TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of VLSI Design, Diagnostics & Architecture

VHDL

A Language for Modeling and Designing Hardware

Dr. Thomas B. Preußer

thomas.preusser@tu-dresden.de

Dresden, July 15, 2013

Goals of this Lecture

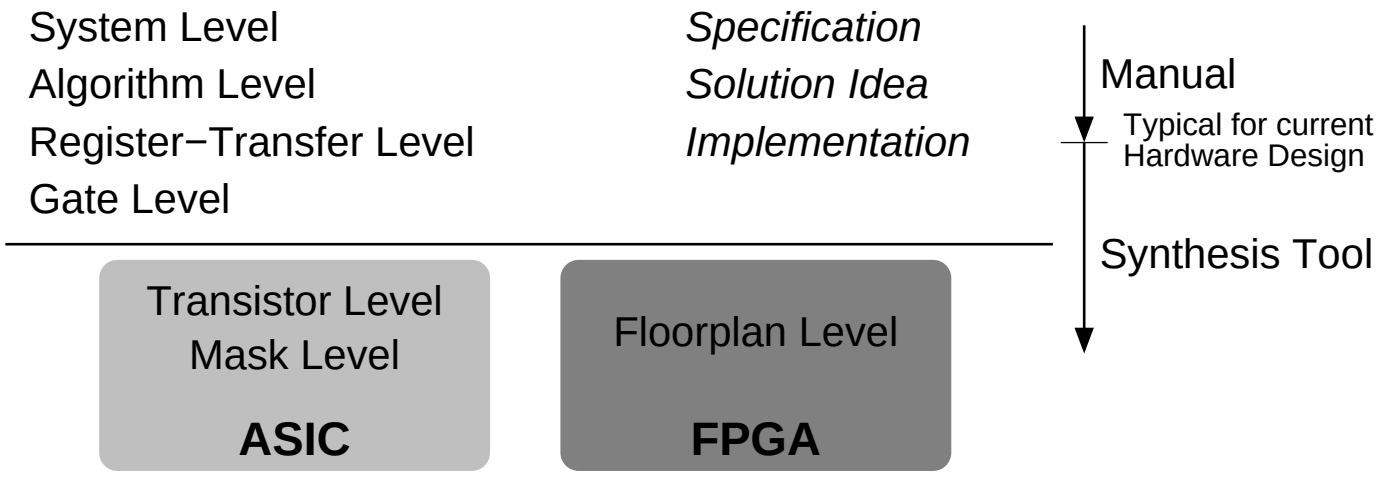
- Answer the questions:
  - Where does VHDL come from?
  - Where is VHDL used?
  - What are FPGAs?
- Give a first impression of the language.
- Provide a reference to consult during the lab.
- Show usage examples.

History

VHDL – Very High Speed Integrated Circuit (VHSIC)
Hardware Description Language

1981	Initiated by US Department of Defense (DoD) for simplifying the reproduction of hardware in new technologies: "Hardware Life Cycle Crisis".
1983	Intermetrics, IBM and TI to design an ADA-based HDL.
1985	Completion of core VHDL in version 7.2.
1986	DoD transfers all rights on VHDL to the IEEE.
1987	VHDL becomes IEEE-Standard 1076-1987.
1987	DoD requires VHDL models for all purchased ASICs.
1988	VHDL becomes ANSI Standard.
1993	IEEE-Standard 1076-1993: still most widely used.
2008	IEEE-Standard 1076-2008: latest major language revision.

Abstractions Levels



- VHDL is suitable for descriptions from the system to the gate level.
- An automated (technology-specific) typically starts at the Register-Transfer Level.
- The synthesis from higher abstraction levels is an active research domain.

Fundamental Paradigms

Structure

- Designs are built from *components*.

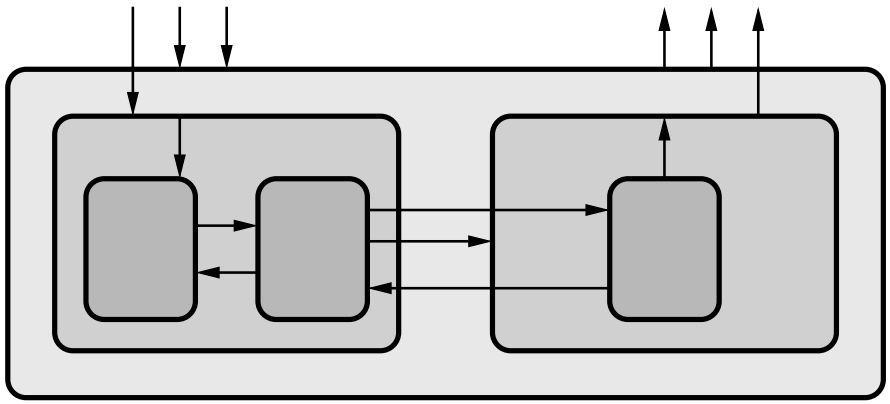
Concurrency

- Implementations are composed of *concurrent* statements.

Fundamental Paradigms

Structure

- Designs are built from *components*.



  - typically: wide, bidirectional interfaces (of *wires*).
  - Hierarchical instantiation tree of components.
  - Wiring within or through instantiating module.

Concurrency

- Implementations are composed of *concurrent* statements.

Fundamental Paradigms

Structure

- Designs are built from *components*.

Concurrency

- Implementations are composed of *concurrent* statements.
  - Order of statements does *not* matter:

```
p <= a xor b;
s <= p xor c;
```

and

```
s <= p xor c;
p <= a xor b;
```

are equivalent!

  - Dependencies are defined by signal connections.

High-Performance Custom Computing

Option	Development Cost & Time	Mask Costs	Power Bill	Design Adaptability
SW on Supercomputers	low	none	huge	cheap
Full-Custom ASIC	huge	huge	low	extremely expensive
Semi-Custom ASIC	large	high	low	very expensive
PLD / FPGA	high	none	low	reasonable

High-Performance Custom Computing

Option	Development Cost & Time	Mask Costs	Power Bill	Design Adaptability
SW on Supercomputers	low	none	huge	cheap
Full-Custom ASIC	huge	huge	low	extremely expensive
Semi-Custom ASIC	large	high	low	very expensive
PLD / FPGA	high	none	low	reasonable

SW

- is ideal for flexible number-crunching applications, but
- requires an extensive infrastructure, and
- produces huge power bills.

High-Performance Custom Computing

Option	Development Cost & Time	Mask Costs	Power Bill	Design Adaptability
SW on Supercomputers	low	none	huge	cheap
Full-Custom ASIC	huge	huge	low	extremely expensive
Semi-Custom ASIC	large	high	low	very expensive
PLD / FPGA	high	none	low	reasonable

SW

- is ideal for flexible number-crunching applications, but
- requires an extensive infrastructure, and
- produces huge power bills.

ASICs

- are only affordable for *very* high-volume systems,
- do not match FPGA performance if budget dictates old technology, and
- are extremely hard and expensive to bugfix or adapt.

High-Performance Custom Computing

Option	Development Cost & Time	Mask Costs	Power Bill	Design Adaptability
SW on Supercomputers	low	none	huge	cheap
Full-Custom ASIC	huge	huge	low	extremely expensive
Semi-Custom ASIC	large	high	low	very expensive
PLD / FPGA	high	none	low	reasonable

SW

- is ideal for flexible number-crunching applications, but
- requires an extensive infrastructure, and
- produces huge power bills.

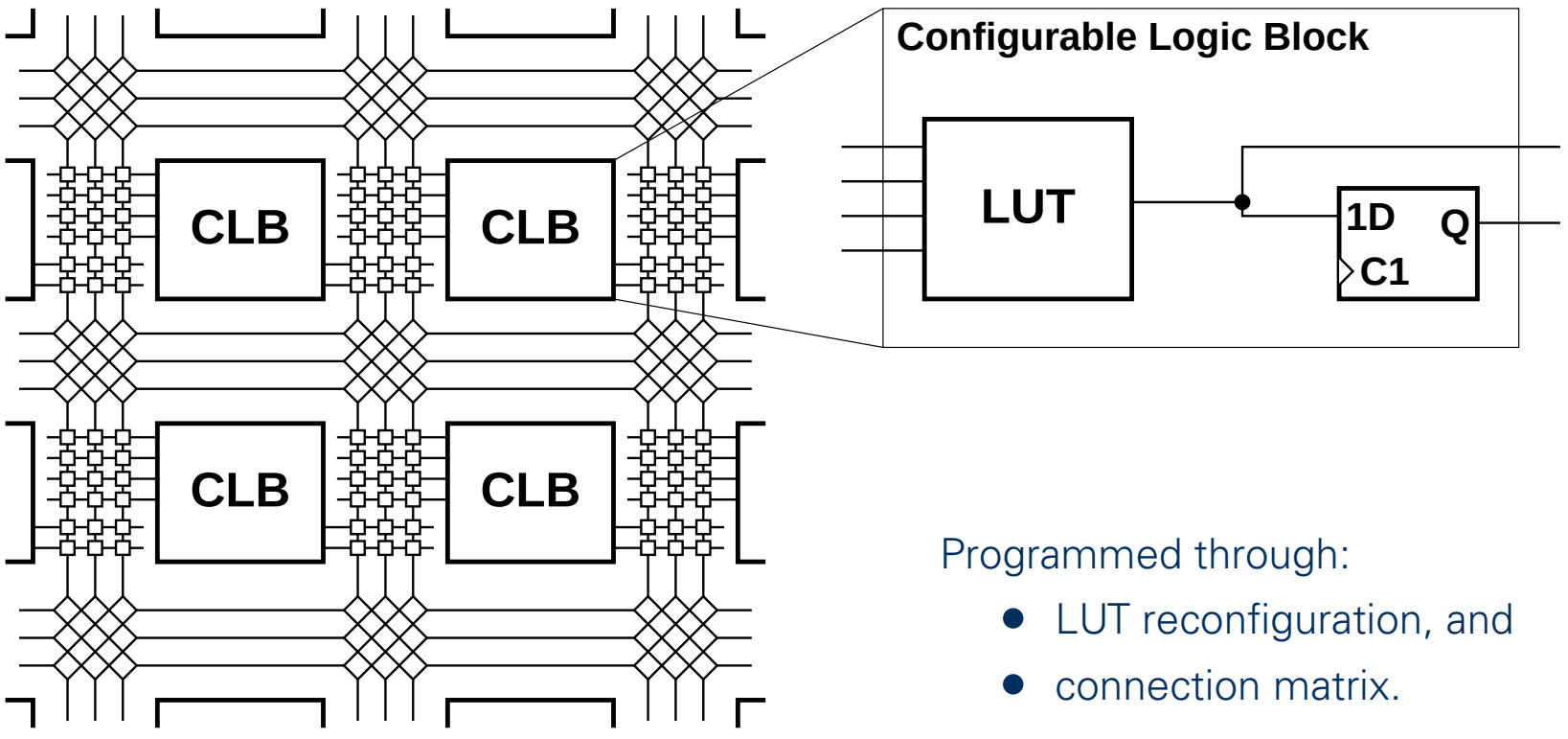
ASICs

- are only affordable for *very* high-volume systems,
- do not match FPGA performance if budget dictates old technology, and
- are extremely hard and expensive to bugfix or adapt.

FPGAs

- offer tremendous performance gains for low power budgets,
- are suited for a mostly stable core computation, and
- exploit *bit-level* optimizations.

FPGA – Field-Programmable Gate Array

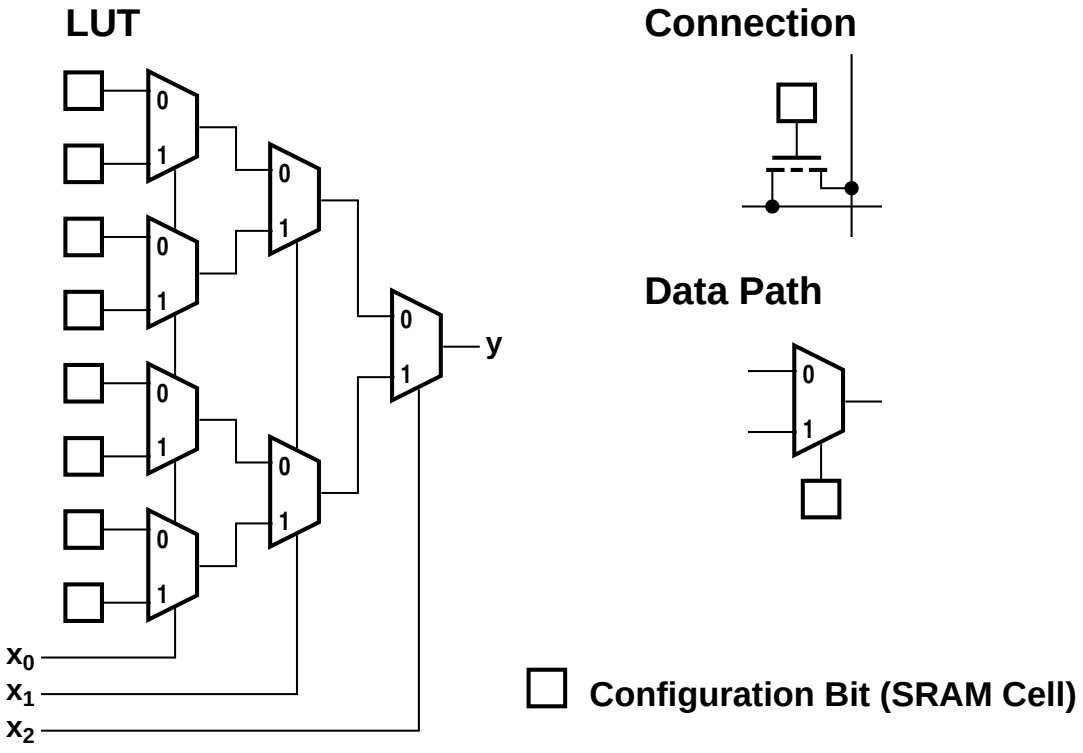


Programmed through:

- LUT reconfiguration, and
- connection matrix.

FPGA – Field-Programmable Gate Array

Configurability



FPGA – Field-Programmable Gate Array

FPGA Design vs. Software

Configurable Hardware:

- \- greater design effort (VHDL, Verilog),
- \- 10× lower clock frequency than standard CPUs,
- \+ fine-grained custom concurrency,
- \+ high computational power per Watt,
- \+ no instruction overhead.

[Xilinx Inc.]

FPGAs offer:

- affordable custom hardware performance (beyond a mass market),
- revisable designs (prototyping, communication protocols), and
- tremendous low-level concurrency.

VHDL – Syntax Overview

Designed after ADA → similarities with Pascal

- Rather wordy syntax.
- Strict separation between interface and implementation:

```
-- Interface
entity FA is
  port(
    a, b, c : in  bit;
    s, cout : out bit;
  );
end FA;
```

```
-- (An) Implementation
architecture FA_1 of FA is
begin
  s    <= a xor b xor c;
  cout <= (a and b) or (a and c)
                    or (b and c);
end FA_1;
```

- Both are stored in *one* `.vhdl`-file.
- Comments start with -- and extend to the end of the line.
- VHDL is case-*insensitive*.

Primitive Data Types

Discrete Enumerations

```
type boolean   is (false, true);
type bit       is ('0', '1');
type character is (NUL, ..., '0', '1', ..., 'A', 'B', ...);
```

Numerical Ranges

```
type integer is range −2147483648 to 2147483647; -- or larger
type real    is range       −1E38 to       +1E38; -- or larger
```

Subtypes with Constrained Ranges

```
subtype natural  is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Compound Data Types: Arrays

Unconstrained Array Types

```
type bit_vector is array(natural  range <>) of bit;
type string     is array(positive range <>) of bit;
```

Constrained Array Types (custom example)

```
-- Immediately constrained Array: byte is not a bit_vector
type    byte is array(7 downto 0) of bit;

-- Constrained Subtype: byte is a bit_vector
subtype byte is bit_vector(7 downto 0);
```

Multidimensional Arrays (custom example)

```
type matrix is
array(natural range <>, natural range <>) of integer;
```

- are different from arrays of arrays!

Compound Data Types: Structures

Records (custom example)

```
type packet is record
                 valid : bit;
                 data  : byte;
               end record;
```

Operators

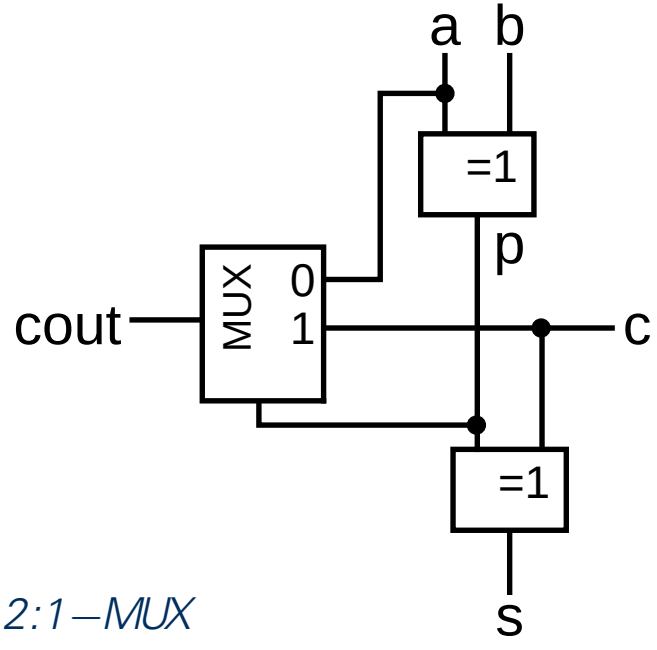
In increasing priority:

Logical Binary	**and or xor nand nor xnor**	– use parentheses
Comparison	= /= < <= > >=	– note the *not equals*
Shift / Rotate	**sll srl sla sra rol ror**	– avoid in favor of . . .
Addition / Concatenation	+ − &	. . . concatenation
Sign	+ −	
Multiplication	∗ / **mod rem**	– beware of . . .
Other Unary	**not abs** ∗∗	. . . hardware complexity

Binary operators with equal priority are left-associative.

Signals

. . . represent typed implementation-specific intermediate nodes:




```
architecture FA_2 of FA is
signal p : bit;  -- "Propagate"
begin
  p    <= a xor b;
  s    <= p xor c;
  cout <= c when p = '1' else a;  -- 2:1-MUX
end FA_2;
```

Concurrent Assignments

. . . describe combinatorial logic:

→ They are fully *parallel* to one another.

→ The specification order does *not* matter:

```
p <= a xor b;        s <= p xor c;
s <= p xor c;   and  p <= a xor b;
```

are equivalent!

Concurrent Assignments: Variants

- Unconditional Assignment
  ```
  s <= p xor c;
  ```
- Conditional Assignment
  ```
  y <= '0' when clr = '1' else
       '1' when set = '1' else
        x;
  ```
  - Prioritized selection of *first* hit.
  - Final alternative necessary for combinatorics (to avoid state).
- Selection Assignment

  ```
  with s select
    y <= a when "00",
         b when "01" | "10",
         c when others;
  ```

  - Balanced Multiplexer.
  - Exhaustive case listing may be completed by **others**.

Processes

... are complex concurrent statements.

- are themselves specified by *sequential* code:
  → The order of the statements *within* a process is relevant!
- have a control flow, which is controlled by:

```
-- Branches
if <cond1> then
  ...
elsif <cond2> then
  ...
end if;
```

```
-- Loops
for i in 0 to N loop
  a(i) <= '0';
end loop;
```

```
-- Case Selections
case s is
  when "00" =>
    ...
  when "01" | "10" =>
    ...
  when others =>
    ...
end case;
```

Combinatorial Processes

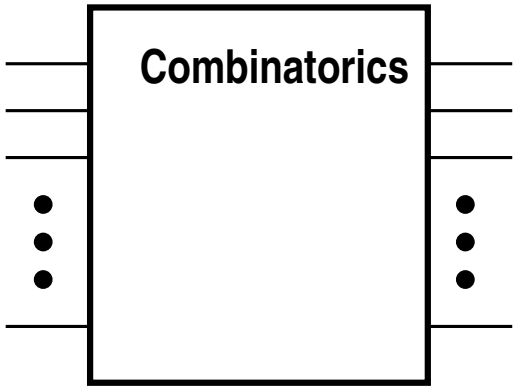
. . . describe the behavior of (complex) combinatorics.

. . . avoid a detailed structural design.

. . . leave the technological optimization to the synthesis tool.

. . . are preferable over manually derived equations as they are:

- more comprehensible, and
- easier to adapt.



Combinatorial Processes: Example

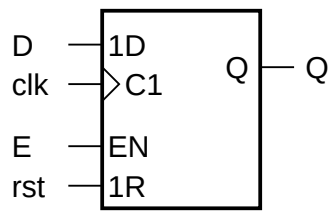
```
architecture FA_3 of FA is
begin  -- / "Sensitivity list"
  process(a, b, c)     -- !!!: ALL inputs must be listed here!
--process(all)         -- Simplified alternative in VHDL 2008
    variable p : bit; -- Local intermediate nodes as variables
  begin
    p := a xor b;      -- !!!: Assignment to variable is DIFFERENT
    s <= c;
    if p = '0' then    -- !!!: Compute ALL outputs on
      cout <= a;       --      ALL control flow paths
    else
      cout <= c;
      s    <= not c;   -- Signal assignments may be overwritten
    end if;
  end process;
end FA_3;
```

Sequential Processes

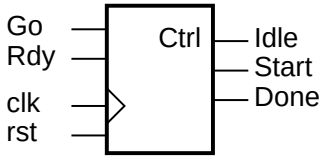
. . . describe the behavior of state registers.

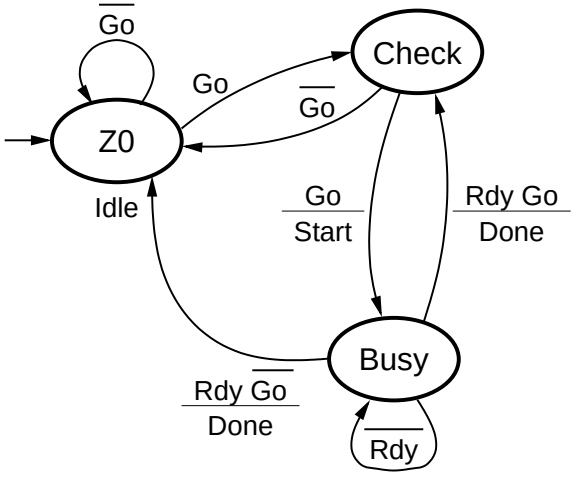
- Example: D-FF with an enable and a synchronous reset

```
process(clk) -- !!!: ONLY inputs that may
begin        --      induce a state transition
  if clk'event and clk = '1' then -- positive clock edge
    if rst = '1' then             -- synchronous reset
      Q <= '0';
    elsif E = '1' then            -- enable
      Q <= D;
    end if;
  end if;
end process;
```

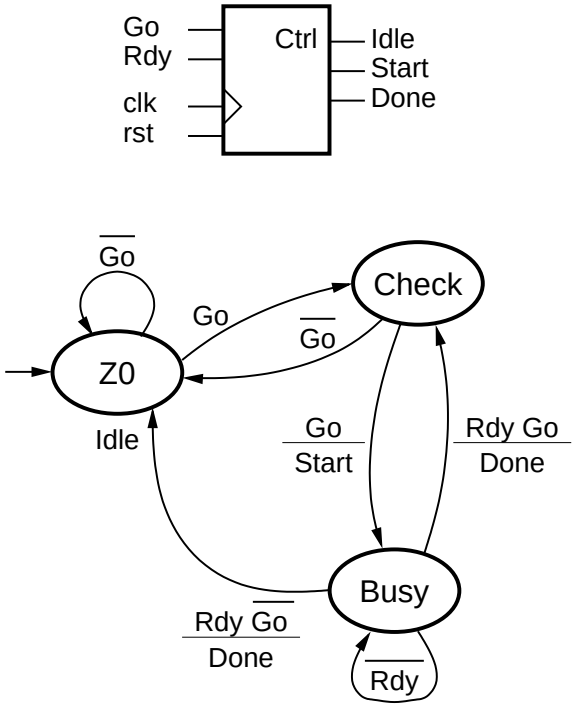


Example: Implementation of a Finite State Machine



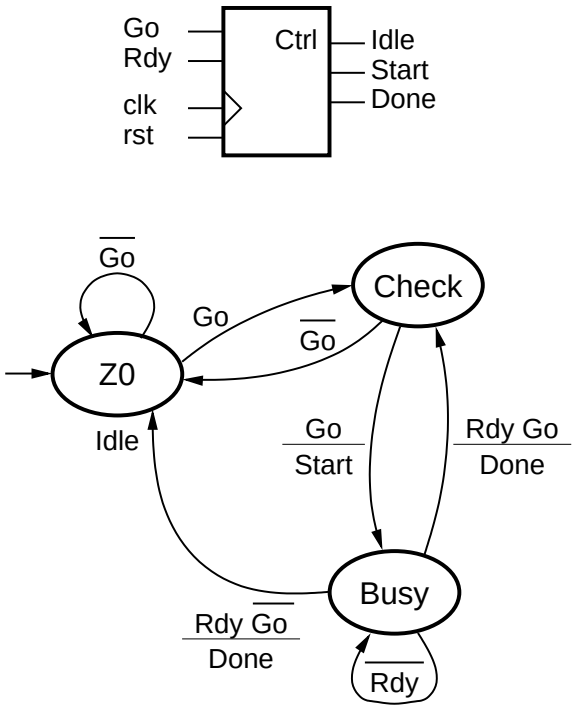


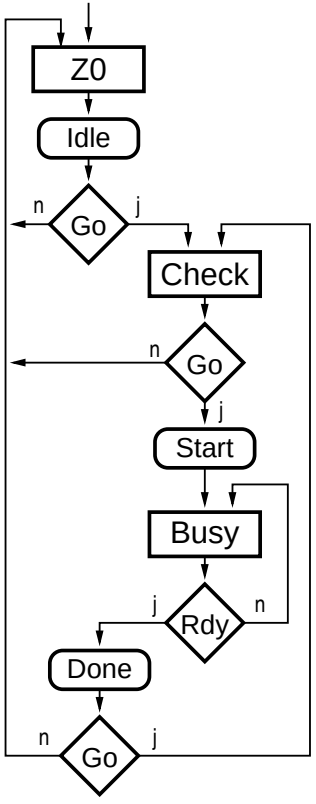
Interface Definition




```
entity ctrl is
 port(
   rst, clk : in  bit; -- Reset, Clock
   Go       : in  bit; -- Start cycle
   Rdy      : in  bit; -- Stop cycle

   Idle     : out bit; -- Idle
   Start    : out bit; -- Start of cycle
   Done     : out bit  -- End of cycle
 );
end ctrl;
```

SM-Chart





- equivalent to a state diagram,
- precise modeling of hierarchical decisions, and
- easy to translate into an HDL.

Types and Signals

```
architecture rtl of ctrl is
  type tState is (Z0, Check, Busy);
  signal State     : tState := Z0; -- state register
  signal NextState : tState;       -- combinatorial computation
begin                              --   of successor state
  ...
end rtl;
```

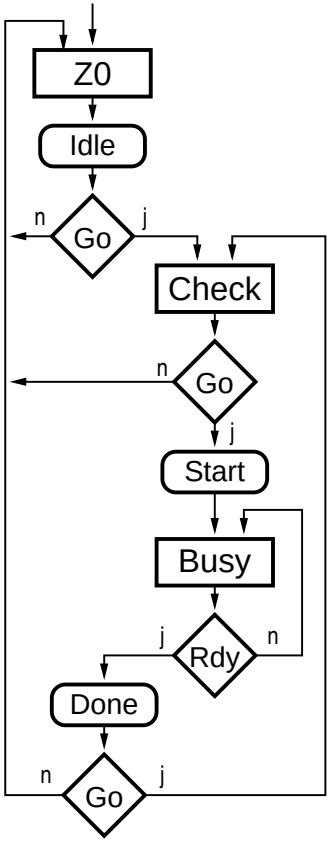
Don't put your foot in it!

Initialization of register signals should be equivalent to their reset assignment! Otherwise inconsistencies between the system states after an initialization (e.g. FPGA-programming) and after a reset are possible. → *Happy Debugging!*
If there is no initialization, synthesis tools try to deduce the initialization state from the reset description – simulators do not!

Registers

```
architecture rtl of strg is
  type tState is (Z0, Check, Busy);
  signal State     : tState := Z0; -- state register
  signal NextState : tState;       -- combinatorial computation
begin                              --   of successor state
  -- Sequential Process
  process(clk)
  begin
    if clk'event and clk = '1' then -- rising edge
      if rst = '1' then             -- synchronous reset
        State <= Z0;
      else
        State <= NextState;
      end if;
    end if;
  end process;
  ...
end rtl;
```

Combinatorics



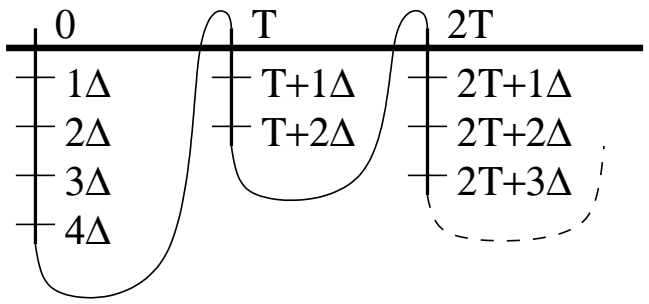

```
-- Combinatorial Process
process(State, Go, Rdy)
begin
  -- Default Assignment
  NextState <= State;
  Idle      <= '0';
  Start     <= '0';
  Done      <= '0';

  case State is
    when Z0 =>
      Idle <= '1';
      if Go = '1' then
        NextState <= Check;
      end if;

    when Check =>
      if Go = '0' then
        NextState <= Z0;
      else
        Start     <= '1';
        NextState <= Busy;
      end if;

    when Busy =>
      if Rdy = '1' then
        Done <= '1';
        if Go = '0' then
          NextState <= Z0;
        else
          NextState <= Check;
        end if;
      end if;

    -- not required here
    -- completes state enumeration
    when others =>
      null;
-- does nothing
  end case;
end process;
```

Delayed Assignment

- . . . requests the change of a signal after a given time:
  `a <= b and c after 20 ns;`
- . . . is good for functional modeling.
- . . . may be used in testbenches for the description of stimuli:
  `clk <= not clk after 50 ns;` — *10–MHz Clock*
- . . . is declined or ignored by synthesis tools.

Semantic Model: Simulation Time

Simulation of parallel hardware in sequential software
$\rightarrow$ two-dimensional simulation time with Δ-cycles:



- Event-driven simulation on signals.
- A simulation time is assigned to every event.
- Every signal has only one *fixed* value throughout a Δ-cycle.

Semantic Model: Execution

1. Initialize every signal to its default value.
2. Set simulation time $t = 0$.
3. Collect all concurrent statements (including processes!) into the set of executable statements.
4. Remove and execute every statement from the set of executable statements. Generate an event for every computed signal value:
   - at $t + \Delta$ if there is no explicit delay information, or
   - at $\lfloor t \rfloor + d$, for **after** d - assignments.
5. Advance the simulation time until the time of the next event or terminate the simulation if there is no further event.
6. Update signal values according to the events scheduled for the advanced simulation time. For every value change, collect all processes with this signal in their sensitivity lists and all concurrent statements with this signal on their right-hand side into the set of executable statements.
7. Goto step 4.

Semantic Model: Consequences

- A missing entry in the sensitivity list of a combinatorial process:
  - prevents the re-computation of the process after a change of this input, which
  - produces:
    - either a state in the form of a latch, or
    - a warning by a well-meaning synthesis tool.
- Further, usually unintended, sources of state within latches are:
  - combinatorial signals without an explicit computation in every control flow path of a process, and
  - conditional concurrent assignments without final **else** branches.

9-valued Logic (IEEE-1164)

'U'	Uninitialized – default value
'X'	Forcing Unknown (often indicates an error in simulations!)
'0'	Forcing 0
'1'	Forcing 1
'Z'	High Impedance (e.g. tristate buffer, open-collector/open-drain)
'W'	Weak Unknown
'L'	Weak 0 (e.g. pull-down resistor)
'H'	Weak 1 (e.g. pull-up resistor)
'–'	Don't Care

- Intuitive meta-values for the debugging by simulation.
- Description of optimization possibilities: **else** '—'.
- Description of transmission gates, e.g. in bus drivers.
- Modeling of external pull-up or pull-down resistors.

Synthesis Hints

The synthesis produces *digital logic* only:

- Internal signals are always '0' or '1'.
- 'L' and 'H' are mapped to '0' or '1'.
- '-' and 'X' are mapped *arbitrarily* to '0' or '1' by the choice of the synthesis tool.
- On the device, 'Z' is eliminated by combinatorics.
  Tristate buffers are utilized in output pins.
- 'U' and 'W' generate an error in explicit assignments.

Simulation Hints

Meta-values have a great relevance for the circuit analysis by simulation:

- 'U' indicates signals that have not been evaluated yet and their dependents.
- '-' describes and *documents* disinterest.
  → Use, whenever possible!
- 'X' signals:
  - the evaluation of undetermined values ('X', 'Z', 'W', '-'), or
  - the driving of a bus with different values → **error!**

Example: 7-Segment-Decoding

```
library IEEE;                      -- Declare IEEE-Library
use IEEE.std_logic_1164.all;       -- Allows use of 9-valued logic

entity seg7dec is
  port(
    dig  : in  std_logic_vector(3 downto 0); -- decimal BCD digit
    seg7 : out std_logic_vector(6 downto 0)  -- H-active 7-segment output
  );
end seg7dec;

architecture seg7dec_impl of seg7dec is
begin
  with dig select
    seg7 <= "1111110" when "0000",
            "0110000" when "0001",
            "1101101" when "0010",
            "1111001" when "0011",
            "0110011" when "0100",
            "1011011" when "0101",
            "1011111" when "0110",
            "1110000" when "0111",
            "1111111" when "1000",
            "1111011" when "1001",
            "-------" when others;
end seg7dec_impl;
```

Example: MUX with Tristate Buffers

```
entity mux4 is
  port(
    sel         : in  std_logic_vector(1 downto 0);
    a, b, c, d  : in  std_logic;
    y           : out std_logic;
  );
end mux4;

architecture mux4_impl of mux4 is
begin
  y <= a when sel = "00" else 'Z';
  y <= b when sel = "01" else 'Z';
  y <= c when sel = "10" else 'Z';
  y <= d when sel = "11" else 'Z';
end mux4_impl;
```

Resolution of multiple bus drivers:

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

Similar tables are used for the computations of **and**, **or**, ...

Hardware-Oriented Arithmetic Data Types

For the representation of unsigned and signed numbers in two's complement with fixed but *arbitrary* bit length:

- Package: **use** IEEE.numeric_std.**all**;
- Definitions:
  unsigned: **type** unsigned **is array**(natural **range** <>) **of** std_logic;
  signed: **type** signed **is array**(natural **range** <>) **of** std_logic;
- . . . only differ in the implementation of sign-dependent arithmetic like $*$ and comparisons like < or >=.
- . . . permit a direct addition of integer:
  Count <= Count + 1;
- . . . enable:
  - a secure control over data widths, and
  - a simple description of bit (slicing) operations.

Type Conversions

- . . . serve the strict enforcement of the type concept,
- . . . do usually *not* induce hardware,
- . . . are implemented by VHDL functions:

```
function to_integer (arg : unsigned)       return natural;
function to_integer (arg : signed)         return integer;
function to_unsigned(arg, size : natural) return unsigned;
function to_signed  (arg  : integer;
                     size : natural) return signed;

-- Pseudo-functions for ''Related Types'':
function signed  (arg : std_logic_vector) return signed;
function unsigned(arg : std_logic_vector) return unsigned;
function signed  (arg : unsigned)         return signed;
...
```

Blocks

- . . . structure the implementation within an architecture, and
- . . . allow the isolation of local declarations:

```
<Label>: block
   [Local declarations]
begin
   ...
end block [Label];
```

Generics

- . . . allow a generic, parametric design:

```
-- Determines the parity of an n-bit argument
entity par_n is
  generic(N : positive := 8); -- Generic with Default
  port(
    arg : in  std_logic_vector(1 to N);
    par : out std_logic
  );
end par_n;

architecture rtl of par_n is
  signal  tmp : std_logic_vector(1 to N);
begin
  tmp(1) <= arg(1);
  xor_chain: for i in 2 to N generate
    tmp(i) <= tmp(i-1) xor arg(i);
  end generate;
  par <= tmp(N);
end rtl;
```

The generate-statement

- ...allows the description of repetitive or conditional substructures, and
- is especially useful in a generic design.

```
<Label>: if <Condition> generate
[ [local declarations]
begin]
   ...                    -- else-branch is only to come with VHDL-2003
end generate [Label];

<Label>: for <Identifier> in <range> generate
[ [local declarations]
begin]
   ...
end generate [Label];
```

Array and Type Attributes

- Simplify the generic description of array accesses:

  Example: **signal** arr : std_logic_vector(7 **downto** 0); ––*Array type*

arr'length	Array length	8
arr'**range**	Index range	7 **downto** 0
arr'reverse_range	Reverse index range	0 **to** 7
arr'left	Left boundary of index range	7
arr'right	Right boundary of index range	0
arr'high	Highest array index	7
arr'low	Lowest array index	0

  - Instead of variable names, type names may also be used.
  - The desired dimension of multidimensional array types has to be selected by argument:
    arr'length(1) and arr'length are equivalent.

Components

- realize a hierarchically structured designs, and
- allow the reuse of available solutions.

```
architecture rtl of xyz is
-- Component declaration (analogous to the declaration of the entity)
component par_n is
  generic(N : positive := 8); -- Generic with Default
  port(
    arg : in  std_logic_vector(1 to N);
    par : out std_logic
  );
end component;
-- Connections signals
signal arg1, arg2 : std_logic_vector(5 downto 0);
signal par1, par2 : std_logic;
...
begin
  -- Instantiation
  p1 : par_n     -- Named assignment of parameters / signals
    generic map(N   => 6)
    port    map(arg => arg1, par => par1);
  p2 : par_n     -- Positional assignment of parameters / signals
    generic map(6)
    port    map(arg2, par2);
  ...
end rtl;
```

Components II

- Open outputs are bound with the help of the keyword **open**.
- A particular implementation can be requested if multiple architectures are available:

```
architecture rtl of xyz is
...
-- work is implicit library to current project
for par1 : par_n use entity work.par_n(rtl_fast);
...
begin
  -- Instantiation
  par1 : par_n  -- Known assignment of parameters / signals
    generic map(N   => 6)
    port    map(arg => arg1, par => par1);
  ...
end rtl;
```

Packages

. . . are useful for the collection of:

- non-local types and constants,
- component declarations,
- functions and procedures.

Separation into interface and implementation:

```
package Paket is
  subtype tALUOp is std_logic_vector(1 downto 0);
  constant ALU_OP_ADD : tALUOp := "00";
  ...
  function max(arg1 : integer; arg2 : integer) return integer;
end package Paket;

package body Paket is
  function max(arg1 : integer; arg2 : integer) return integer is
  begin
    if arg1>arg2 then return arg1; end if;
    return arg2;
  end;
end package body Paket;
```

Functions and Procedures

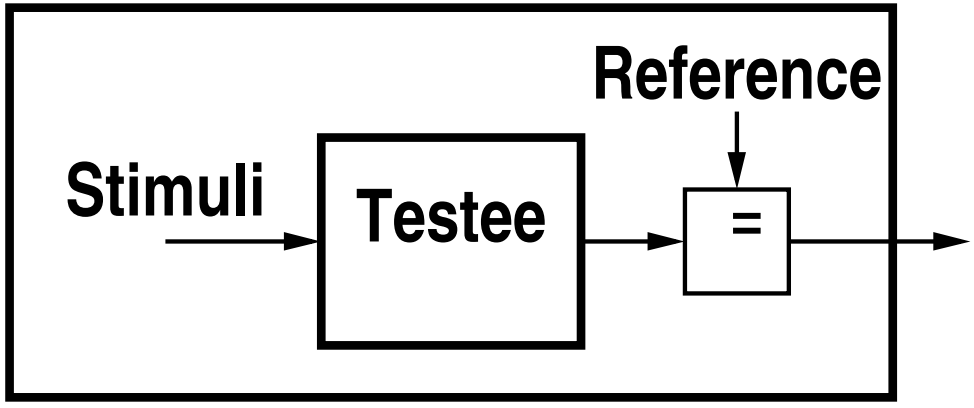
- are a powerful utility for behavioral models but
- their support by synthesis tools is limited:
  - → e.g. broad support for the computation of constant or combinatorial expressions.
- Parameters are passed in different modes:

Mode	Class	Parameter	
		Procedure	**Function**
in[1]	**constant**[2]	Expression	Expression
	signal	Signal	Signal
	variable	Variable	⊘
out / inout	**signal**	Signal	⊘
	variable[2]	Variable	⊘

[a] Default
[b] Default for Mode

Test Benches

- serve the design evaluation and verification by simulation,
- take their target design as component (device under test - DUT), and
- implement a behavioral model for:
  - the generation of stimuli (input signals) as well as
  - the checking of the output signals (e.g. by comparison).



Endless Processes

- are declared without a sensitivity list,
- are executed in an implicit infinite loop, and
- have to be interrupted by **wait** statements:
  wait on <Sensitivity list>;
  wait for <Time>;
  wait until <Condition>;
  wait; —— *forever*
  wait for a Δ-cycle with: **wait for** 0 ns;
- only a few fix **wait** constructs can be synthesized:
  e.g.: **wait until** clk'event **and** clk = '1'; ——*only once in a process*

Assertions

- are used for checking
  - static assumptions, e.g. about generics, or
  - dynamic assumptions during the simulation.
- are not synthesized into logic.

```
assert <Condition>                                  -- Assertion
  [report   <String>]                               -- Message
  [severity (note|warning|error|failure)];          -- Severity
```

- If the assertion does not apply:
  - the message is output if it is provided, and
  - the simulation aborts depending on the declared severity.

Assertions: Example

```
architecture tb_impl of tb is
  signal arg : std_logic_vector(1 to 4);
  signal par : std_logic;
begin
  par : par_n
    generic map(N => 4)
    port    map(arg, par);

  process
  begin
    for i in 0 to 15 loop
      arg <= to_stdlogicvector(i, 4);
      wait for 10ns;
      assert par = (arg(1) xor arg(2) xor arg(3) xor arg(4))
        report "Parity Mismatch"
        severity error;
    end loop;
    report "Test complete";
    wait; -- forever
  end process;
end tb_impl;
```

Hints for Synthesis

- Good solutions demand a good overview!
  Structure → Block Diagrams
  Behavior → SM-Charts
- Design preferably on a high abstraction level.
  The synthesis tool usually does a better technology mapping.
- Buffer asynchronous inputs (external, other clock domains) with FFs!
  Unless there is *definitely* only *one* I/O-register-path,
  inconsistencies due to run-time differences may occur.
- FPGA: It is better to use more FFs instead of complex logic.
  Every logic cell has one FF.
  One-hot state encoding is usually favorable for non-trivial state machines.

Meta-Values

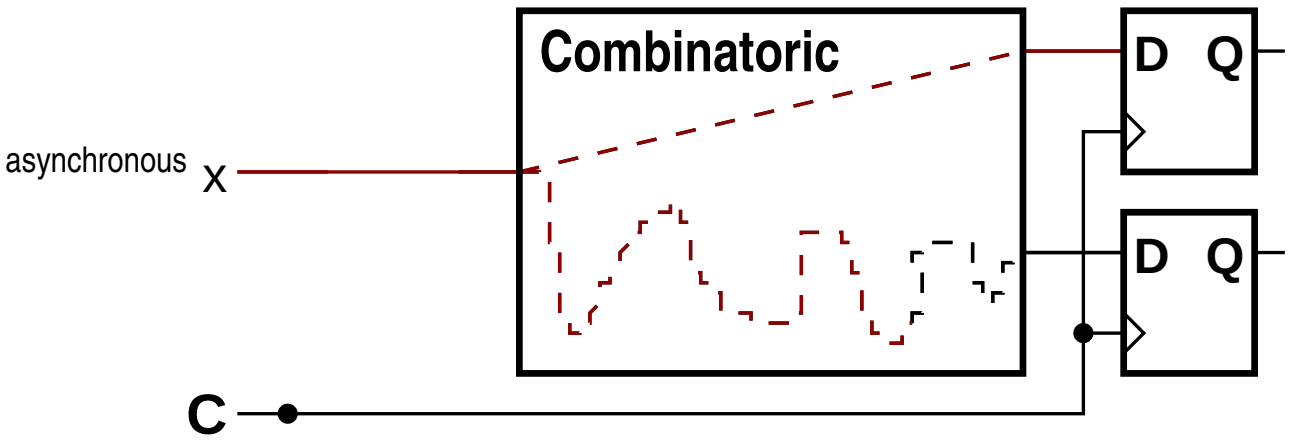
Useful functions:

Function	Simulation	Synthesis
Is_X(s)	s maps unambiguously to '0' or '1'	false
Is_X('X')	true	false
to_X01	Mapping to '0', '1' or 'X' ('U', 'X', 'W', '-', 'Z')	Identity
rising_edge(clk)	clk'event **and** to_X01(clk) = '1' **and** to_X01(clk'last_value) = '0'	

Use Don't Cares intensively! They:

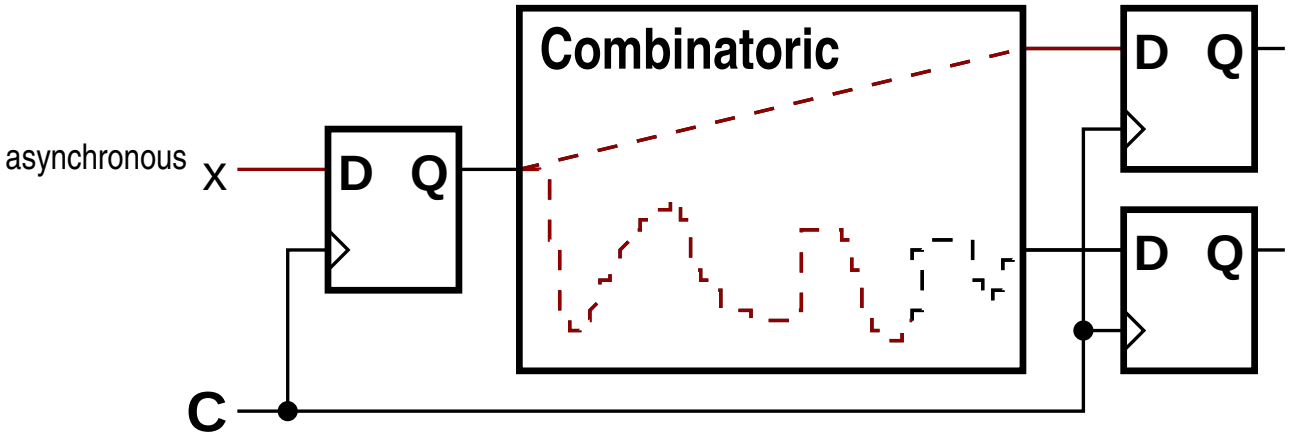
- reveal the use of seemingly irrelevant signal values by 'X's instead of a normal but wrong signal value in the simulation.
- document irrelevant signal states, and
- open optimization opportunities for the synthesis tool.

Asynchronous Inputs



- Input switches at an *arbitrary* time (with respect to the device clock).
- Propagation through the combinatorics by the next active edge is *not* assured.
- Inconsistencies in the system state may arise from different path lengths.

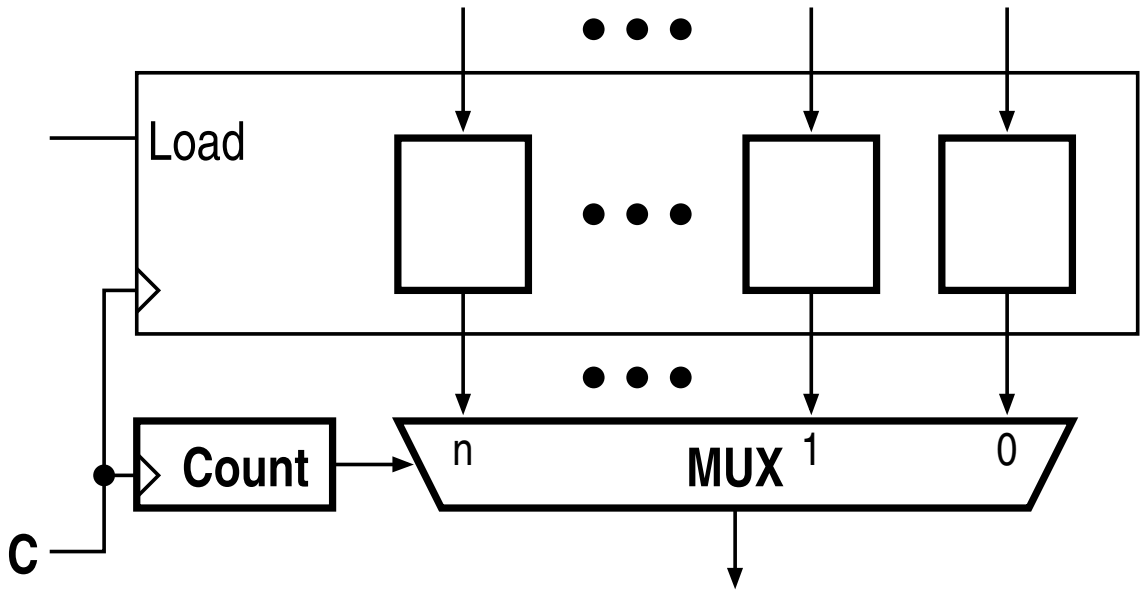
Asynchronous Inputs



- Register-register delays are determined by the synthesis tool.
  → A warning is generated if the clock constraint cannot be satisfied.
- Often, there is a second FF-stage for masking meta-stabilities.
- Note that also Reset buttons are asynchronous inputs.

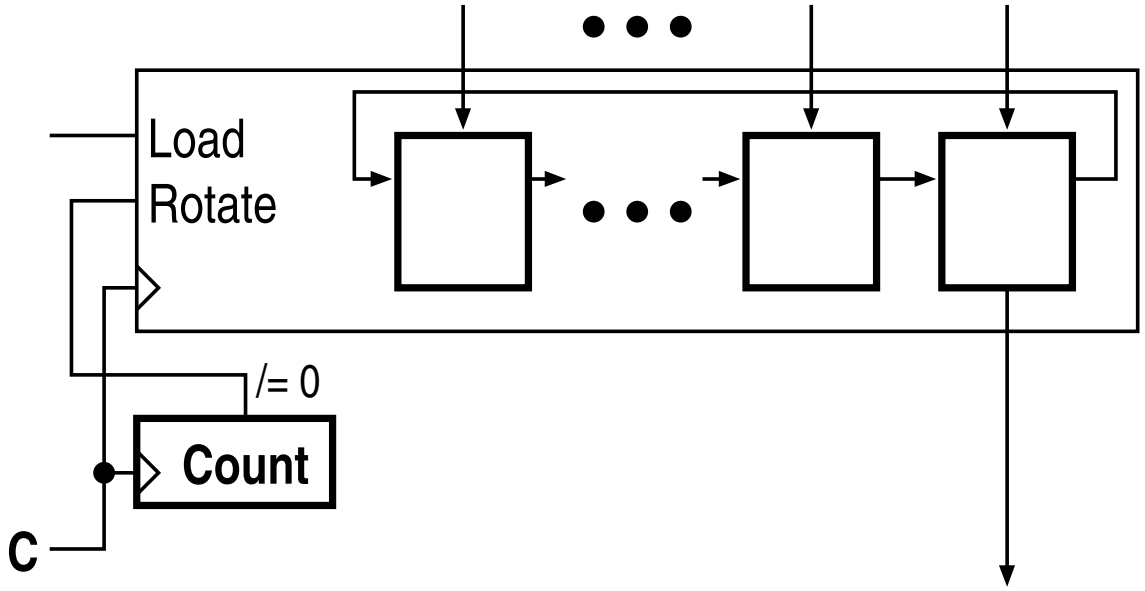
Iterations over Arrays

- Array access by index increment implies a complex multiplexer circuit or enable logic.



Iterations over Arrays

- Often better:
  Simple regular structure with the help of a shift register.



Application:
Parallelization, Serialization, Iteration over digits (arithmetic), . . .

Wide Gates and Comparators

Keep implementations maintainable by the use of array attributes:

```
-- Wide OR
y <= '1' when x /= (x'range => '0') else '0';

-- Wide AND
y <= '1' when x  = (x'range => '1') else '0';

-- Wide XOR
process(x)
  variable t : std_logic;
begin
  t := '0'; -- Neutral initialization
  for i in x'range loop
    t := t xor x(i);
  end loop;
  y <= t;   -- Final result assignment (signal)
end process;

-- Comparator
y <= '1' when x = to_unsigned(42, x'length) else '0';
```

Classic Detection of End-of-Count

```
-- Counts: 0 .. CNT_CNT-1
process(clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      Cnt <= (others => '-');
    else
      if Init = '1' then
        Cnt <= (others => '0');
      elsif Step = '1' then
        Cnt <= Cnt + 1;
      end if;
    end if;
  end if;
end process;

-- AND over *all* (possibly inverted) digits of Cnt
Done <= '1' when Cnt = CNT_CNT-1 else '0';
```

Optimized Detection of End-of-Count

```
-- Counts: 0 .. CNT_CNT-1
process(clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      Cnt <= (others => '-');
    else
      if Init = '1' then
        Cnt <= (others => '0');
      elsif Step = '1' then
        Cnt <= Cnt + 1;
      end if;
    end if;
  end if;
end process;

-- AND over all expected '1' of Cnt (only for up-counter!)
Done <= '1' when (Cnt or not to_unsigned(CNT_CNT-1, Cnt'length))
                  = (Cnt'range => '1') else '0';
```

Detection of End-of-Count by Change of Sign

```
-- Counts: CNT_CNT-2 .. -1
--  -> Cnt must have an extra bit for the sign!
process(clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      Cnt <= (others => '-');
    else
      if Init = '1' then
        Cnt <= to_signed(CNT_CNT-2, Cnt'length);
      elsif Step = '1' then
        Cnt <= Cnt - 1;
      end if;
    end if;
  end if;
end process;

-- No combinatorics!
Done <= Cnt(Cnt'left);
```

Literature

- Roth, Charles H., Jr.: *Digital Systems Design Using VHDL*, PWS Publishing Company, 1998.
- Smith, Douglas J.: *HDL Chip Design: A Practical Guide for Designing, Synthesizing & Simulating ASICs & FPGAs Using VHDL or Verilog*, Doone Publishing, 1996.
- *Hamburger VHDL-Archiv*, `http://tams.informatik.uni-hamburg.de/research/vlsi/vhdl/`, especially chapter *Documentation*.