

Embedded Systems

10



REVIEW: VHDL

- HDL = hardware description language
- VHDL = VHSIC hardware description language
- VHSIC = very high speed integrated circuit

- Initiated by US Department of Defense
- 1987 IEEE Standard 1076
- Reviews of standard: 1993, 2000, 2002, 2008

- Standard in (European) industry

- Extension: VHDL-AMS, includes analog modeling

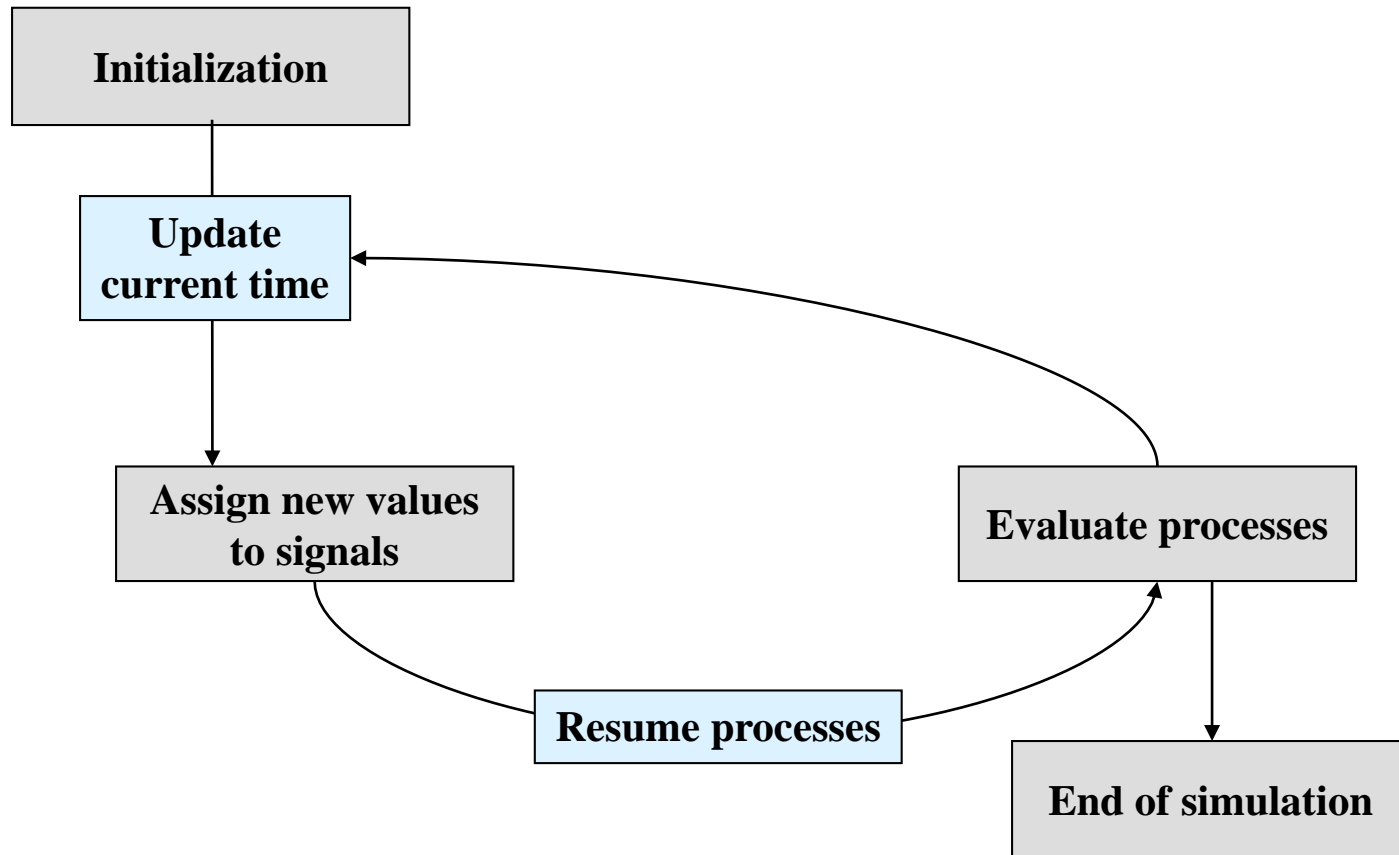
REVIEW: Semantics of VHDL:

Basic concepts

- „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.

REVIEW: 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;
```

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;
```

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:
 - $s \leq a$ **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.
 - $s \leq '1'$ **after** 10 ns;
 - Insert $(s, '1', t_{curr} + 10 \text{ ns})$ 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 $(p_i, t_{curr} + 20 \text{ ns})$ 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;
```

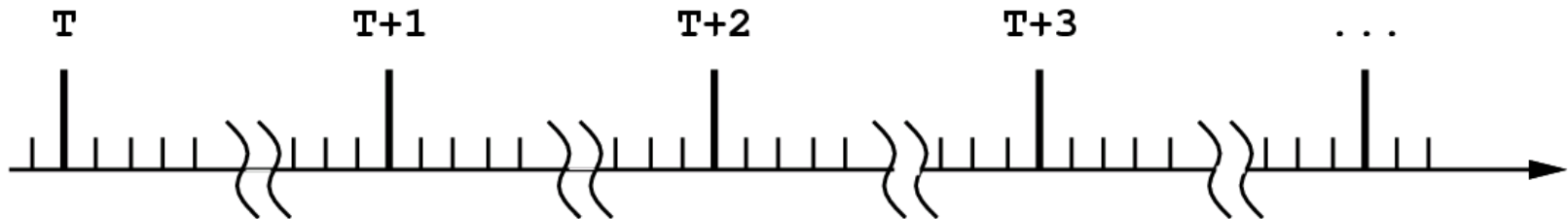
```
    wait for 15 ns;
```

```
end process;
```

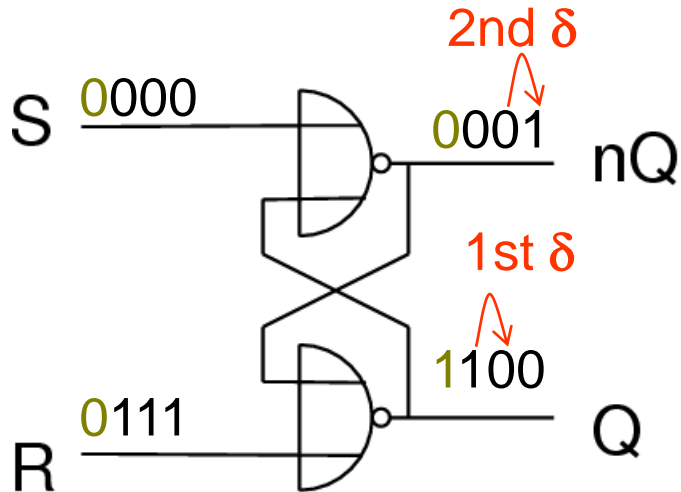
```
end architecture;
```

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 - Simulation of an RS-Flipflop



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

```
entity 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.

„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
- 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 is used to compute final result.

Abstraction of electrical signals

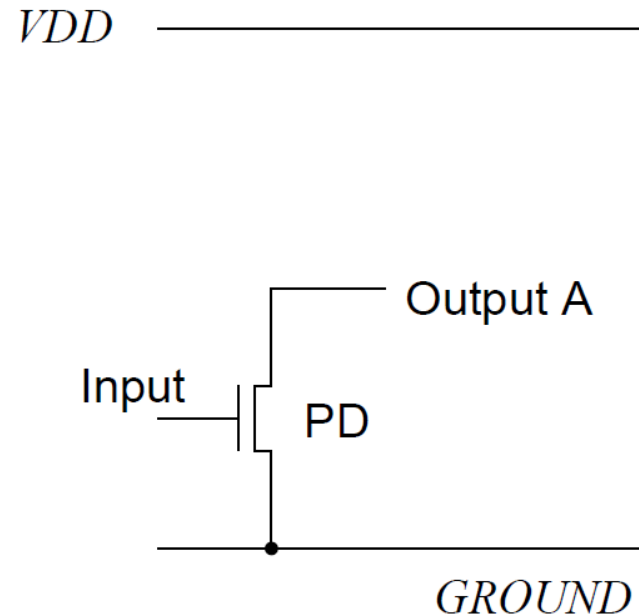
- Complete analog simulation at the circuit level would be time-consuming
 - ☞ We try to use digital values and DE simulation as long as possible
 - ☞ However, using just 2 digital values would be too restrictive
- ☞ We introduce the distinction between:
 - the **logic level** (as an abstraction of the voltage) and
 - the **strength** (as an abstraction of the current drive capability) of a signal.
- The two are encoded in **logic values**.

1 signal strength

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

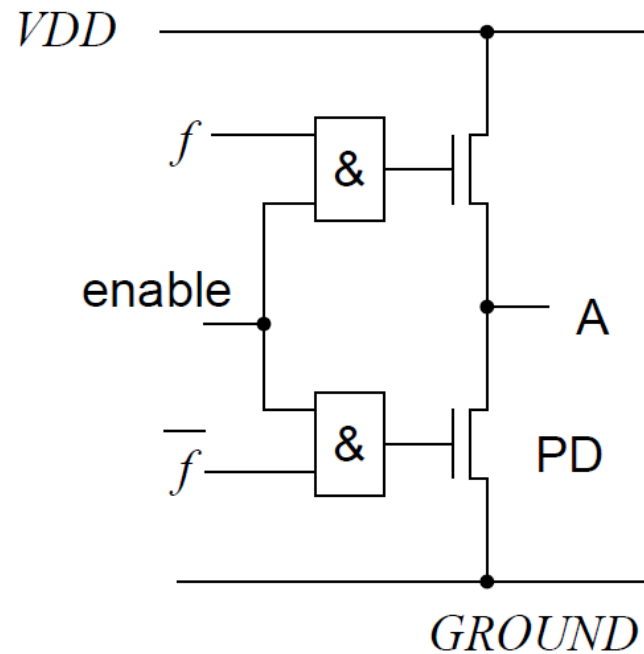
2 signal strengths

- Many subcircuits can effectively disconnect themselves from the rest of the circuit (they provide “high impedance” values to the rest of the circuit).
- Example: subcircuits with open collector



Input = '0' -> A disconnected

TriState circuits

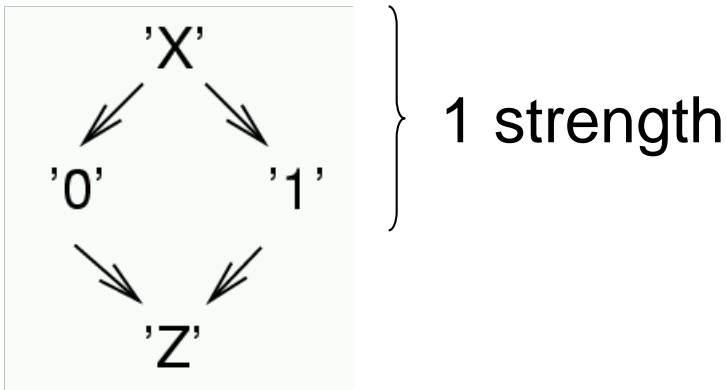


enable = '0' -> A disconnected

☞ We introduce signal value 'Z', meaning “high impedence”

2 signal strengths (cont'ed)

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



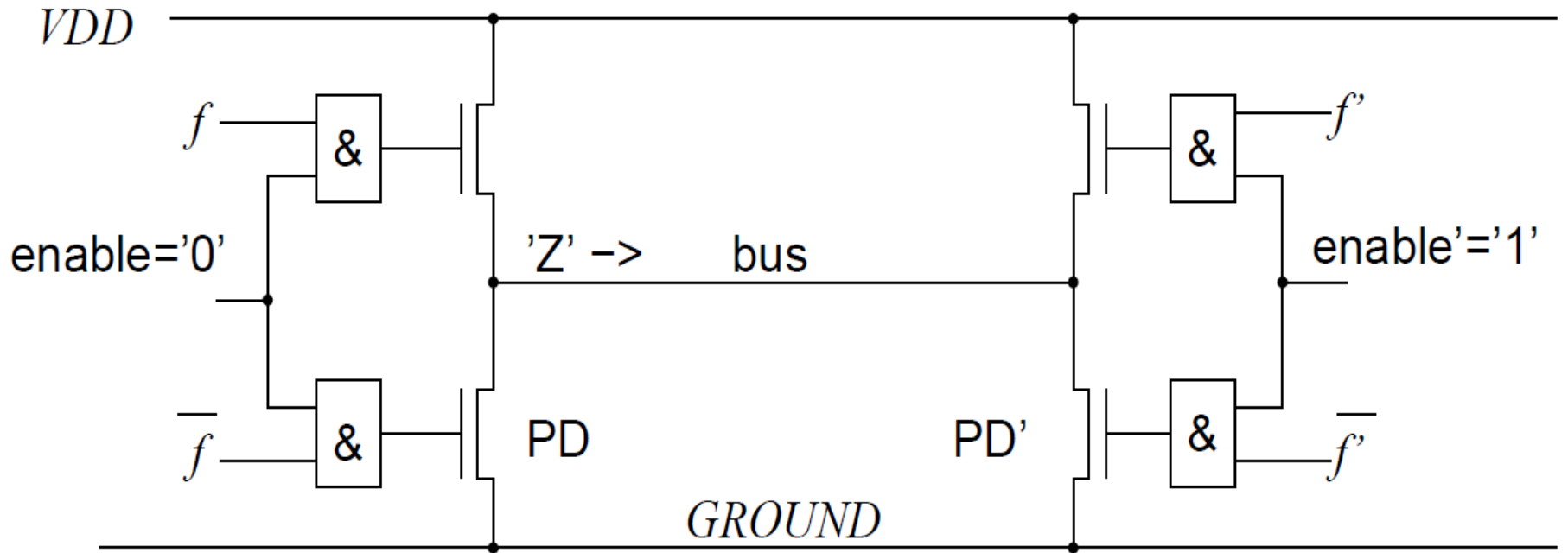
Hasse diagram

According to the partial order in the diagram, $\#$ returns the smallest element at least as large as the *two arguments* (“Sup”).

In order to define $\#('0', '1')$, we introduce 'X', denoting an undefined signal level.

'X' has the same strength as '0' and '1'.

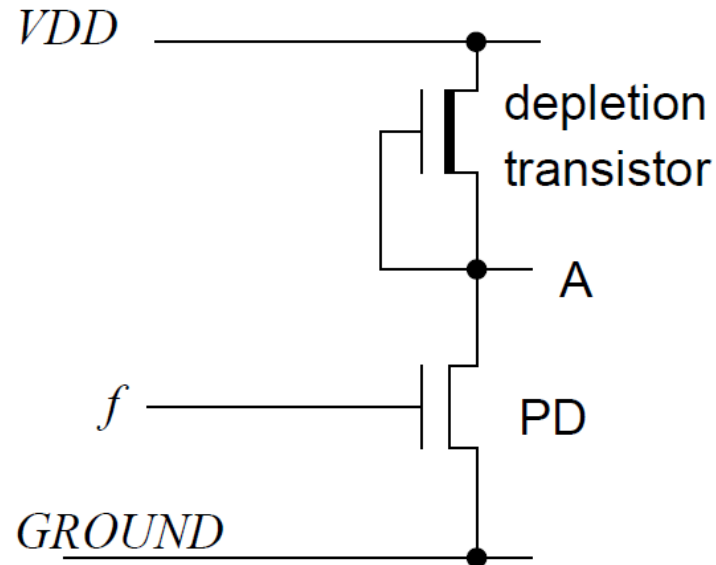
Application example



signal value on bus = #(value from left subcircuit, value from right subcircuit)

#('Z', value from right subcircuit) = value from right subcircuit
 “as if left circuit were not there”.

3 signal strengths





Depletion transistor contributes a weak value to be considered in the #-operation for signal A

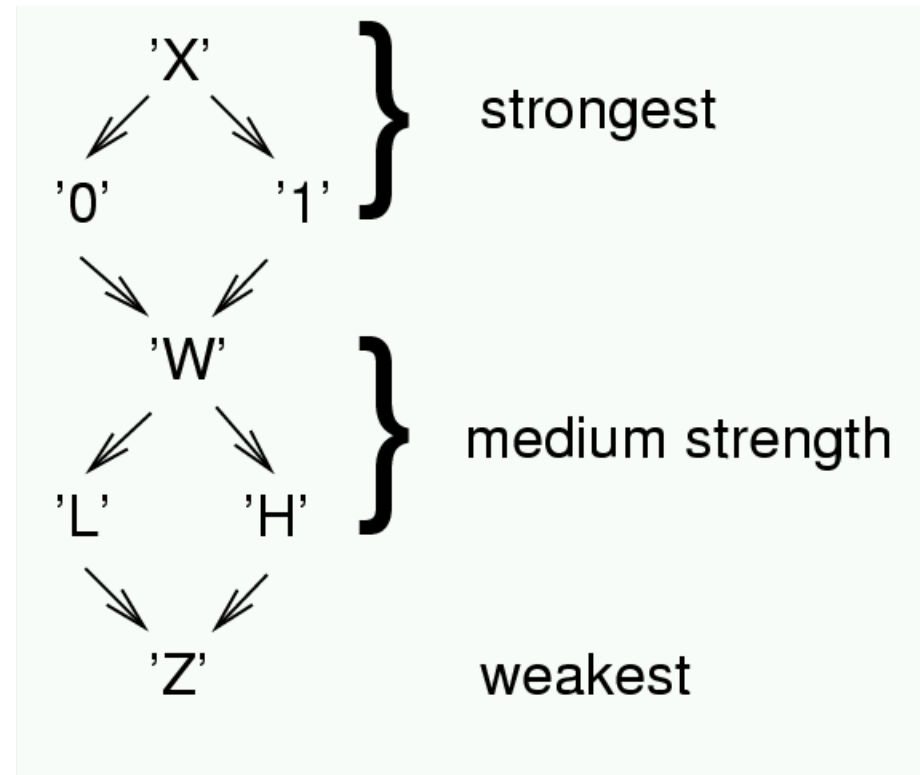
☞ Introduction of 'H',

denoting a weak signal of the same level as '1'.

$\#('H', '0') = '0'$; $\#('H', 'Z') = 'H'$

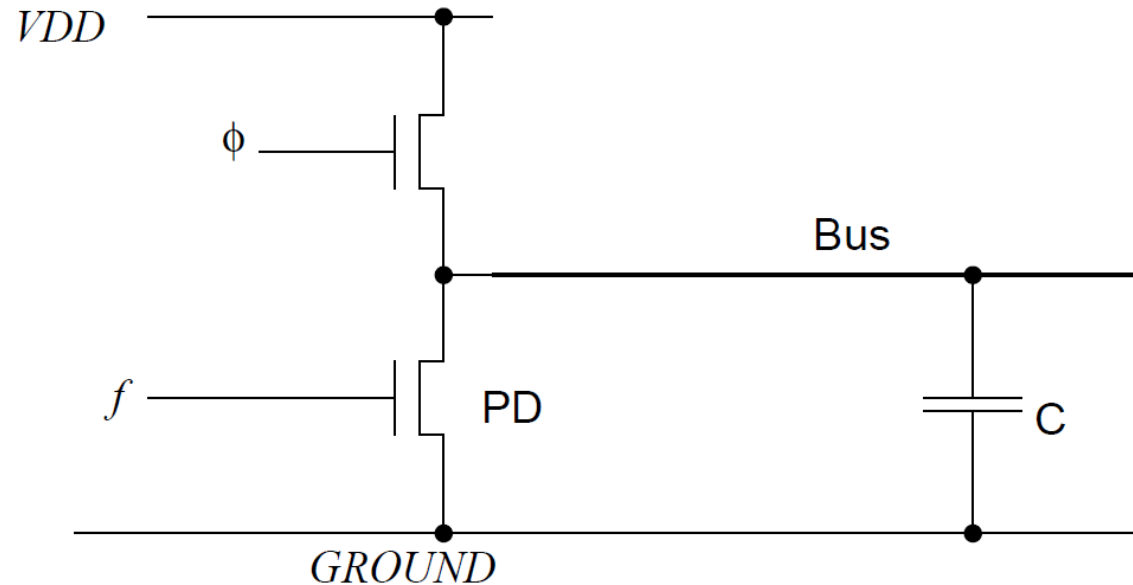
3 signal strengths

- 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': $\#('L', '1') = '1'$; $\#('L', 'Z') = 'L'$;
-  Introduction of 'W', denoting a weak signal of undefined level 'X': $\#('L', 'H') = 'W'$; $\#('L', 'W') = 'W'$;
- # reflected by the partial order shown.



4 signal strengths (1)

- pre-charging:





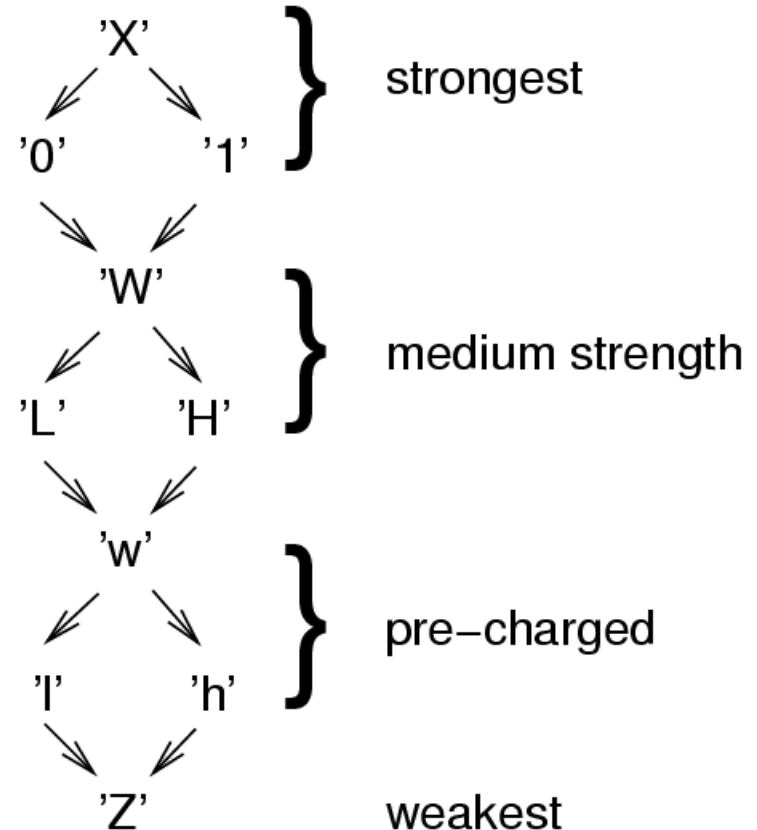
Pre-charged '1'-levels weaker than any of the values considered so far, except 'Z'.

☞ Introduction of 'h', denoting a very weak signal of the same level as '1'.

$\#('h', '0') = '0'$; $\#('h', 'Z') = 'h'$

4 signal strengths (2)

- There may also be weak signals of the same level as '0'
-  Introduction of 'l', denoting a very weak signal of the same level as '0': $\#('l', '0') = '0'$; $\#('l', 'Z') = 'l'$;
-  Introduction of 'w', denoting a very weak signal of the same level as 'W': $\#('l', 'h') = 'w'$; $\#('h', 'w') = 'w'$; ...
- # reflected by the partial order shown.



IEEE 1164

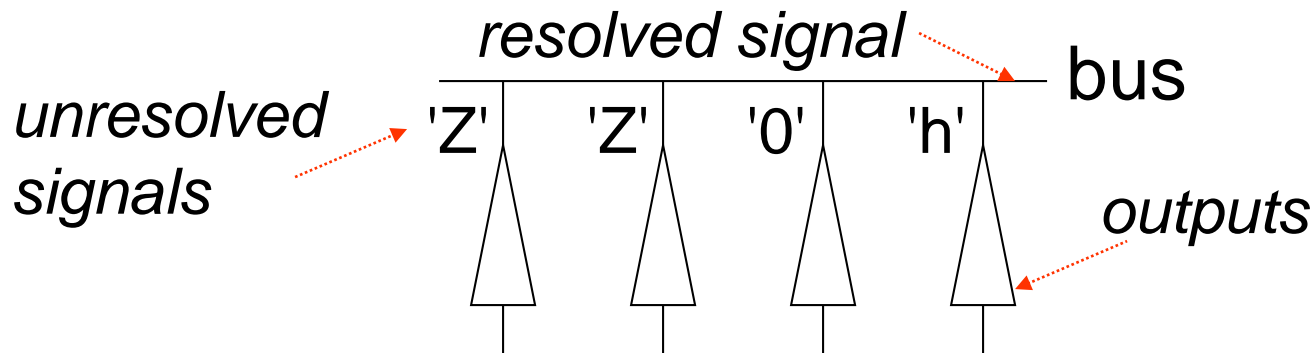
- VHDL allows user-defined value sets.
- Each model could use different value sets (unpractical)
- Definition of standard value set according to standard IEEE 1164:

{'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.

Outputs tied together

In hardware, connected outputs can be used:



Modeling in VHDL: resolution functions

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

```
subtype std_logic is resolved std_ulogic;
```

Resolution function for IEEE 1164

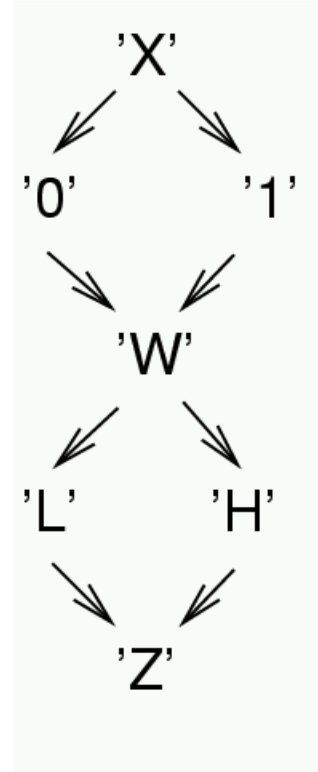
```
type std_ulogic_vector is array(natural range<>)of std_ulogic;  
  
function resolved (s:std_ulogic_vector) return std_logic is  
  variable result: std_ulogic:='Z'; --weakest value is default  
  begin  
    if (s'length=1) then return s(s'low) --no resolution  
    else for i in s'range loop  
      result:=resolution_table(result,s(i))  
    end loop  
    end if;  
    return result;  
end resolved;
```

Resolution function for IEEE 1164

```

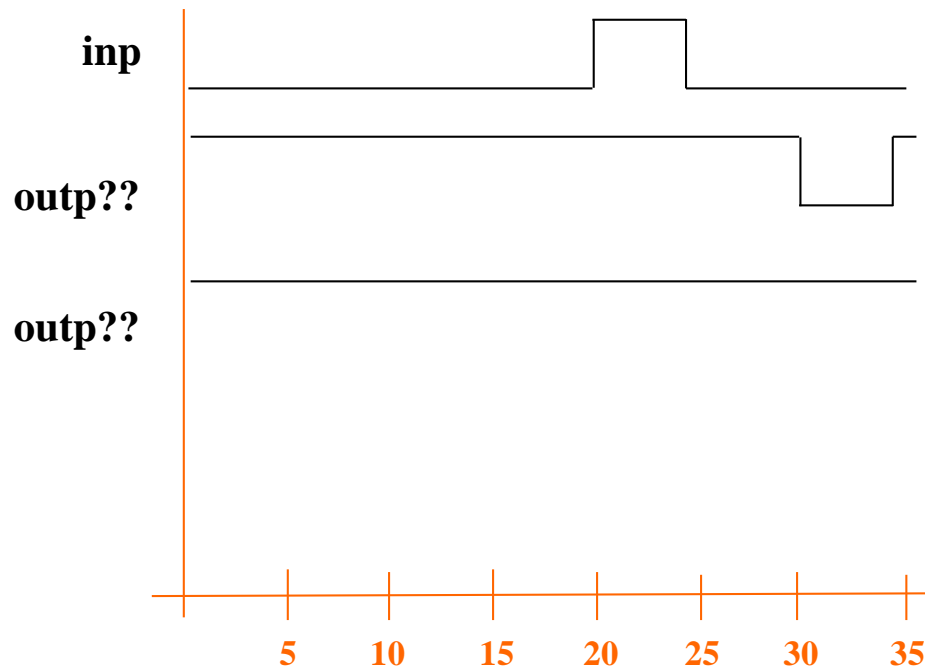
constant resolution_table : stdlogic_table := (
--U  X  0  1  Z  W    L  H  -
('U', 'U', 'U', 'U', 'U', 'U',  'U', 'U', 'U'),  --| U |
('U', 'X', 'X', 'X', 'X', 'X',  'X', 'X', 'X'),  --| X |
('U', 'X', '0', 'X', '0', '0',  '0', '0', 'X'),  --| 0 |
('U', 'X', 'X', '1', '1', '1',  '1', '1', 'X'),  --|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

```



Inertial and transport delay model

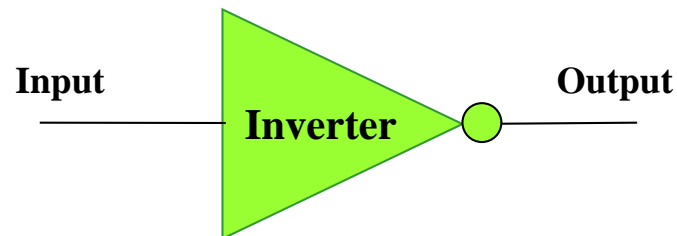
- Example for signal assignment:
`outp <= not inp after 10 ns;`



Inertial and transport delay model

Two delay models in VHDL:

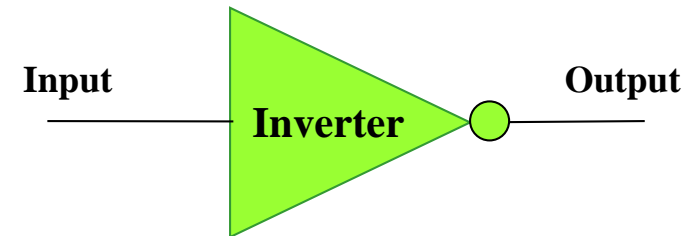
- **Inertial delay** („träge Verzögerung“)
- **Transport delay** („nichtträge Verzögerung“)



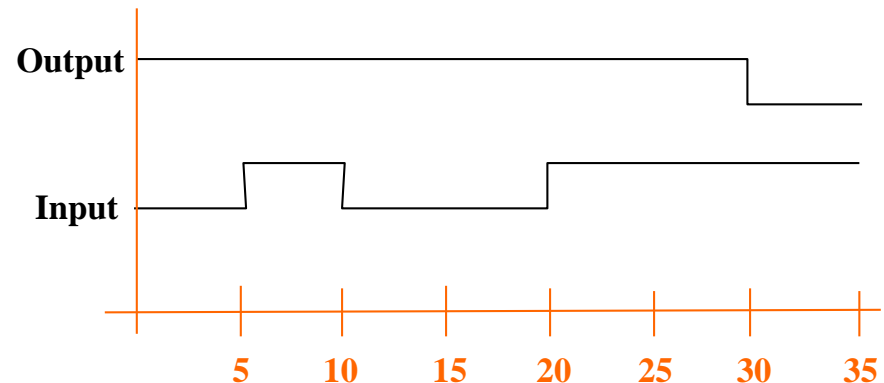
- Inertial delay model is motivated by the fact that physical gates absorb short pulses (spikes) at their inputs (due to internal capacities)

Inertial delay model

- ... is the **default** model
- Absorbs pulses at the inputs which are shorter than the delay specified for the gate / operation

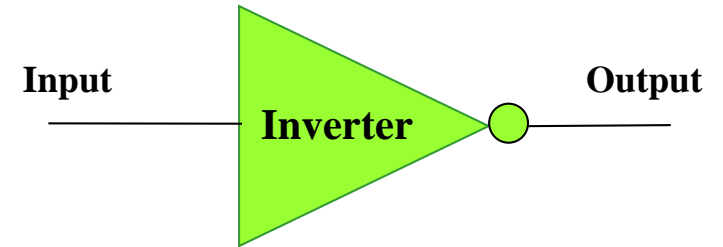


-- **INERTIAL** is the default
Output <= NOT input AFTER 10 ns;

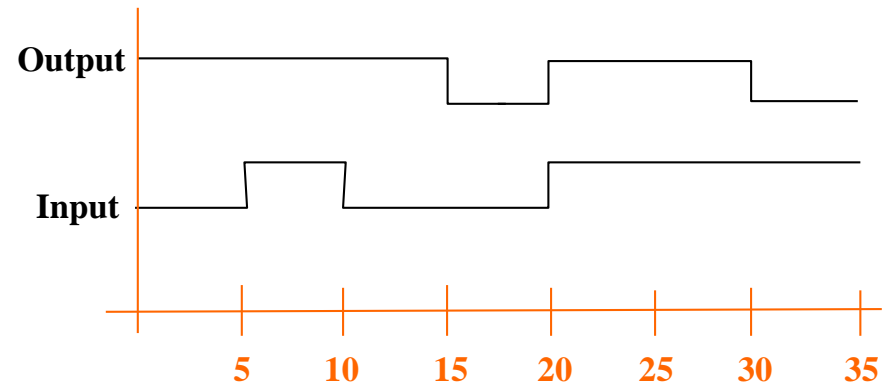


Transport delay model

- Transmits all pulses at the inputs ideally



-- TRANSPORT must be specified
Output <= TRANSPORT NOT input AFTER 10 ns;



Inertial and transport delay model

```
entity DELAY is
end DELAY;
```

```
architecture RTL of DELAY is
```

```
  signal A, B, X, Y: bit;
```

```
begin
```

```
  p0: process (A, B)
```

```
  begin
```

```
    Y <= A nand B after 10 ns;
```

```
    X <= transport A nand B after 10 ns;
```

```
  end process;
```

```
  p1: process
```

```
  begin
```

```
    A <= '0', '1' after 20 ns, '0'
```

```
      after 40 ns, '1' after 60 ns;
```

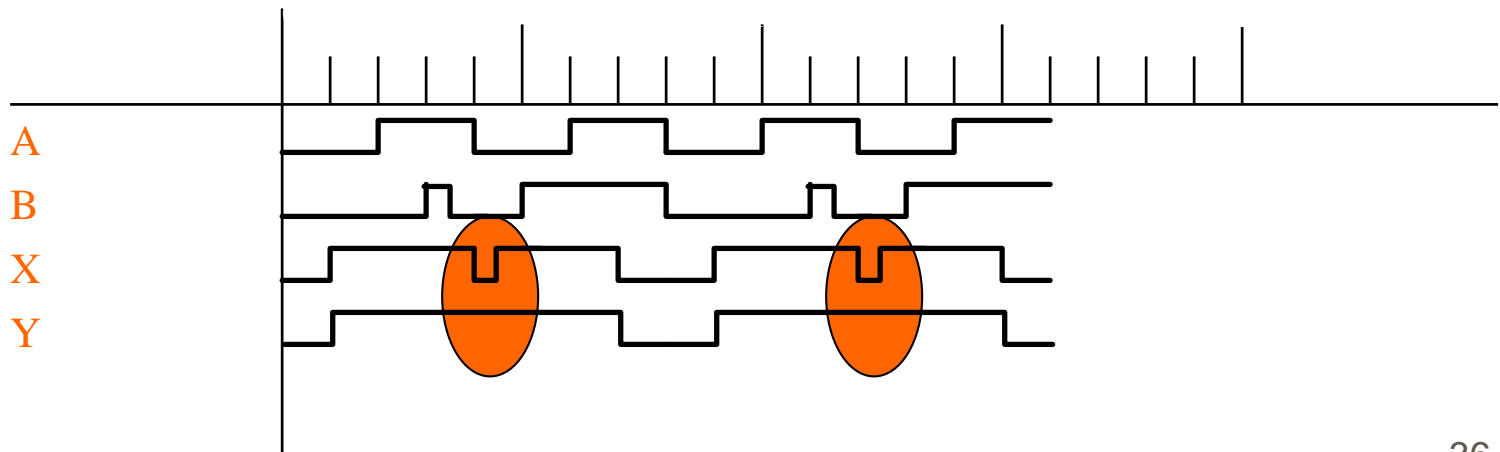
```
    B <= '0', '1' after 30 ns, '0'
```

```
      after 35 ns, '1' after 50 ns;
```

```
    wait for 80 ns;
```

```
  end process
```

```
end RTL;
```



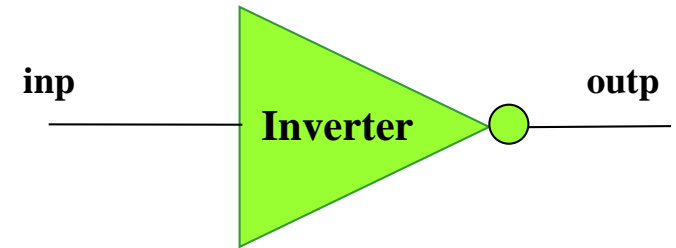
Semantics of transport delay model

Signal assignments change transaction list.

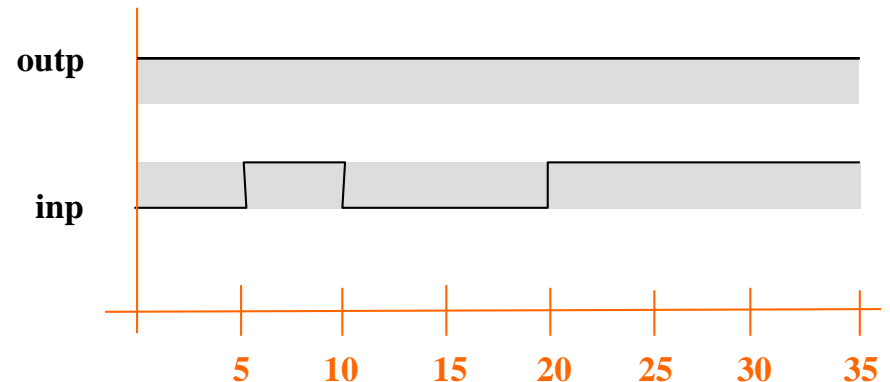
- Before transaction (s, t_1, v_1) is inserted into transaction list, all transactions in the transaction list (s, t_2, v_2) with $t_2 \geq t_1$ are removed from transaction list.

Example for transport delay model

```
inv : process(inp)
begin
  if inp='1' then
    outp <= transport `0` after 20 ns;
  elsif inp='0' then
    outp <= transport `1` after 12.5 ns
  end if;
end process inv;
```



- Transaction list:
 - At 5ns:
(outp, 25ns, `0`)
 - At 10 ns:
(outp, 22.5ns, `1`), (outp, 25ns, `0`)
Remove (outp, 25ns, `0`)!
→ (outp, 22.5ns, `1`)



Semantics of inertial delay model

- Semantics for more general version of inertial delay statement:
 - Inertial delay absorbs pulses at the inputs which are shorter than **the delay specified for the gate / operation**.
 - Key word **reject** permits absorbing only pulses which are shorter than specified delay:
 - Example:
 - `outp <= reject 3 ns inertial not inp after 10 ns;`
 - Only pulses smaller than 3 ns are absorbed.
 - `outp <= reject 10 ns inertial not inp after 10 ns;`
and
`outp <= not inp after 10 ns;`
are equivalent.

Semantics of inertial delay model

- Rule 1 as for transport delay model:
Before transaction (s, t_1, v_1) is inserted into transaction list, all transactions in the transaction list (s, t_2, v_2) with $t_2 \geq t_1$ are removed from transaction list.
- Rule 2 removes also some transactions with times $< t_1$:
 - Suppose the time limit for reject is rt .
 - Transactions for signal s with time stamp in the intervall $(t_1 - rt, t_1)$ are removed.
 - Exception:
If there is in $(t_1 - rt, t_1)$ a subsequence of transactions for s immediately before (s, t_1, v_1) which also assign value v_1 to s , then these transactions are preserved.