and trouble-free synthesis ...

Good VHDL synthesis code shall

- model circuits at the RTL (register transfer level) throughout,
- collect combinational and sequential logic in separate processes,
- have all memorizing process statements conform with our skeleton,
- prefer concurrent, conditional and selected signal assignments for combinational logic,
- have all memoryless process statements coded according to the rules.

Warning

The emergence of unplanned for latches or other bistables during synthesis always points to bad code.
How to write portable synthesis code (SystemVerilog)

For the sake of code portability and trouble-free synthesis ...

Good SystemVerilog synthesis code shall

- model circuits at the RTL (register transfer level) throughout,
- collect combinational and sequential logic in separate processes,
- use `always_ff` blocks exclusively for memorizing behavior (`always_latch` blocks for level-sensitive clocking),
- use continuous assigns or `always_comb` blocks for combinational logic.
How to write portable synthesis code *(SystemVerilog)*

For the sake of code portability and trouble-free synthesis ...

**Good SystemVerilog synthesis code shall**

- model circuits at the RTL *(register transfer level)* throughout,
- collect combinational and sequential logic in separate processes,
- use `always_ff` blocks exclusively for memorizing behavior *(`always_latch` blocks for level-sensitive clocking)*,
- use continuous `assigns` or `always_comb` blocks for combinational logic.

**Warning**

The emergence of unplanned for latches or other bistables during synthesis always points to bad code.
Example: a Mealy-type state machine

Refer to transparency mealy5st.vhd or .sv for code!

Observe

1. The enumerated type for states.
2. The default state and output assignments.
3. The `when state if input then action` construct that assigns values to output and next state.
4. The tying up of parasitic states.
Example: a Moore-type state machine

Refer to transparency moore6st.vhd or .sv for code!

Observe

1. The combination of symbolic state identifiers and one-hot state codes imposed by the user.
2. The default state and output assignments (as before).
3. The placement of all output assignments outside the if input then clauses throughout. This makes the code describe a Moore machine!
4. Output assignments can be stated before or after each of the if input then clauses with no difference.
5. The tying up of parasitic states (as before).
Example: a Medvedev-type state machine

Refer to transparencies graycnt.vhd and grayconv.vhd or .sv for code!

Observe

1. The simplicity and elegance of the combinational process.
2. The usage of two functions bintogray and graytobin for code conversion described in a user-defined package grayconv.vhd.
3. The clause use work.grayconv.all that makes that package available.
More on good FSM coding practices

**Recommendation**

- Try to decompose large FSMs into a bunch of smaller ones that cooperate.
- Adhere to hierarchical and modular design.
- Consider using counters instead of long state chains.

**Recommendation**

The various processes that make up for an FSMs are best included in the same architecture body as the datapath they command.

Shutting the FSM into an entity of its own just inflates the code and the effort for coding and maintenance.
Synthesis of ROMs

Innocent approach: Declare a storage array as if the code were intended for simulation and assume the synthesizer will take care of all the rest.

Example: 4bit-binary to seven-segment display decoder.

-- unsupported VHDL coding style
......
-- address of array must be of type integer or natural

```
p_memless : process (Binary4Code_D)
  variable address : natural range 0 to 15;
  type array16by7 is array(0 to 15) of std_logic_vector(1 to 7)
  constant SEGMENT_LOOKUP_TABLE : array16by7 :=
    ("1111110","0110000","1101101","1111001",
     "0110011","1011011","0011111","1110000",
     "1111111","1110011","1110111","0011111",
     "1001110","0111101","1001111","1000111");
begin
  -- use binary input as index, look up in table, and assign to segment output
  address := to_integer(unsigned(Binary4Code_D));
  Segment7Code_D <= SEGMENT_LOOKUP_TABLE(address);
end process p_memless;
......
```
Synthesis of RAMs I

Innocent approach: Declare a storage array as if the code were intended for simulation and assume the synthesizer will take care of all the rest.

Example: 64 byte read-write memory. – unsupported VHDL coding style

```vhd
    type array64by8 is array(0 to 63) of std_logic_vector(7 downto 0);
    signal Storage_D : array64by8;
```

Will not synthesize into a RAM because the behavior so defined is a far cry from actual RAM macrocells and their interfaces. Automated synthesis would hardly churn out a safe and synchronous gate-level circuit either.
Synthesis of RAMs II

Second attempt: Instantiate RAM, state macrocell generator to be used, pass on all further specifications in a generic map.

-- unsupported VHDL coding style

......
u39: cmosram01
generic map ( NUMBER_OF_WORDS => 64, WORD_WIDTH => 8, DATA_INPUT_OUTPUT_SEPARATE => false )
port map ( CLK => Clk_C, WRENA => RamWrite_S, ADDR => RamAddress_D, DATIO => RamData_D );

......

▶ Realistic, but not currently feasible due to the lack of standardization and the absence of interfaces between VHDL synthesis and proprietary macrocell generators.
Synthesis of RAMs III (VHDL)

Final attempt: instantiate RAM macrocell like any other component.

-- supported VHDL coding style
.....
u39: myram64by8
   port map ( CLK => Clk_C, WRENA => RamWrite_S,
               ADDR => RamAddress_D, DATIO => RamData_D );
.....

Observation

The necessary design views of a macrocell (simulation model, schematic icon, detailed layout, etc.) must be obtained from outside the VHDL environment.

The chip designer must either
- gain access to the process-specific macrocell generator software or
- commission the silicon foundry do so for him.
Synthesis of RAMs III (SystemVerilog)

Final attempt: instantiate RAM macrocell like any other component.

// supported SystemVerilog coding style
......
myram64by8 u39 ( .CLK(Clk_C), .WRENA(RamWrite_S),
                  .ADDR(RamAddress_D), .DATIO(RamData_D) );
......

Observation

The necessary design views of a macrocell (simulation model, schematic icon, detailed layout, etc.) must be obtained from outside the SystemVerilog environment.

The chip designer must either

- gain access to the process-specific macrocell generator software or
- commission the silicon foundry do so for him.
## Synthesis of RAMs and ROMs

<table>
<thead>
<tr>
<th>Look up table (LUT) (memoryless)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>desired hardware organization</td>
<td>random logic</td>
</tr>
<tr>
<td>function must be modeled</td>
<td>as an array-type constant</td>
</tr>
<tr>
<td></td>
<td>or with logic equations</td>
</tr>
<tr>
<td></td>
<td>ROM (tiled layout)</td>
</tr>
<tr>
<td></td>
<td>by instantiating a ROM</td>
</tr>
<tr>
<td></td>
<td>macrocell as a component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data storage array (memorizing)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>desired hardware organization</td>
<td>register file built from</td>
</tr>
<tr>
<td></td>
<td>flip-flops or latches</td>
</tr>
<tr>
<td>function must be modeled</td>
<td>as an array of (clocked)</td>
</tr>
<tr>
<td></td>
<td>storage registers</td>
</tr>
<tr>
<td></td>
<td>RAM (tiled layout)</td>
</tr>
<tr>
<td></td>
<td>by instantiating a RAM</td>
</tr>
<tr>
<td></td>
<td>macrocell as a component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Common characteristics of implemented circuit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>efficient when data quantity is small</td>
<td>techn.-specific softw. required</td>
</tr>
<tr>
<td>no pre-synthesis simulation from</td>
<td>code amenable to retargeting</td>
</tr>
<tr>
<td>yes post-synthesis simulation from</td>
<td>pre-synthesis simulation from</td>
</tr>
<tr>
<td>small RTL source code gate-level model</td>
<td>large macrocell generator</td>
</tr>
<tr>
<td>no extra behavioral model idem</td>
<td>manual rework needed</td>
</tr>
</tbody>
</table>

© Hubert Kaeslin Microelectronics Design Center ETH Zürich
Insight gained

A storage array can be implemented
  - as on-chip macrocell,
  - as off-chip part,
  - as a RAM,
  - assembled from flip-flops or
  - from latches.

Deciding among those options has far-reaching consequences for circuit performance, system architecture, and design effort.

Conclusion

Spontaneous incorporation of macrocells is neither a practical nor really a desirable proposition for RTL synthesis because it would deprive designers of control over a circuit’s architecture.
What is an “optimal” circuit?

- Meets all user-defined performance targets at the lowest hardware costs.

Figure: Trade-offs between size and performance for a hypothetical circuit.
Timing quantities related to synthesis

**Timing constraint**  User-defined bound (upper or lower) for a timing quantity (propagation delay, contamination delay, clock period, etc.) that the final circuit must meet.

**Slack**  Difference between target specified and actual circuit delay, e.g.

\[ t_{sl} = t_{lp \ max} - t_{lp} \text{ (combinational ckt)} \text{ or} \]
\[ t_{sl} = T_{clk} - t_{ss} \text{ (sequential ckt)} \]
Timing quantities related to synthesis

**Timing constraint**  User-defined bound (upper or lower) for a timing quantity (propagation delay, contamination delay, clock period, etc.) that the final circuit must meet.

**Slack**  Difference between target specified and actual circuit delay, e.g.
\[ t_{sl} = t_{lp\ max} - t_{lp} \text{ (combinational ckt)} \]  or
\[ t_{sl} = T_{clk} - t_{ss} \text{ (sequential ckt)}. \]

**Negative slack** indicates synthesis has failed to meet timing target ➞

1. Try varying synthesizer directives.
2. Rework RTL synthesis code.
3. Rework circuit architecture.
Timing constraints are not part of the HDL standards …

- All timing-related constructs get ignored during synthesis, e.g.
  - ... after tpd    (VHDL)
  - wait for 3.9 ns  idem
  - ... #3.9ns ...  (SystemVerilog)

These serve to model the behavior of existing circuits, not to impose target requirements for the synthesis process.
Timing constraints are not part of the HDL standards ...

- All timing-related constructs get ignored during synthesis, e.g.
  - ... after tpd (VHDL)
  - wait for 3.9 ns *idem*
  - ... #3.9ns ... (SystemVerilog)

These serve to model the behavior of existing circuits, not to impose target requirements for the synthesis process.

- Timing constraints that could guide synthesis and optimization have never been adopted in the IEEE 1076 and 1800 standards.

Unsupported constructs:

**VHDL:**

\[
\text{Oup}_D \leftarrow \text{A}_a_D + \text{B}_b_D \text{ with}\_\text{delay}\_\text{no}\_\text{more}\_\text{than} \ 1.7 \text{ ns};
\]

**SystemVerilog:**

\[
\text{assign} \ \#\text{max}\#1.7\text{ns} \ \text{Oup}_D = \text{A}_a_D + \text{B}_b_D;
\]
... but must be expressed with scripting languages instead

- The HDL code is complemented with user-defined timing constraints along with other synthesis control statements.
- Proprietary language extensions or scripting languages such as Tcl must be used.

Tcl syntax examples for timing constraints:

```tcl
create_clock -period 15 [get_ports Clk_CI]
set_input_delay -max 6 -clock Clk_CI [get_ports Inp_DI]
```
... but must be expressed with scripting languages instead

- The HDL code is complemented with user-defined timing constraints along with other synthesis control statements.
- Proprietary language extensions or scripting languages such as Tcl must be used.

Tcl syntax examples for timing constraints:

```tcl
create_clock -period 15 [get_ports Clk_CI]
set_input_delay -max 6 -clock Clk_CI [get_ports Inp_DI]
```

- Tcl requires time spans to be expressed as multiples of a predefined time unit, 1 ns in this example.
- The same timing constraints are to be reused later during design verification.

For your first encounter with RTL synthesis, you may prefer to skip further details and come back later.
How to formulate timing constraints I

- An upper bound for the delay from one register to the next gets imposed by $T_{clk} \rightarrow$ Indicating a clock period is mandatory and straightforward.

![Diagram showing timing constraints](image)

- a) `create_clock -period T clk [get_ports Clk_CI]

**Figure:** Most synthesis tools accept timing constraints in terms of $T_{clk}$, $t_{idel}$ and $t_{odel}$.
How to formulate timing constraints II

- Constraining input- and output paths is more tricky because it is possible to look at input/output timing from two different perspectives:
  - “Egocentric” view indicates how much time is available to the circuit under construction.
  - “Altruistic” view quantifies the amount of time that must be set aside for the surrounding circuitry.
How to formulate timing constraints II

- Constraining input- and output paths is more tricky because it is possible to look at input/output timing from two different perspectives:
  - “Egocentric” view indicates how much time is available to the circuit under construction.
  - “Altruistic” view quantifies the amount of time that must be set aside for the surrounding circuitry.

**Note**

Most EDA tools adopt the “altruistic” view mainly because target clock can be altered without having to numerically readjust all I/O timing constraints.

**Caution**

Do not get confused by inexpressive or ill-defined names!
How to formulate input timing constraints

- data-valid window of upstream circuitry during which circuit under construction must catch data at its input
- time span left for input circuit under construction to settle (propagation through input logic plus register set-up)
- data-valid window begins
- data-valid window ends

```
set_input_delay $t_{idel}$ -clock Clk_CI [get_ports Inp_DI]
set_input_delay -max $t_{idel\ max}$ -clock Clk_CI [get_ports Inp_DI]
b) set_input_delay -min $t_{idel\ min}$ -clock Clk_CI [get_ports Inp_DI]
```

**Figure:** Input constraints $t_{idel\ max}$ and $t_{idel\ min}$ as understood by synthesis tools.
How to formulate output timing constraints

- Data types
  - Finite state machines and sequential subcircuits in general
  - RAM and ROM macrocells
  - Timing constraints
- Limitations and caveats
  - How to establish a register transfer level model step by step

**Motivation and background**

**Key concepts and constructs of VHDL**

**Key concepts and constructs of SystemVerilog**

**Automatic circuit synthesis from HDL models**

**Conclusions**

---

**Data types**

**Finite state machines and sequential subcircuits in general**

**RAM and ROM macrocells**

**Timing constraints**

**Limitations and caveats**

**How to establish a register transfer level model step by step**

---

**How to formulate output timing constraints**

- **data-call window of downstream circuitry** during which circuit under construction must provide valid output data
- **time span left for output circuit under construction to settle** (propagation delay through register and output logic)

1. **data-call window ends**
2. **data-call window begins**
3. active clock edge
4. **set_output_delay**

- **todel** -clock Clk_CI [get_ports Oup_DO]
- **todel max** -clock Clk_CI [get_ports Oup_DO]
- **todel min** -clock Clk_CI [get_ports Oup_DO]

**Figure:** Output constraints \( t_{d_{del\,max}} \) and \( t_{d_{del\,min}} \) as understood by synthesis tools.
## Cross reference for I/O timing constraints

<table>
<thead>
<tr>
<th>Event</th>
<th>Symbol</th>
<th>Quantity (altruistic view)</th>
<th>Synopsys term</th>
</tr>
</thead>
<tbody>
<tr>
<td>relating to the interface with the upstream circuitry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3) data-valid window begins</td>
<td>$t_{su ; inp}$</td>
<td>setup time of circuit under construction</td>
<td>maximal input delay</td>
</tr>
<tr>
<td></td>
<td>$\leq T_{clk} - t_{pd ; upst}$</td>
<td>clock-to-output prop. delay of upstream circuitry</td>
<td>maximum input delay</td>
</tr>
<tr>
<td></td>
<td>$t_{pd ; upst} = t_{idel ; max}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) data-valid window ends</td>
<td>$t_{ho ; inp}$</td>
<td>hold time of circuit under construction</td>
<td>minimal input delay</td>
</tr>
<tr>
<td></td>
<td>$\leq t_{cd ; upst}$</td>
<td>clock-to-output cont. delay of upstream circuitry</td>
<td>minimum input delay</td>
</tr>
<tr>
<td></td>
<td>$t_{cd ; upst} = t_{idel ; min}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>relating to the interface with the downstream circuitry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4) data-call window begins</td>
<td>$t_{pd ; oup}$</td>
<td>clock-to-output prop. delay of circuit under construction</td>
<td>output delay</td>
</tr>
<tr>
<td></td>
<td>$\leq T_{clk} - t_{su ; dnst}$</td>
<td>setup time of downstream circuitry</td>
<td>maximum output delay</td>
</tr>
<tr>
<td></td>
<td>$t_{su ; dnst} = t_{odel ; max}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1) data-call window ends</td>
<td>$t_{cd ; oup}$</td>
<td>clock-to-output cont. delay of circuit under construction</td>
<td>output delay</td>
</tr>
<tr>
<td></td>
<td>$\geq t_{ho ; dnst}$</td>
<td>hold time of downstream circuitry</td>
<td>minus minimum output delay</td>
</tr>
<tr>
<td></td>
<td>$t_{ho ; dnst} = - t_{odel ; min}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dealing with special subcircuits I

- Adders, multipliers, and other (high-performance) arithmetic units.
- Padframe (core ↔ pads interconnect).
- Clock distribution network (typically buffered trees).
- Clock gating circuitry (for low power).
- Data synchronizers (at clock boundaries).
- Scan paths (auxiliary structures for circuit testing).

What do all these subcircuits have in common?

Each must conform to a precisely defined pattern at the gate level, mimicking the desired behavior alone does not suffice! Limitation: Boolean optimization tools are not designed to handle such "non-logic" circuits.
Dealing with special subcircuits I

- Adders, multipliers, and other (high-performance) arithmetic units.
- Padframe (core ↔ pads interconnect).
- Clock distribution network (typically buffered trees).
- Clock gating circuitry (for low power).
- Data synchronizers (at clock boundaries).
- Scan paths (auxiliary structures for circuit testing).

What do all these subcircuits have in common?

Each must conform to a precisely defined pattern at the gate level, mimicking the desired behavior alone does not suffice!

Limitation: Boolean optimization tools are not designed to handle such “non-logic” circuits.
Dealing with special subcircuits II

► What are the options when tight control is sought?
(over a subcircuit’s gate-level construction)
► Use dedicated design automation tools,
► fall back to schematic entry, or
► write a parametrized structural VHDL model.

► As for arithmetic circuits, take advantage of proven synthesis models
(e.g. Synopsys DesignWare).

Hint
Do not reoptimize subcircuits so obtained as critical properties may deteriorate. Use “Don't touch” synthesis directives to prevent logic optimization from altering structurally critical subcircuits.

---

1Example: Clock tree generation is postponed to the physical design phase and handled by specialized EDA software there.
Why write RTL synthesis models?

- VHDL is perfectly suitable for coding a data processing algorithm.
- Yet, do not expect an EDA tool to accept a purely behavioral model and to turn that into a circuit design of acceptable performance, size, and energy efficiency.

Conclusion

The fun and the burden of architecture design rests with the hardware developer.
How to write an RTL model step by step 1

**VHDL construct (SystemVerilog construct)**

1. Draw a fairly detailed block diagram of the architecture devised.
2. Check where you can take advantage of DesignWare models.
3. Organize the design such as to confine critical propagation paths to within design entities (modules).
4. Identify macrocells such as RAMs and ROMs and prepare for generating the necessary design views outside the HDL environment.
5. Identify all registers and loosely collect the combinational operations in between into clouds.
6. For each cloud, specify the operations in mathematical terms.
7. Specify what is to happen during each clock cycle (schedule).
How to write an RTL model step by step II

8. For each FSM, find out what type is most appropriate.

9. Capture each register in a memoryizing process statement
   (always_ff block).

10. For each cloud, decide how many processes to use.
    Prefer concurrent signal assignments (continuous assignments)
    for simpler operations.

11. Declare the data items that run back and forth between the various
    processes as signals (variables) and decide on appropriate types.

12. Only now translate your draft into actual HDL code.
    • Adhere to the code patterns established in this text.
    • Watch out for special signals such as clock, reset, enable etc.
    • The schedule defines the various subfunctions in full detail.
    • Fill in don’t care entries wherever possible.
    • Follow the recommendations in this text.
Translating an RTL diagram into HDL code (VHDL)
Translating an RTL diagram into HDL code (SystemVerilog)
Hardware model writing versus programming

- Writing code for HDL synthesis is not the same as writing software for a program-controlled computer!
- Always think in terms of circuit hierarchies and simultaneous activities (concurrent processes) rather than in terms of instruction sequences!
Hardware model writing versus programming

- Writing code for HDL synthesis is not the same as writing software for a program-controlled computer!
- Always think in terms of circuit hierarchies and simultaneous activities (concurrent processes) rather than in terms of instruction sequences!

Golden rule

- Establish a block diagram of your architecture first, then code what you see!
Hardware model writing versus programming

- Writing code for HDL synthesis is not the same as writing software for a program-controlled computer!
- Always think in terms of circuit hierarchies and simultaneous activities (concurrent processes) rather than in terms of instruction sequences!

Golden rule

- Establish a block diagram of your architecture first, then code what you see!

Caution

Do not ignore warnings and error messages from the synthesizer unless you understand what they mean!
Conclusions I

VHDL and SystemVerilog universally adopted due to paying benefits:

+ Top-down design methodology using a single standard language.
+ RTL synthesis does away with all lower-level schematic drawings in a typical VLSI design hierarchy. \(\Rightarrow\) Saves time and effort.
+ HDLs enable sharing, reusing and porting of subfunctions and -circuits in a parametrized form. \(\Rightarrow\) More useful than schematics.
+ Automatic technology mapping postpones the committment to a specific fabrication process until late in the design cycle.
+ Allows for retargetting between field-programmable logic and ASICs.
Conclusions II

+ VHDL and SystemVerilog support the coding of simulation testbenches.

More on this in chapter 5 “Functional Verification”.

© Hubert Kaeslin Microelectronics Design Center ETH Zürich
Circuit Modeling with Hardware Description Languages 226 / 227
Conclusions III

The limitations are relatively minor:

- Learning to master VHDL or SystemVerilog may be daunting.
- Only a subset is amenable to synthesis. \(\rightarrow\) No serious problem.
- Lack of agreement between tool vendors on
  - what constructs the synthesis subset ought to include,
  - how to capture timing constraints and synthesis directives, and
  - when to support new constructs introduced with past std revisions.

- A gap remains between system design and actual hardware design. Manual translation from a behavioral model to RTL synthesis code is inefficient and prone to errors. Will high-level synthesis help?
Conclusions III

The limitations are relatively minor:

- Learning to master VHDL or SystemVerilog may be daunting.

~ Only a subset is amenable to synthesis. → No serious problem.

- Lack of agreement between tool vendors on
  ▶ what constructs the synthesis subset ought to include,
  ▶ how to capture timing constraints and synthesis directives, and
  ▶ when to support new constructs introduced with past std revisions.

- A gap remains between system design and actual hardware design. Manual translation from a behavioral model to RTL synthesis code is inefficient and prone to errors. Will high-level synthesis help?

Concluding remark

HDL synthesis does not do away with architecture design!