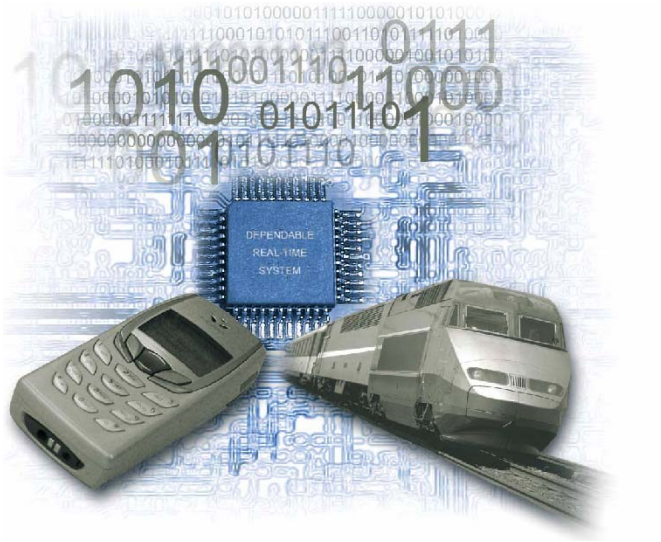
Embedded Systems 9



Exam registration

- Exam registration through HISPOS open now.
- If you cannot register through HISPOS
  → send email to finkbeiner@cs.uni-sb.de
- Deadline: December 5

Message Sequence Charts

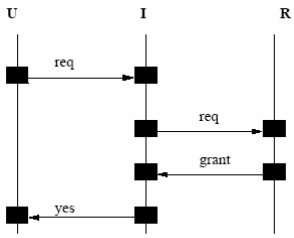
REVIEW

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Communicating finite state machines	StateCharts, StateFlow		SDL, MSCs
Data flow model ⊂ **Computational graphs**			Kahn process networks, SDF Petri nets
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	
Discrete event (DE) model	VHDL, Simulink	Only experimental systems, e.g. distributed DE in Ptolemy	

MSC: Example

REVIEW

- user (U) sends a request to an interface (I) to gain access to a resource R
- interface in turn sends a request to the resource, receives “grant” as a response
- Sends “yes” to U.



Basic MSCs

REVIEW

Consider the Σ-labeled poset $Ch = (E, \leq, \lambda)$ where

- E : set of **events**
- $\leq \subseteq E \times E$: **causality relation**
- $(E, \leq)$ is a **partially ordered set** (poset)
- $\lambda : E \rightarrow \Sigma$ is a **labeling function** with set of actions Σ

- $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$ „events in which p takes part“
- $E_{p!q} = \{e \mid e \in E_p \text{ and } \lambda(e) = p!q(m) \text{ for some } m \in M\}$
- $E_{p?q} = \{e \mid e \in E_p \text{ and } \lambda(e) = p?q(m) \text{ for some } m \in M\}$

Definition basic MSCs

REVIEW

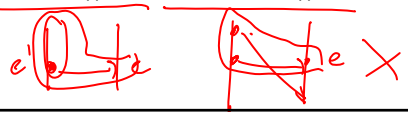
An **MSC over** (P, M, Act) is a Σ-labeled poset $Ch = (E, \leq, \lambda)$ that satisfies:

1. *All events that a process takes part in are linearly ordered; each process is a sequential agent:*

   $\leq_p$ is a linear order for each p, where $\leq_p$ is $\leq$ restricted to $E_p \times E_p$.

2. *Messages must be sent before they can be received:*

   Let $\lambda(e) = p?q(m)$, then $|\downarrow(e) \cap E_{p?q}| = |\downarrow(e) \cap E_{q!p}|$ and there exists $e' \in \downarrow(e)$ such that $\lambda(e') = q!p(m)$ and $|\downarrow(e) \cap E_{q!p}| = |\downarrow(e') \cap E_{q!p}|$



Definition basic MSCs

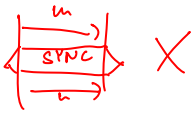
REVIEW

3. *There are no dangling communication edges in an MSC; all sent messages have also been received:*

   For every p,q with $p \neq q$, $| E_{p?q} | = | E_{q!p} |$

4. *Causality relation between the events in an MSC is completely determined by the order in which the events occur within each process and communication relation relating send-receive pairs:*

   $\leq = (\leq_P \cup R_P)^*$, where $\leq_P = \cup_{p \in P} \leq_P$ and $R_P = \cup_{p,q \in P, p \neq q} R_{(p,q)}$



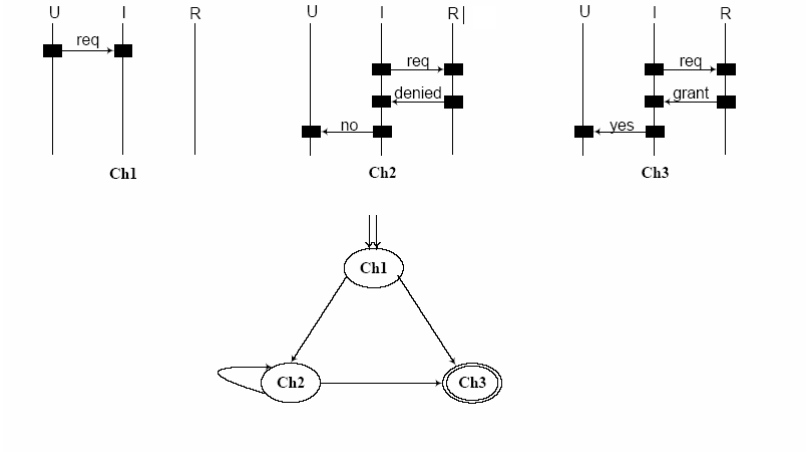
HMSCs

REVIEW

- HMSC is a finite state automaton whose states are labeled with MSCs over (*P, M, Act*).
  - Results in finite specifications involving choice, concatenation and iteration operations over a finite set of seed MSCs.
- [in general, specification can be hierarchical, i.e. a state of the automaton can be labeled by an HMSC instead of an MSC.
  Here:
  flattened HMSCs, *message sequence graphs* (MSGs).]

Example (1)

REVIEW



Properties

REVIEW

- asynchronous concatenation of two charts is also a chart.
- synchronous concatenation of two charts may not result in a chart.
- Asynchronous concatenation may lead to non-regular languages
- **Theorem:** The intersection of two MSGs (with asynchronous concatenation) is undecidable.

Mandatory vs. provisional behavior

REVIEW

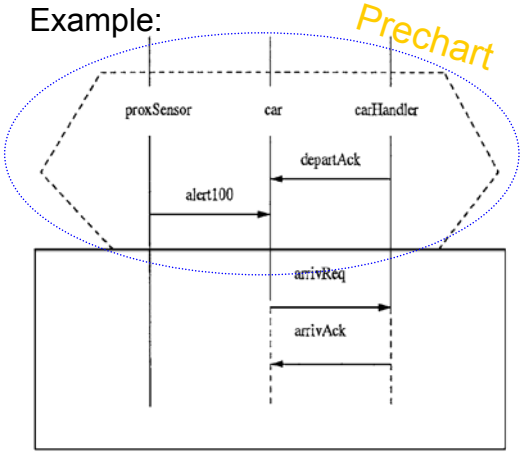
Level	Mandatory (solid lines)	Provisional (dashed lines)
Chart	All runs of the system satisfy the chart	At least one run of the system satisfies the chart
Location	Instance must move beyond location/time	Instance run need not move beyond loc/time
Message	If message is sent, it will be received	Receipt of message is not guaranteed
Condition	Condition must be met; otherwise abort	If condition is not met, exit subchart

Universal LSC with Prechart

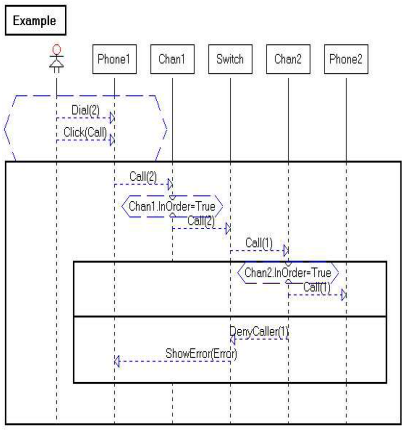
REVIEW

Precharts describe conditions that must hold for the main chart to apply.

Example:



Composition of LSCs



Additional concepts not covered here

- Instantaneous messages
- Method calls and returns
- Time, Timer events
- Coregions
- …

VHDL

VHDL

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Communicating finite state machines	StateCharts, StateFlow		SDL, MSCs
Data flow model ⊂ **Computational graphs**			Kahn process networks, SDF Petri nets
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	
Discrete event (DE) model	VHDL, Simulink	Only experimental systems, e.g. distributed DE in Ptolemy	

VHDL

- HDL = hardware description language
- VHDL = VHSIC hardware description language
- VHSIC = very high speed integrated circuit
  - Consortium which developed VHDL (Intermetrics Inc., IBM, Texas Instruments)
  - Early 80's, initiated by US Department of Defense
- Modeling of digital circuits
- 1987 IEEE Standard 1076
- Reviews of standard: 1993, 2000, 2002, 2008

⇒ Standard in (European) industry

- Extension: VHDL-AMS, includes analog modeling

Goals

- Main goal was modeling of digital circuits
  - Modelling at various levels of abstraction
  - Technology-independent

  ⇒ Re-Usability of specifications
  - Standard

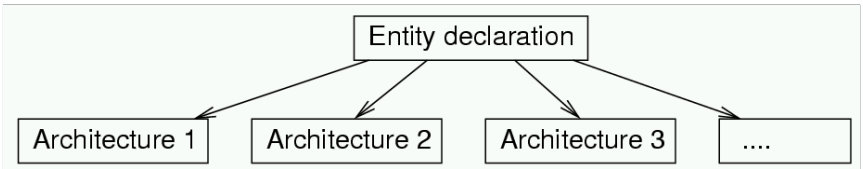
  ⇒ Portability (different synthesis and analysis tools possible)
  - Validation of designs based on the same description language for different levels of abstraction
- **Powerful** description language
- Contains also many aspects of imperative programming languages

⇒ VHDL is able to describe software, too.

- **Here**: Only some aspects of VHDL, not complete language.

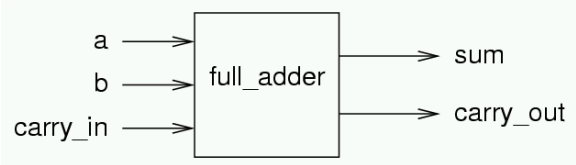
Entities and architectures

- Each design unit is called an **entity**.
- Entities are comprised of **entity declarations** and one or several **architectures.**



Each architecture includes a model of the entity. By default, the most recently analyzed architecture is used. The use of another architecture can be requested in a **configuration**.

Example: full adder
- Entity declaration -



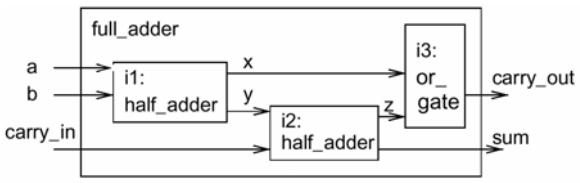
- **Entity declaration:**

```
entity full_adder is
port(a, b, carry_in: in Bit;  -- input ports
  sum,carry_out: out Bit); --output ports
end full_adder;
```

Example: full adder
- Architecture with behavioural body

```
architecture behavior of full_adder is
 begin
  sum       <= (a xor b) xor carry_in after 10 Ns;
  carry_out <= (a and b) or (a and carry_in) or
                  (b and carry_in)       after 10 Ns;
 end behavior;
```


Example: full adder
- structural body




```
architecture structure of full_adder is
  component half_adder
    port (in1,in2:in Bit; carry:out Bit; sum:out Bit);
  end component;
  component or_gate
    port (in1, in2:in Bit; o:out Bit);
  end component;
 signal x, y, z: Bit;      -- local signals
  begin                    -- port map section
   i1: half_adder port map (a, b, x, y);
   i2: half_adder port map (y, carry_in, z, sum);
   i3: or_gate    port map (x, z, carry_out);
  end structure;
```

Example: full adder - Architectures

- Architectures describe implementations of entities.
- For component half_adder we need
  - An entity, e.g.

```
entity half_adder
   port (in1,in2:in Bit; carry:out Bit; sum:out Bit);
end half_adder;
```

  - (At least) one architecture
    - This architecture may contain components, too.
- Architectures and their components can define a hierarchy of arbitrary depth.

Structural and behavioural descriptions

- Structural descriptions use component instances.
- Behavioural descriptions describe behaviour without defining the structure of the system.
- Mixtures are **possible**.
- Mixtures are **needed**, at least for the leaves in structural hierarchy.
- Structural hierarchy is essential for a compact and clear modelling of large (hardware) systems.
- To define semantics of VHDL, we can assume that the structural hierarchy is „flattened“, i.e., we can assume w.l.o.g. that we have just an behavioural description.

Essential elements of behavourial descriptions: Processes

- Behavioural descriptions consist of a set of concurrently executed (explicit or implicit) processes.

- Explicit processes:
  Syntax:
  [label:]
  process[(sensitivity list)]
  declarations
  begin
  statements
  end process *[label]*

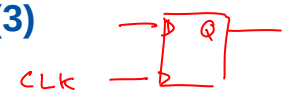
Processes – Examples (1)


```
architecture RTL of NANDXOR is
begin
   process
   begin
        if (C='0') then
           D <= A nand B after 5 ns;
        else
           D <= A and B after 10 ns;
        end if;
        wait on A, B, C;
   end process;
end RTL;
```

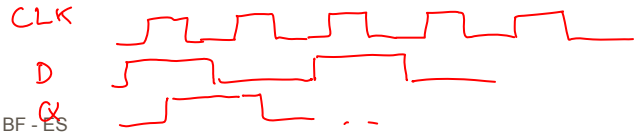
Processes – Examples (2)

```
signal clk : std_logic;

...

clk_gen : process
begin
   clk <= 0;
   wait for 5 ns;
   clk <= 1;
   wait for 5 ns;
end process clk_gen;
```

Processes – Examples (3)




```
architecture RTL of DFF is
begin
   p : process
   begin
        if (clk'event) and (clk='1') then
           Q <= D;
        end if;
        wait on clk;
   end process p;
end RTL;
```



Processes - Execution

- Processes are not allowed to have subprocesses (no hierarchy of processes).
- Processes are executed sequentially until a wait statement is encountered.
- Processes are reactivated according to conditions of wait-statements.
- Different types of wait-statements

Wait-statements

Four possible types of **wait**-statements:

- **wait on** *signal list*;
  - wait until at least one of the signals in signal list changes;
  - Example: **wait on** a;
- **wait until** *condition*;
  - wait until condition is met;
  - Example: **wait until** c='1';
- **wait for** *duration*;
  - wait for specified amount of time;
  - Example: **wait for** 10 ns;
- **wait;**
  - suspend indefinitely

Processes - Sensitivity lists

- Sensitivity lists are a shorthand for a single **wait on**-statement at the end of the process body:
- **process** (x, y)
  begin
  prod <= x and y ;
  end process;

is equivalent to

- **process**
  begin
  prod <= x and y ;
  wait on x,y;
  end process;

Essential elements of behavourial descriptions
Signal assignments

- Signal assignments outside processes can be viewed as implicit processes:

  a <= b **and** c **after** 10 ns

  is equivalent to

  process(b, c)
  begin
  a <= b **and** c **after** 10 ns
  end

Essential elements of behavourial descriptions
Constants, signals and variables

- **Constants**
  - the value of a constant cannot be changed.
- **Examples**:

```
constant PI : real := 3.1415;
constant DEFAULT : bit_vector(0 to 3) :=„1001“;
constant PERIOD : time := 100 ns;
```


Essential elements of behavourial descriptions
Constants, signals, and variables

- Variables
  - Variables are declared locally in processes (and procedures / functions) and are only visible in this scope.
- Signals
  - Can be viewed as a wire
  - Signals cannot be declared in processes (procedures / functions), but in architectures (outside processes).
- Syntax:
  - `variable_assignment ::=`
    `target := expression`
    - Example:
      Sum := 0
  - `signal_assignment ::=`
    `target <= [ delay_mechanism ] waveform_element`
    `{ , waveform_element }`
  - `waveform_element ::=`
    `value_expression [ after time_expression ]`
    - Example:
      Inpsig <= ´0´, ´1´after 5 ns, ´0´ after 10 ns, ´1´ after 20 ns;

Variable versus signal assigment

- Variable assignments are performed **sequentially** and directly after their occurence,
- Signal assignments are performed **concurrently**, i.e. they are (sequentially) collected until the process is stopped and are performed in parallel after all processes are stopped.

```
signal a : std_logic :=
   '0';
signal b : std_logic :=
   '1';
...
swap : process
variable c : std_logic := `1`;
variable d : std_logic := `0`;
begin
   a <= b; b <= a;
   c := d; d := c;
   wait on a, b;
end process swap;
```

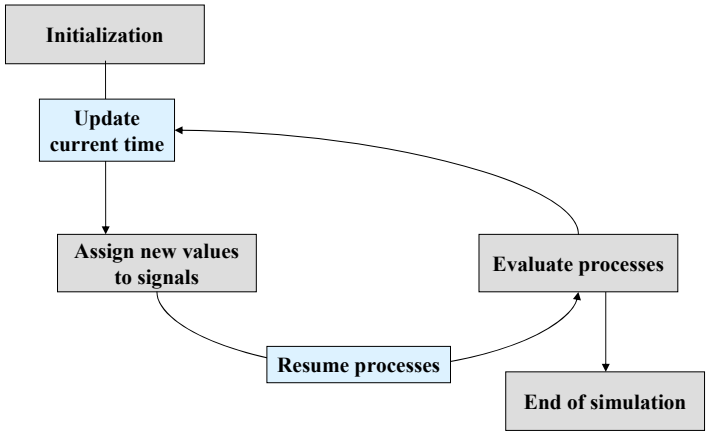

Semantics of VHDL: Basic concepts

- Similar concepts as in StateCharts.
- „Discrete event driven simulation“

- Step-based semantics as in StateCharts:
  - Computation as a series of basic steps
  - Time does not necessarily proceed between two steps
  - Like superstep semantics of StateCharts
- Concurrent assignments (of signals) like concurrent assignments in StateCharts.

⇒ Steps consist of two stages.

Overview of simulation



Transaction list and process activation list

- Transaction list
  - For signal assignments
  - Entries of form (*s, v, t*) meaning „signal *s* is set to value *v* at time *t*“
  - Example: (clock, ´1´, 10 ns)
- Process activation list
  - For reactivating processes
  - Entries of form (p_i, *t*) meaning „process p_i resumes at time *t*“.

Initialization

- At the beginning of initialization, the current time, t_{curr}, is assumed to be 0 ns.
- An initial value is assigned to each signal.
  - Taken from declaration, if specified there, e.g.,
    - **signal** s : std_ulogic := `0`;
  - Otherwise: First value in enumeration for enumeration based data types, e.g.
    - **signal** s : std_ulogic with **type** std_ulogic **is** (`U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `-`); ⇒ initial value is `U`
  - This value is assumed to have been the value of the signal for an infinite length of time prior to the start of the simulation.
- Initialization phase executes each process exactly once (until it suspends).
- During execution of processes: Signal assignments are collected in transaction list (**not** executed immediately!) – more details later.
- If process stops at „wait for"-statement, then update process activation list – more details later.
- After initialization the time of the next simulation cycle (which in this case is the first simulation cycle), t_{next} is calculated:
  - Time t_{next} of the next simulation cycle = earliest of
    1. time high (end of simulation time).
    2. Earliest time in transaction list (if not empty)
    3. Earliest time in process activation list (if not empty).

Example

```
architecture behaviour of example is
    signal a : std_logic := `0`;
    signal b : std_logic := `1`;
    signal c : std_logic := `1`;
    signal d : std_logic := `0`;
begin
    swap1: process(a, b)
    begin
        a <= b after 10 ns;
        b <= a after 10 ns;
    end process;

    swap2: process
    begin
        c <= d;
        d <= c;
        wait for 15 ns;
    end process;

end architecture;
```

$t_{curr} = 0$

Transaction list:

(a, 1, 10 ns)
(b, 0, 10 ns)
(c, 0, 0 ns)
(d, 1, 0 ns)

Process activation list:

(swap2, 15 ns)

Signal assignment phase – first part of step

- Each simulation cycle starts with setting the current time to the next time at which changes must be considered:
- $t_{curr} = t_{next}$
- This time t_{next} was either computed during the initialization or during the last execution of the simulation cycle. Simulation terminates when the current time would exceed its maximum, time'high.
- For all (s, v, t_{curr}) in transaction list:
  - Remove (s, v, t_{curr}) from transaction list.
  - s is set to v.
- For all processes p_i which wait on signal s:
  - Insert (p_i, t_{curr}) in process activation list.
- Similarly, if condition of „**wait until**"-expression changes value.

Example

```
architecture behaviour of example is
    signal a : std_logic := `0`;
    signal b : std_logic := `1`;
    signal c : std_logic := `1`;
    signal d : std_logic := `0`;
begin
    swap1: process(a, b)
    begin
        a <= b after 10 ns;
        b <= a after 10 ns;
    end process;

    swap2: process
    begin
        c <= d;
        d <= c;
        wait for 15 ns;
    end process;

end architecture;
```

$t_{curr} = 0$

$c := 0$

$d := 1$

TL: (a, 1, 10ns)
(b, 0, 10ns)

PAL: (swap2, 15ns)

Process execution phase – second part of step (1)

- Resume all processes p_i with entries (p_i, t_{curr}) in process activation list.
- Execute all activated processes „in parallel“ (in fact: in arbitrary order).
- Signal assignments
  - are collected in transaction list (**not** executed immediately!).
  - Examples:
    - *s* <= *a* **and** *b*;
      - Let *v* be the conjunction of current value of *a* and current value of *b*.
      - Insert (*s, v,* t_{curr}) in transaction list.
    - *s* <= ´1´ **after** 10 ns;
      - Insert (*s*, ´1´, t_{curr} + 10 ns) into transaction list.
- Processes are executed until wait statement is encountered.
- If process p_i stops at „wait for“-statement, then update process activation list:
  - Example:
    - p_i stops at „**wait for** 20 ns;“
    - Insert (p_i, t_{curr} + 20 ns) into process activation list

Process execution phase – second part of step (2)

If some process reaches last statement and

- does not have a sensitivity list and
- last statement is not a wait statement,

then it continues with first statement and runs until wait statement is reached.

- When all processes have stopped, the time of the next simulation cycle t_{next} is calculated:
  - Time t_{next} of the next simulation cycle = earliest of
    1. time'high (end of simulation time).
    2. Earliest time in transaction list (if not empty)
    3. Earliest time in process activation list (if not empty).
- Stop if t_{next} = time'high and transaction list and process activation list are empty.

Example

```
architecture behaviour of example is
    signal a : std_logic := `0`;
    signal b : std_logic := `1`;
    signal c : std_logic := `1`;
    signal d : std_logic := `0`;
begin
    swap1: process(a, b)
    begin
        a <= b after 10 ns;
        b <= a after 10 ns;
    end process;

    swap2: process
    begin
        c <= d;
        d <= c;
        wait for 15 ns;
    end process;

end architecture;
```

$t_{curr} = 10$ ns

a := 1
b := 0

PAL: (swap1, 10ns)
(swap2, 15ns)

TL: ∅

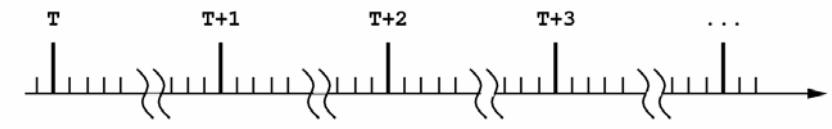
$t_{cur} = 10$ ns

TL: (a, 0, 20ns)
(b, 1, 20ns)

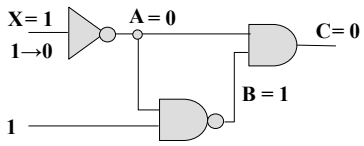
PAL: (swap2, 15ns)

Delta delay

- As for StateCharts (super step semantics!) time does not necessarily proceed between two steps.
- Several (potentially an infinite number of) steps can take place at the same time t_{curr}.
- **Notion**: Signal assignments which take place at the same time in two consecutive steps are separated by one „**delta delay**“.



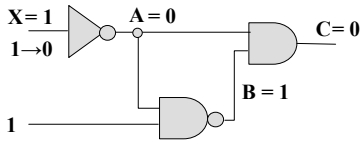
Delta delay - Example



Simulation time does not proceed due to delta delays!

Current time	Delta delay	Event
0 ns	1	-- evaluation of inverter -- (A, 1, 0 ns)

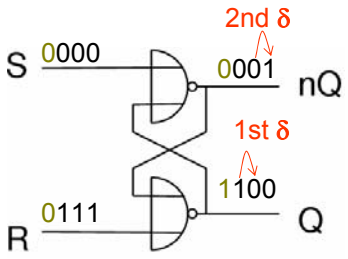
Delta delay - Example



Simulation time does not proceed due to delta delays!

Current time	Delta delay	Event
0 ns	1	-- evaluation of inverter -- (A, 1, 0 ns)
	2	-- evaluation of AND and NAND -- (B, 0, 0ns), (C, 1, 0ns)
	3	-- evaluation of AND -- (C, 0, 0ns)

Delta delay - Simulation of an RS-Flipflop



	0ns	0ns+δ	0ns+2δ
R	1	1	1
S	0	0	0
Q	1	0	0
nQ	0	0	1

```
entitiy RS_Flipflop is
    port (R, S : in std_logic;
          Q, nQ : inout std_logic);
end RS_FlipFlop;

architecture one of RS_Flipflop is
    begin
    process (R,S,Q,nQ)
    begin
          Q := R nor nQ;
          nQ := S nor Q;
    end process;
end one;
```

δ cycles reflect the fact that no real gate comes with zero delay.

Semantics of VHDL Details

- What happens, if the same signal is written more than once in one step?
- Inertial and transport delay model
- Some additional language elements
- Recursive description of parameterized hardware

„Write-write-conflicts“

```
signal s : bit;
...
p : process
begin
   ...
   s <= '0';
   ...
   s <= '1';
   wait for 5 ns;
end process p;
```

- **Case 1:**
  Write-write-conflicts are restricted to the same process
  (i.e. they occur inside the same process)
  - Then the second signal assignment overwrites the first one.
  - This is the only case of „non-concurrency“ of signal assignments
  - Note that writing to different signals occurs concurrently, however!

„Write-write-conflicts“

```
signal s : dt;
...
s<= v1;
...
p : process
begin
   ...
   s <= v2;
   ...
end process p;

q : process
begin
   ...
   s <= v3;
   ...
end process q;
```

- **Case 2:**
  Write-write-conflicts between different processes (explicit or implicit processes)
  - If there is no **„resolution function“** for the data type *dt,* then writing the same signal by different processes in the same step is **forbidden**.
  - If there is a resolution function, then the resolution function computes the value of s at time t_{curr}:
    - Value for s in the current step is computed for each process separately,
    - „resolution function“ for different values is used to compute final result.
- In the following:
  Data type std_ulogic with resolution function
  ⇒ data type std_logic

Multi-valued logic and standard IEEE 1164

- How many logic values for modeling?
- Two ('0' and '1') or more?
- If real circuits have to be described, some abstraction of the resistance (inversely-related to the strength) is required.

⇒ We introduce the distinction between:

  - the **logic level** (as an abstraction of the voltage) and
  - the **strength** (as an abstraction of the current drive capability) of a signal.

- Both logic level and strength are encoded in **logic values**.

1 signal strength

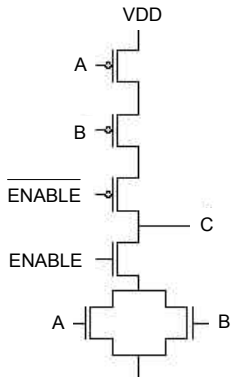
- Logic values '0' and '1'.
- Both of the same strength.
- Encoding false and true, respectively.

- No meaningful “resolution function” possible, if `0` and `1` are written to the same signal at the same time.

2 signal strengths (1)

- Many subcircuits can be effectively disconnected from the rest of the circuit (they provide „high impedance“ values to the rest of the circuit).
- Example: subcircuits with tri-state outputs.

Example: Tristate NOR

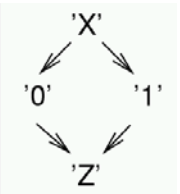


ENABLE = `0`
⇒ C is disconnected from the rest of the circuit

☞ We introduce signal value 'Z', meaning „high impedance “

2 signal strengths (2)

- We introduce an operation #, which generates the effective signal value whenever two signals are connected by a wire (“resolution”).
- #('0','Z')='0'; #('1','Z')='1'; '0' and '1' are „stronger“ than 'Z'



1 strength

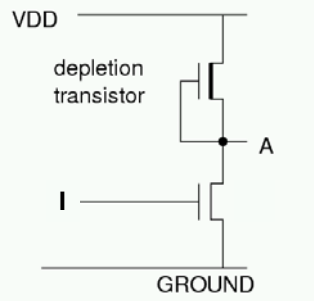
According to the partial order in the diagram, # returns the *larger of the two arguments*.

In order to define #('0','1'), we introduce 'X', denoting an undefined signal level.
'X' has the same strength as '0' and '1'.

3 signal strengths

Current set of values insufficient for describing real circuits:

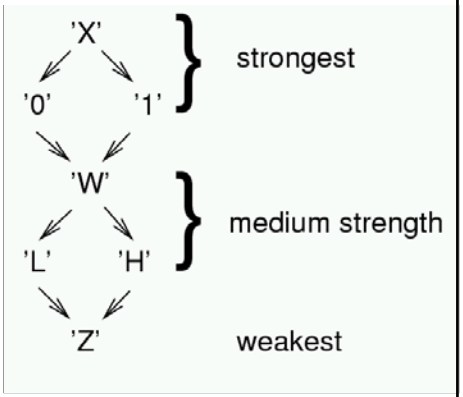
Example:
nMOS-Inverter



Depletion transistor (resistor) contributes a weak value to be considered in the #-operation for signal A
☞ Introduction of 'H', denoting a weak signal of the same level as '1'.
#('H', '0')='0'; #('H,'Z') = 'H'

3 signal strengths

- There may also be weak signals of the same level as '0'
- ☞ Introduction of 'L', denoting a weak signal of the same level as '0':
#('L', '0')='0'; #('L,'Z') = 'L';
- ☞ Introduction of 'W', denoting a weak signal of the same level as 'X':
#('L', 'H')='W'; #('L,'W') = 'W';
- # reflected by the partial order shown.



IEEE 1164

- VHDL allows user-defined value sets.

⇒ Each model could use different value sets (unpractical)

⇒ Definition of standard value set according to standard IEEE 1164:

{'0', '1', 'Z', 'X', 'H', 'L', 'W', 'U', '-'}

- First seven values as discussed previously.
- 'U': un-initialized signal; used by simulator to initialize all not explicitly initialized signals:
  type std_ulogic **is** (`U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `-`);
- '-': is used to specify don't cares:
  - Example: **if** a /= '1' **or** b/='1' **then** f <= a **exor** b; **else** f <= '-';
  - '-' may be replaced by arbitrary value by synthesis tools.