Discrete Event Models

Jian-Jia Chen
(slides are based on
Peter Marwedel)
TU Dortmund, Informatik 12
Germany

2017年 10 月 25日

Models of computation considered in this course

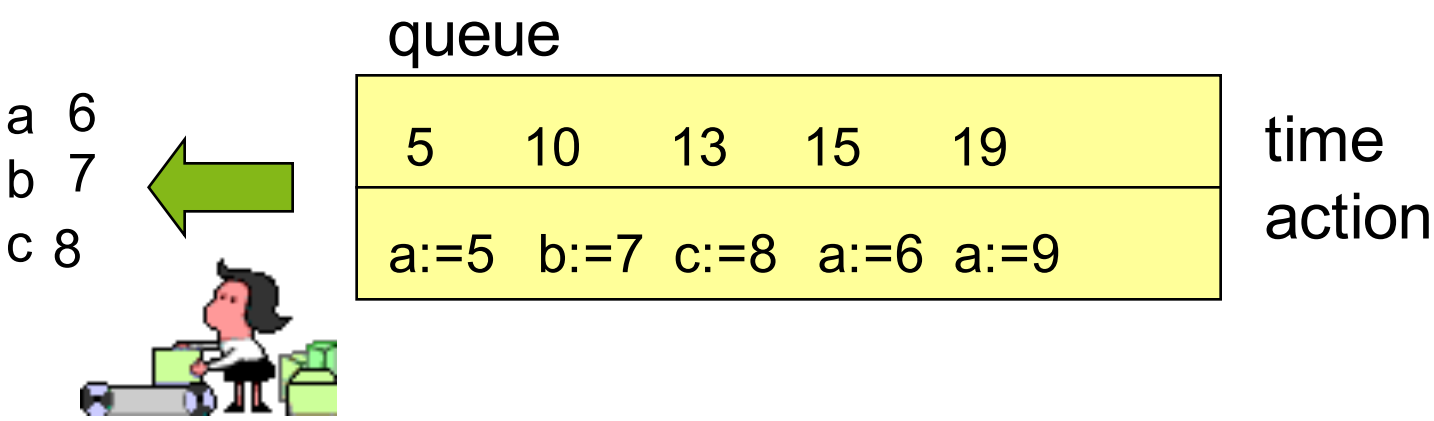
Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous
Undefined components	Plain text, use cases | (Message) sequence charts	
Communicating finite state machines	StateCharts	SDL
Data flow	Scoreboarding + Tomasulo Algorithm (☞ Comp.Archict.)	Kahn networks, SDF
Petri 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



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)

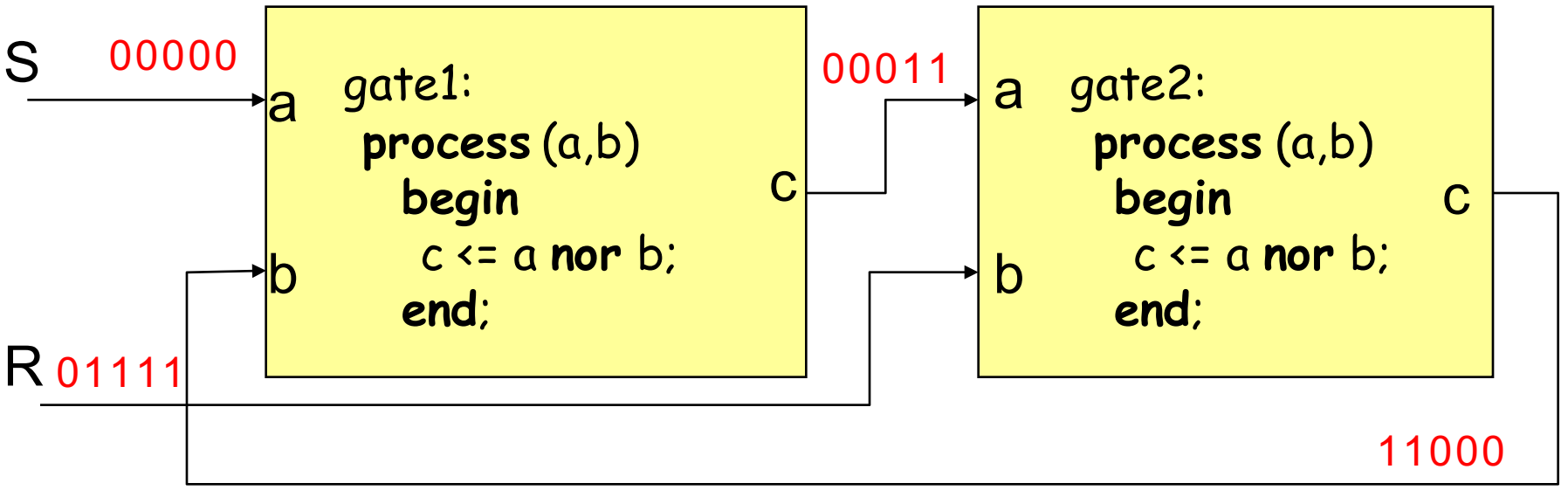
1987: revised version became IEEE standard 1076

1992: revised IEEE standard

1999: VHDL-AMS: includes analog modeling

2006: Major extensions

Simple example (VHDL notation)



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.

VHDL processes

Delays allowed:

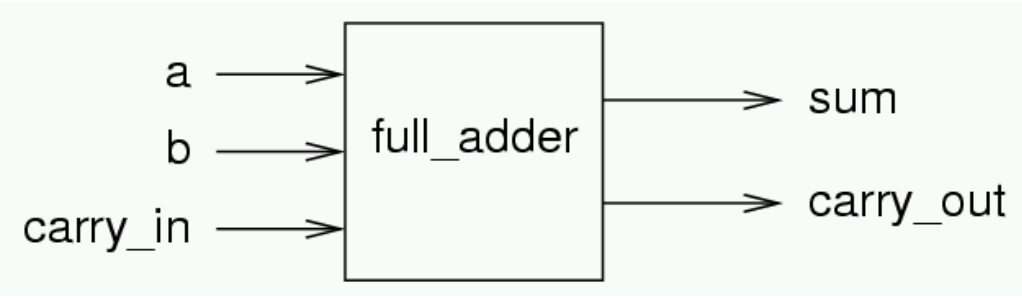
```
process (a,b)
 begin
   c <= a nor b after 10 ns;
 end;
```

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

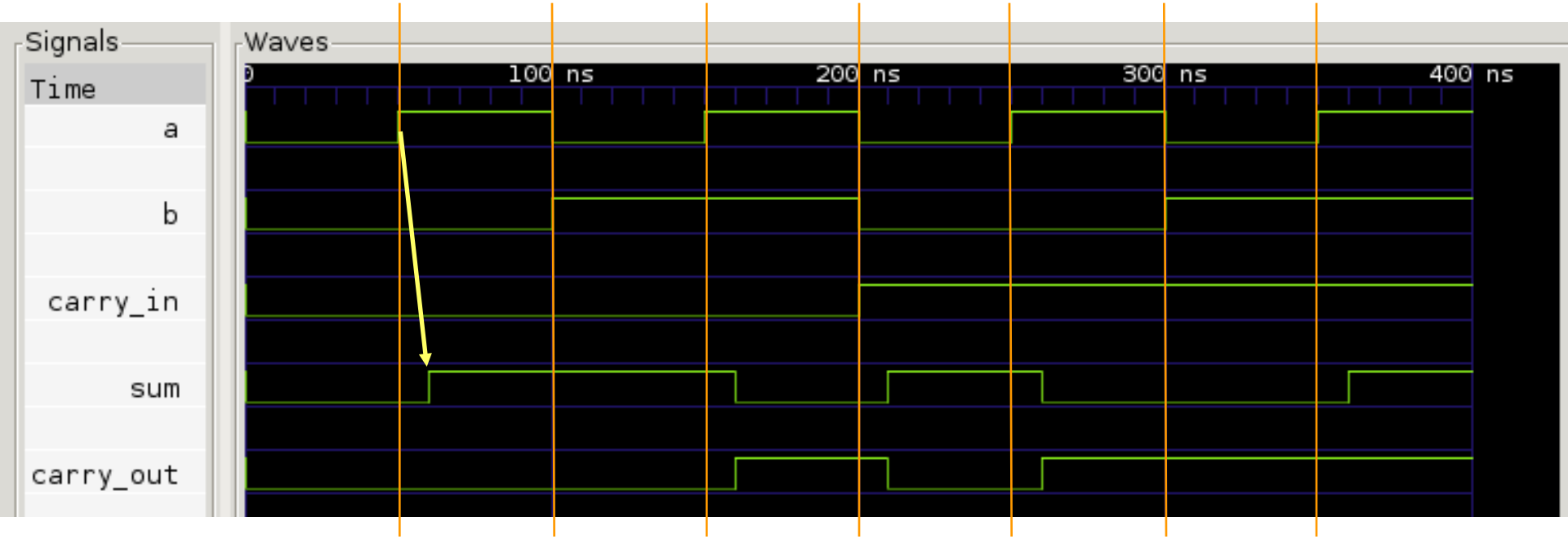
The full adder as an example




```
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

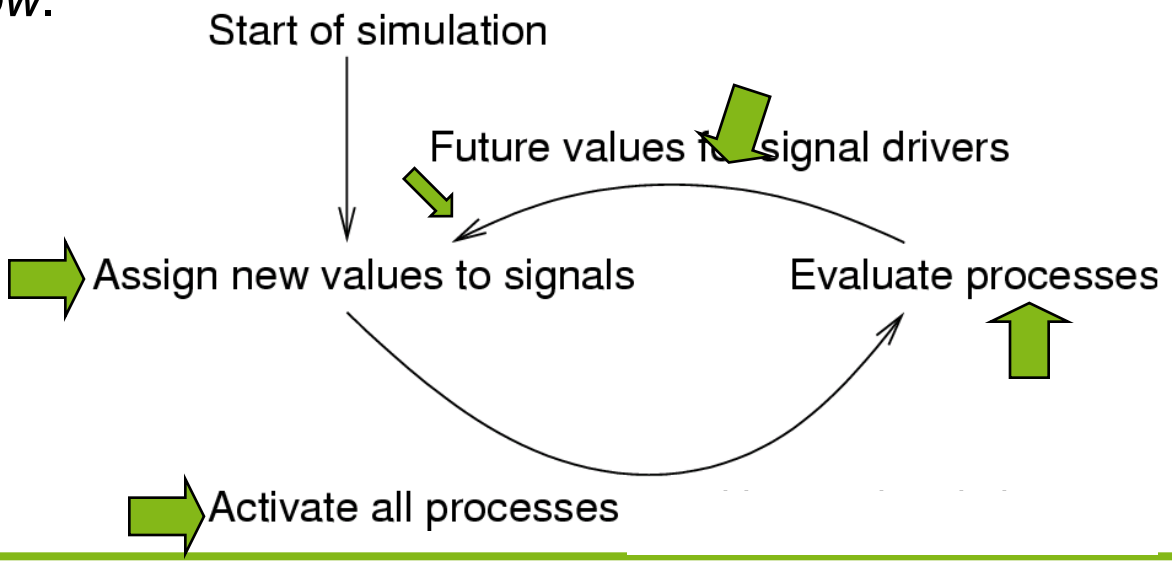
According to the original VHDL standards document:

- 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.

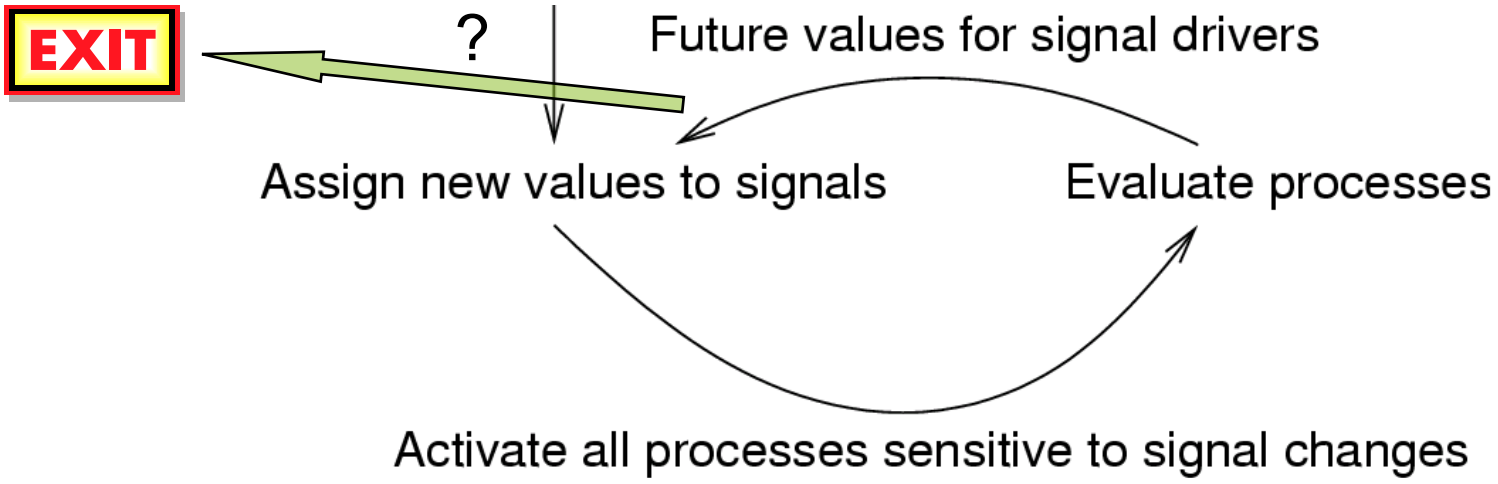
- *The … effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. …*
- *Each … process … is executed until it suspends.*
- *The time of the next simulation cycle (… in this case … the 1st cycle), T_n is calculated according to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



VHDL semantics: The simulation cycle (1)

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

a) Stop if T_n= time‘high or “nothing else is to be done” at T_n. Otherwise, the current time, T_c is set to T_n.



VHDL semantics: The simulation cycle (2)

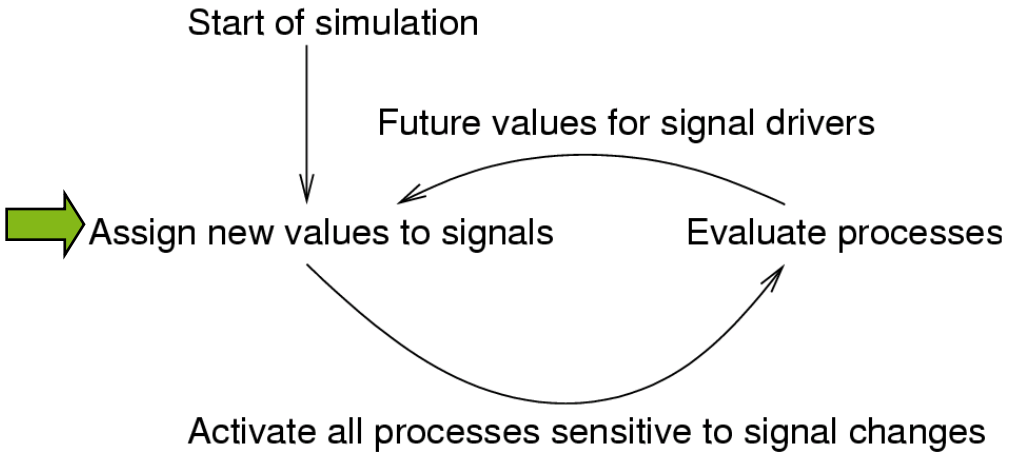
b) Each active explicit signal in the model is updated. (Events may occur as a result.)

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)….

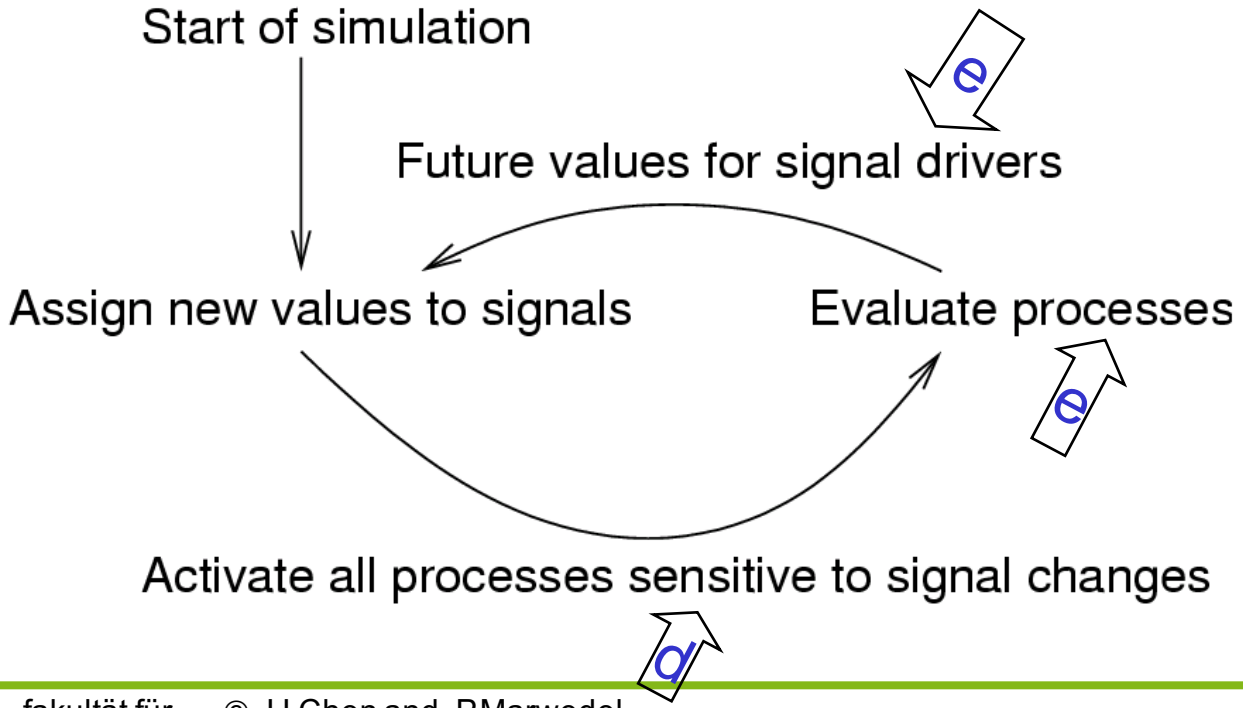


VHDL semantics: The simulation cycle (3)

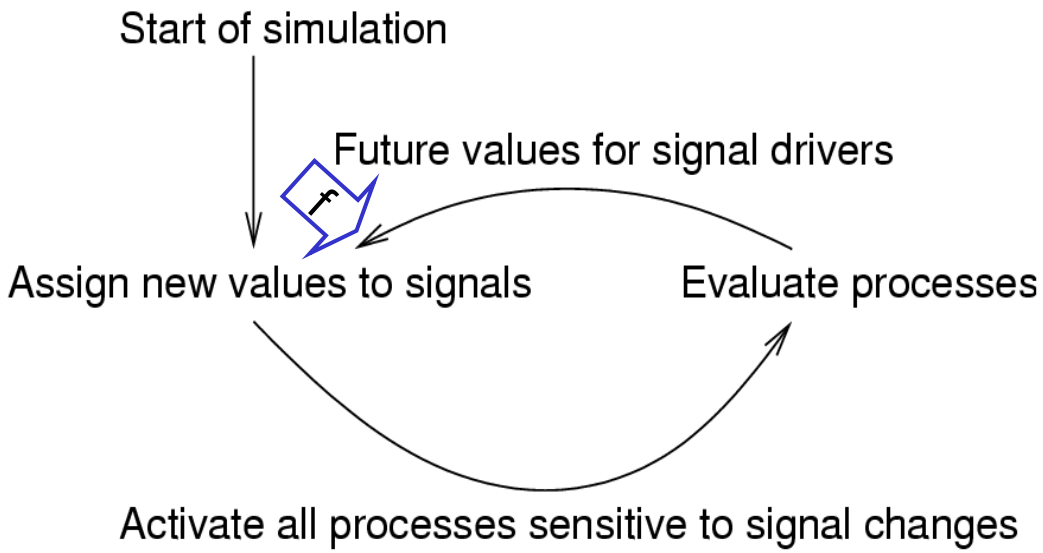
d) ∀ P sensitive to s: if event on s in current cycle: P resumes.

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

*Generates future values for signal drivers.



VHDL semantics: The simulation cycle (4)



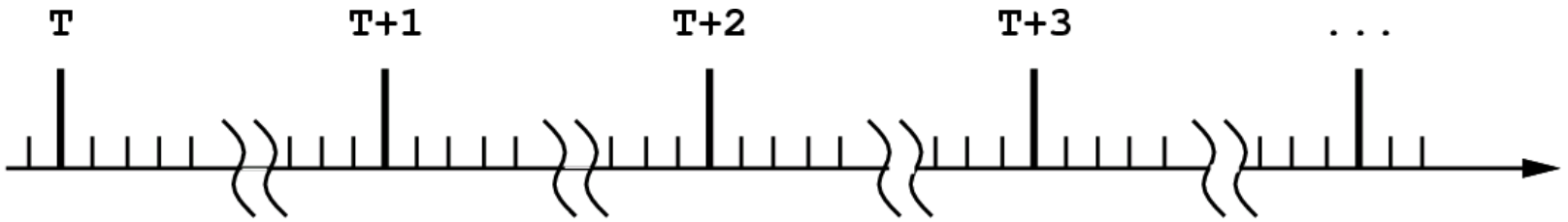
f) Time T_n of the next simulation cycle = earliest of

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 $T_n = T_c$.

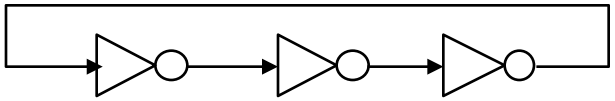
δ-simulation cycles

…
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:

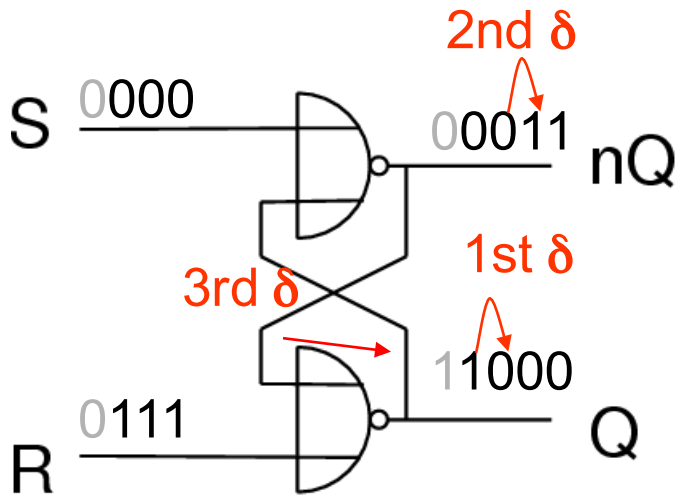


In fact, simulation of delay-less hardware loops might not terminate (don't even advance T_c).

What is the delta cycle of the following circuit?



δ-simulation cycles
Simulation of an RS-Flipflop




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

	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

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

Jian-Jia Chen
(slides are based on
Peter Marwedel)
TU Dortmund, Informatik 12
Germany

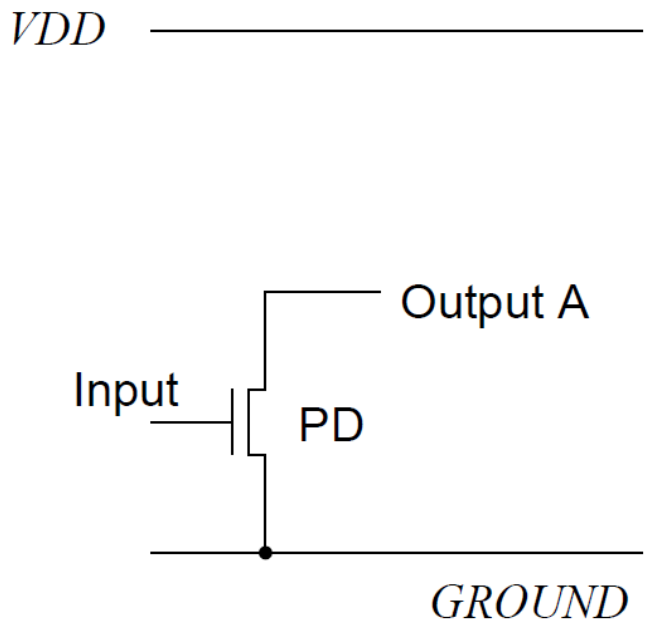
2017年 11 月 06 日

Abstraction of electrical signals

- 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)

2 signal strengths

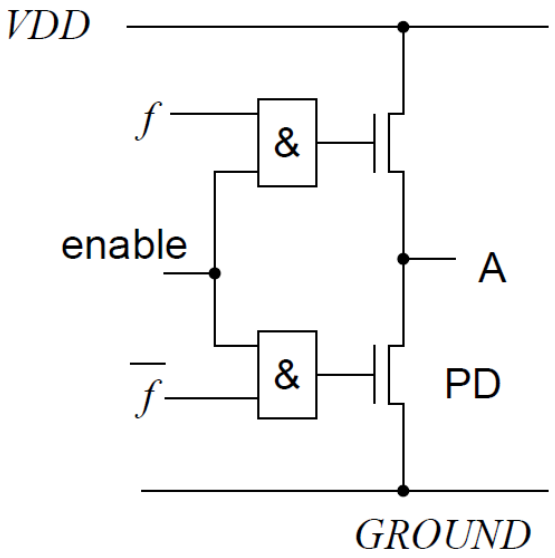
Many subcircuits can effectively disconnect themselves from the rest of the circuit (they provide “high impedance“ values to the rest of the circuit). Example: subcircuits with open collector or tri-state outputs.



Input = ’0’ –> A disconnected

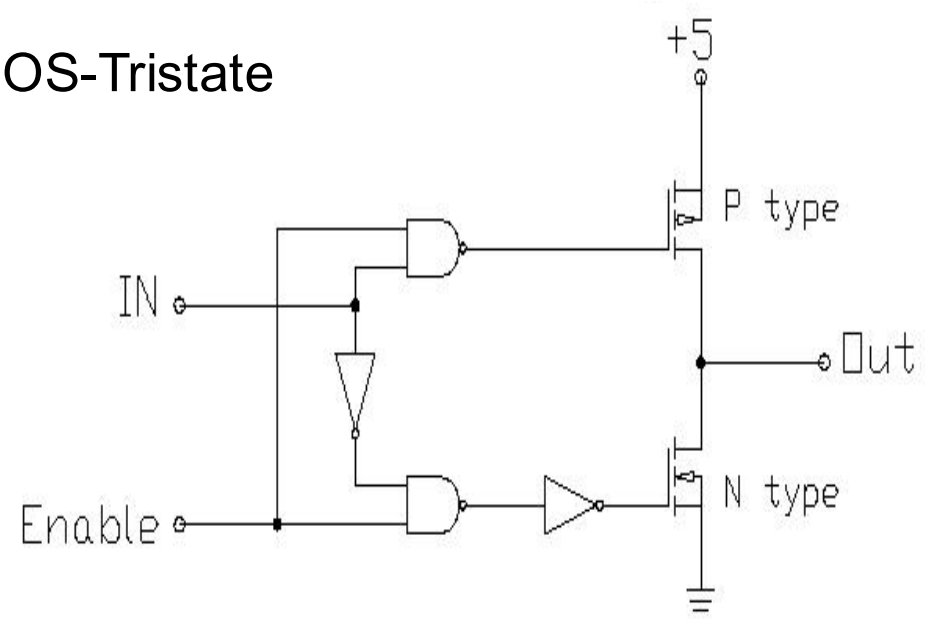
TriState circuits

nMOS-Tristate



enable = ’0’ –> A disconnected

CMOS-Tristate

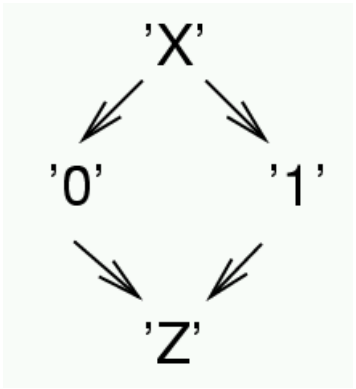


Source: http://www-unix.oit.umass.edu/~phys532/lecture3.pdf

☞ We introduce signal value 'Z‘

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'



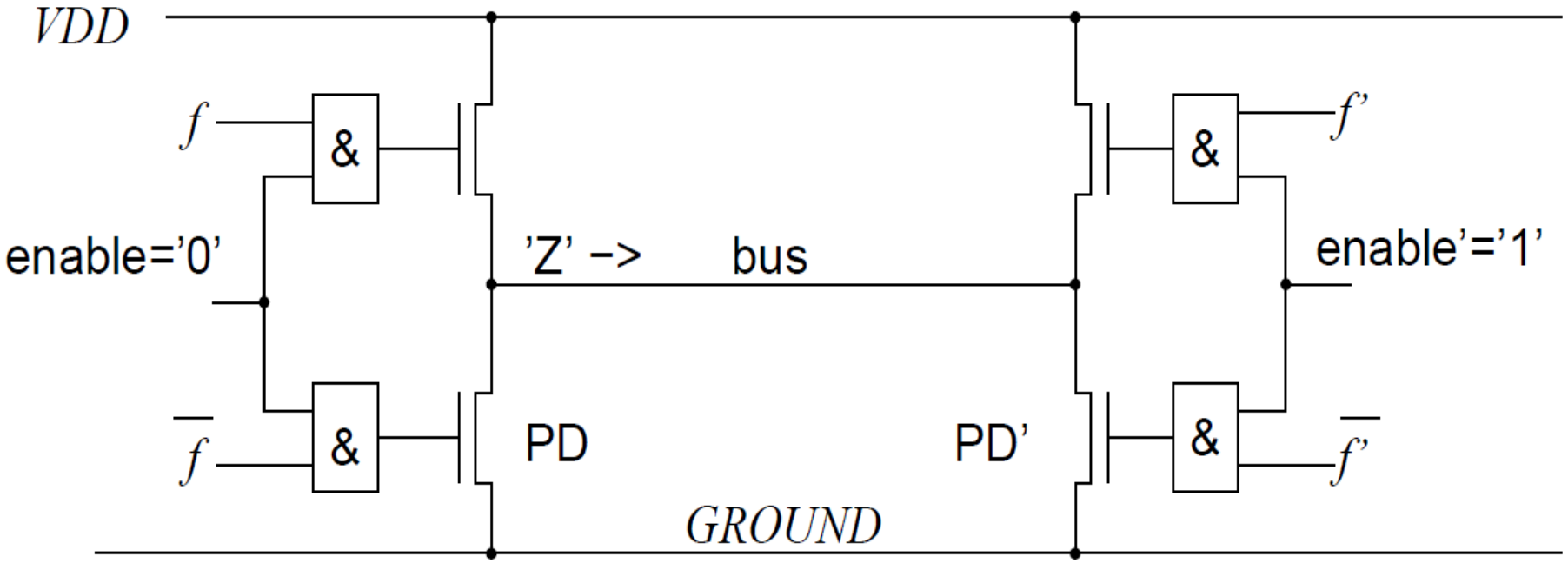
1 strength

Hasse diagram

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

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

Application example



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

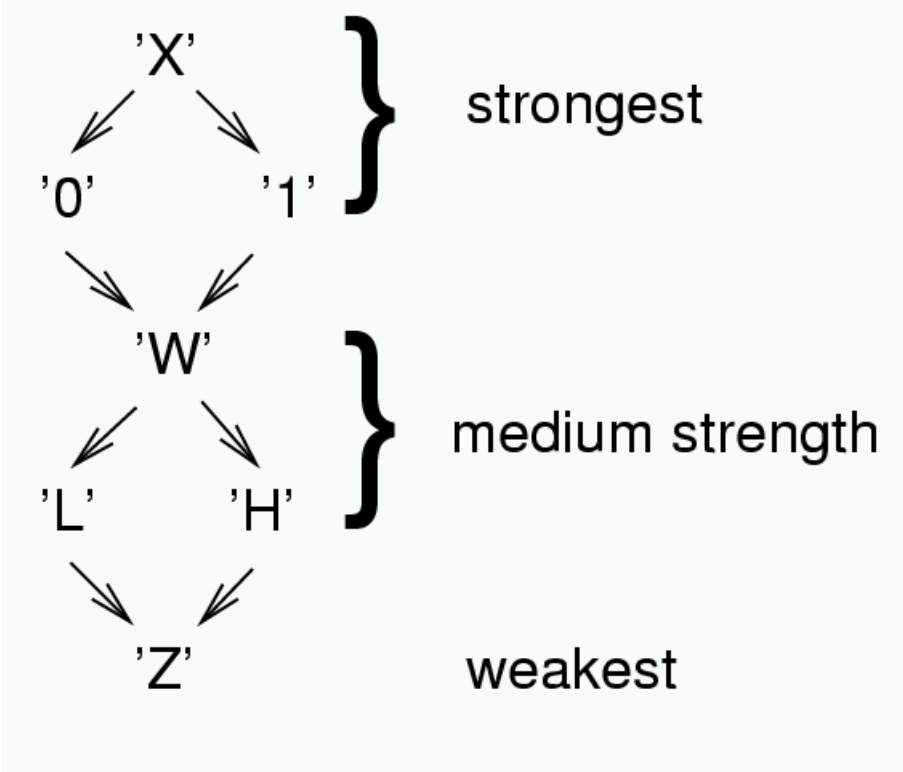
"as if left circuit were not there".

3 signal strengths

There may also be weak signals of the same level as '0'

- Introduction of 'L‘ #('L', '1')=‘1'; #('L','Z') = 'L';
- Similarly for ‘H‘ #(‘H', '1')=‘1'; #(‘H','Z') = ‘H';
- Introduction of ‘W‘: #('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.

☞: Everything said about 7-valued logic applies.

'U': un-initialized signal; used by simulator for signals that have not been explicitly initialized.

'-': denotes input don't care.

Input don‘t care

'–' denotes **input don't care.**

Suppose:

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

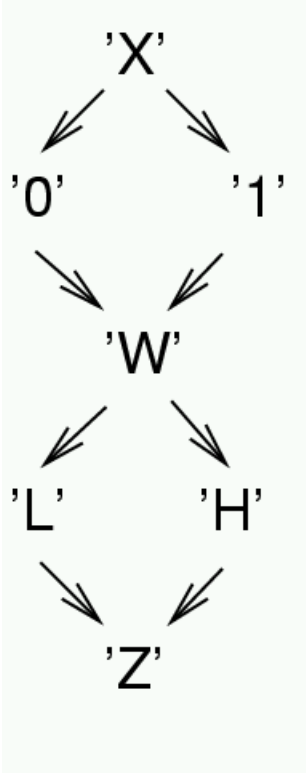
Then, we could like specifying this in VHDL as

```
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;
```

(„input don‘t care -“ was with different behaviour before *VHDL’2006*).

Using # (=sup) in resolution functions in VHDL

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



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 approach
- Application to IEEE 1164