technische universität
dortmund

fakultät für informatik
informatik 12

Discrete Event Models

Peter Marwedel
TU Dortmund, Informatik 12
Germany

2012年 11 月 06 日

Models of computation considered in this course

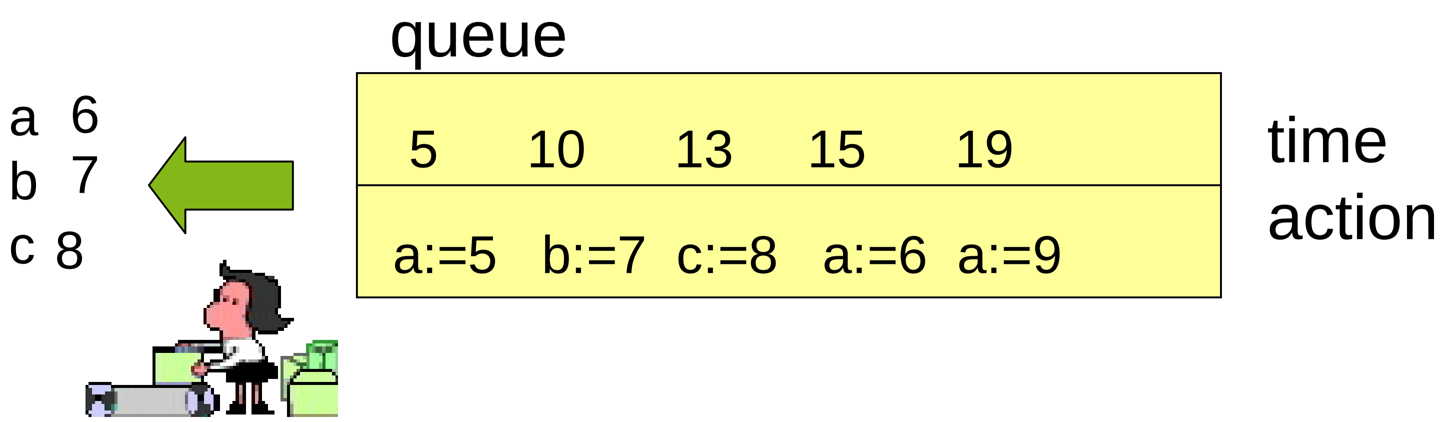
Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases	| (Message) sequence charts	
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorithm (☞ Comp.Archict.)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA	

* Classification is based on implementation of VHDL, Verilog, SystemC with central queue

Discrete event semantics

Basic discrete event (DE) semantics

- Queue of future actions, sorted by time
- Loop:
  - Fetch next entry from queue
  - Perform function as listed in entry
    - May include generation of new entries
- Until termination criterion = true



time	5	10	13	15	19
action	a:=5	b:=7	c:=8	a:=6	a:=9

HDLs using discrete event (DE) semantics

Used in hardware description languages (HDLs):

Description of concurrency is a must for HW description languages!

- Many HW components are operating concurrently
- Typically mapped to “processes“
- These processes communicate via “signals“
- Examples:
  - MIMOLA [Zimmermann/Marwedel], ~1975 …
  - ….
  - VHDL (very prominent example in DE modeling)
    One of the 3 most important HDLs:
    VHDL, Verilog, SystemC

VHDL

VHDL = VHSIC hardware description language

VHSIC = very high speed integrated circuit

1980: Def. started by US Dept. of Defense (DoD) in 1980

1984: first version of the language defined, based on ADA
(which in turn is based on PASCAL)

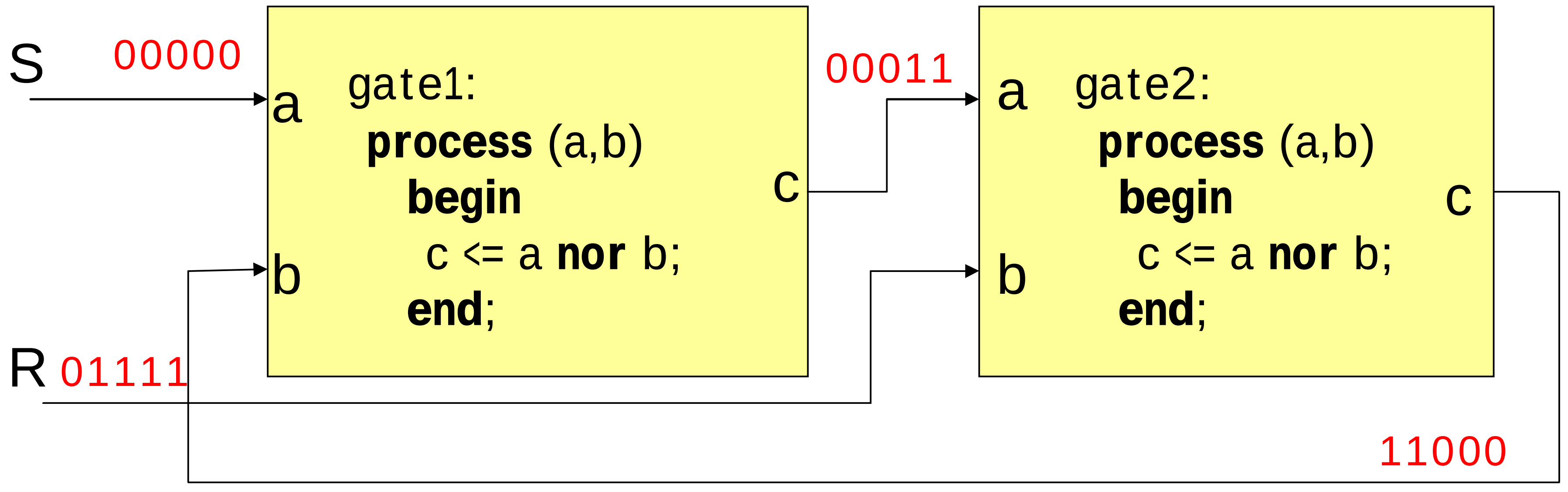
1987: revised version became IEEE standard 1076

1992: revised IEEE standard

1999: VHDL-AMS: includes analog modeling

2006: Major extensions

Simple example (VHDL notation)



```
gate1:
  process (a,b)
    begin
      c <= a nor b;
    end;
```

```
gate2:
  process (a,b)
    begin
      c <= a nor b;
    end;
```

Processes will wait for changes on their input ports.

If they arrive, processes will wake up, compute their code and deposit changes of output signals in the event queue and wait for the next event.

If all processes wait, the next entry will be taken from the event queue.

00	00	00	01	01
01	11	10	10	10

VHDL processes

Delays allowed:

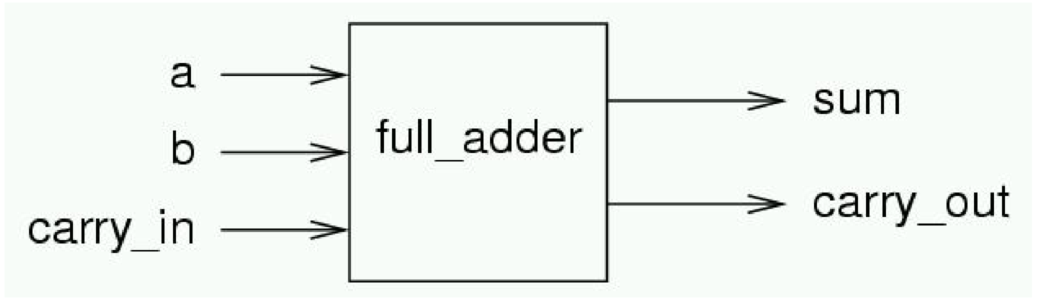
```
process (a,b)
 begin
   c <= a nor b after 10 ns;
 end;
```

Equivalent to

```
process
 begin
  c <= a nor b after 10 ns;
  wait on a,b;
 end;
```

- <=: signal assignment operator
- Each executed signal assignment will result in **adding** entries in the projected waveform, as indicated by the (optional) delay time
- Implicit loop around the code in the body
- Sensitivity lists are a shorthand for a single **wait on**-statement at the end of the process body

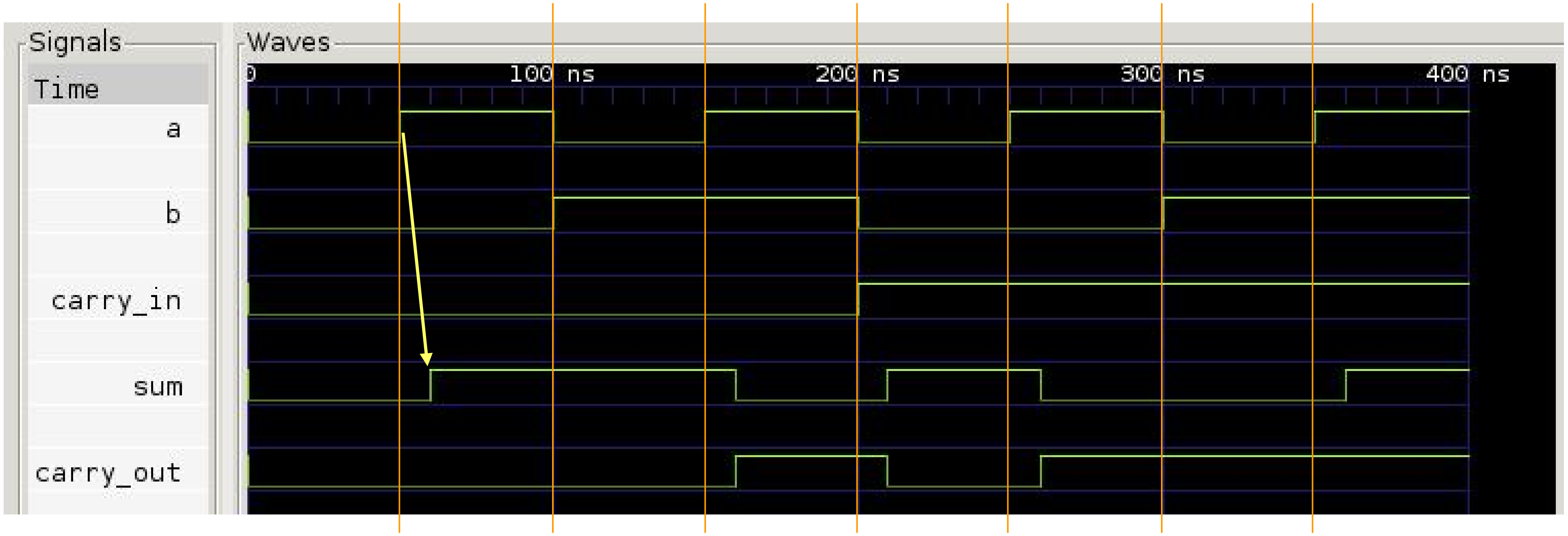
The full adder as an example



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

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;
```

The full adder as an example
- Simulation results -



VHDL semantics: global control

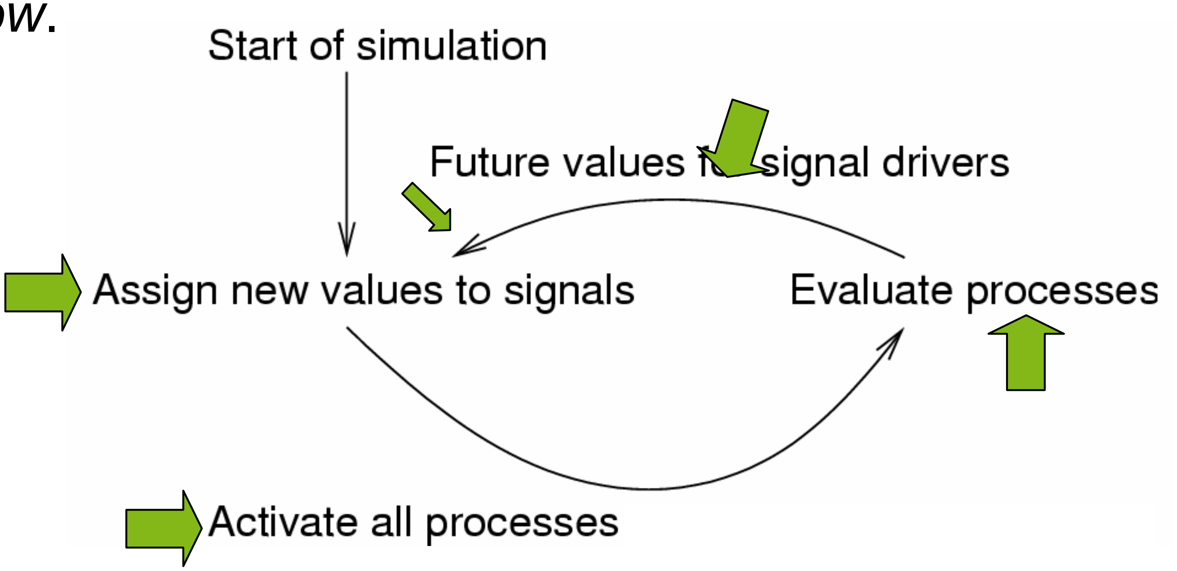
According to the original VHDL standards document:

- The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model.
- Initialization phase executes each process once.

VHDL semantics: initialization

At the beginning of initialization, the current time, T_c is 0 ns.

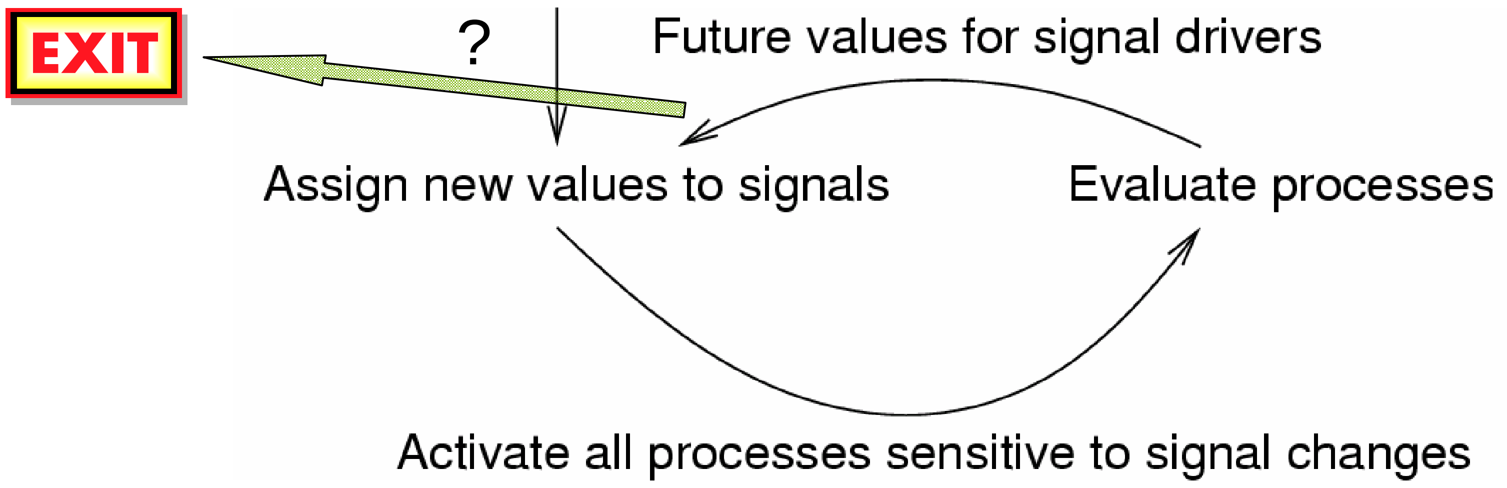
- *The … effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. …*
- *Each ... process … is executed until it suspends.*
- *The time of the next simulation cycle (… in this case … the 1st cycle), T_n is calculated according to the rules of step f of the simulation cycle, below.*


Start of simulation
Future values for signal drivers
Assign new values to signals
Evaluate processes
Activate all processes

VHDL semantics: The simulation cycle (1)

According to the standard, the simulation cycle is as follows:

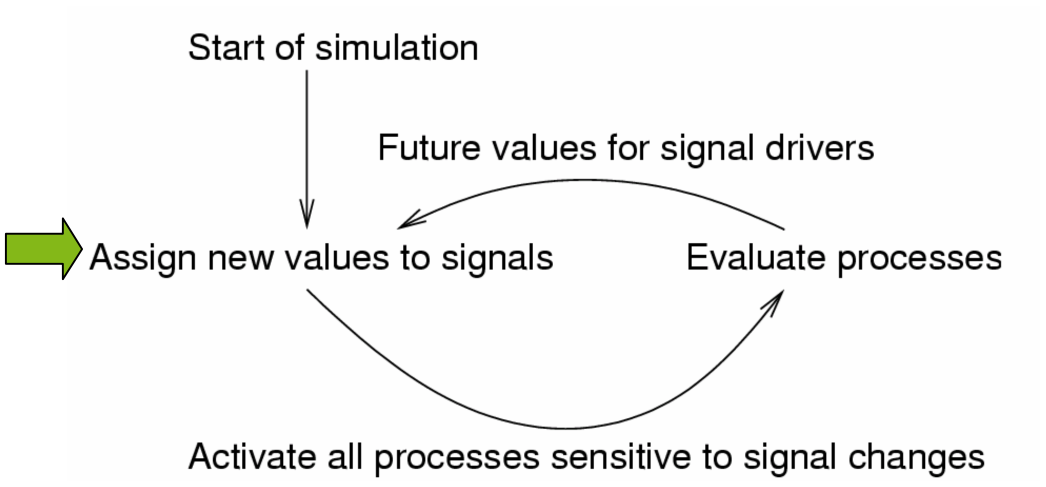
a) Stop if T_n= time'high
and "nothing else is to be done" at T_n.
The current time, T_c is set to T_n.


EXIT
?
Future values for signal drivers
Assign new values to signals
Evaluate processes
Activate all processes sensitive to signal changes

VHDL semantics: The simulation cycle (2)

b) ***Each active explicit signal in the model is updated. (Events may occur as a result.)***
Previously computed entries in the queue are now assigned if their time corresponds to the current time T_c.
New values of signals are not assigned before the next simulation cycle, at the earliest.
Signal value changes result in events ☞ enable the execution of processes that are sensitive to that signal.

c) ..

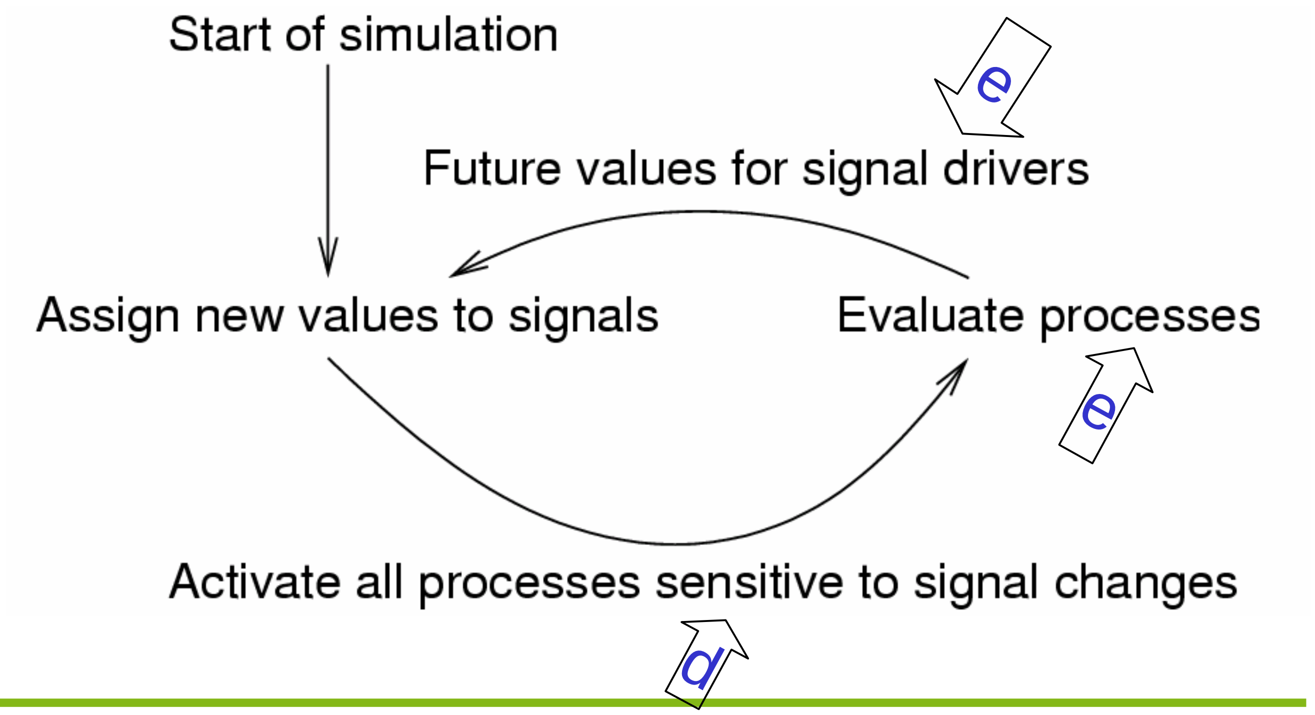

Start of simulation
Future values for signal drivers
Assign new values to signals
Evaluate processes
Activate all processes sensitive to signal changes

VHDL semantics: The simulation cycle (3)

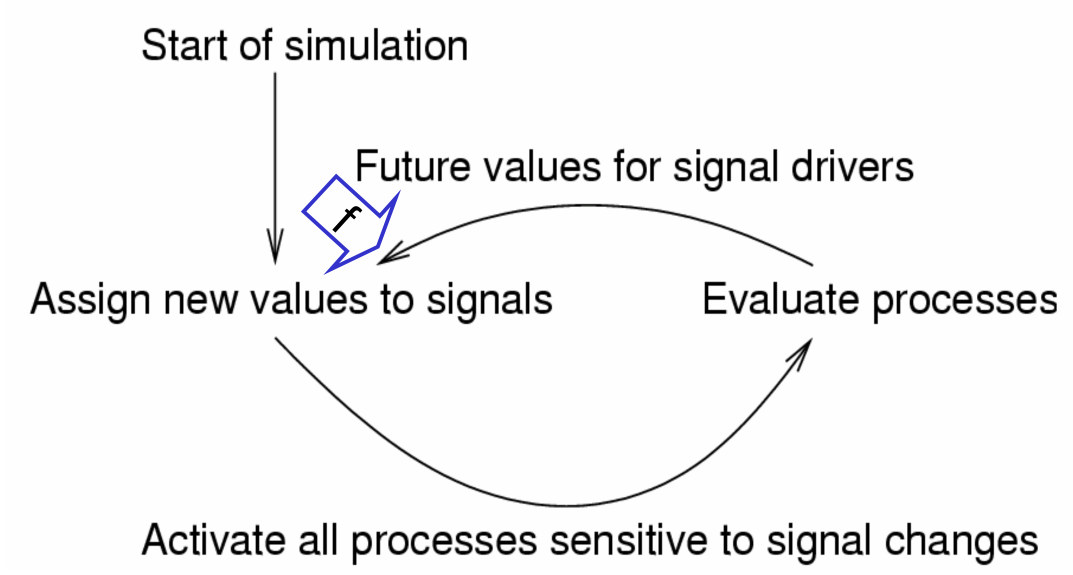
d) ∀ *P* sensitive to *s*: if event on *s* in current cycle: *P* resumes.

e) Each ... process that has resumed in the current simulation cycle is executed until it suspends*.

*Generates future values for signal drivers.


Start of simulation
e
Future values for signal drivers
Assign new values to signals
Evaluate processes
e
Activate all processes sensitive to signal changes
d

VHDL semantics: The simulation cycle (4)


Start of simulation
Future values for signal drivers
f
Assign new values to signals
Evaluate processes
Activate all processes sensitive to signal changes

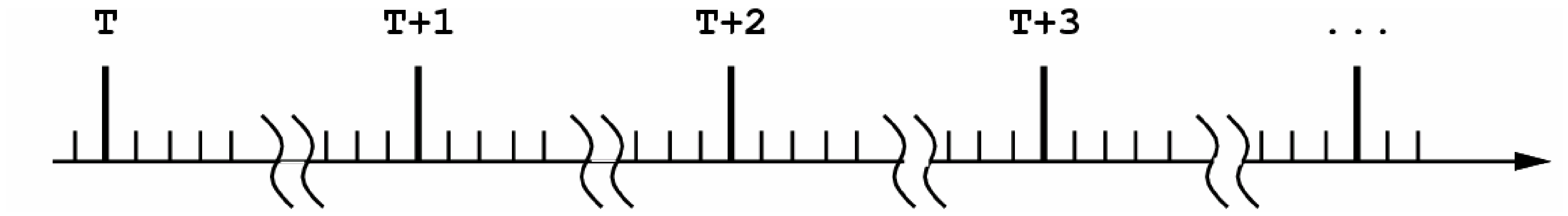
f) Time T_n of the next simulation cycle = earliest of

1. time‘high (end of simulation time).
2. The next time at which a driver becomes active
3. The next time at which a process resumes (determined by **wait for** statements).

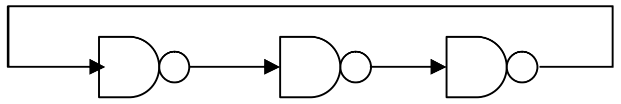
Next simulation cycle (if any) will be a delta cycle if $T_n = T_c$.

δ-simulation cycles

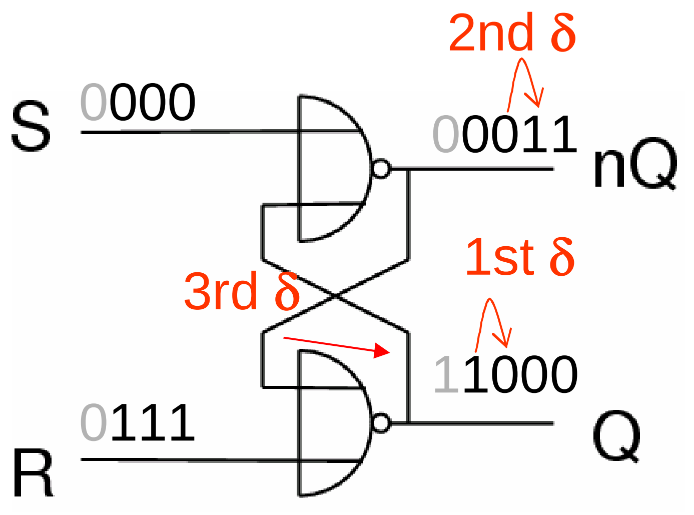
…
Next simulation cycle (if any) will be a delta cycle if $T_n = T_c$.
Delta cycles are generated for delay-less models.
There is an arbitrary number of δ cycles between any 2 physical time instants:



In fact, simulation of delay-less hardware loops might not terminate (don't even advance T_c).



δ-simulation cycles
Simulation of an RS-Flipflop



	0ns	0ns+δ	0ns+2δ	0ns+3δ
R	1	1	1	1
S	0	0	0	0
Q	1	0	0	0
nQ	0	0	1	1

```
gate1:
  process (S,Q)
   begin
     nQ <= S nor Q;
   end;
gate2:
  process (R,nQ)
   begin
     Q <= R nor nQ;
   end;
```

δ cycles reflect the fact that no real gate comes with zero delay.
☞ should delay-less signal assignments be allowed at all?

δ-simulation cycles and determinate simulation semantics

Semantics of

a <= b;

b <= a; ?

Separation into 2 simulation phases results in determinate semantics (☞ StateMate).

technische universität
dortmund

fakultät für informatik
informatik 12

Multi-valued logic and standard IEEE 1164

Peter Marwedel
TU Dortmund,
Informatik 12

2012年 11 月 06 日

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 (as we will see)

How many logic values for modeling ?

Two ('0' and '1') or more?

If real circuits have to be described, some abstraction of the driving 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.

The two are encoded in **logic values**.

☞ CSA (connector, switch, attenuator) - theory [Hayes]

1 signal strength

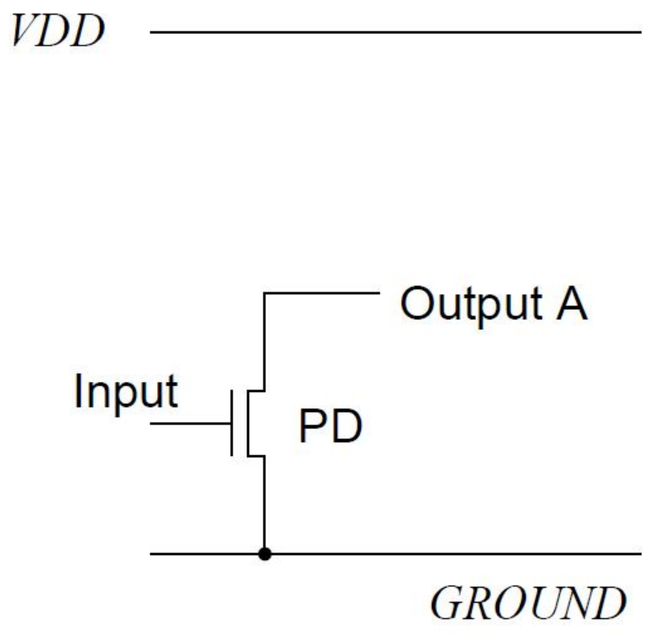
Logic values '0' and '1'.

Both of the same strength.

Encoding false and true, respectively.

2 signal strengths

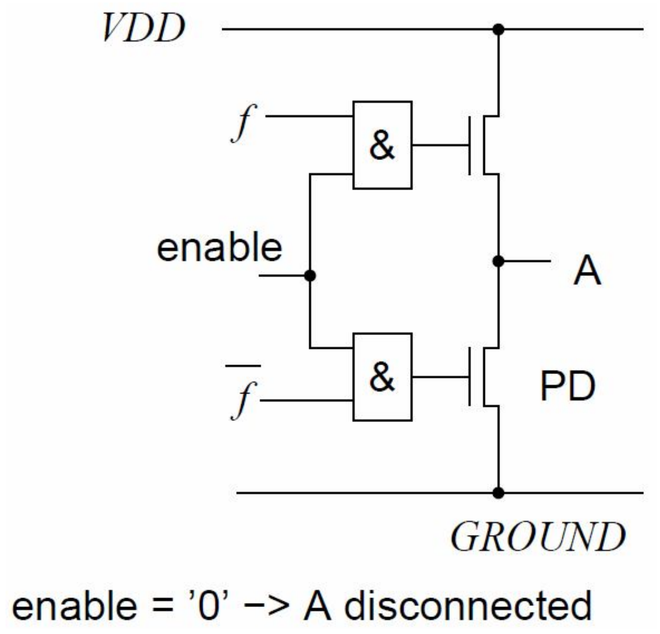
Many subcircuits can effectively disconnect themselves from the rest of the circuit (they provide “high impedance“ values to the rest of the circuit). Example: subcircuits with open collector or tri-state outputs.



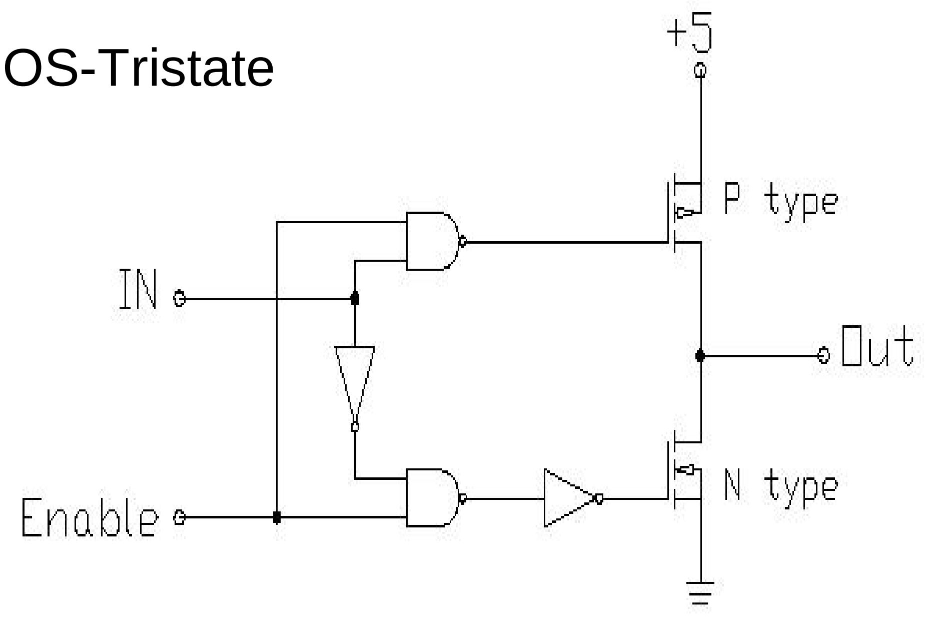
Input = ’0’ –> A disconnected

TriState circuits

nMOS-Tristate



CMOS-Tristate

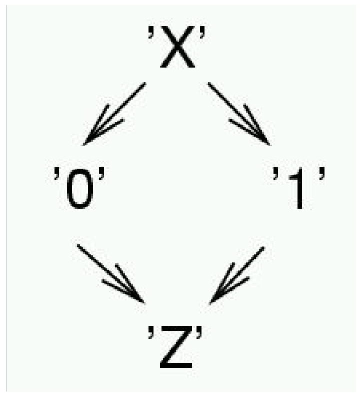


Source: http://www-unix.oit.umass.edu/~phys532/lecture3.pdf

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

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'



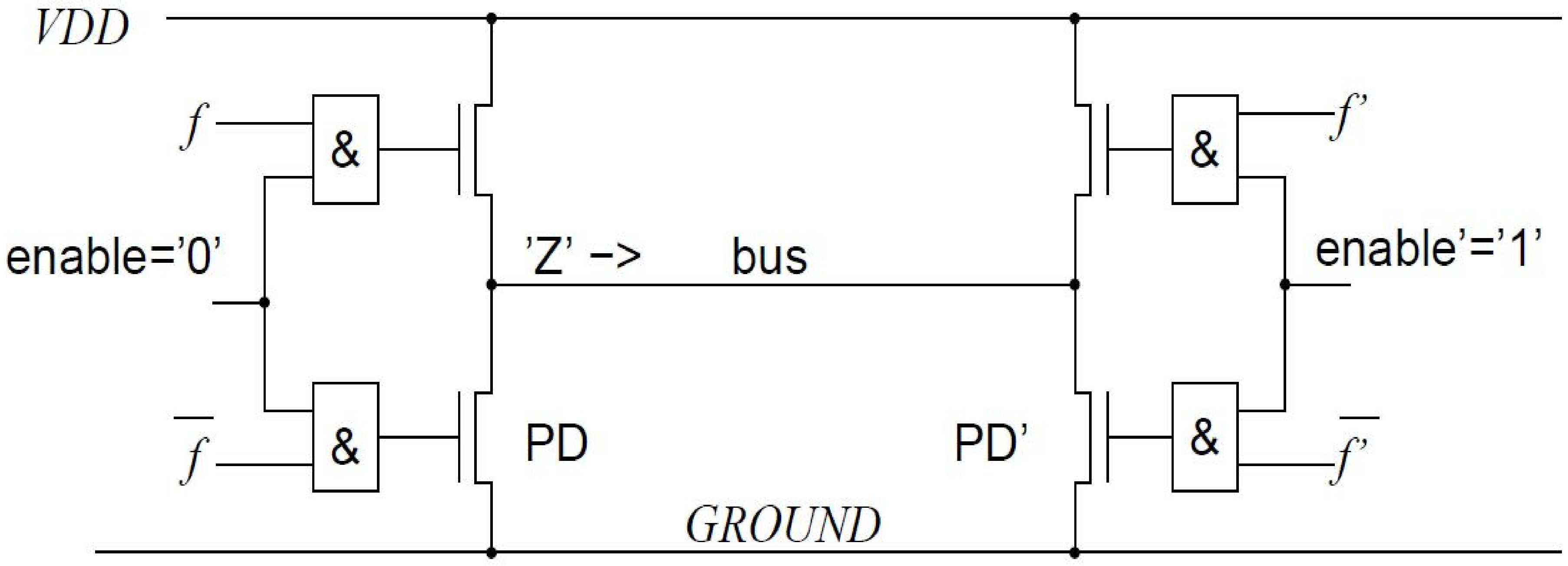
1 strength

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

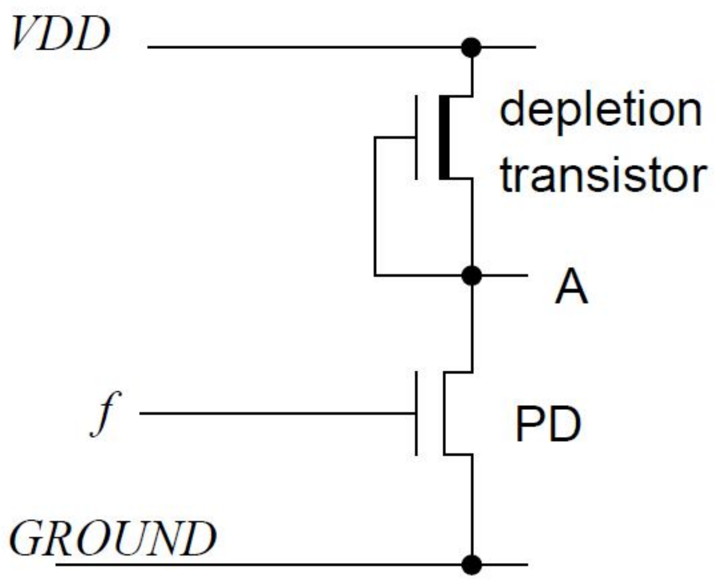


signal value on bus = #(value from left subcircuit, value from right subcircuit)
#('Z', value from right subcircuit)
value from right subcircuit

"as if left circuit were not there".

3 signal strengths

Current values insufficient for describing real circuits:



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'

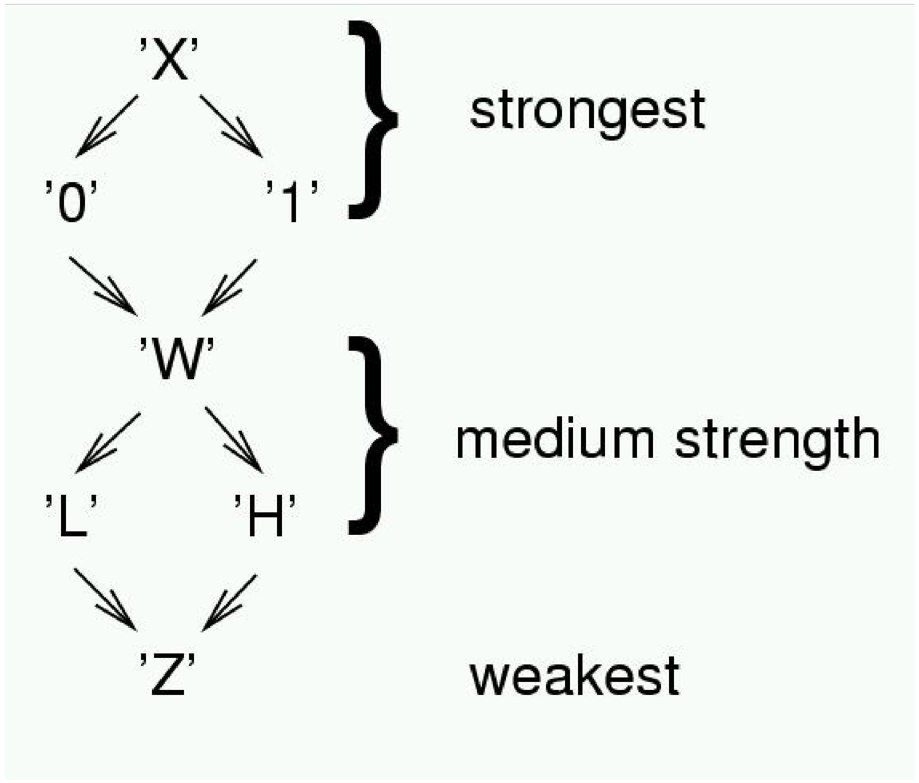
3 signal strengths

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';

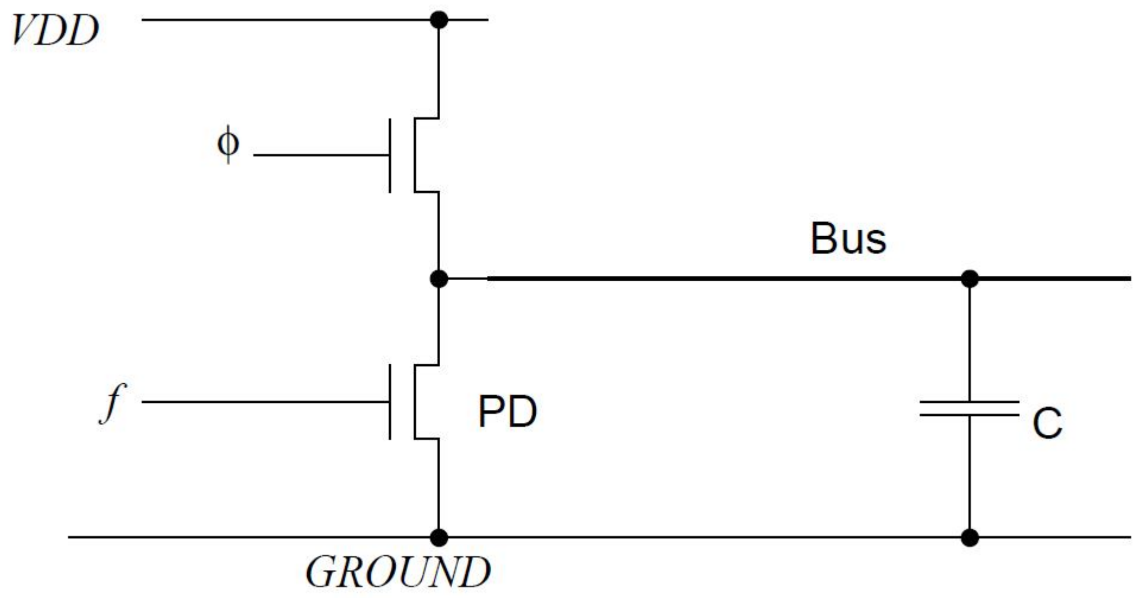
☞ Introduction of 'W', denoting a weak signal of undefined level 'X': #('L', 'H')='W'; #('L','W') = 'W';

# reflected by the partial order shown.



4 signal strengths (1)

Current values insufficient for describing 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'

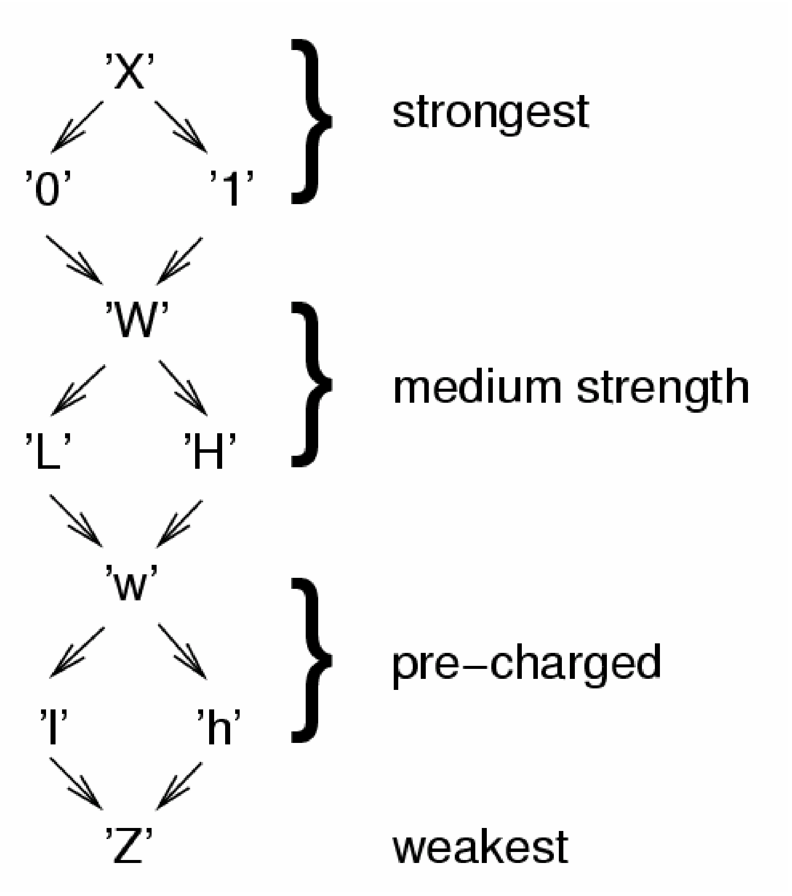
4 signal strengths (2)

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';

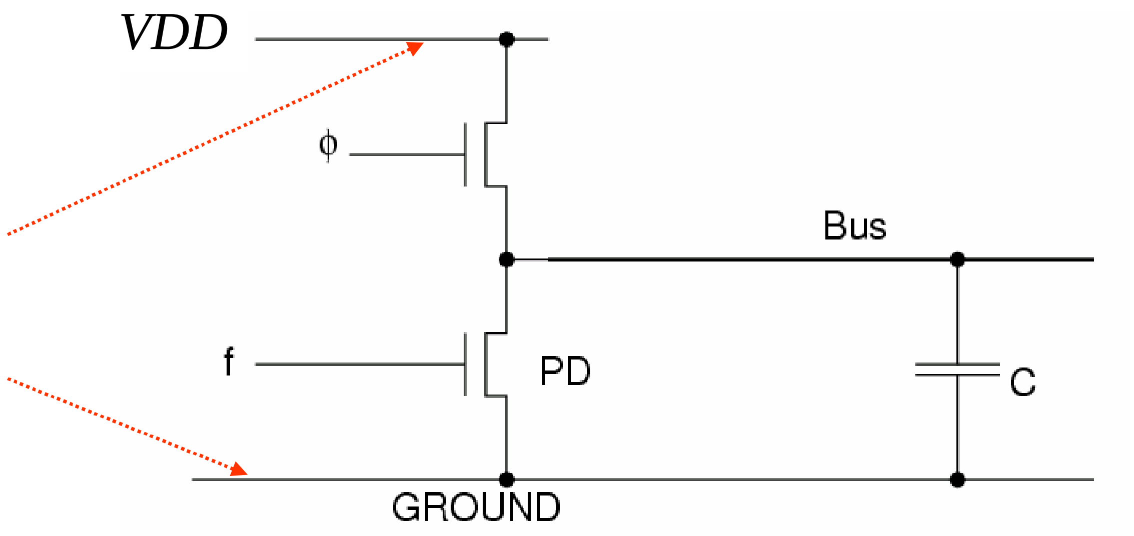
☞ Introduction of 'w', denoting a very weak signal of the same level as 'W': #('l', 'h')='w'; #('h','w') = 'w'; ...

# reflected by the partial order shown.



5 signal strengths

Current values insufficient for describing strength of supply voltage



Supply voltage stronger than any voltage considered so far.

☞ Introduction of 'F0' and 'F1', denoting a very strong signal of the same level as '0 ' and '1'.

☞ Definition of 46-valued logic, also modeling uncertainty (Coelho); initially popular, now hardly used.

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.

☞: Everything said about 7-valued logic applies.

☞: Combination of pre-charging and depletion transistors cannot be described in IEEE 1164.

'U': un-initialized signal; used by simulator to initialize all not explicitly initialized signals.

Input don‘t care

`'-'` denotes **input don't care.**

Suppose:

$f(a,b,c) = a\overline{b} + bc$ except for a=b=c='0' where f is undefined

Then, we could like specifying this in VHDL as

f <= **select** a & b & c

  '1' **when** "10-" -- first term

  '1' **when** "-11" -- second term

  'X' **when** "000" -- 'X' ≙ ('0' or '1') here (output don't care)

  '0' **otherwise;**

Simulator would check if a & b & c = "10-", i.e. if c='-'.

Since c is never assigned a value of '-', this test would always fail. Simulator does not know that **`'-'`** means either **`'1'`** or **`'0'`**, since it does not include any special handling for **`'-'`**, (*at least not for pre-VHDL’2006*).

Function std_match

Special meaning of **'-'** can be used in special function std_match.

if std_match(a&b&c,"10-")
is true for any value of **c**, but this does not enable the use of the compact **select** statement.

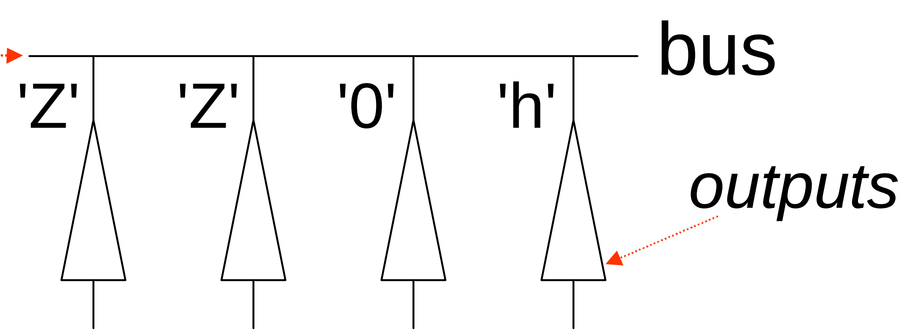
- ☞ The flexibility of VHDL comes at the price of less convenient specifications of Boolean functions.

VHDL’2006 has changed this: **'-'** can be used in the “intended” way in case selectors

Outputs tied together

In hardware, connected outputs can be used:

Resolution function used for assignments to bus, if bus is declared as std_logic.



Modeling in VHDL: resolution functions

type std_ulogic **is** ('U', 'X','0', '1', 'Z', 'W', 'l', 'h', '-');

subtype std_logic **is** resolved std_ulogic;

-- involve function resolved for assignments to std_logic

Resolution function for IEEE 1164

```
type std_ulogic_vector is array(natural range<>)of std_ulogic;

function resolved (s:std_ulogic_vector) return …
 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;
```

Using # (=sup) in resolution functions

```
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')    --| - |
);
```

$$\begin{tikzcd}
 & \text{'X'} \arrow[dl] \arrow[dr] & \\
\text{'0'} \arrow[dr] & & \text{'1'} \arrow[dl] \\
 & \text{'W'} \arrow[dl] \arrow[dr] & \\
\text{'L'} \arrow[dr] & & \text{'H'} \arrow[dl] \\
 & \text{'Z'} &
\end{tikzcd}$$

This table would be difficult to understand without the partial order

Summary

Discrete event models

- Queue of future events, fetch and execute cycle, commonly used in HDLs
- processes model HW concurrency
- signals model communication
- **wait**, sensitivity lists
- the VHDL simulation cycle
    - δ cycles, determinate simulation

Multiple-valued logic

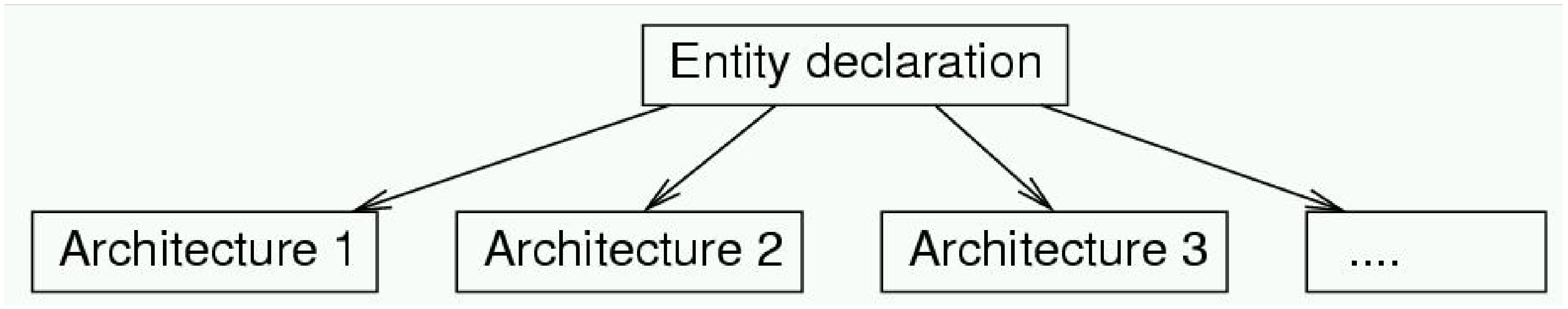
- General CSA approach
- Application to IEEE 1164

More in depth:
Elements of VHDL
+
other DE-based languages

(Optional material)

Entities and architectures

- In VHDL, HW components correspond to "entities"
- Entities comprise processes
- Each design unit is called an **entity**.
- Entities are comprised of **entity declarations**



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**.

The full adder as an example - Architectures -

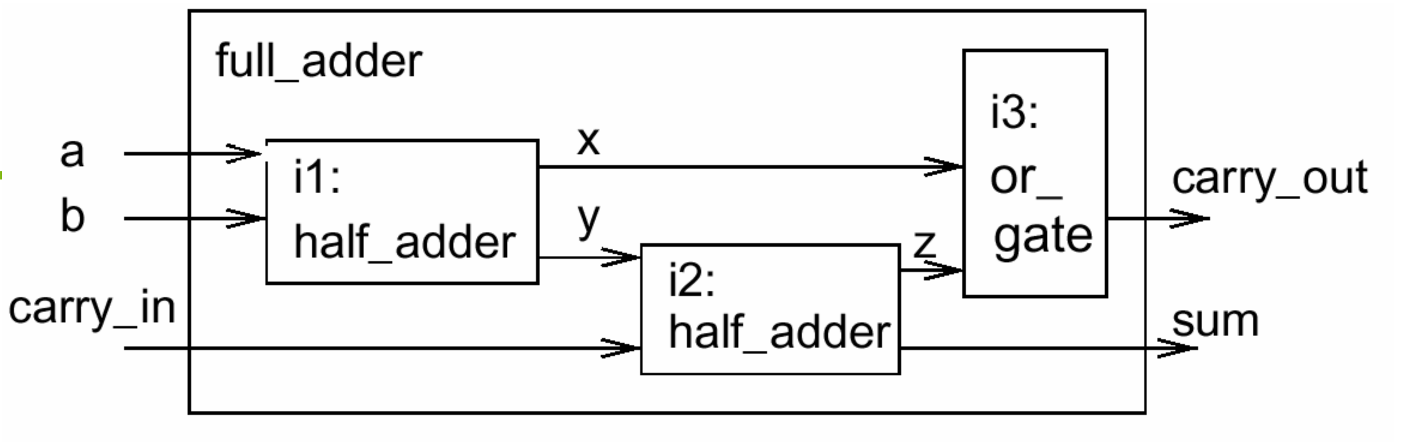
Architecture = Architecture header + architectural bodies

Architectural bodies can be
- **behavioral bodies** or - **structural bodies**.

Bodies not referring to hardware components are called behavioral bodies.

```
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;
```

Structural bodies



```
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;
```

Assignments

2 kinds of assignments:

- <u>**Variable assignments**</u>
  Syntax: *variable* := *expression;*
- <u>**Signal assignments**</u>
  Syntax:
  signal <= *expression*;
  signal <= *expression* **after** *delay*;
  signal <= **transport** expression **after** delay;
  signal <= **reject** time **inertial** expression **after** delay;

Possibly several assignments to 1 signal within 1 process.

For each signal there is one **driver** per process.
Driver stores information about the **future** of signal,
the so-called **projected waveform**.

Adding entries in the projected waveform

- Each executed signal assignment will result in **adding** entries in the projected waveform, as indicated by the delay time, e.g.:

  output <= ‘0’ **after** 5 ns, ‘1’ **after** 10 ns;

1. Transport delay

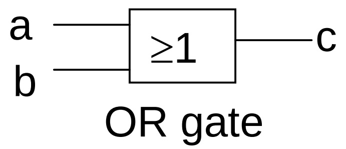
signal <= **transport** expression **after** delay;

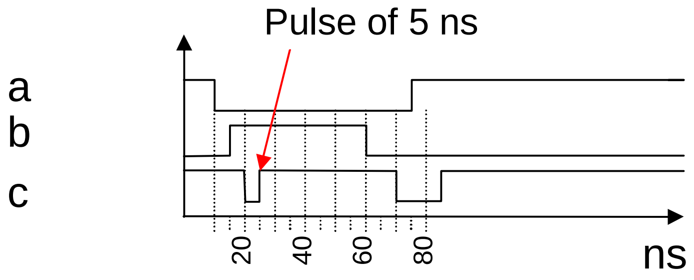
This corresponds to models for *simple wires*

Pulses will be propagated, no matter how short they are.

Example:

c <= **transport** a or b **after** 10 ns;





1. Transport delay (2)

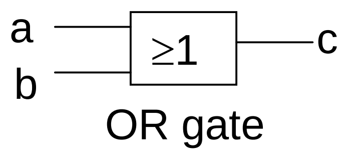
“All old transactions that are projected to occur at or after the time at which the earliest of the new transactions is projected to occur are deleted from the projected output waveform” [VHDL LRM, chap. 8.4]

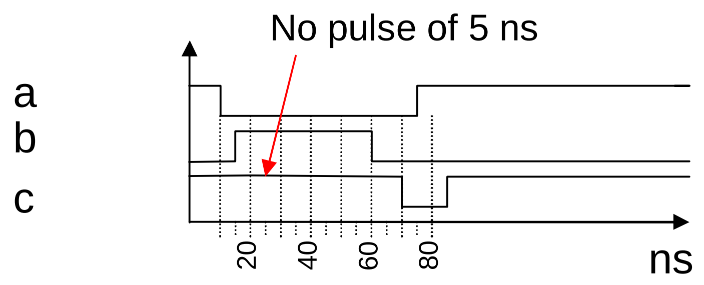
2. Inertial delay

- By default, inertial delay is assumed.
- Suppression of all “spikes” shorter than the delay, resp. shorter than the indicated suppression threshold.
- Inertial delay models the behavior of gates.

Example:

c <= a or b **after** 10 ns;





- Tricky rules for removing events from projected waveform

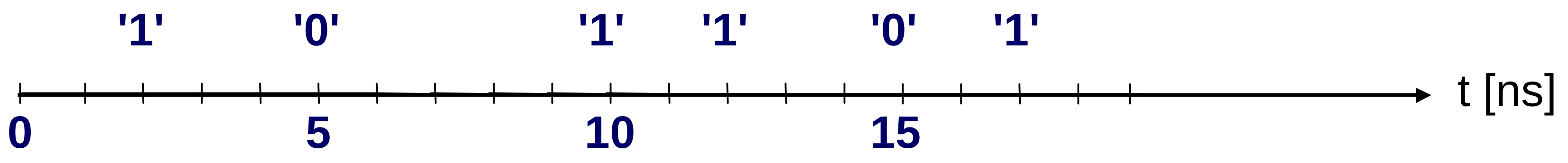
2. Inertial delay (2)

- "*All old transactions that are projected to occur at or after the time at which the earliest of the new transactions is projected to occur are deleted from the projected output waveform*"
- The new transactions are then appended
- "*All of the new transactions are marked*
- *An old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction;*
- *The transaction that determines the current value of the driver is marked;*
- *All unmarked transactions … are deleted from the projected output waveform*"

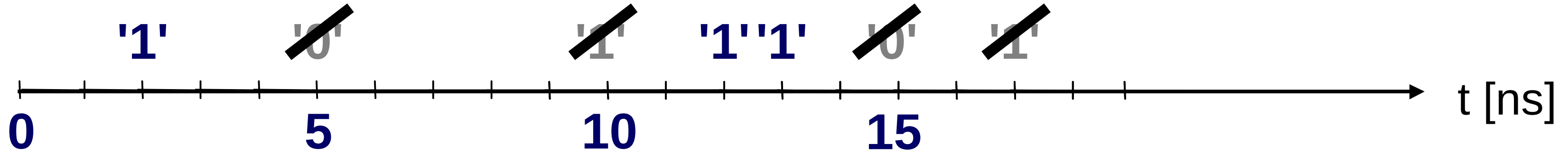
[VHDL LRM, chap. 8.4]

2. Inertial delay (3)

Assume that we are executing a signal assignment
output <= ‘1’ **after** 11 ns at time t=2 ns and the projected waveform is:



- Transactions to occur at or after 13 ns are deleted from the output waveform
- The new transactions are then appended
- All of the new transactions are marked
- Transactions immediately preceding a marked transaction and their value component is the same as that of the marked transaction;
- The transactions that determines the current value of the driver is marked;
- All unmarked transactions … are deleted from the projected output waveform



Spikes are suppressed; not immediately obvious if
a) this suppresses all spikes and b) removes all unnecessary transactions.

Wait-statements

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

- **wait on** *signal list***;**
  - wait until signal 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

VHDL: Evaluation

- Behavioral hierarchy (procedures and functions),
- Structural hierarchy: through structural architectures, but no nested processes,
- No specification of non-functional properties,
- No object-orientation,
- Static number of processes,
- Complicated simulation semantics,
- Too low level for initial specification,
- Good as an intermediate “Esperanto“ or ”assembly” language for hardware generation.

Using C for ES Design: Motivation

- Many standards (e.g. the GSM and MPEG-standards) are published as C programs
  - ☞ Standards have to be translated if special hardware description languages have to be used
- The functionality of many systems is provided by a mix of hardware and software components
  - ☞ Simulations require an interface between hardware and software simulators unless the same language is used for the description of hardware and software

☞ Attempts to describe software and hardware in the same language. Easier said than implemented.
Various C dialects used for hardware description.

Drawbacks of a C/C++ Design Flow

- C/C++ is *not* created to design hardware !
- C/C++ does not support
  - Hardware style communication - Signals, protocols
  - Notion of time - Clocks, time sequenced operations
  - Concurrency - Hardware is concurrent, operates in ||
  - Reactivity - Hardware is reactive, responds to stimuli, interacts with its environment (requires handling of exceptions)
  - Hardware data types - Bit type, bit-vector type, multi-valued logic types, signed and unsigned integer types, fixed-point types
- Missing links to hardware during debugging

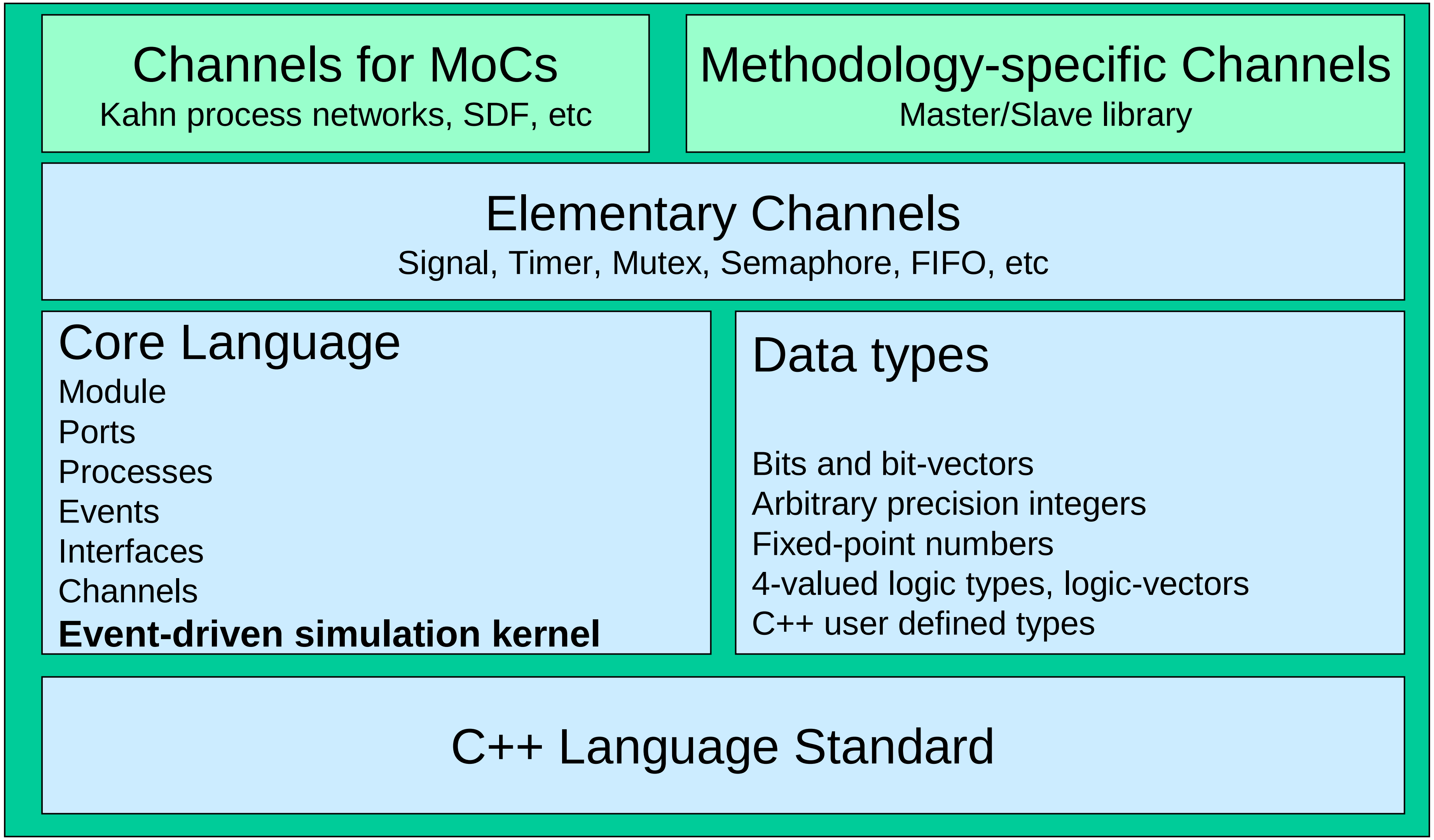
SystemC: Required features

Requirements, solutions for modeling HW in a SW language:

- **C++ class library including required functions.**
- **Concurrency:** via processes, controlled by sensitivity lists* and calls to wait primitives.
- **Time:** Floating point numbers in SystemC 1.0.
  Integer values in SystemC 2.0;
  Includes units such as ps, ns, µs etc*.
- **Support of bit-datatypes**: bitvectors of different lengths; 2- and 4-valued logic; built-in resolution*)
- **Communication:** plug-and-play (pnp) channel model, allowing easy replacement of intellectual property (IP)
- Determinate behavior not guaranteed.

* Good to know VHDL ☺

SystemC language architecture



Transaction-based modeling

Definition: "***Transaction-level modeling*** *(TLM) is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture.*
Communication mechanisms such as buses or FIFOs are modeled as channels, and are presented to modules using SystemC interface classes. Transaction requests take place by calling interface functions of these channel models, which encapsulate low-level details of the information exchange.
At the transaction level, the emphasis is more on the functionality of the data transfers - what data are transferred to and from what locations - and less on their actual implementation, that is, on the actual protocol used for data transfer.
This approach makes it easier for the system-level designer to experiment, for example, with different bus architectures (all supporting a common abstract interface) without having to recode models that interact with any of the buses, provided these models interact with the bus though the common interface."

Grötker et al., 2002

Verilog

HW description language competing with VHDL

Standardized:

- IEEE 1364-1995 (Verilog version 1.0)
- IEEE 1364-2001 (Verilog version 2.0)
- Features similar to VHDL:
  - Designs described as connected entities
  - Bitvectors and time units are supported
- Features that are different:
  - Built-in support for 4-value logic and for logic with 8 strength levels encoded in two bytes per signal.
  - More features for transistor-level descriptions
  - Less flexible than VHDL.
  - More popular in the US (VHDL common in Europe)

SystemVerilog

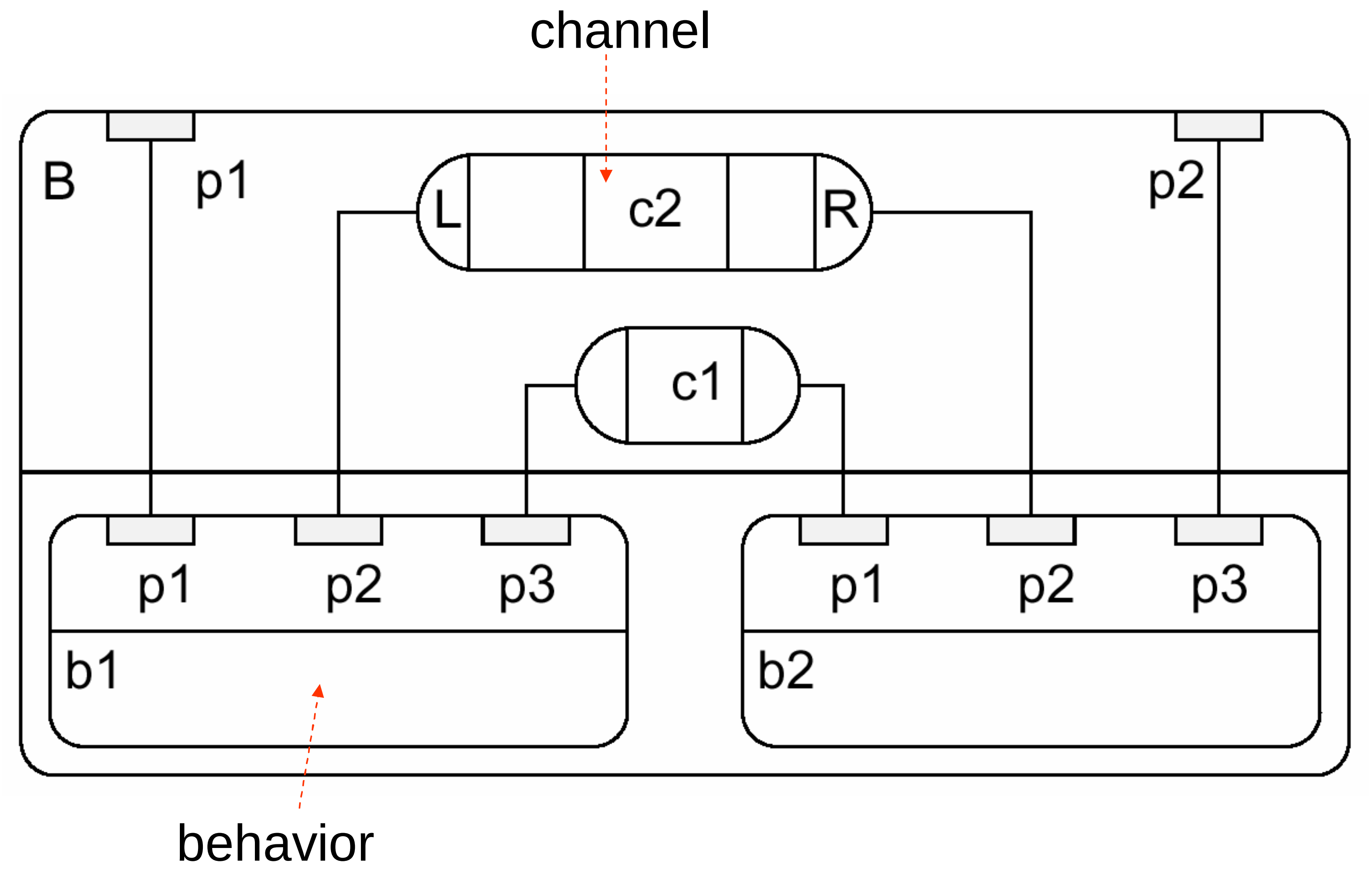
Corresponds to Verilog versions 3.0 and 3.1. Includes:

- Additional language elements for modeling behavior
- C data types such as **`int`**
- Type definition facilities
- Definition of interfaces of HW components as entities
- Mechanism for calling C/C++-functions from Verilog
- Limited mechanism for calling Verilog functions from C.
- Enhanced features for describing the testbench
- Dynamic process creation.
- Interprocess communication and synchronization
- Automatic memory allocation and deallocation.
- Interface for formal verification.

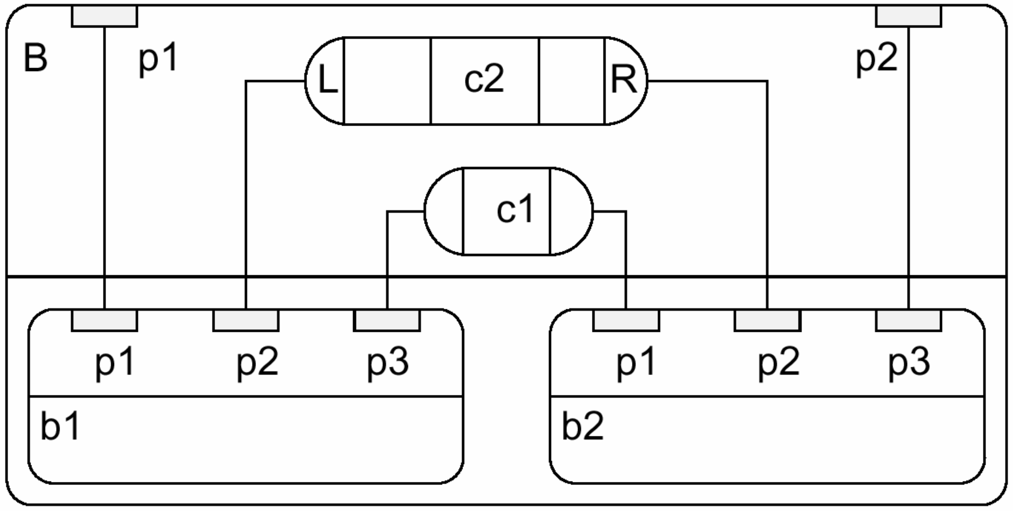
SpecC [Gajski, Dömer et. al. 2000]

- SpecC is based on the clear separation between communication and computation. Enables „*plug-and-play*“ for system components; models systems as hierarchical networks of behaviors communicating through channels.
- Consists of behaviors, channels and interfaces.
- **Behaviors** include ports, locally instantiated components, private variables and functions and a public main function.
- **Channels** encapsulate communication. Include variables and functions, used for the definition of a communication protocol.
- **Interfaces:** linking behaviors and channels.
  Declare communication protocols (*defined* in a channel).

Example

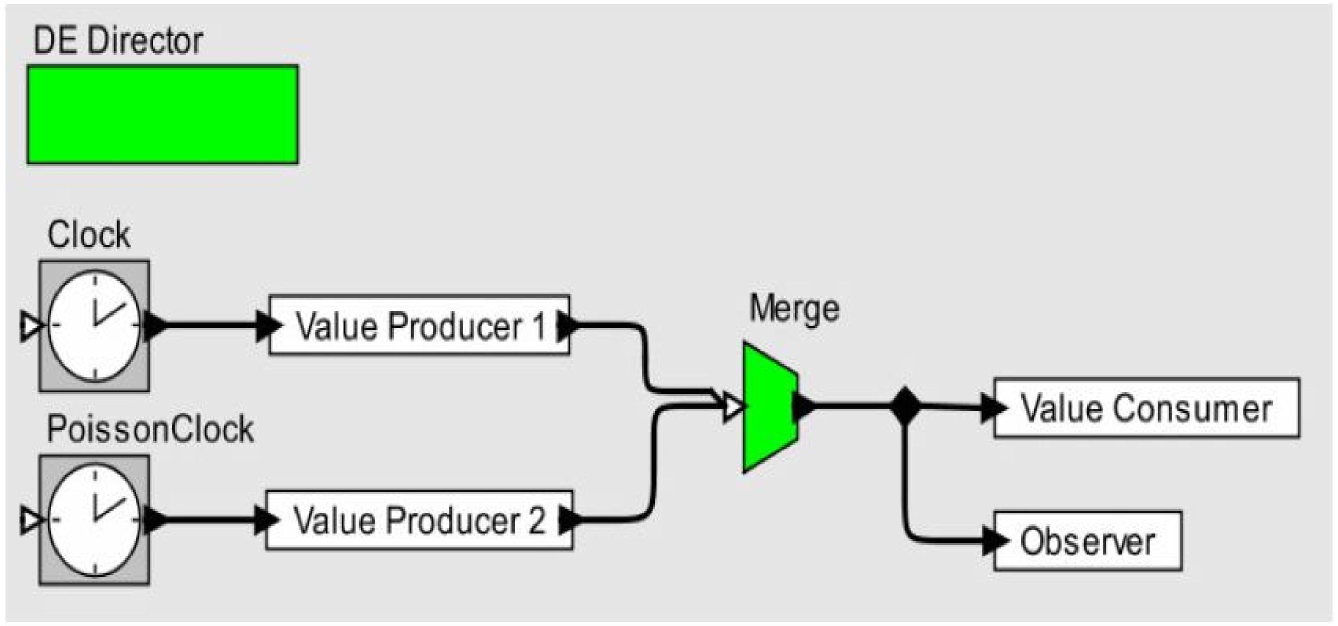


SpecC-Example



```
interface L {void Write(int x);};
interface R {int Read (void);};
channel C implements L,R
 { int Data; bool Valid;
   void Write(int x)  {Data=x; Valid=true;}
   int Read(void) {while (!Valid) waitfor(10); return (Data);}
  };
behavior B1 (in int p1, L p2, in int p3)
  {void main(void)    {/*...*/ p2.Write(p1);}  };
behavior B2 (out int p1, R p2, out int p3)
  {void main(void)   {/*...*/ p3=p2.Read();   }  };
behavior B(in int p1, out int p2)
  { int c1; C c2; B1 b1(p1,c2,c1); B2 b2 (c1,c2,p2);
    void main (void)
    {par {b1.main();b2.main();}}
   };
```

Observer Pattern using Discrete Events



Messages have a (semantic) time, and actors react to messages chronologically. Merge now becomes deterministic.