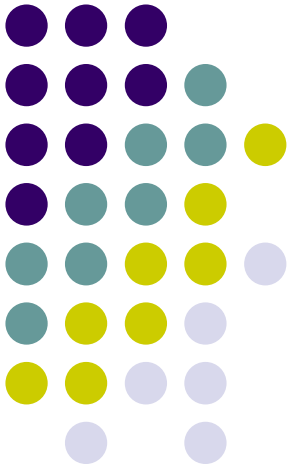
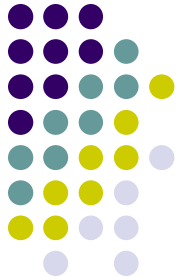
Chapter 2 Logic Simulation

電路邏輯模擬

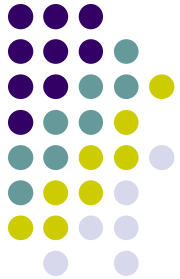


Simulation



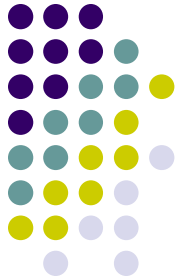
- Purposes
  - Verification
  - Debugging
  - Studying design alternative (cost/speed)
  - Computing expected behavior for test patterns
- Simulation-based design verification
  - To check correct operations:
    - e.g. delays of critical paths
    - free of critical races & oscillation
  - Problem is that tests are hand crafted; Very hard to prove that a test is complete.
  - Formal and assertion-based verification required

Modeling for Circuit Simulation



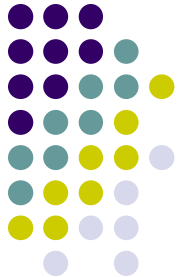
- Circuit models
  - Modeling levels
    - Behavioral, logic, switch, timing, circuit
  - Modeling description (languages)
- Signal models
  - Logic value models
  - Timing value models
- Choices of models determine the complexity and accuracy of simulation

Level of Circuit Modeling (1/2)



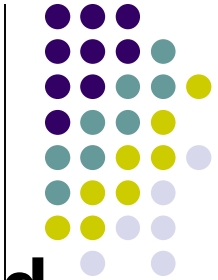
- Electronic system level
  - Software+hardware
  - Transaction/cycle-accurate functions
  - C/C++, SystemC, SystemVerilog, etc.
- Register-Transfer-Level (RTL)
  - Define bit and timing (almost) accurate architecture for sign-off
  - VHDL and Verilog
- Logic/cell/gate level
  - Interconnected Boolean gates
    - AND, OR, NOR, NAND, NOT, XOR, Flip-flops, Transmission gates, buses, etc.
  - Suitable for logic design, verification and test

Level of Circuit Modeling (2/2)



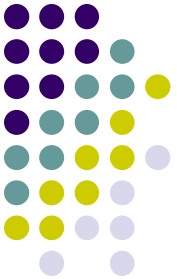
- Switch level
  - Interconnects of ideal transistor switch
  - Need transistor size, node R and C to determine logic value
    - Zero delay in timing
  - Suitable for full-custom high-performance ASIC
- Timing level
  - Use transistors with detailed device models
  - Calculate charge/discharge current with transistor's voltage-current model and obtain node voltage as a function of time
  - Mainly for post-PR timing verification, e.g., Timemill™
- Circuit level
  - Lowest level, ultimate in simulation accuracy
  - Obtain timing by solving the equations relating branch/loop current and node voltage
  - Critical timing analysis for digital circuits
  - Mixed-signal circuit simulation

Logic States for Simulation



- **Two states (0, 1) for combinational and sequential circuits with known initial states.**
- **Three states (0, 1, X) for sequential circuits with unknown initial states**
  - X (unknown state) for cases when the logic value cannot be determined
  - X can be either 0 or 1.
  - Sources: uninitialized FF, bus, memory, multi-cycle paths, etc.

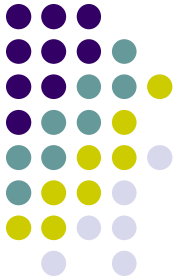
Logic Operations with X



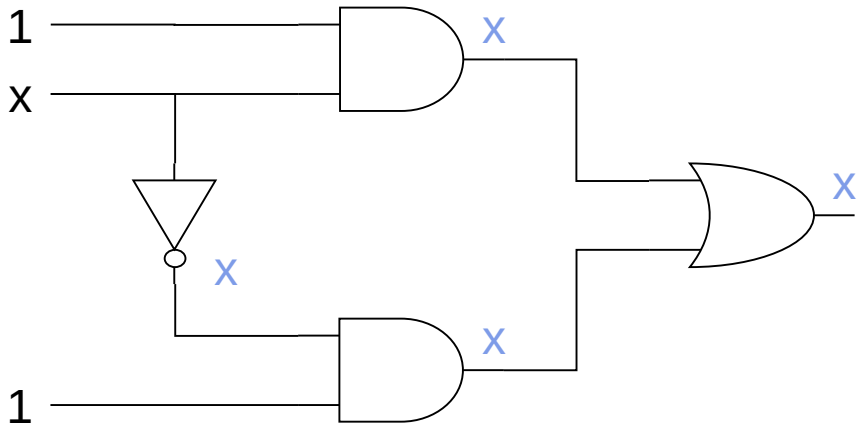
AND	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

OR	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

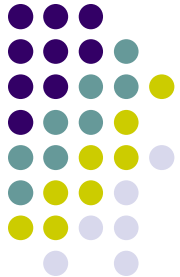
Problems with 3-Valued Logic



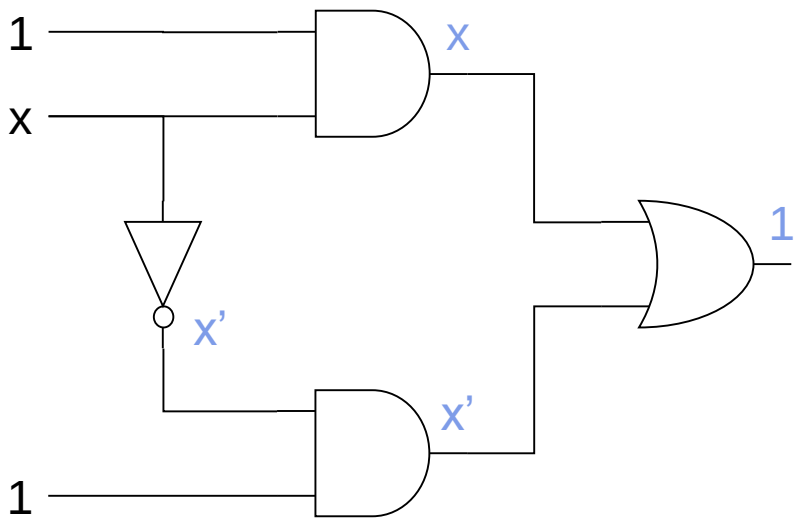
- May cause information loss
  - Fail determining the logic value even though that value can be easily determined
  - Example:
    The output is evaluated as x even though it should be 1 regardless of the actual value of x



Symbolic Simulation of “x”

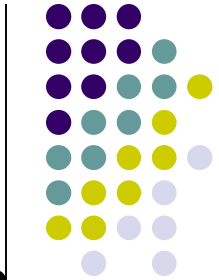


- Treat “x” as a signal
  - NOT x = x’
  - x + x’ = 1



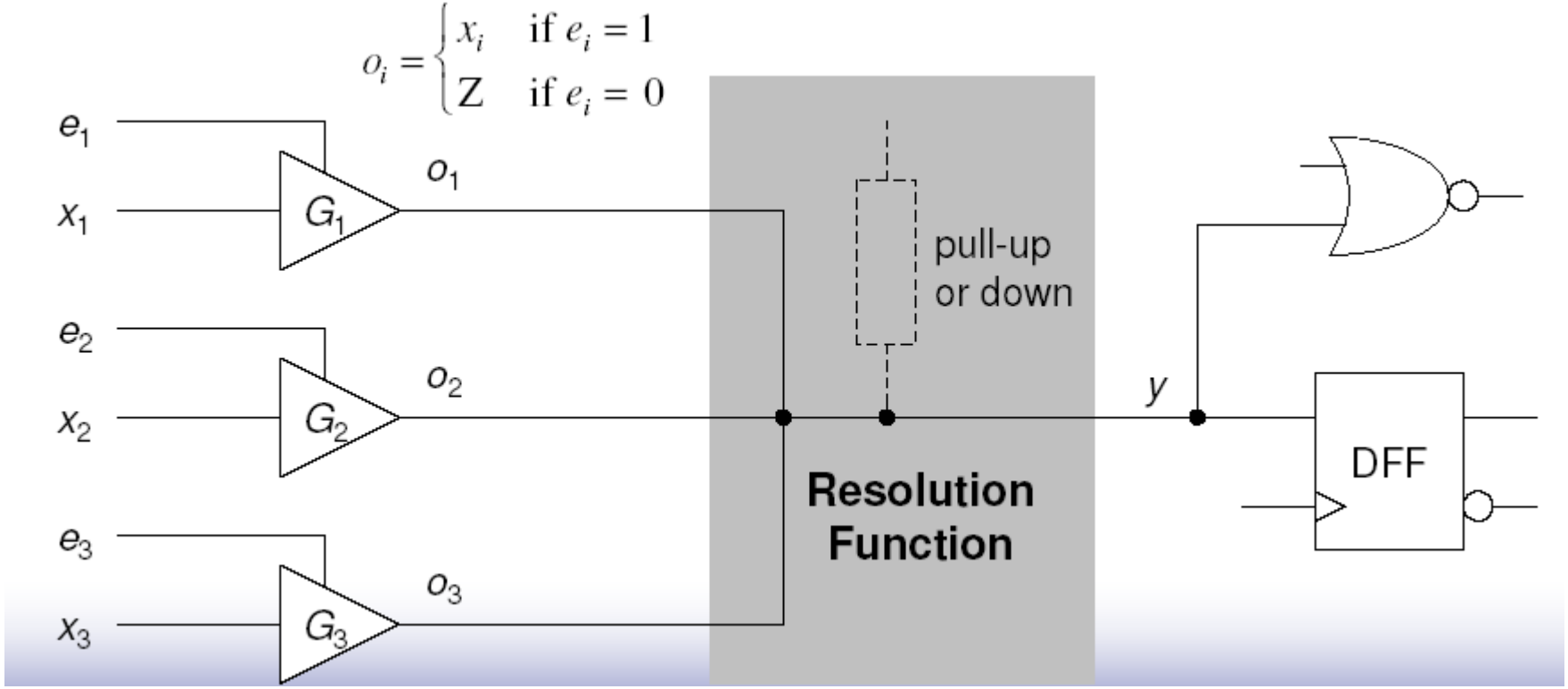
- Problem
  - There can be multiple sources of X, e.g., Flip-flops
  - One “x” for each unknown value (x1, x2, …)
  - Impractical for large circuits, e.g., x1 + x2 = ?

High-Impedence State Z



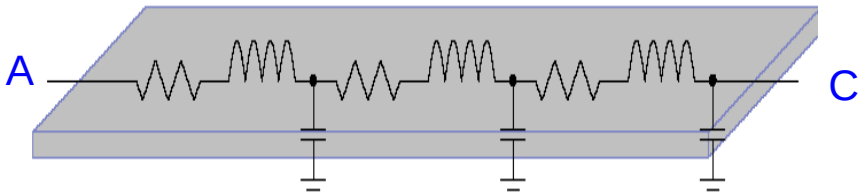
- Floating state: a node w/o a path conducting to either Vdd or Gnd
- Logic state of Z is interpreted dynamically
  - Single floating node
    - Same as its driven value before becoming floating
  - A set of floating nodes get connected
    - Depends on charge sharing, may become uncertain
  - A floating node connected to Vdd/Gnd becomes 1/0
  - When multiple source drive a floating node, the value depends on the strength of the driving logics.
- Most MOS circuits containing dynamic logic require four states (0,1, x, z) for simulation

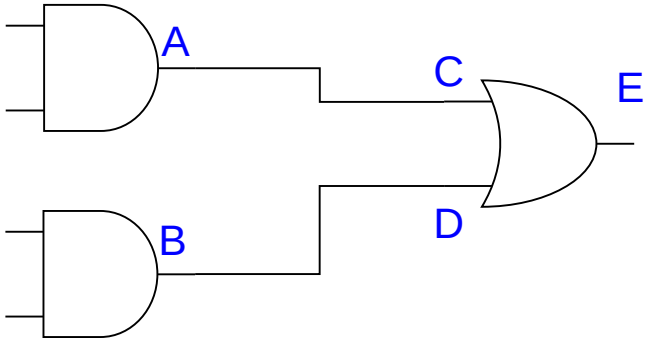
An Example of High-Z Bus



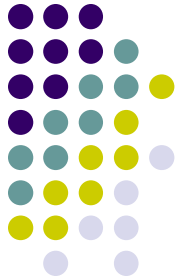
Delay (Timing) Models

- Delay of a logic element
  - Time between an input change (cause) and the output change (effect), e.g. C->E or D->E
  - Called gate delay, pin-pin delay, or switching delay
- Interconnect delay
  - Time between the generation of a signal transition at a gate output and its arrival at the input of a fanout gate, e.g. A->C, or B-> D
  - Or called switching delay
  - Consider R, C (L) effects



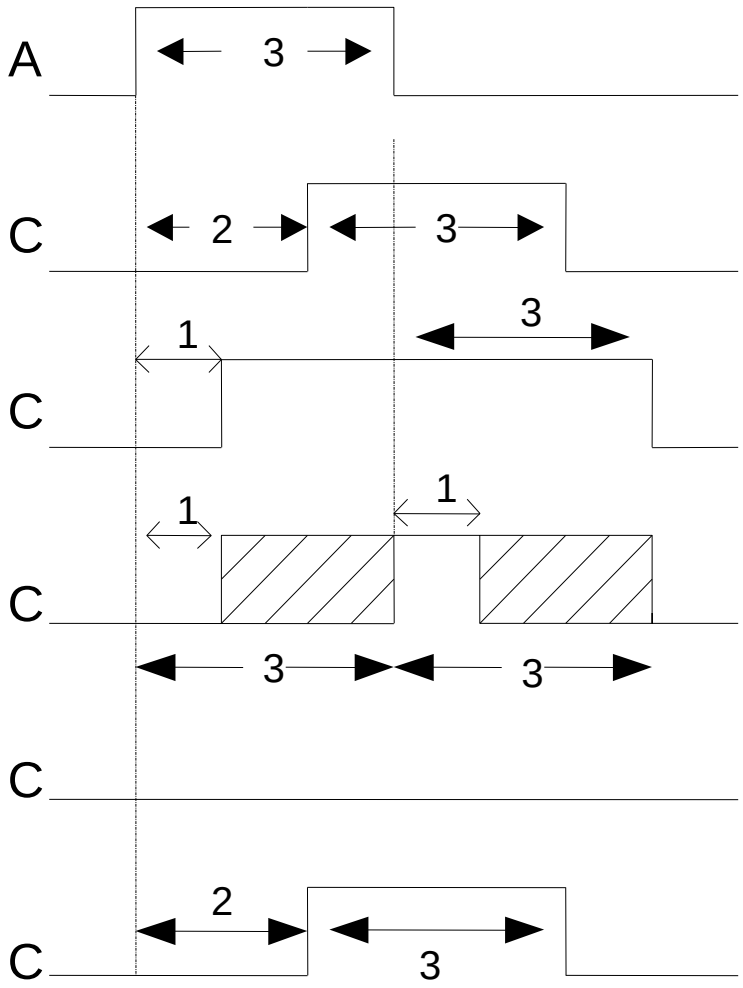


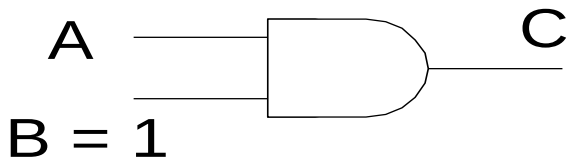
Terms for Cell Delay Models



- Zero and unit delay
- Rise (fall) delay
  - Gate delays of different final output states
- Inertia delay
  - Minimum pulse width to cause a transition
  - Used for filtering input/output pulse
  - Input inertia delay: minimum pulse width for input
  - Output inertia delay: minimum pulse width for output
- Min/Max Delay
  - The minimum or maximum bound of a gate delay
- Transition time
  - Time for a signal to transit from 0 to 1 or 1 to 0.

Delay Models Examples





transport delay = 2
(transition-independent)

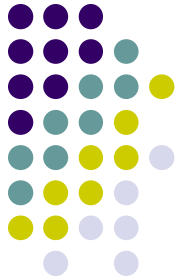
rise delay = 1
fall delay = 3

min-max delay
(transition independent)
$1 \leq d \leq 3$

input inertia delay = 4

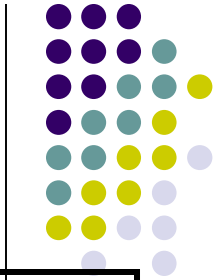
input inertia delay = 2
& transport delay = 2

Common Cell Delay Models



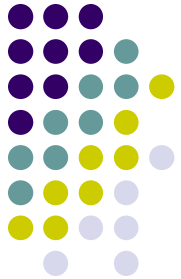
- Table-based
  - A pin-pin min/max rise/fall delay of a cell = f(CL, Tr)
  - CL=output load
  - Tr=input transition time
- Current-source based
  - A voltage-controlled current source I(Vi, Vo)
  - I: Vdd to Gnd current
  - Vi: input voltage
  - Vo: output voltage
  - More accurate in terms of noise, but more CPU intensive
- Interconnect delays
  - Elmore delay

Modeling Levels and Signals



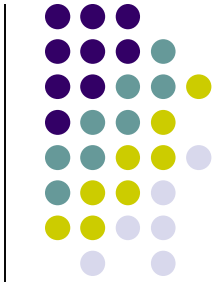
level	Circuit Description	Signal	Timing Resolution	Application
ESL	**SystemC**	**0,1**	**transaction**	**system**
Behavior	**HDL**	**0,1**	**cycle**	**architecture**
Logic	**gate-level HDL**	**0, 1, X, Z (with signal strength)**	**zero, unit, multiple cell delays**	**logic design and test**
Switch	**transistor, RC interconnects**	**0, 1, X (with signal strength)**	**zero, possible gross-grain timing**	**full-custom logic verification**
Timing	**same as above (SPICE)**	**Analog**	**fine-grain time**	**timing verification**
Circuit	**same as above (SPICE)**	**Analog**	**continuous time**	**timing/analog verification**

Types of Logic Simulators



- Compiled-driven simulators
  - The compiled code is generated from an RTL or gate-level description of the circuit
  - Simulation is simply execution of the compiled code
- Event-driven Simulators
  - Simulate only those signals with value changes
  - Only propagate necessary events (value changes)

Compiled Simulation



- **A circuit is simulated by executing a compiled code of the circuit.**
- Levelization
  - to ensure that a signal is evaluated after all its sources are evaluated

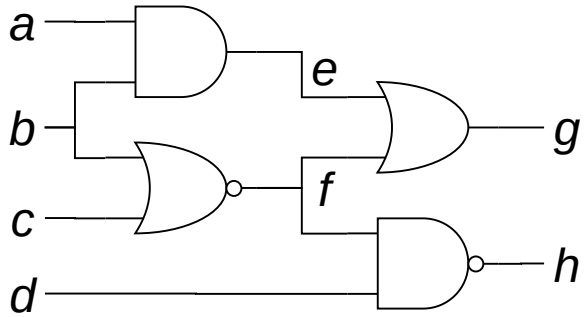
Levelization

•Assign all PI's level 0

• The level of gate G is

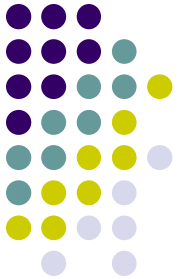
Lg = 1 + max(L1,L2,…)

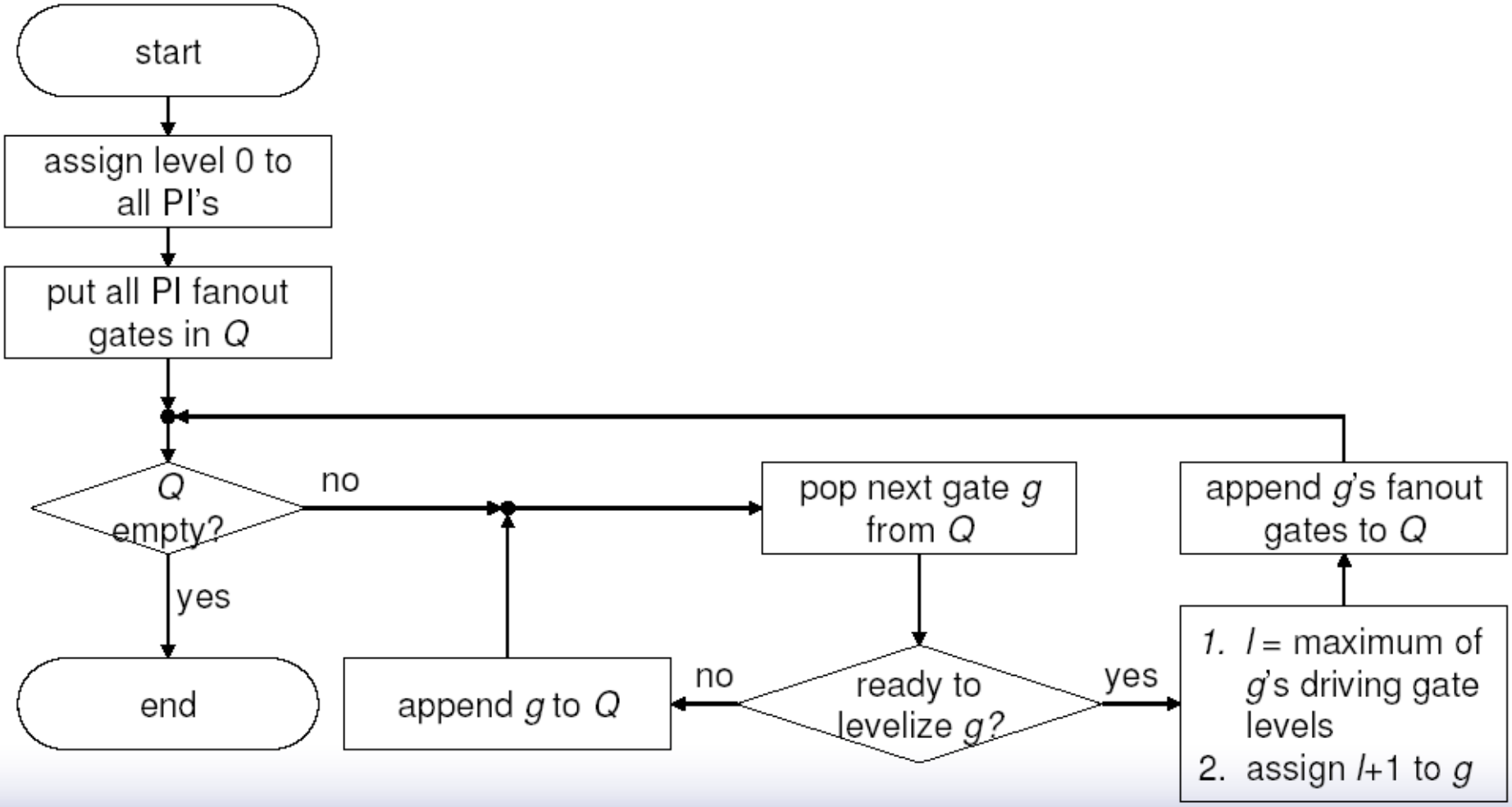
where Li's are G's input gates



- level 0: *a*, *b*, *c*, *d*
- level 1: *e*, *f*
- level 2: *g*, *h*

Flow of Levelization





Example of Levelization

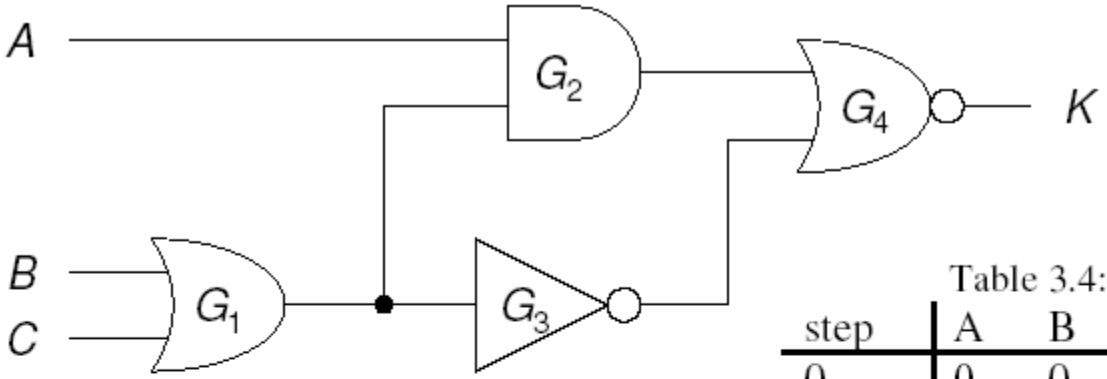
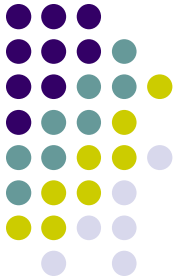


Table 3.4: The levelization process of circuit

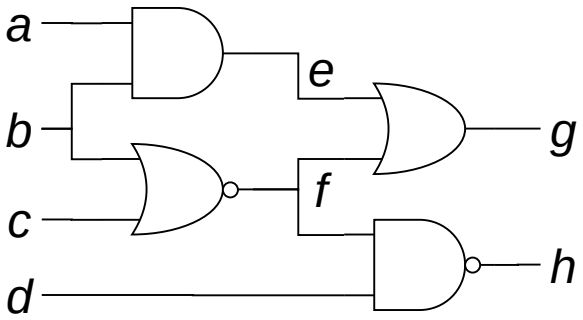
step	A	B	C	G_1	G_2	G_3	G_4	Q
0	0	0	0					$<G_2, G_1>$
1	0	0	0					$<G_1, G_2>$
2	0	0	0	1				$<G_2, G_3>$
3	0	0	0	1	2			$<G_3, G_4>$
4	0	0	0	1	2	2		$<G_4>$
5	0	0	0	1	2	2	3	$<>$

- The following orders are produced
  - $G_1 => G_2 => G_3 => G_4$
  - $G_1 => G_3 => G_2 => G_4$

Compiled Simulation – cont'd



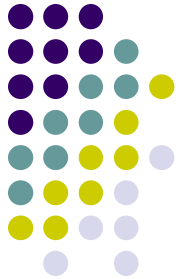
- Code generation & execution



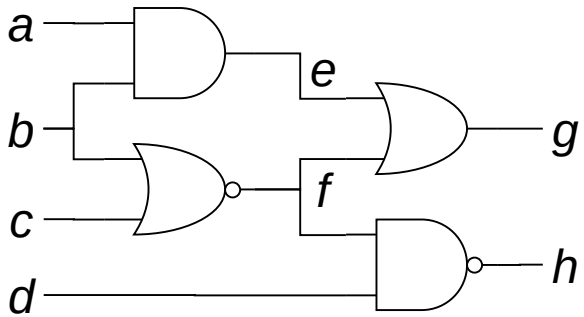

```
while (1) {
  read_in(a,b,c,d);
  e = AND(a,b);
  f = NOR(b,c);
  g = OR(e,f);
  h = NAND(d,f);
  print_out(g,h);
}
```

- Very effective for 2-state (0,1) simulation
  - Can be compiled directly into machine codes
- Mainly for functional verification, where timing is irrelevant

Compiled Simulation – Example in C

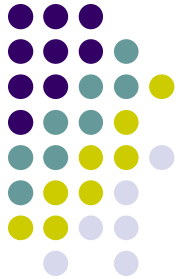


- Code generation in C



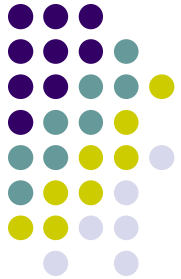

```
#include <stdlib.h>
int main(){
  unsigned int a=0xF; //1111
  unsigned int b=0xA; //1010
  unsigned int c=0x8; //1000
  unsigned int d=0x7; //0111
  unsigned int e, f, g, h;
  e = a&b;
  f = ~(b|c);
  g = e|f;
  h = ~(d&f);
  printf("g,h=%X,%X", g, h);
}
```

Problems with Compiled Simulation

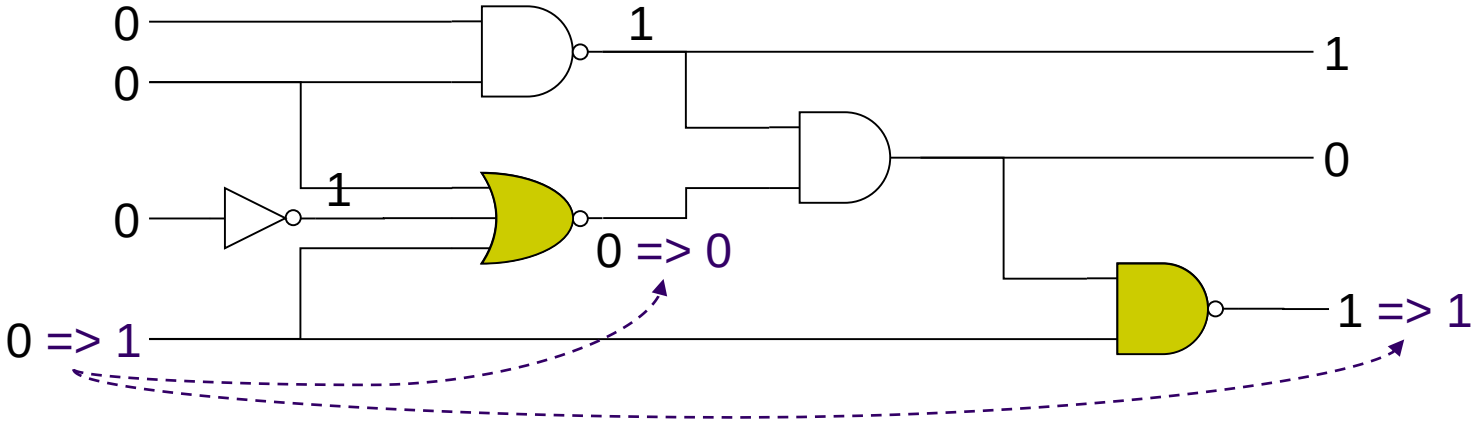


- Zero-delay model
  - Timing problems, e.g., glitches and races, cannot be modeled
- Simulation time could be long
  - Proportional to Ω(input vectors$\times$number of gates)
  - Entire circuit is evaluated even though typically only 1-10% of signals change at any time
  - Note RTL compiled simulation is different and fast, since branching can be used.

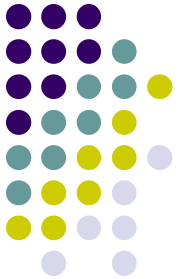
Event-Driven Simulation

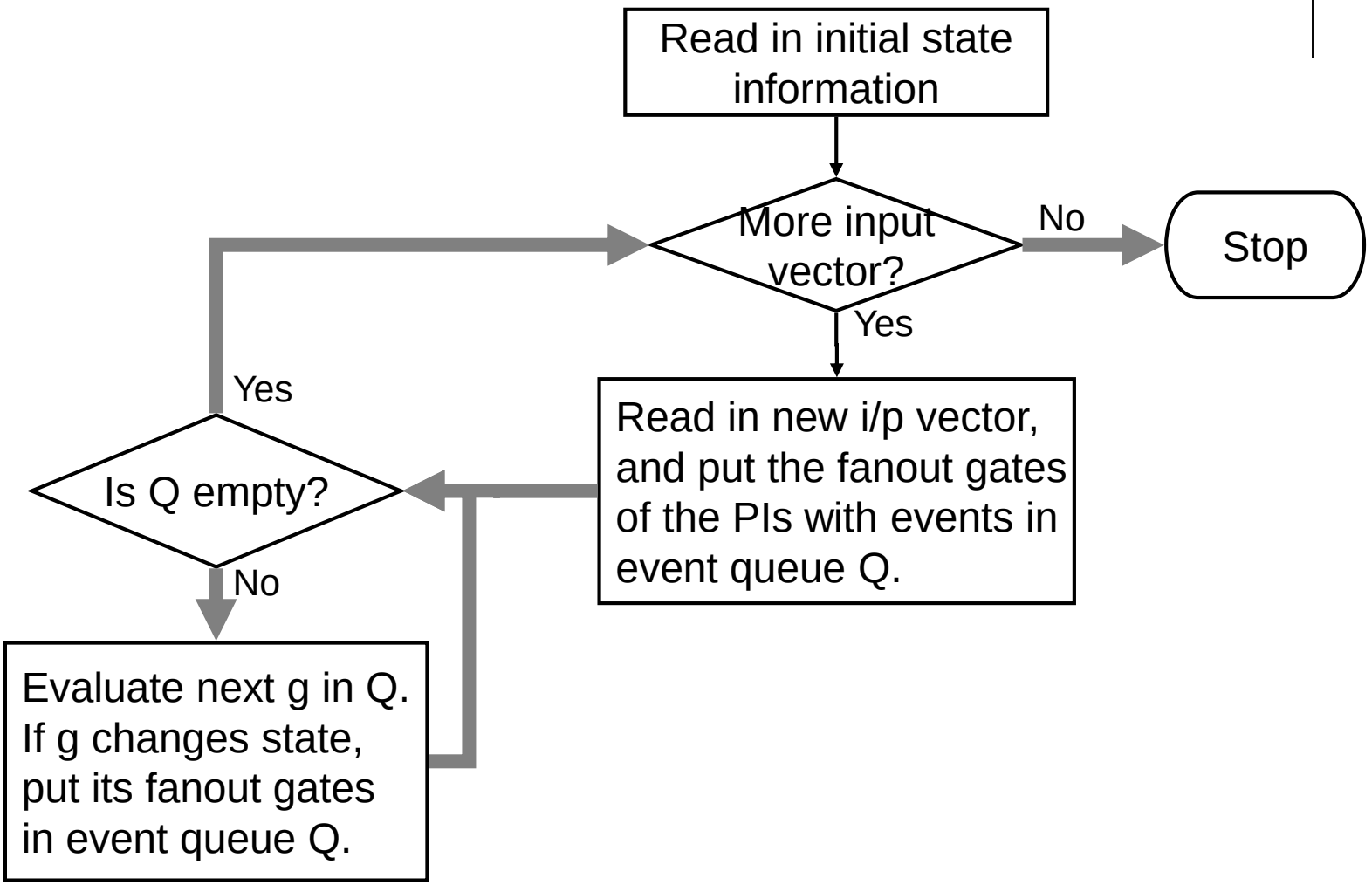


- An event is a change in value of a signal line
- An event-driven simulator evaluates a gate (element) only if one or more events occur at its inputs
  - Only does the necessary amount of work
  - Follows the path of signal flow

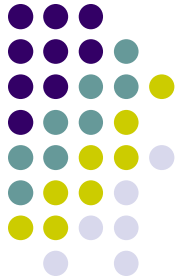


Zero-Delay Event-Driven Simulation



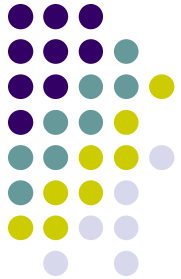


Gate Evaluation – Table Lookup



- The most straightforward and easy way to implement
  - For binary logic, 2^n entries for n-input logic element
  - May use the input value as table index
  - Table size increases exponentially with the number of inputs
- Could be inefficient for multi-valued logic
  - A k-symbol logic system requires a table of 2^{mn} entries for an n-input logic element
  - $m = \log_2 k$
  - Table indexed by mn-bit words

Gate Evaluation – Input Scanning



- Assume that only dealing w/ AND, OR, NAND, and NOR primitive gates
- These gates can be characterized by controlling value *c* and inversion *i*
  - The value of an input is said to be controlling if it determines the gate output value regardless of the values of other inputs

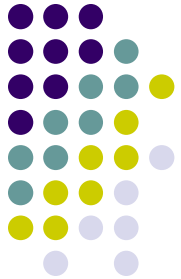
	c	***i***
AND	**0**	**0**
OR	**1**	**0**
NAND	**0**	**1**
NOR	**1**	**1**

Input Scanning – cont'd

I/P of a 3-input primitive gate			O/P
c	x	x	$c \oplus i$
x	c	x	$c \oplus i$
x	x	c	$c \oplus i$
c'	c'	c'	$c' \oplus i$

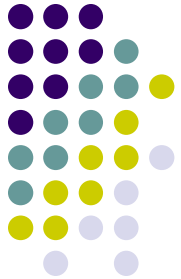
```
Evaluate(G,c,i){
  u_values = false;
  for every input value v of G{
    if (v == c) return c⊕i;
    if (v == x) u_values = true;
  }
  if (u_values) return x;
  return c'⊕i;
}
```

Gate Evaluation – Input Counting



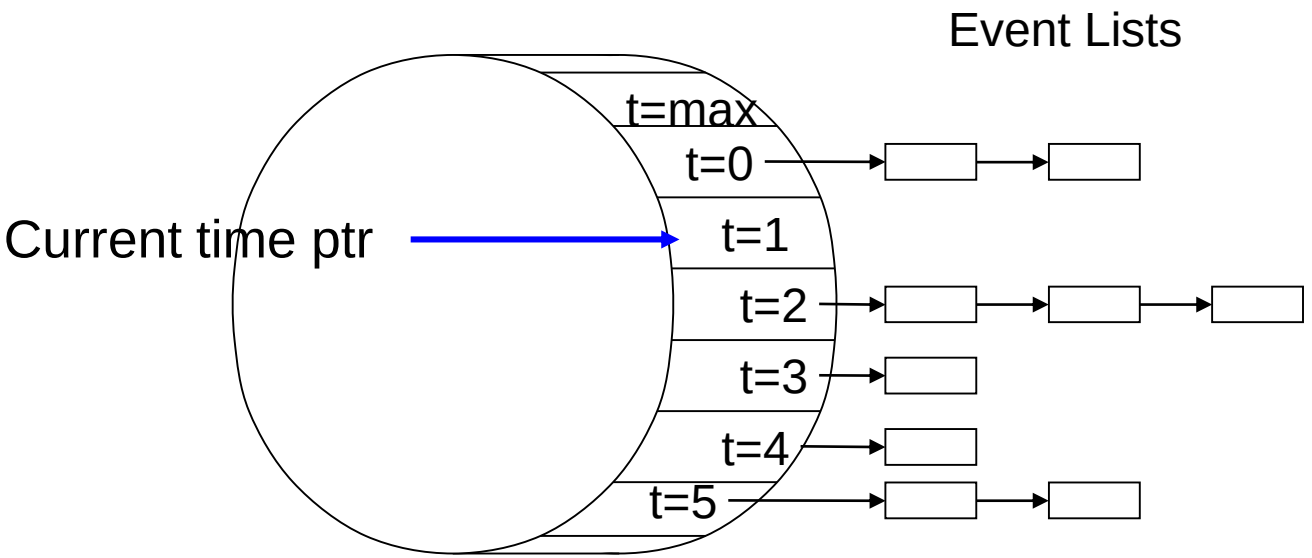
- To evaluate the output, it's sufficient to know
  - Whether any input equals c
  - If not, whether any input equals x
- Simply maintain c_count & x_count
  - Example: AND gate
    - 0 => 1 at one input: c_count--
    - 0 => x at one input: c_count--, x_count++
- ```
  Evaluate(G,c,i){
    if (c_count > 0) return c⊕i;
    if (x_count > 0) return x;
    return c'⊕i;
  }
  ```
  ```

Event-Driven Simulation with Delays



- While (event list is not empty){
  t = next time in list;
  process entries for time t;
  }
- The key is to construct a queue entry for every time point

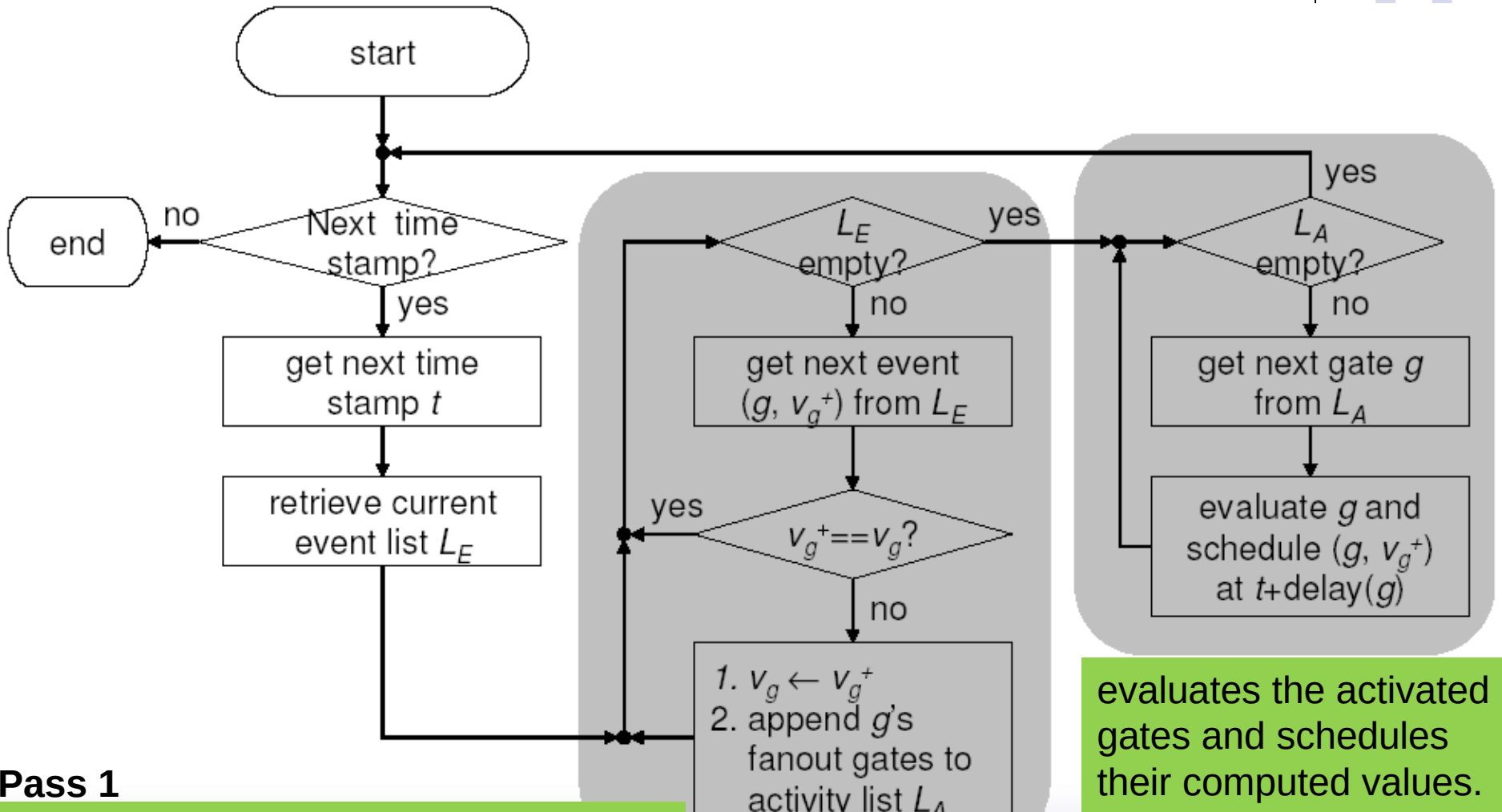
Time wheel



- Max units is the largest delay experienced by any event
  - All gates + interconnects
  - A total of max+1 slots

Flow of Simulation with Delays

Algorithm 1 (two-pass)



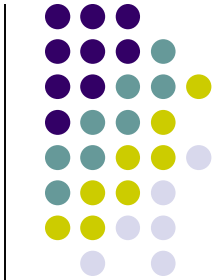
Pass 1

retrieves the entries from event list & determine the activated gates.

evaluates the activated gates and schedules their computed values.

Pass 2

Simulation with Delays



Algorithm 1

Activated **= ϕ**

For every event (*g*, v_g^+) at list of time *t* { *//*from L_E

if ($v_g^+ \neq v_g$) *// v_g* is the original value at signal *g* {

$v_g = v_g^+$;

for every *j* on fanout list of *g* {

update input value of *j;*

add *j* to L_A *if j is not a member of* L_A;

} /* for */

} /* if */

} /* for */

For every *g* ∈ L_A {

v_g^+ = evaluate (*g*);

schedule (*g*, v_g^+) for time *t+delay(g)*;

}

Two-Pass Algorithm

Example

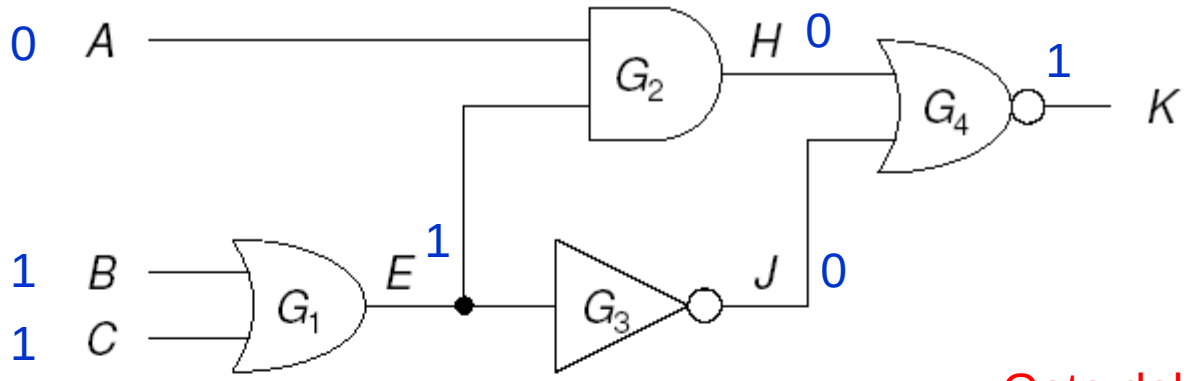
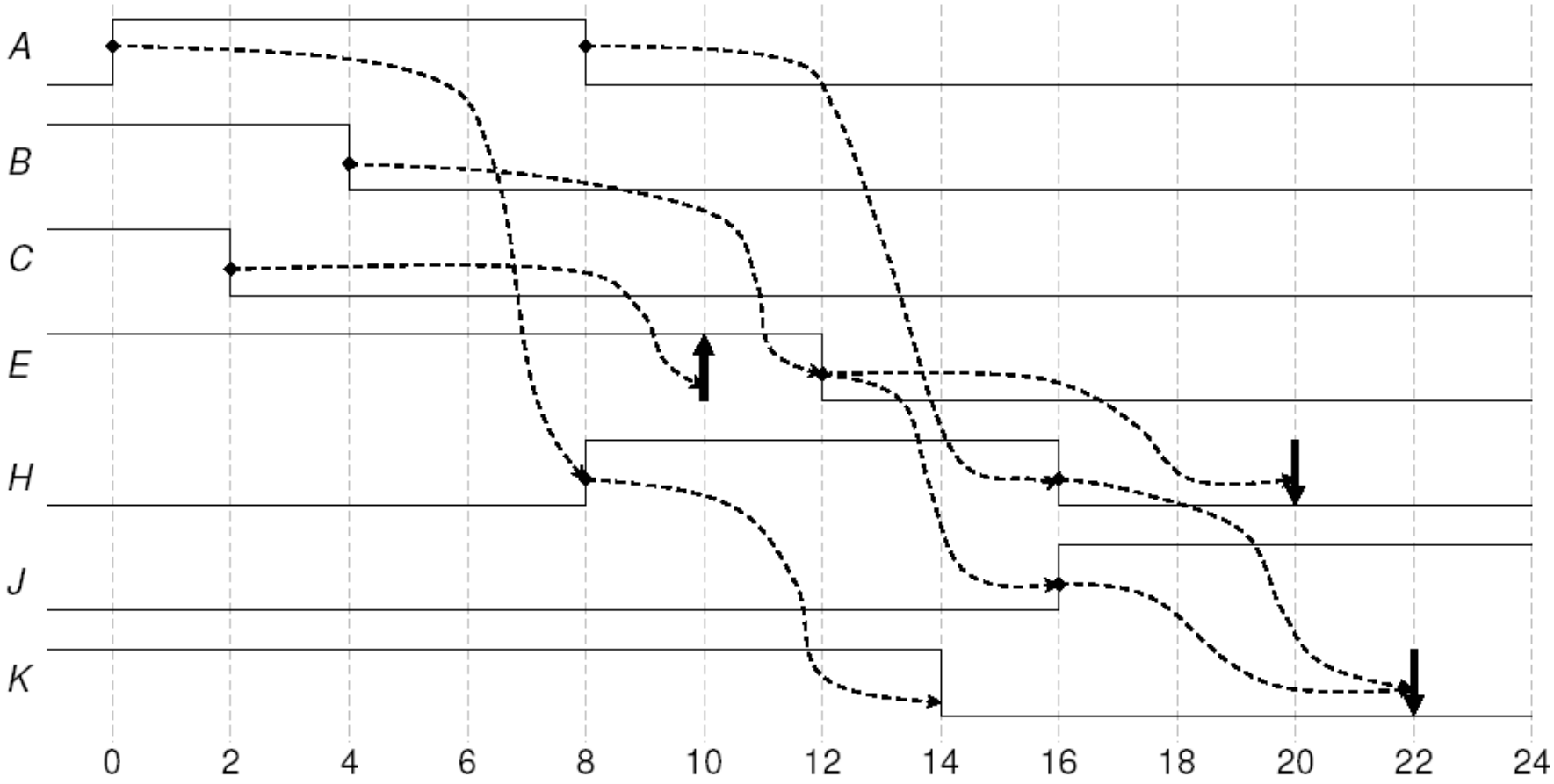


Table 3.5: Two-pass event-driven simulation

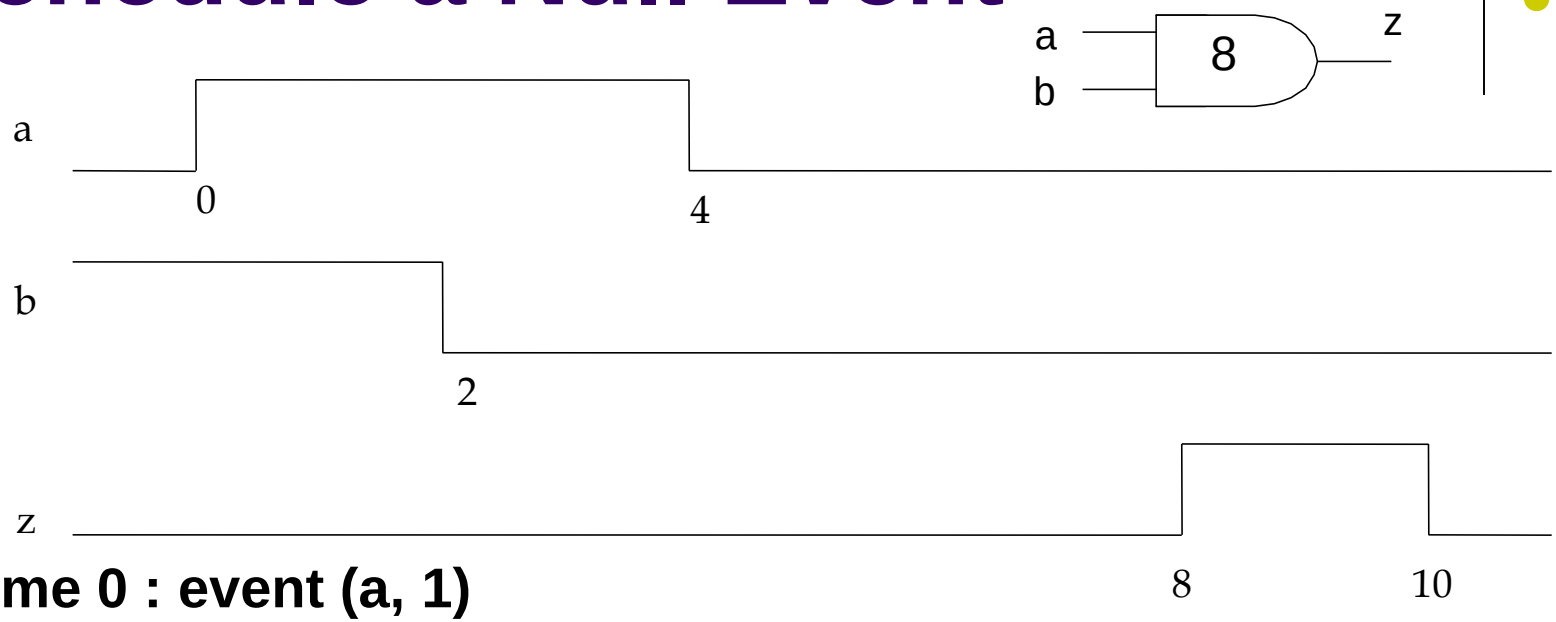
Time	L_E	L_A	Scheduled events
0	{(**A**,**1**)}	{G_2}	{(H,1,8)}
2	{(**C**,**0**)}	{G_1}	{(E,1,10)}
4	{(**B**,**0**)}	{G_1}	{(E,0,12)}
8	{(**A**,**0**),(H,1)}	{G_2,G_4}	{(H,0,16),(K,0,14)}
10	{(E,1)}		
12	{(E,0)}	{G_2,G_3}	{(H,0,20),(J,1,16)}
14	{(K,0)}		
16	{(H,0),(J,1)}	{G_4}	{(K,0,22)}
20	{(H,0)}		
22	{(K,0)}		

Gate delay
G1: 8ns
G2: 8ns
G3: 4ns
G4: 6ns

Example (Cont.)



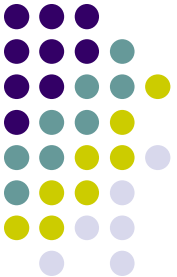
Example of Algorithm 1 to schedule a Null Event



- **Time 0 : event (a, 1)**
  evaluate z=1 ⇒ (z,1) scheduled for time 8
- **Time 2 : event (b, 0)**
  evaluate z=0 ⇒ (z, 0) scheduled for time 10
- **Time 4 : event (a, 0)**
  evaluate z=0 ⇒ (z, 0) scheduled for time 12

The last scheduled event (at t=12) is not a real event!!

An Improved Algorithm



Change Pass 2 to:

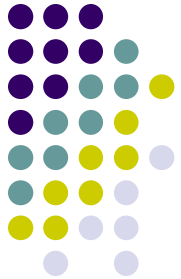
For every $j \in$ ***Activated*** {

v_j' = evaluate (j);
if ($v_j' \neq$ ***lsv(j)***) *(lsv: last saved value)*
{
schedule (j, v_j') for time t+$d(j)$;
lsv(j) = v_j';

}

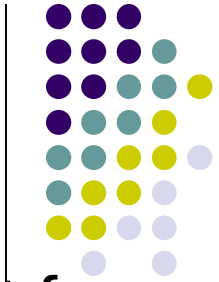
}

Two Pass V.S. One Pass Algorithm



- Two-pass strategy performs the evaluations only after all the concurrent events have been retrieved
  - to avoid repeated evaluations of gates having multiple input changes.
- Experience shows, however, that most gates are evaluated as a result of only one input change.
- One-pass strategy:
  - Evaluates a gate as soon as it is activated
  - Avoids the overhead of building the Activated set

One Pass Algorithm



For every event (g, v_g^+) pending at current time t {

$v_g = v_g^+$;

for every j on the fanout list of g {

update input values of j;

v_j^+ = evaluate (j);

if ($v_j^+ \neq v_j$) {

schedule (j, v_j^+) for time $t+d(j)$;

$v_j = v_j^+$;

}

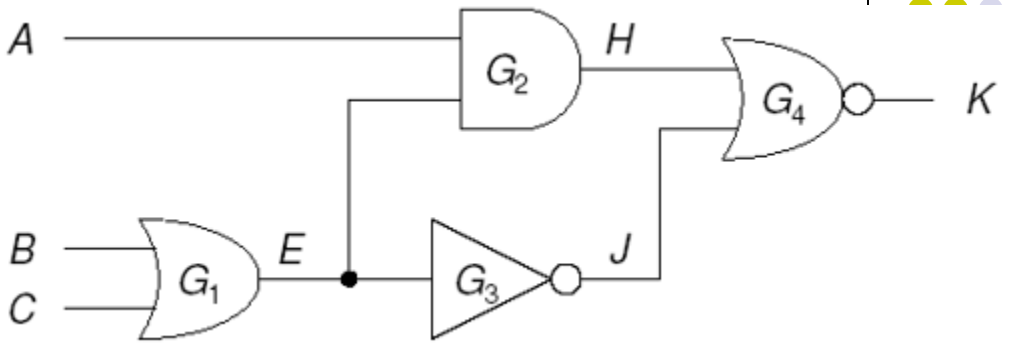
}

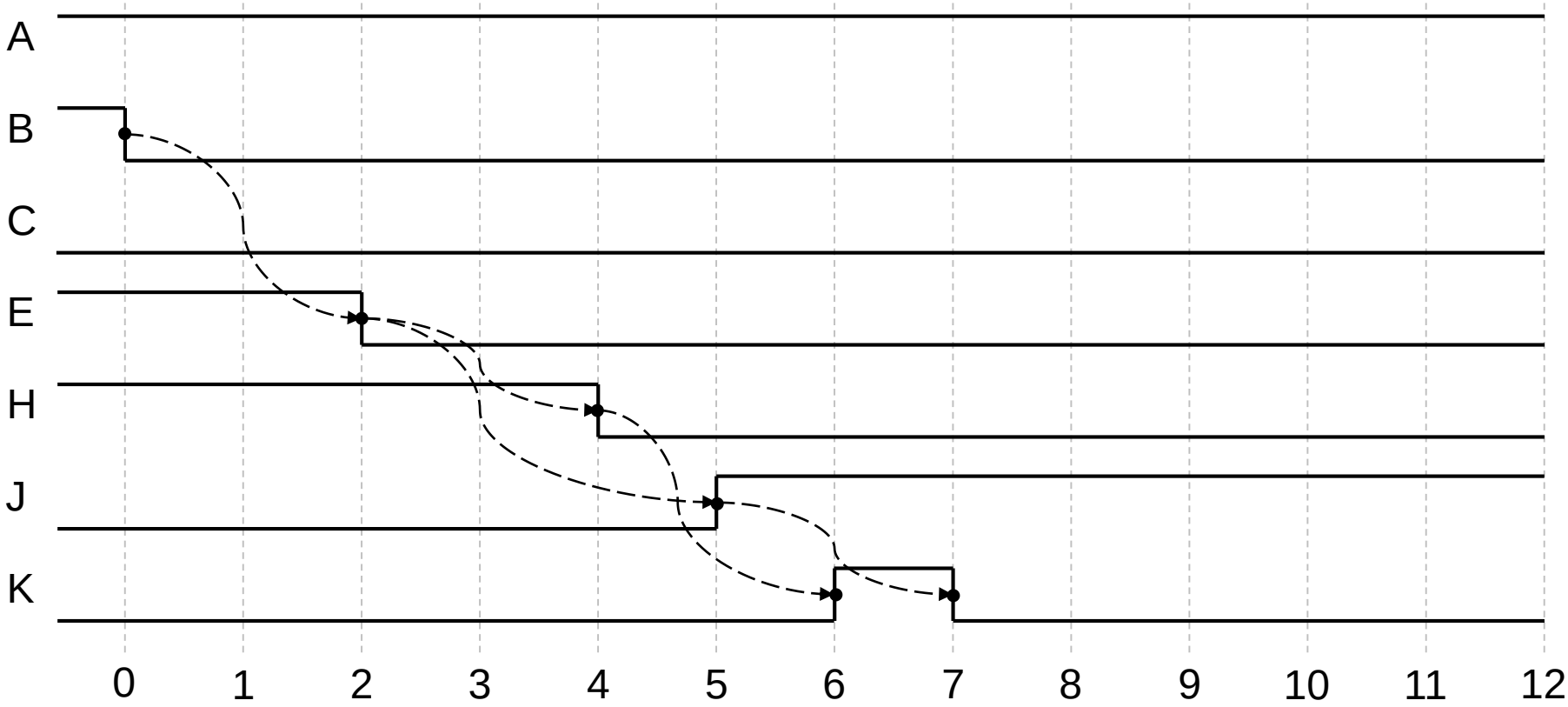
}

An Example of Hazards

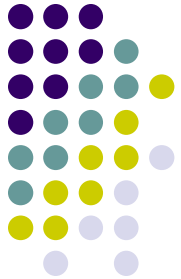
Hazards

- Unwanted transient pulses or glitches





Type of Hazards



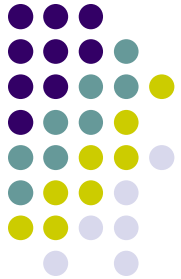
- Static or dynamic
  - A static hazard refers to the transient pulse on a signal line whose static value does not change
  - A dynamic hazard refers to the transient pulse during a 0-to-1 or 1-to-0 transition
- 1 or 0

Static Hazard Detection

- Extra encoding can be used to detect hazards during logic simulation.
  - Note that hazards only occur during signal transition
  - Two consecutive vectors are considered simultaneously
- The following is the 6-valued encoding for a pair of vectors.
  - For example, 0->1 transition (R) is encoded as 0X1.

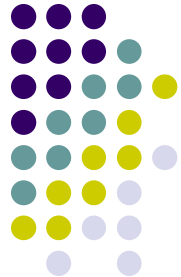
Value	Sequence(s)	Meaning
0	000	Static 0
1	111	Static 1
0/1, R	{001,011} = 0x1	Rise (0 to 1) transition
1/0, F	{110,100} = 1x0	Fall (1 to 0) transition
0*	{000,010} = 0x0	Static 0-hazard
1*	{111,101} = 1x1	Static 1-hazard

6-Valued Logic for Static Hazard Analysis

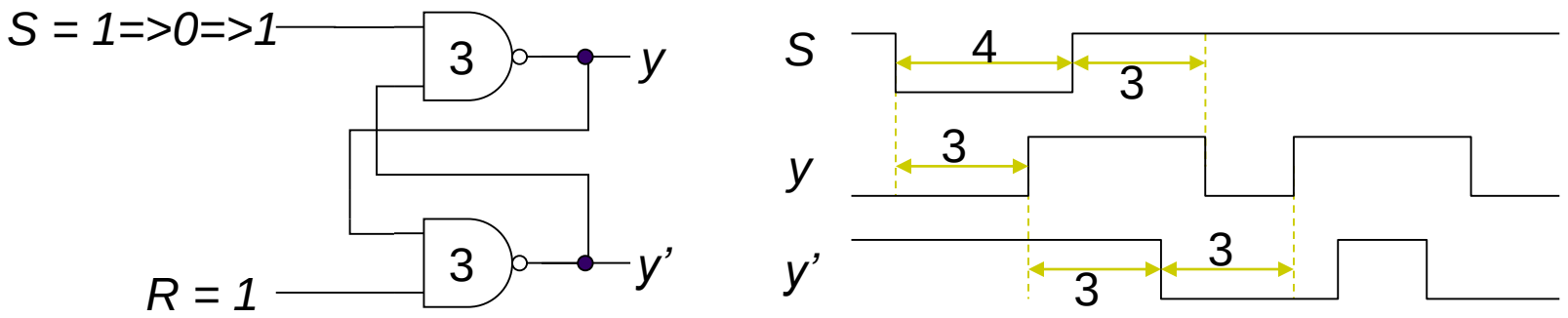


AND	0	1	R	F	0*	1*
0	0	0	0	0	0	0
1	0	1	R	F	0*	1*
R	0	R	R	0*	0*	R
F	0	F	0*	F	0*	F
0*	0	0*	0*	0*	0*	0*
1*	0	1*	R	F	0*	1*

Oscillation



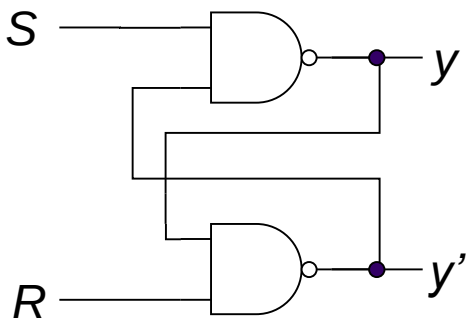
- Oscillating circuits will result in repeated scheduling & processing of the same sequence of events

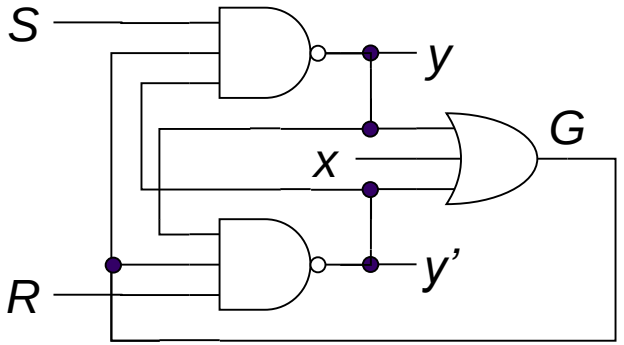


- Oscillation control takes appropriate action upon detection of oscillation

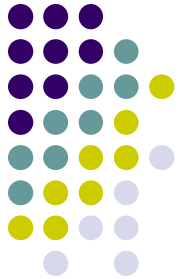
Local Oscillation Control

- identify conditions that causes oscillations in specific sub-circuits, e.g., latches, flip-flops
  - For an oscillating latch, the appropriate corrective action is to set *y* = *y'* = x (unknown)
- Oscillation control via modeling
  - Example: when *y*=*y'*=0 (oscillation condition, also implying S=R=1), *G* = x causes *y* = *y'* = x and stops oscillation



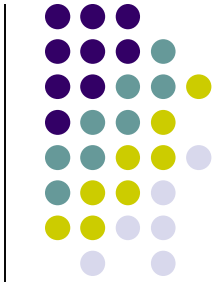


Global Oscillation Control



- Detection of global oscillation is computationally infeasible
  - Requires detecting cyclic sequences of values for any signal in the circuit
- A typical procedure is to count the number of events occurring after any primary input change
  - Oscillation is “assumed” if the number exceeds the specified limit

Simulation Engines



- Motivation
  - Logic simulation is time consuming.
- Simulation engines are special-purpose hardware for speeding up logic simulation.
  - Usually attached to a general-purpose host computer through, for example, VME/PCI bus.
  - FPGA-based logic emulation
- Use parallel and/or distributed processing architectures.