Week 3: Course Material

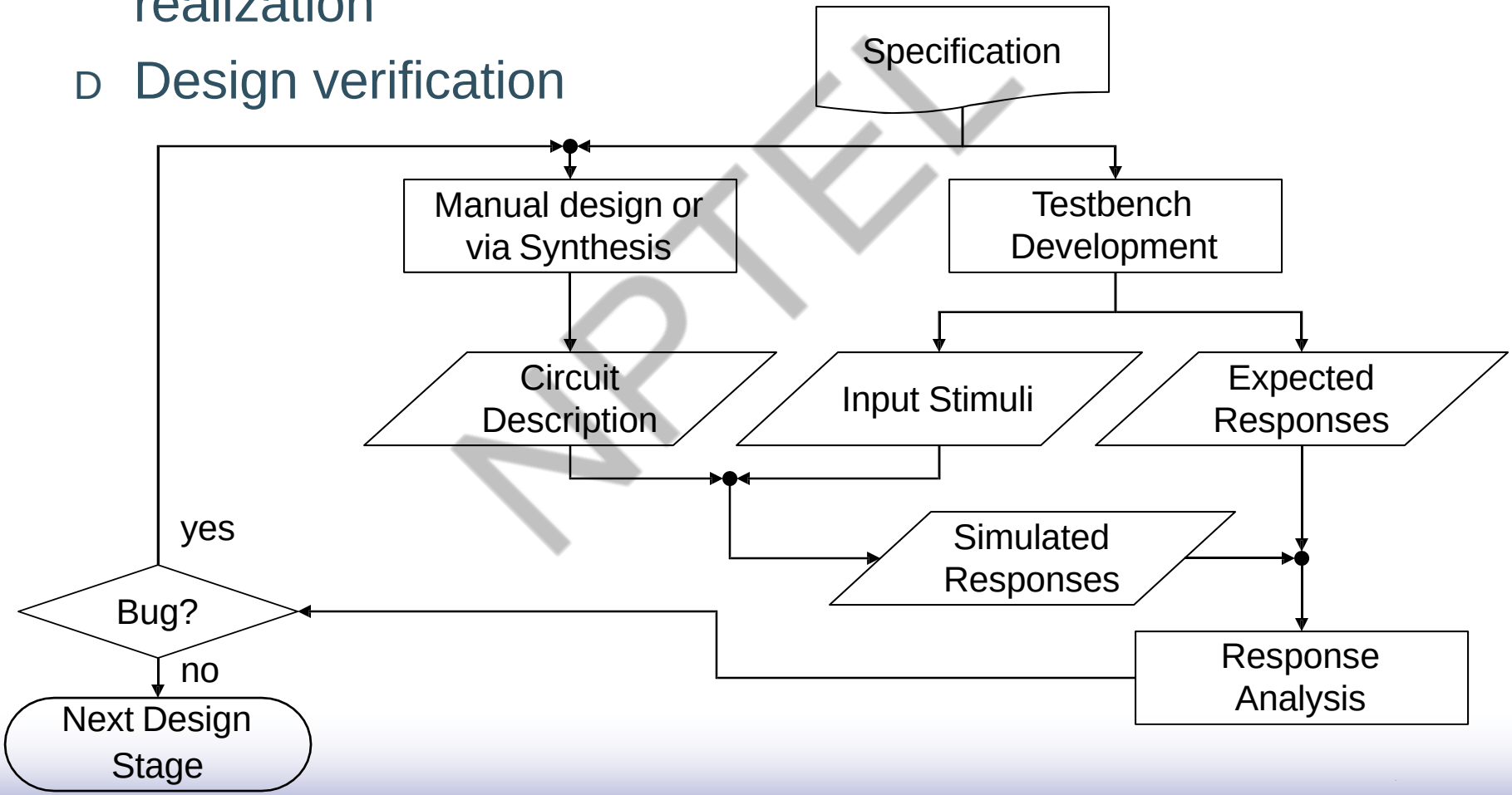
Logic and Fault Simulation
Lecture 11

Logic and Fault Simulation

- **Introduction**
- Simulation models
- Logic simulation
- Fault simulation
- Concluding remarks

Logic Simulation

- Predict the behavior of a design prior to its physical realization
- Design verification



Fault Simulation

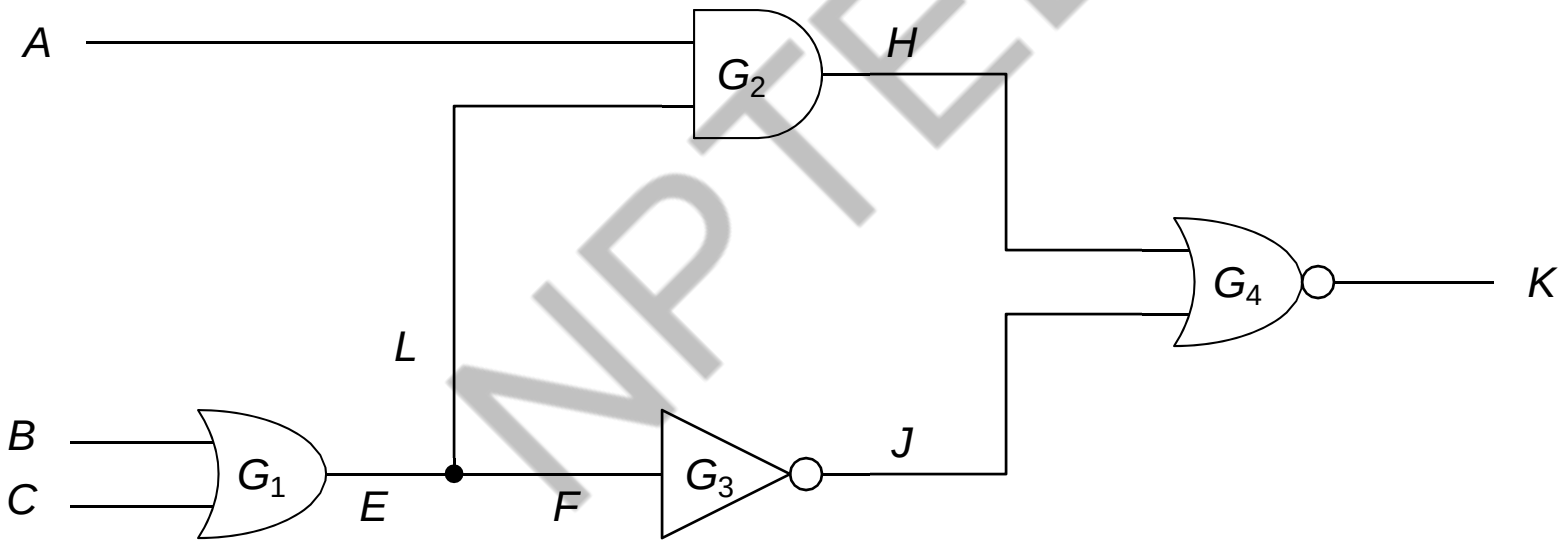
- Predicts the behavior of faulty circuits
  - As a consequence of inevitable fabrication process imperfections
- An important tool for test and diagnosis
  - Estimate fault coverage
  - Fault simulator
  - Test compaction
  - Fault diagnosis

Logic and Fault Simulation

- Introduction
- **Simulation models**
- Logic simulation
- Fault simulation
- Concluding remarks

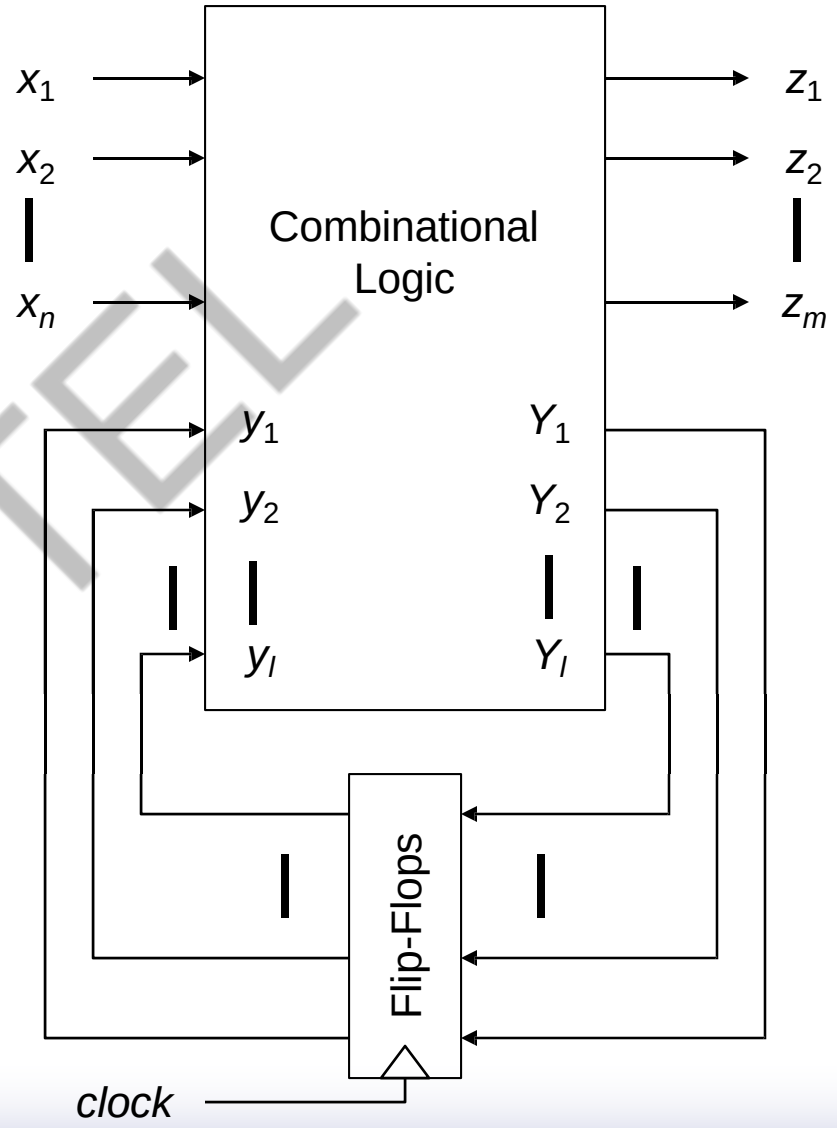
Gate-Level Network

- The interconnections of logic gates



Sequential Circuits

- The outputs depend on both the current and past input values



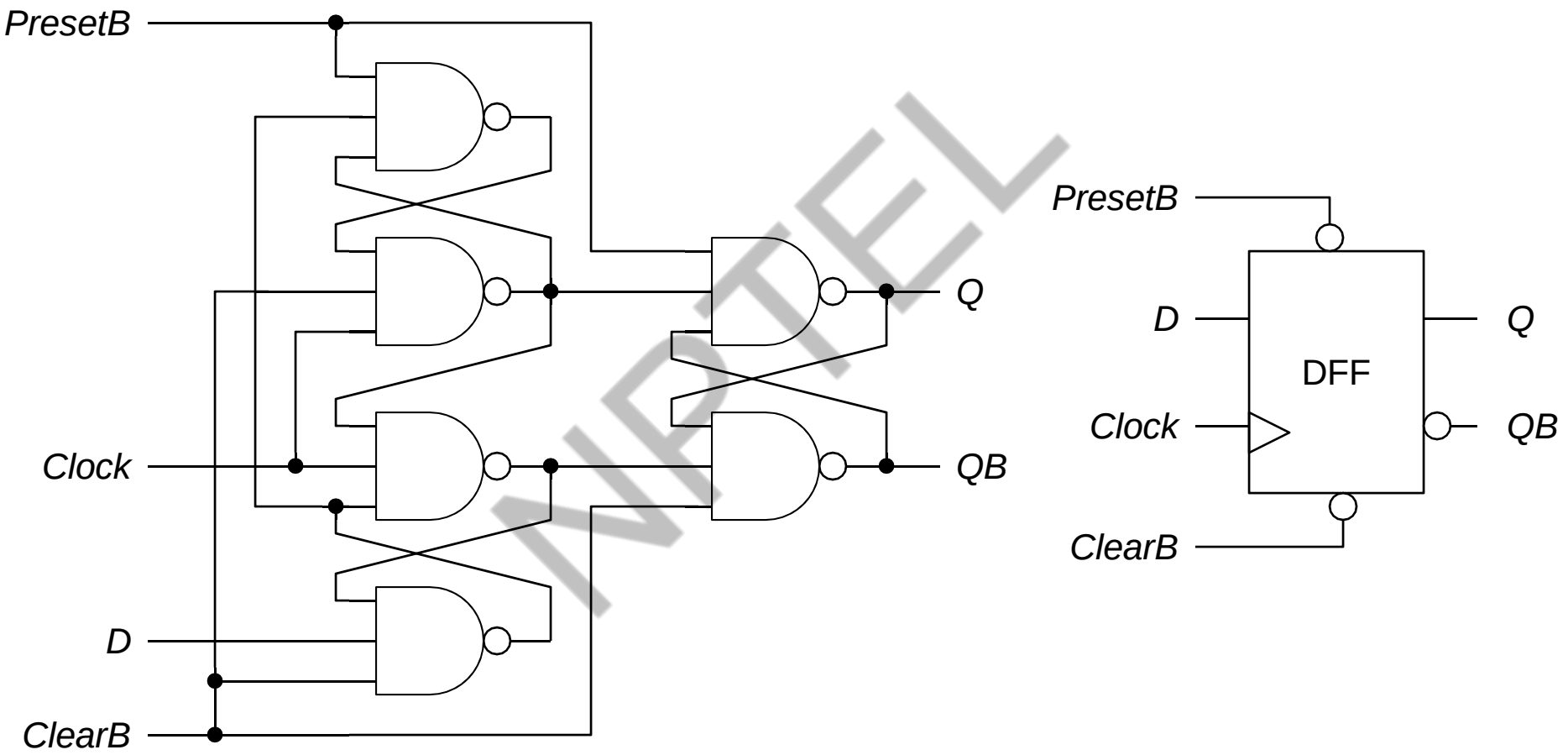
x_i: primary input (PI)

z_i: primary output (PO)

y_i: pseudo primary input (PPI)

Y_i: pseudo primary output (PPO)

A Positive Edge-Triggered D-FF



Logic Symbols

- The most commonly used are 0, 1, *u* and *Z*
- 1 and 0
  - *true* and *false* of the two-value Boolean algebra
- *u*
  - Unknown logic state (maybe 1 or 0)
- *Z*
  - High-impedance state
  - Not connected to V_{dd} or ground

Ternary Logic

- Three logic symbols: 0, 1, and *u*

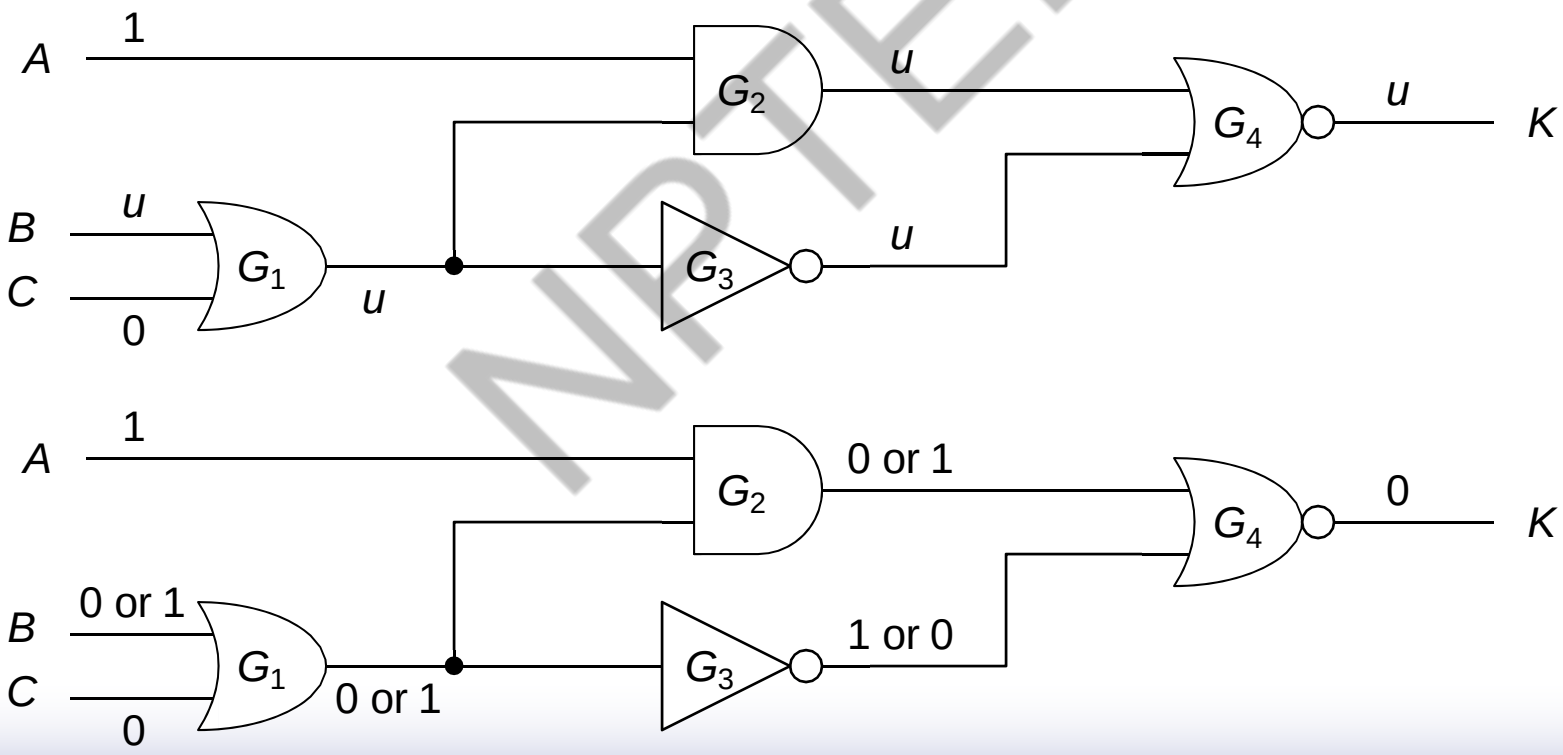
AND	0	1	*u*
0	0	0	0
1	0	1	*u*
u	0	*u*	*u*

OR	0	1	*u*
0	0	1	*u*
1	1	1	1
u	*u*	1	*u*

NOT	0	1	*u*
	1	0	*u*

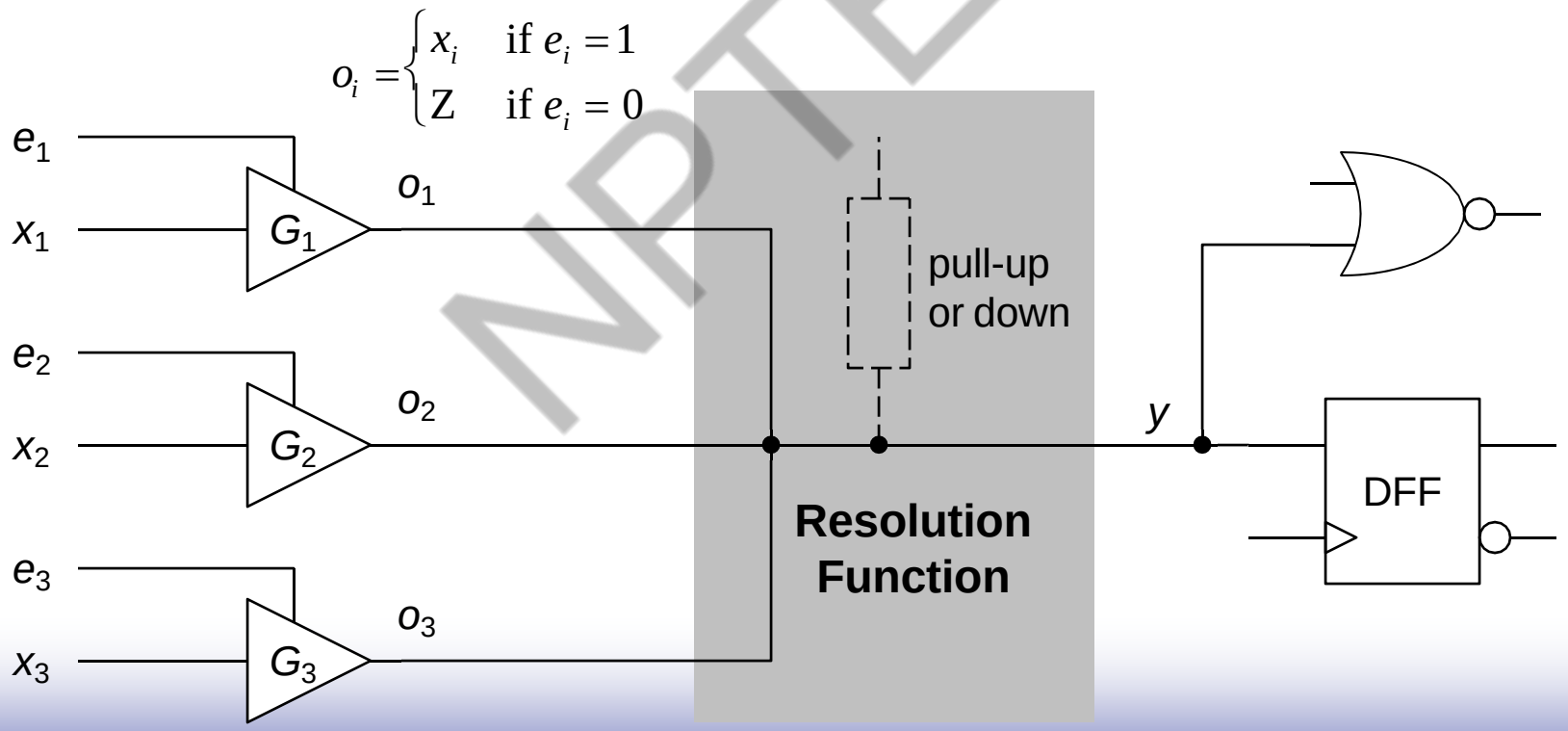
Information Loss of Ternary Logic

- Simulation based on ternary logic is pessimistic
- A signal may be reported as unknown when its value can be uniquely determined as 0 or 1



High-Impedance State Z

- Tri-state gates permit several gates to time-share a common wire, called bus
- A signal is in high-impedance state if it is connected to neither V_{dd} nor ground



Resolving Bus Conflict

- Bus conflict occurs if at least two drivers drive the bus to opposite binary values
- To simulate tri-state bus behavior, one may insert a resolution function for each bus wire
  - May report only the occurrence of bus conflict
  - May utilize multi-valued logic to represent intermediate logic states (including logic signal values and strengths)

Logic Element Evaluation Methods

- Choice of evaluation technique depends on
  - Considered logic symbols
  - Types and models of logic elements
- Commonly used approaches
  - Truth table based
  - Input scanning
  - Input counting
  - Parallel gate evaluation

Truth Table Based Gate Evaluation

- The most straightforward and easy 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
    - $m = \lceil \log_2 k \rceil$
    - Table indexed by mn-bit words

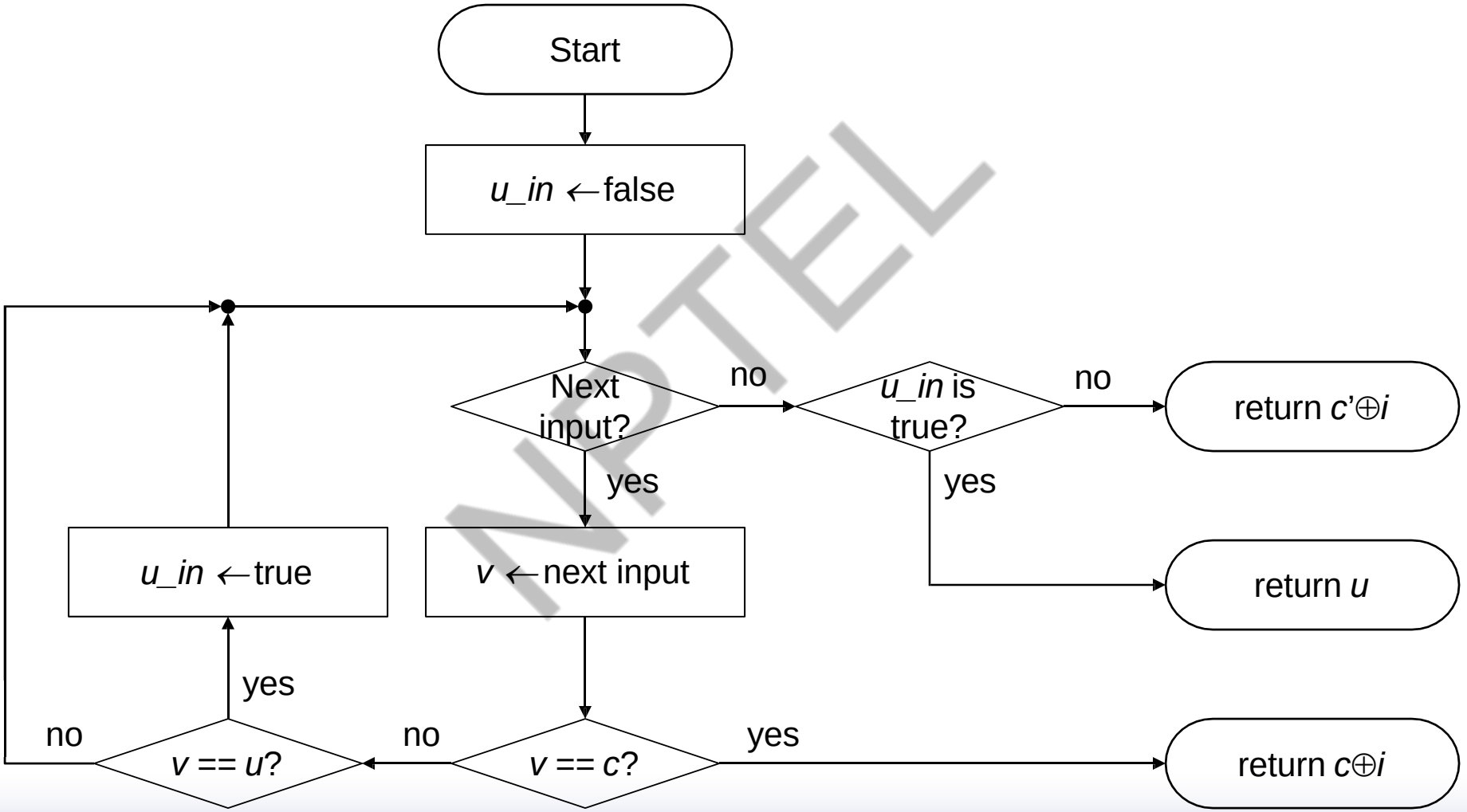
Input Scanning

- The gate output can be determined by the types of inputs
  - If any of the inputs is the controlling value, the gate output is $c \oplus i$
  - Otherwise, if any of the inputs is u, the gate output is u
  - Otherwise, the gate output is $c' \oplus i$

Table 3.2: The c (controlling) and i (inversion) values of basic gates

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

Input Scanning - cont'd

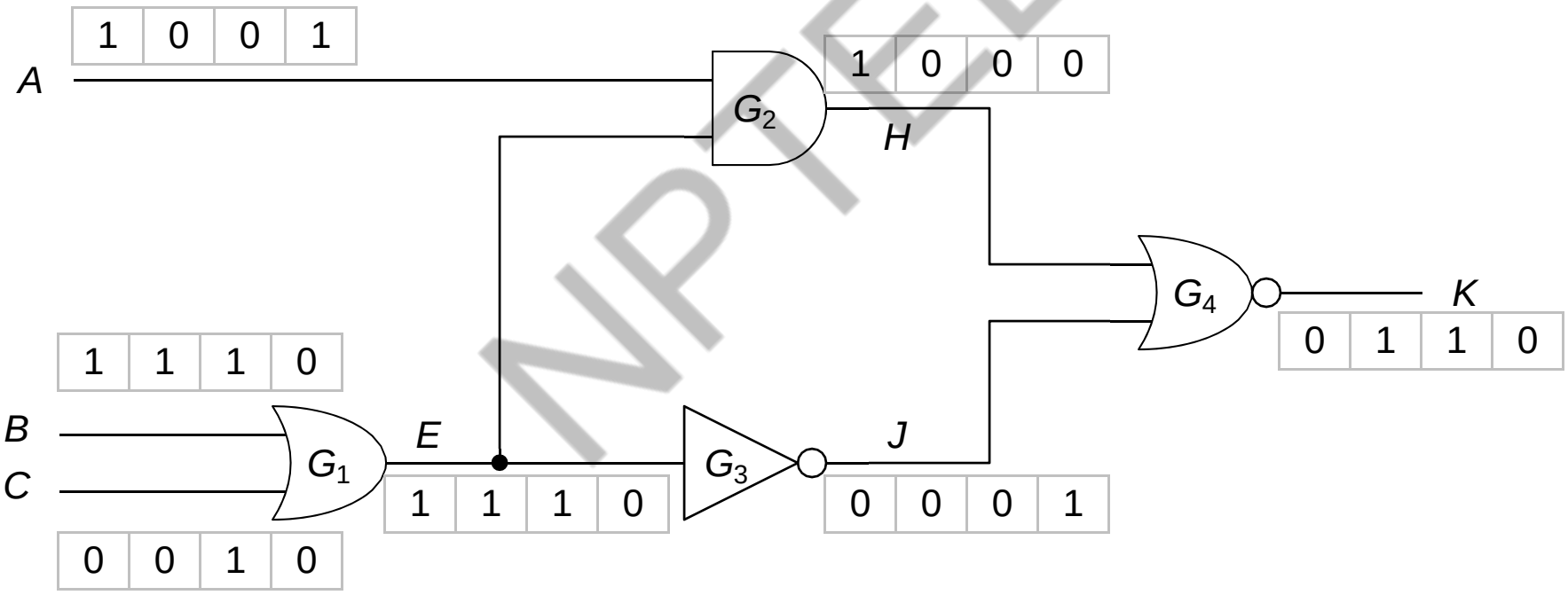


Input Counting

- Keep the counts of controlling and unknown inputs
  - *c_count*: the number of controlling inputs
  - *u_count*: the number of unknown inputs
- Update counts during logic simulation
  - Example:
    One input of a NAND switches from 0 to *u*
    - *c_count* --
    - *u_count* ++
- Same rules as input scanning used to evaluate gate outputs

Parallel Gate Evaluation

- Exploit the inherent concurrency in the host computer
  - A 32-bit computer can perform 32 logic operations in parallel



Logic and Fault Simulation (contd.)

Lecture 12

Multi-Valued Parallel Gate Evaluation

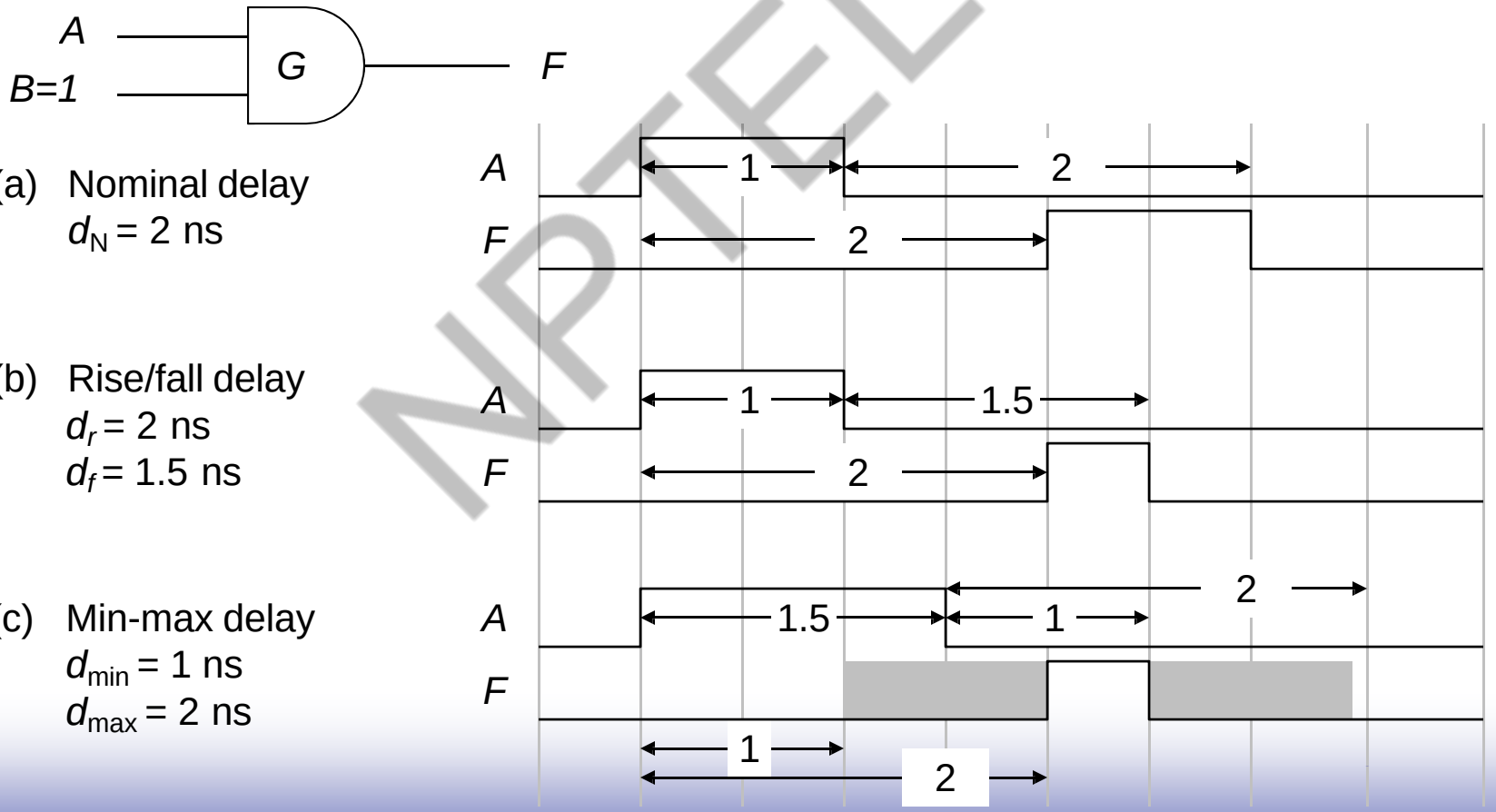
- Use ternary logic as example
  - Assume
    - w-bit wide word
    - Symbol encoding: v_0 = (00), v_1 = (11), v_u = (01)
  - Associate with each signal X two words, X_1 and X_2
    - X_1 stores the first bits and X_2 the second bits of the w copies of the same signal
  - AND and OR operations are realized by applying the same bitwise operations to both words
    - C = OR(A,B) ==> C_1 = OR(A_1,B_1) and C_2 = OR(A_2,B_2)
  - Complement requires inversion
    - C = NOT(A) ==> C_1 = NOT(A_2) and C_2 = NOT(A_1)

Timing Models

- Transport delay
- Inertial delay
- Wire delay
- Function element delay model

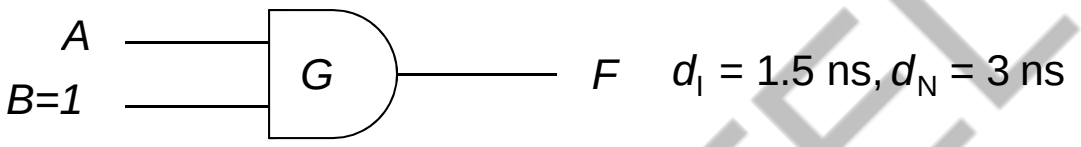
Transport Delay

- The time duration it takes for the effect of gate input changes to appear at gate outputs



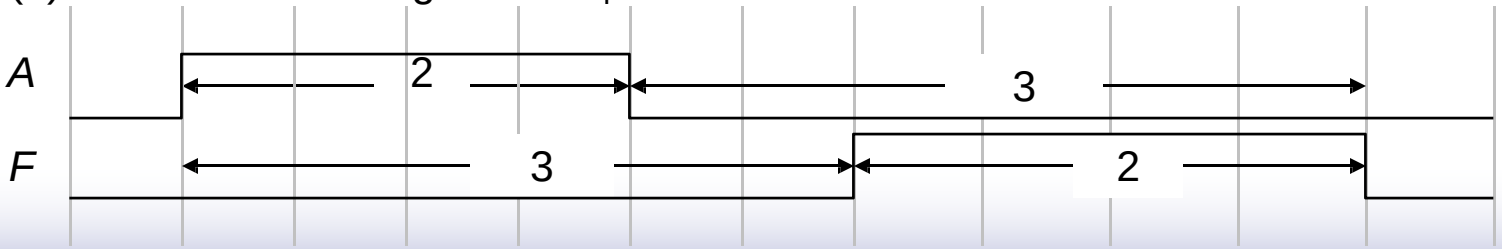
Inertial Delay

- The minimum input pulse duration necessary for the output to switch states



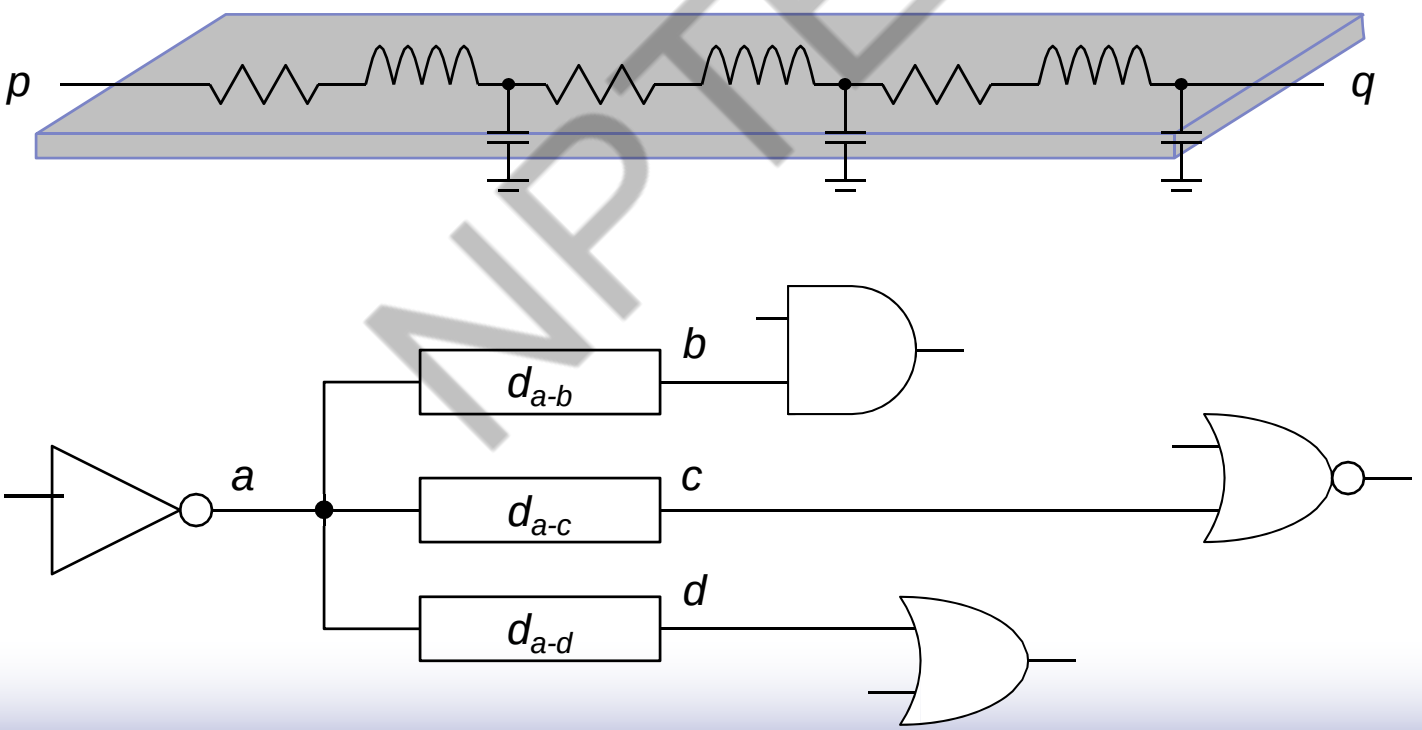
(a) Pulse duration less than d_I

(b) Pulse duration longer than d_I



Wire Delay

- Wires are inherently resistive and capacitive
- It takes finite time for a signal to propagate along a wire



Functional Element Delay Model

- For more complicated functional elements like flip-flops

Table 3.3: The D flip-flop I/O delay model

Input condition				Present state	Outputs		Delays (ns)		
D	*Clock*	*PresetB*	*ClearB*	*q*	*Q*	*QB*	to *Q*	to *QB*	Comments
X	*X*	↓	0	0	↑	↓	1.6	1.8	Asynchronous preset
X	*X*	0	↓	1	↓	↑	1.8	1.6	Asynchronous clear
1	↑	0	0	0	↑	↓	2	3	*Q*: 0→1
0	↑	0	0	1	↓	↑	3	2	*Q*: 1→0

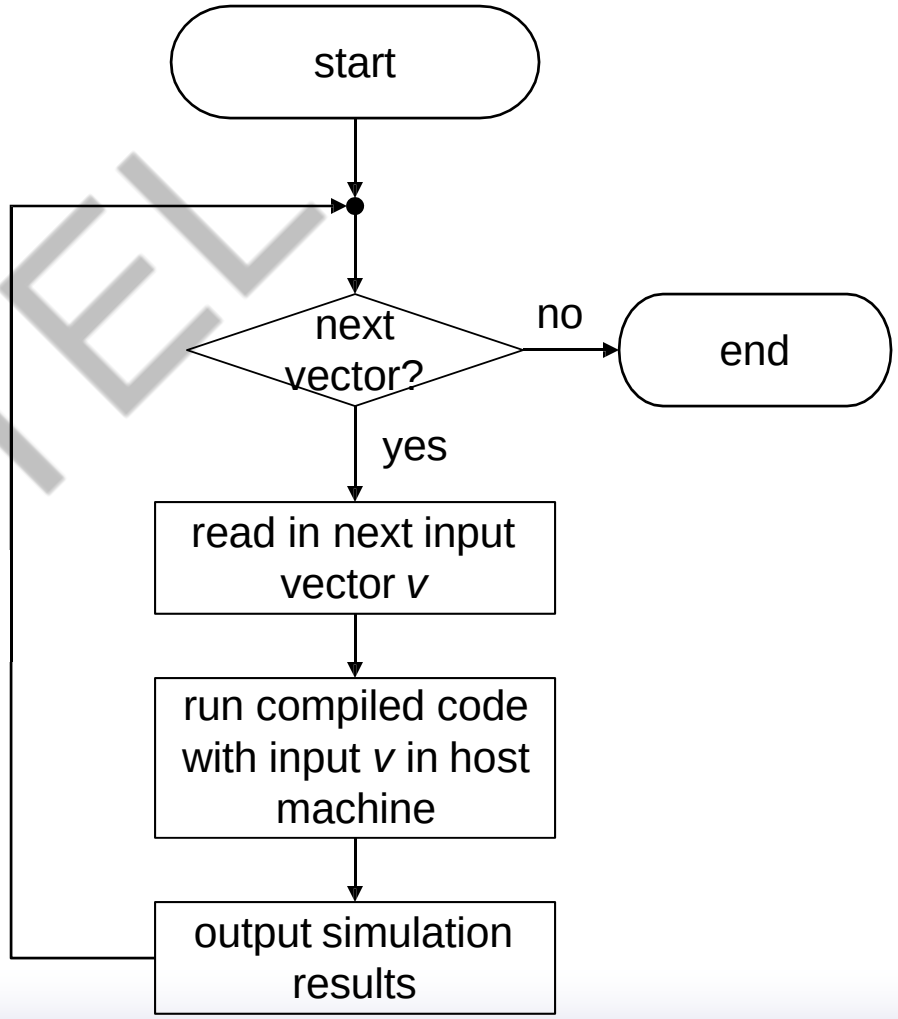
X Š indicates donÕtcare

Logic and Fault Simulation

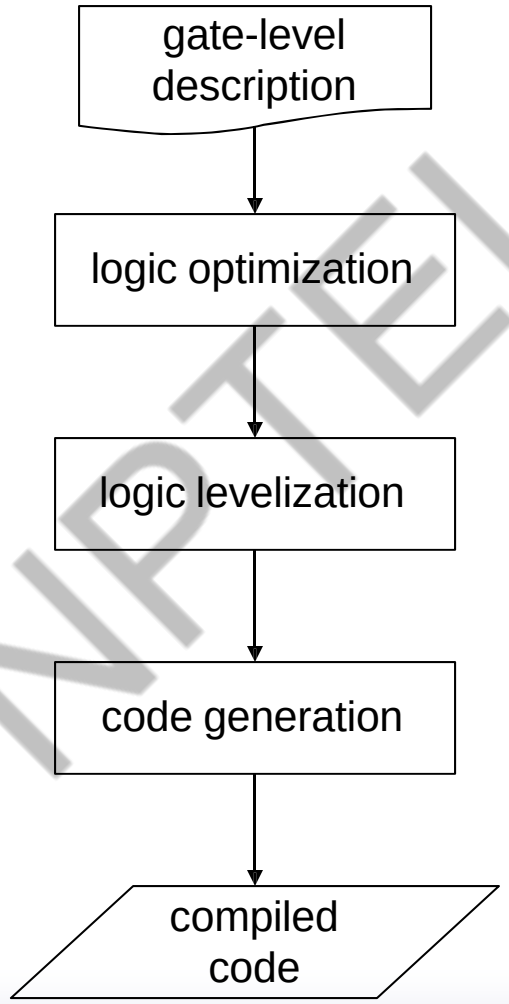
- Introduction
- Simulation models
- **Logic simulation**
- Fault simulation
- Concluding remarks

Compiled Code Simulation

- Translate the logic network into a series of machine instructions that model the gate functions and interconnections

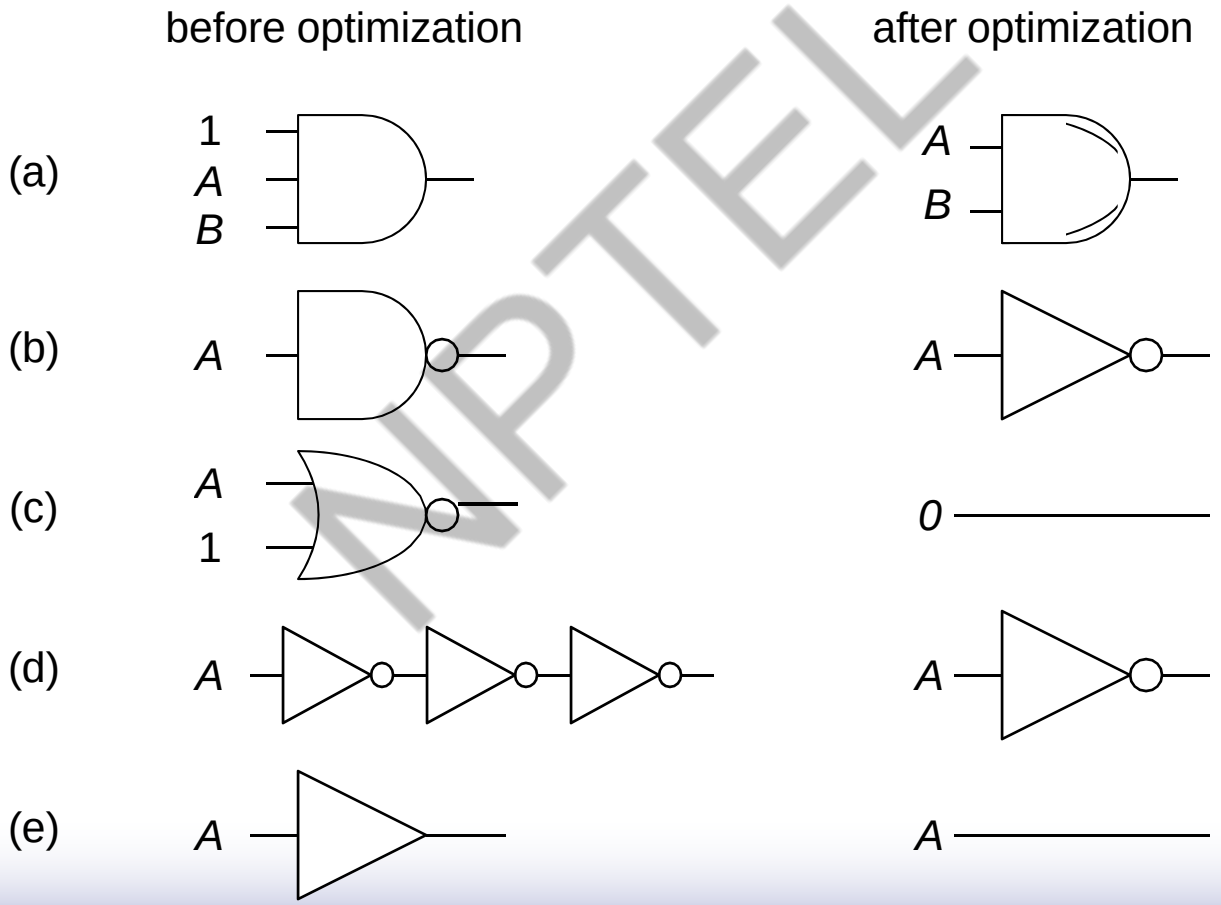


Compiled Code Generation Flow



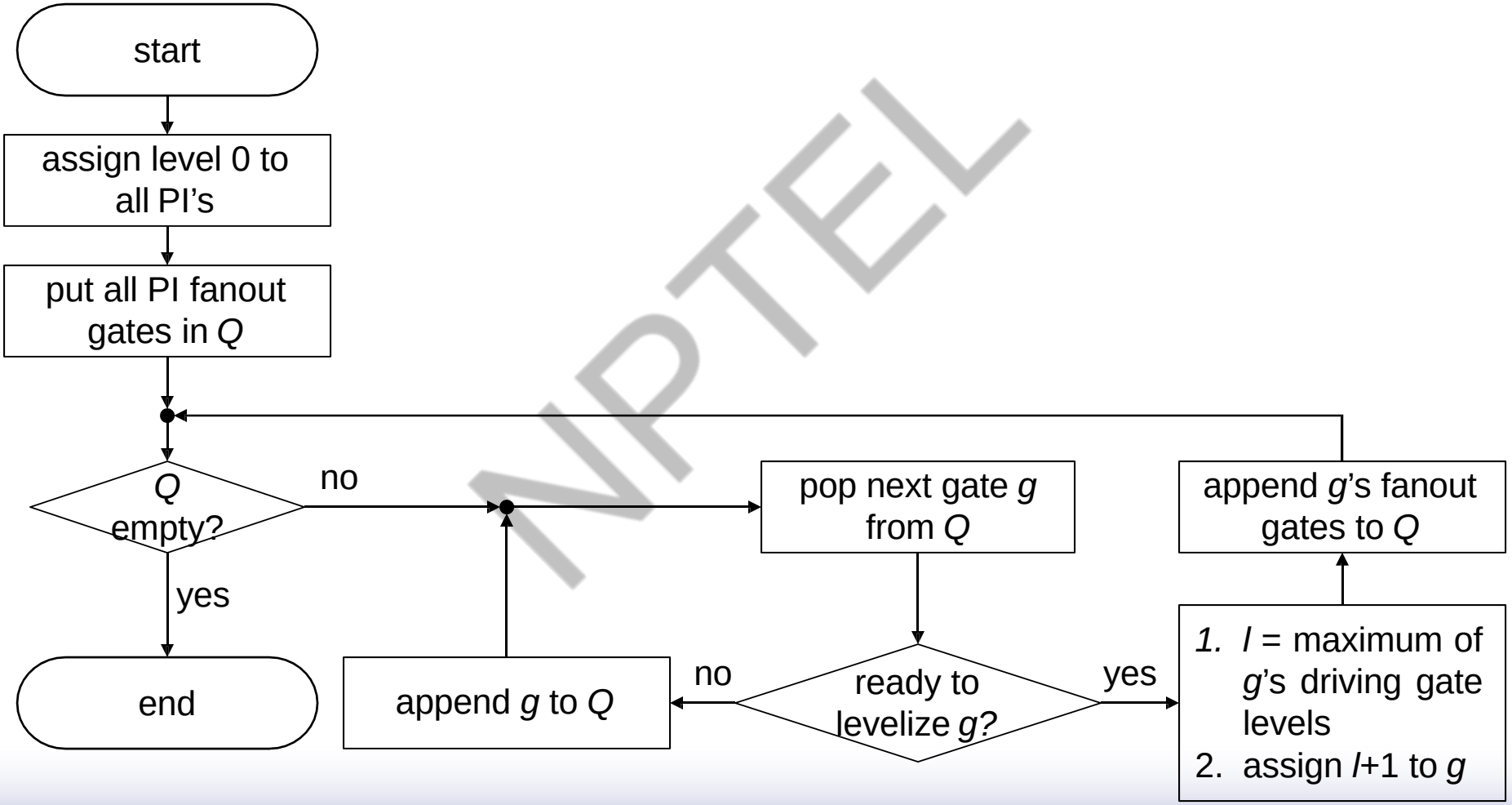
Logic Optimization

- Enhance the simulation efficiency



Logic Levelization

- Determine the order of gate evaluations



Example

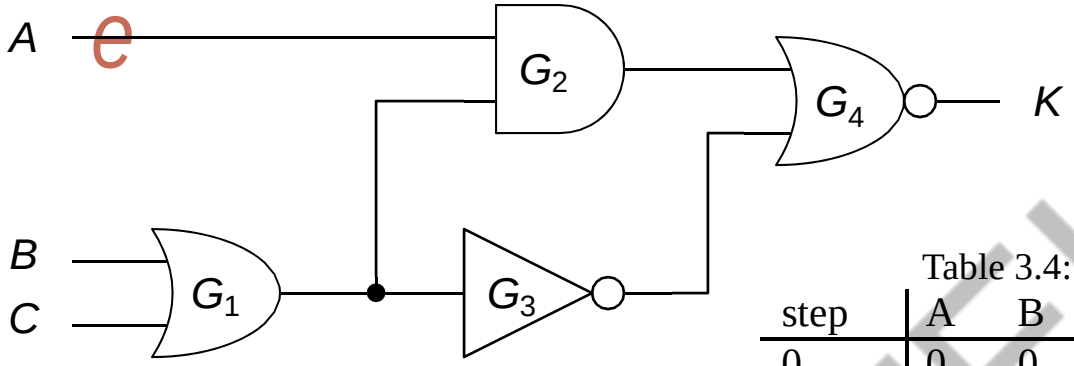


Table 3.4: The levelization process of circuit

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

Code Generation

- High-level programming language source code
  - Easier to debug
  - Can be ported to any target machine that has the compiler
  - Limited in applications due to long compilation times
- Native machine code
  - Generate the target machine code directly
  - Higher simulation efficiency
  - Not as portable

Code Generation - cont'd

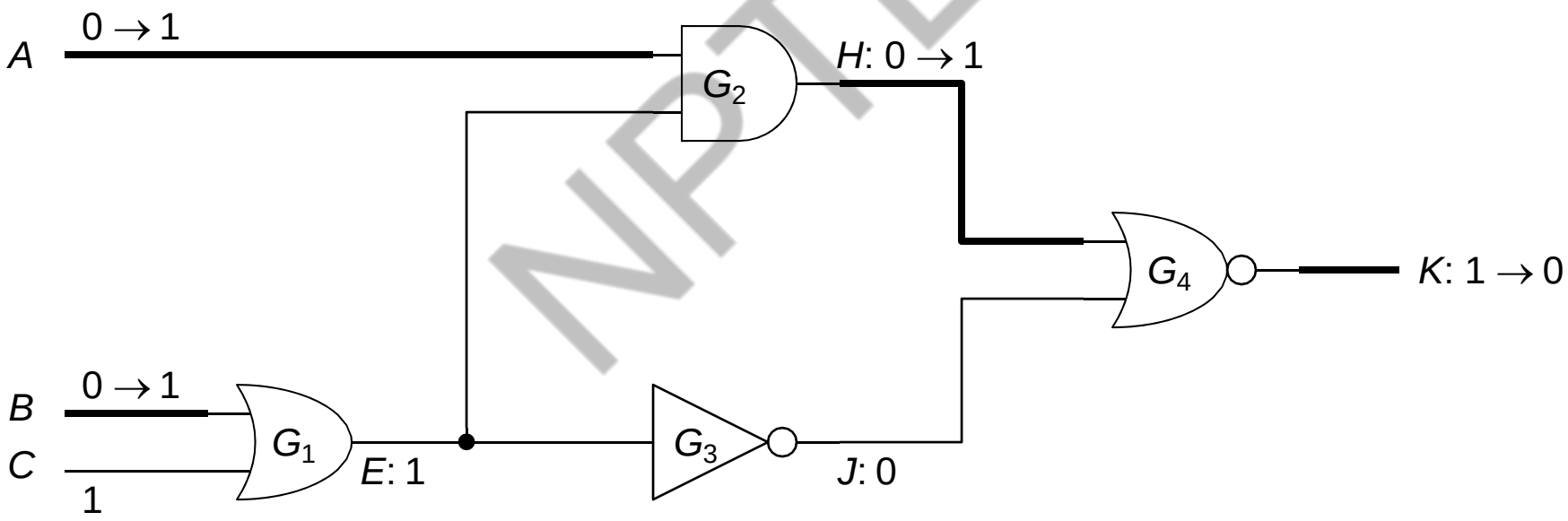
- Interpreted code
  - The target machine is a software emulator
  - The codes are interpreted and executed one at a time
  - Best portability and maintainability
  - Reduced performance

Logic and Fault Simulation (contd.)

Lecture 13

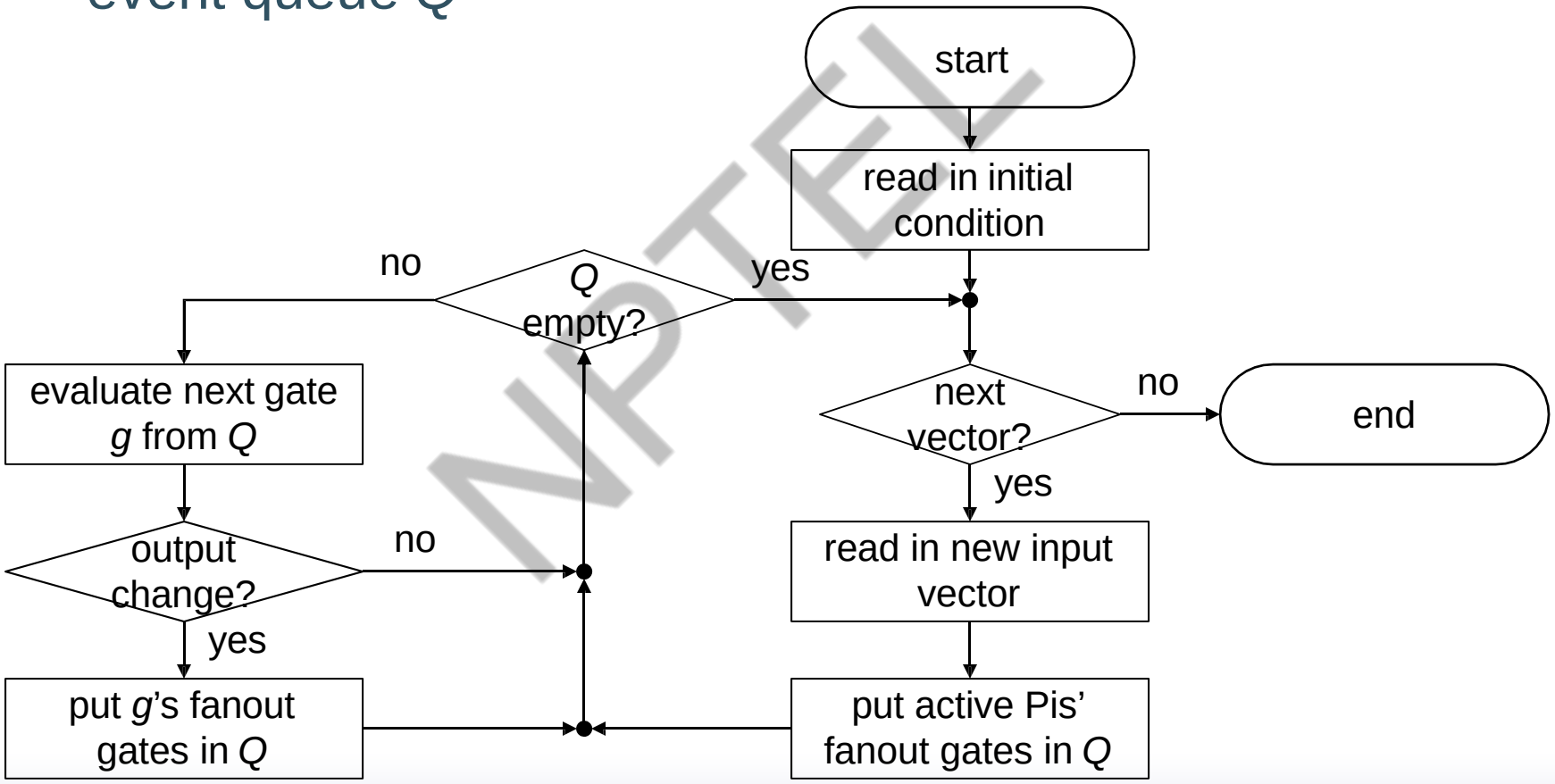
Event-Driven Simulation

- Event: the switching of a signal's value
- An event-driven simulator monitors the occurrences of events to determine which gates to evaluate



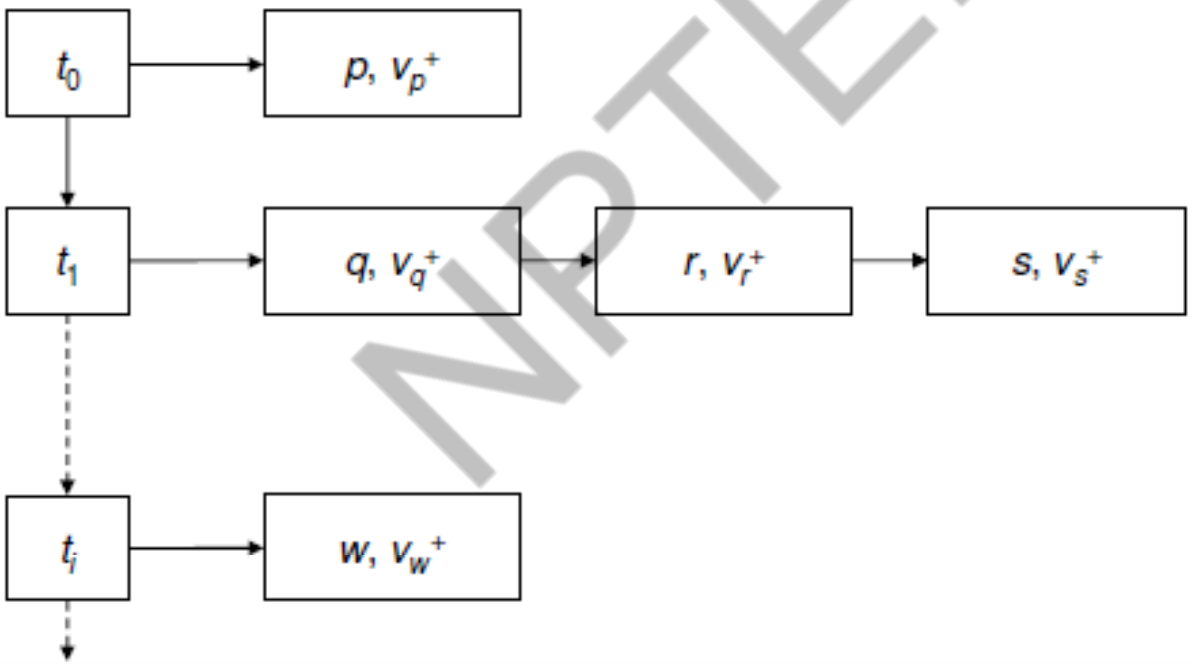
Zero-Delay Event-Driven Simulation

- Gates with events at their inputs are places in the event queue *Q*

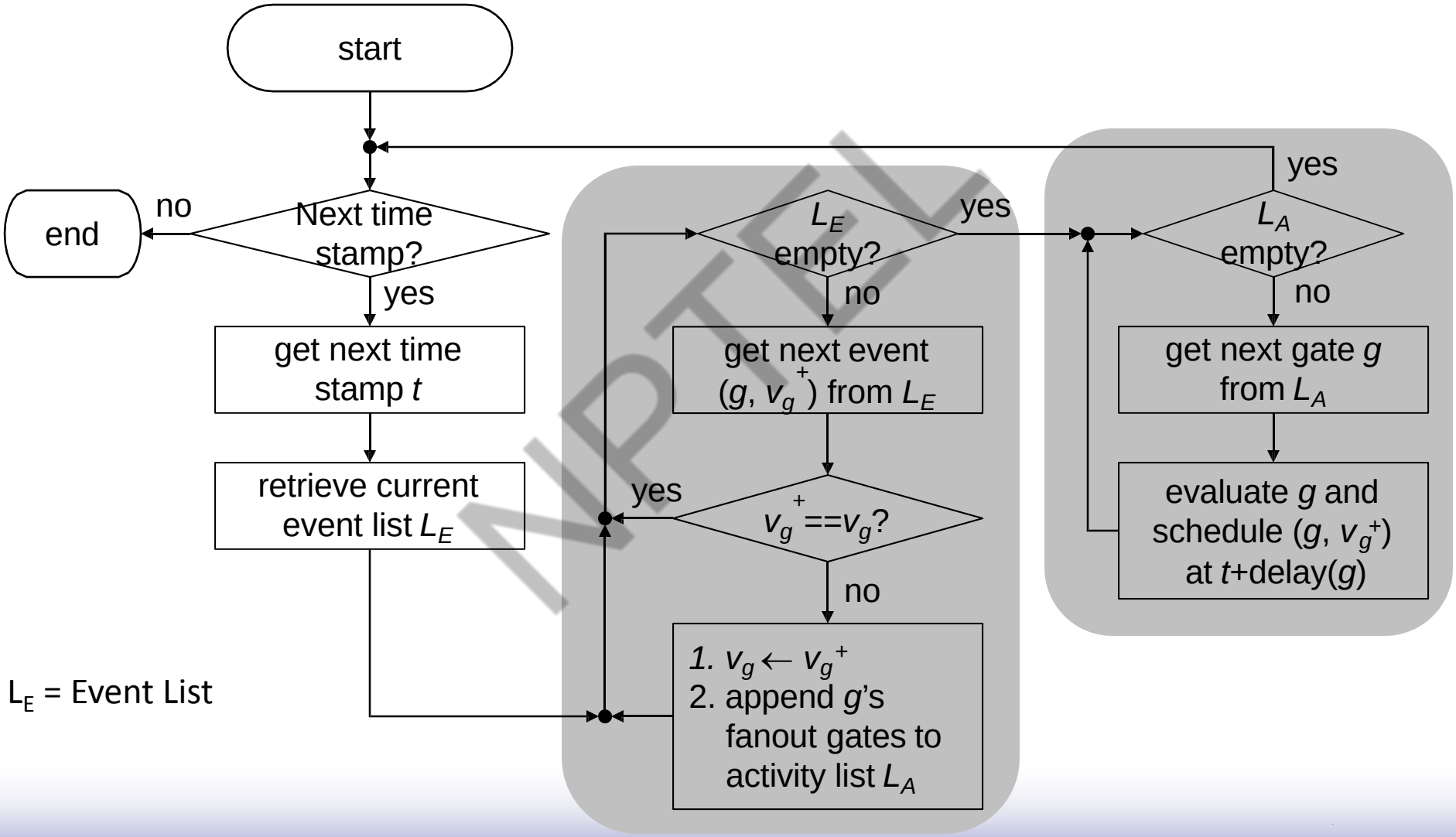


Nominal-Delay Event-Driven Simulation

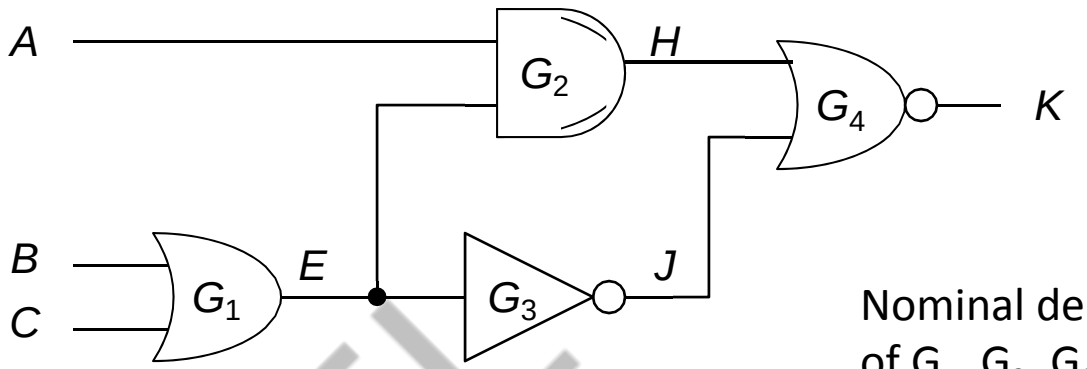
- Need a smarter scheduler than the event queue
  - Not only which gates but also when to evaluate



Two-Pass Event-Driven Simulation



Example

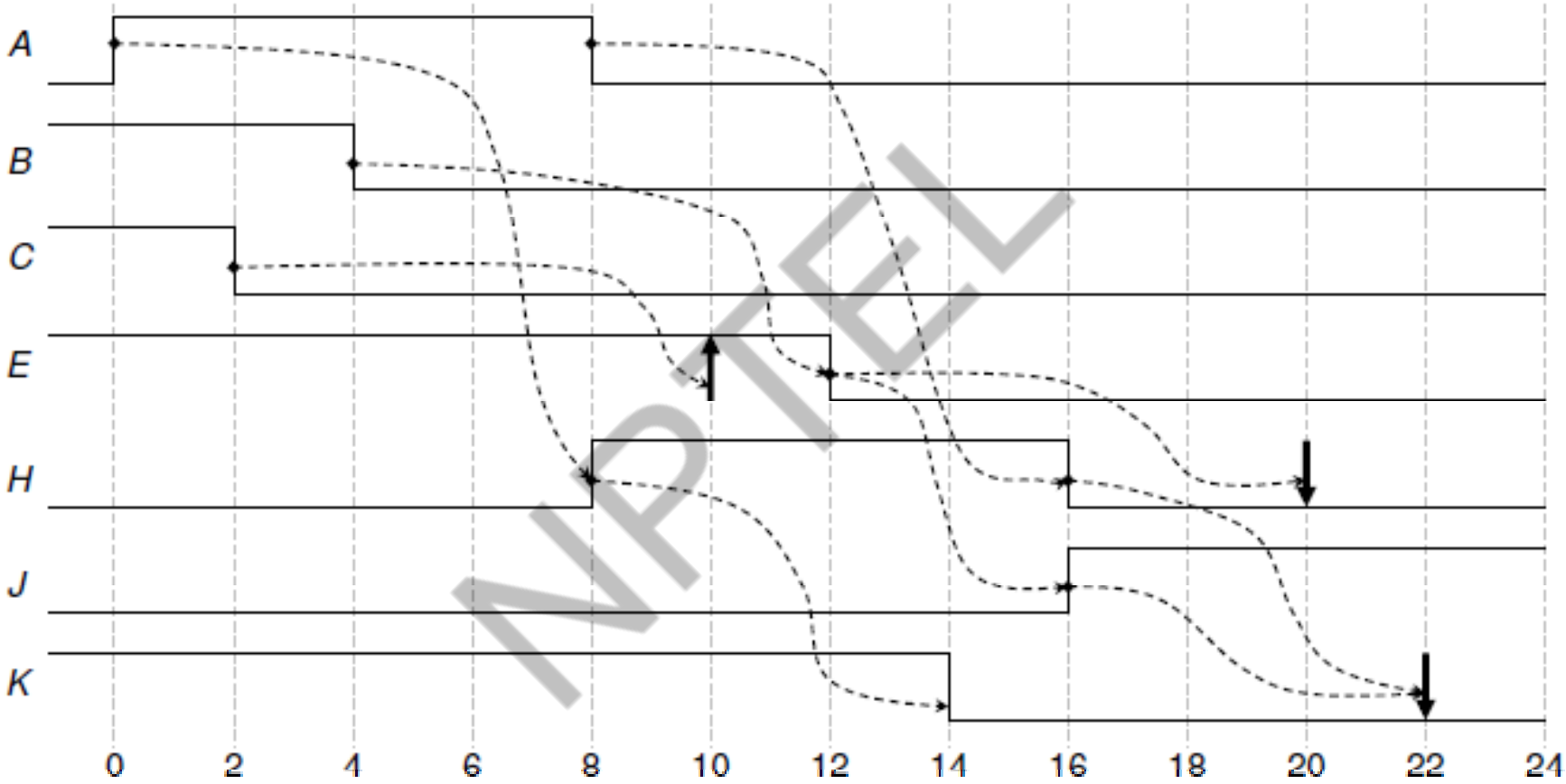


Nominal delays of G_1, G_2, G_3, G_4 are 8, 8, 4, 6 ns

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)}		

Example - cont'd

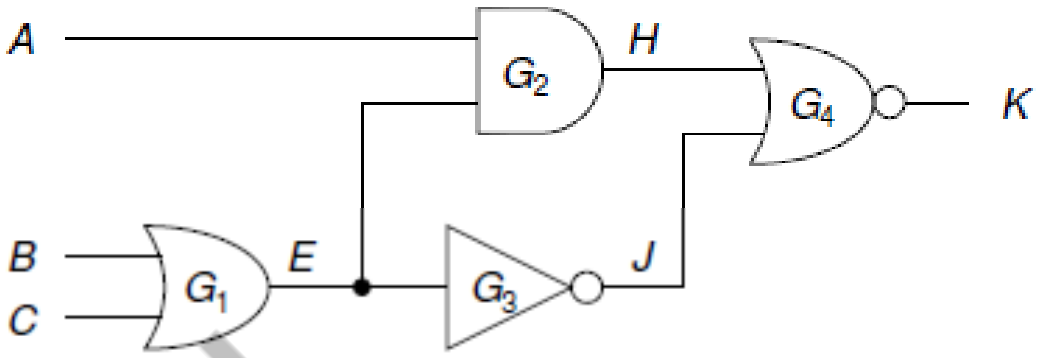


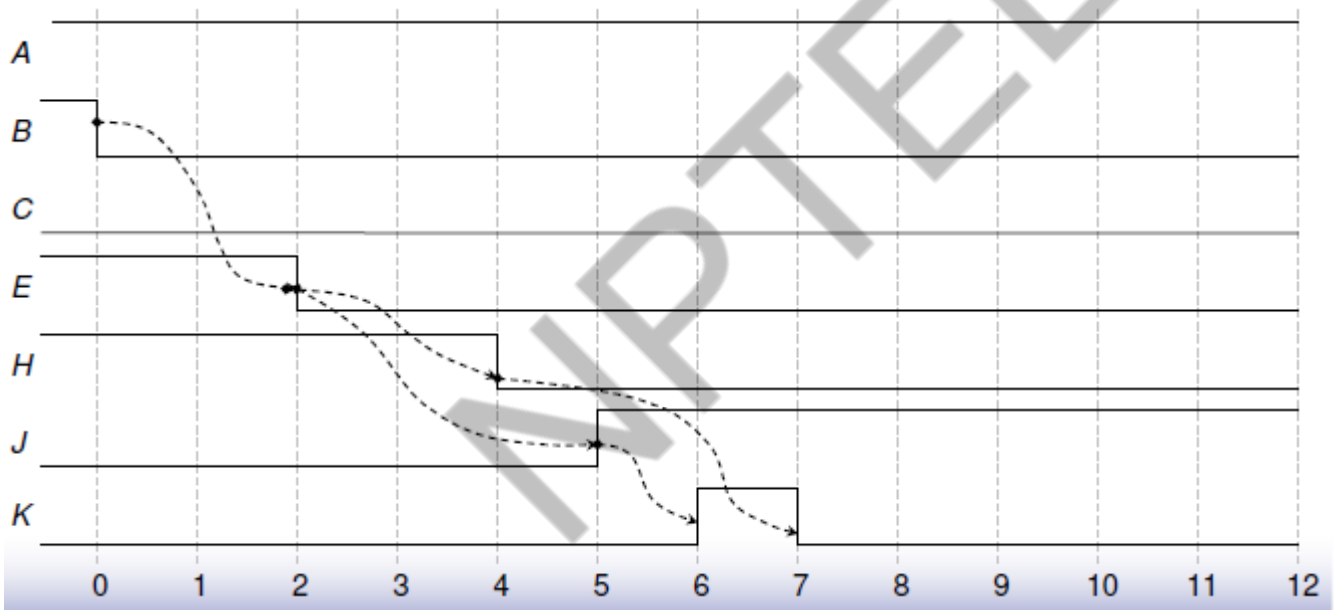
Compiled-Code vs. Event-Driven Simulation

- Compiled-code
  - Cycle-based simulation
  - High switching activity circuits
  - Parallel simulation
  - Limited by compilation times
- Event-driven
  - Implementing gate delays and detecting hazards
  - Low switching activity circuits
  - More complicated memory management

Hazards

- Unwanted transient pulses or glitches





INV delay = 3ns
Others = 2ns

Types 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 1-hazard Static 0-hazard Dynamic 1-hazard Dynamic 0-hazard

Static Hazard Detection

- Let $V^1 = v_1^1 v_2^1 \cdots v_n^1$ and $V^2 = v_1^2 v_2^2 \cdots v_n^2$ be two consecutive input vectors
- Add a new vector $V^+ = v_1^+ v_2^+ .... v_n^+$ according to the following rule

$$v_i^+ = \begin{cases} v_i^1 & \text{if } v_i^1 = v_i^2 \\ u & \text{if } v_i^1 \neq v_i^2 \end{cases}$$

- Simulate the $V^1V^+V^2$ sequence using ternary logic
- Any signal that is 1*u*1 or 0*u*0 indicates the possibility of a static hazard.

Multi-Valued Logic for Hazard Detection

- 6-valued logic for static hazard detection
- 8-valued logic for dynamic hazard detection
- Worst case analysis

Table 3.6: Multi-valued logic for hazard detection

Symbol	Interpretation	6-valued logic	8-valued logic
0	Static 0	{000}	{0000}
1	Static 1	{111}	{1111}
R	Rise transition	{001,011}=0*u*1	{0001,0011,0111}
F	Fall transition	{100,110}=1*u*0	{1110,1100,1000}
0*	Static 0-hazard	{000,010}=0*u*0	{0000,0100,0010,0110}
1*	Static 1-hazard	{111,101}=1*u*1	{1111,1011,1101,1001}
R*	Dynamic 1-hazard		{0001,0011,0101,0111}
F*	Dynamic 0-hazard		{1000,1010,1100,1110}

Logic and Fault Simulation

- Introduction
- Simulation models
- Logic simulation
- **Fault simulation**
- Concluding remarks

Logic and Fault Simulation (contd.)

Lecture 14

Fault Simulation

- Introduction
- Serial Fault Simulation
- Parallel Fault Simulation
- Deductive Fault Simulation
- Concurrent Fault Simulation
- Differential Fault Simulation
- Fault Detection
- Comparison of Fault Simulation Techniques
- Alternative to Fault Simulation
- Conclusion

Introduction

What is fault simulation?

- Given
  - A circuit
  - A set of test patterns
  - A fault model
- Determine
  - Faulty outputs
  - Undetected faults
  - Fault coverage

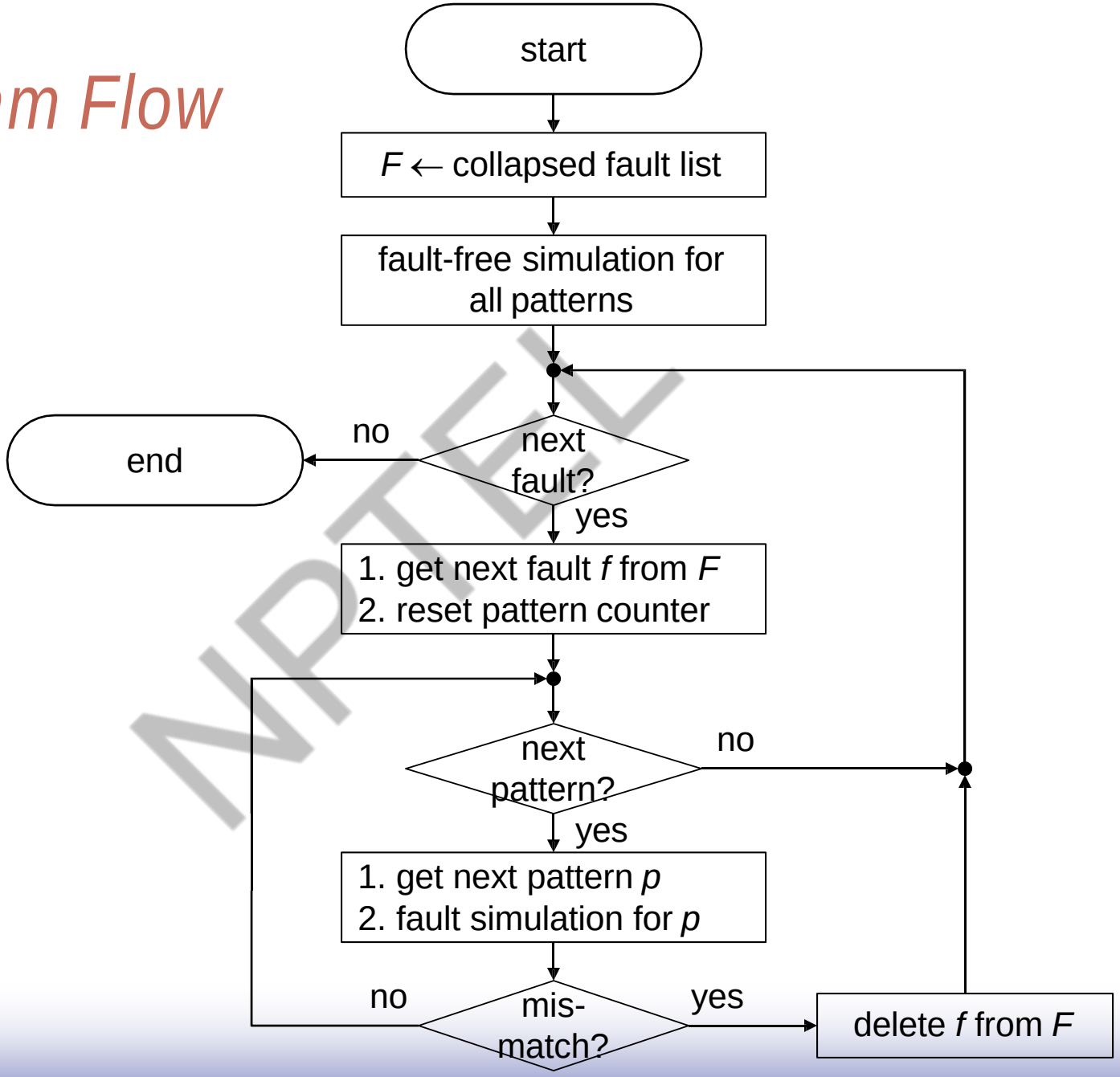
Time Complexity

- Proportional to
  - n: Circuit size, number of logic gates
  - p: Number of test patterns
  - f : Number of modeled faults
- Since *f* is roughly proportional to *n*, the overall time complexity is $O(pn^2)$

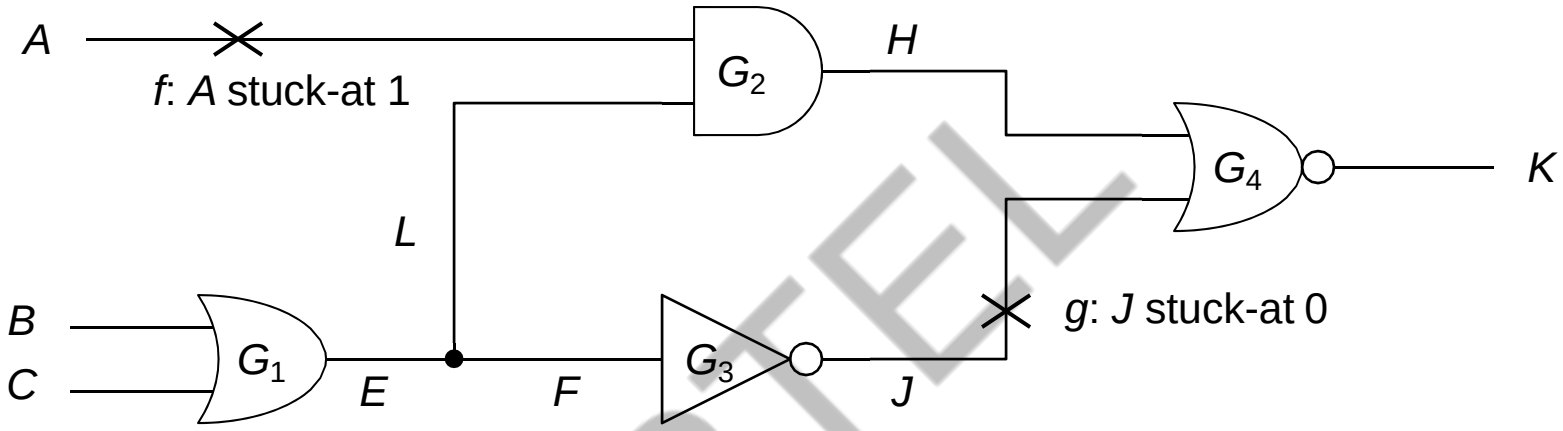
Serial Fault Simulation

- First, perform fault-free logic simulation on the original circuit
  - Good (fault-free) response
- For each fault, perform fault injection and logic simulation
  - *Faulty* circuit response

Algorithm Flow



Example



Pat. #	Input			Internal					Output		
	A	*B*	*C*	*E*	*F*	*L*	*J*	*H*	K_{good}	K_f	K_g
P1	0	1	0	1	1	1	0	0	1	**0**	1
P2	0	0	1	1	1	1	0	0	1	**0**	1
P3	1	0	0	0	0	0	1	0	0	0	**1**

Fault Dropping

- Halting simulation of the detected fault
- Example
  - Suppose we are to simulate P_1, P_2, P_3 in order
  - Fault *f* is detected by P_1
  - Do not simulate *f* for P_2, P_3
- For fault grading
  - Most faults are detected after relatively few test patterns have been applied
- For fault diagnosis
  - Avoided to obtain the entire fault simulation results

Pro and Con

Advantages

- Easy to implement
- Ability to handle a wide range of fault models

  (stuck-at, delay, Br, …)

Disadvantages

- Very slow

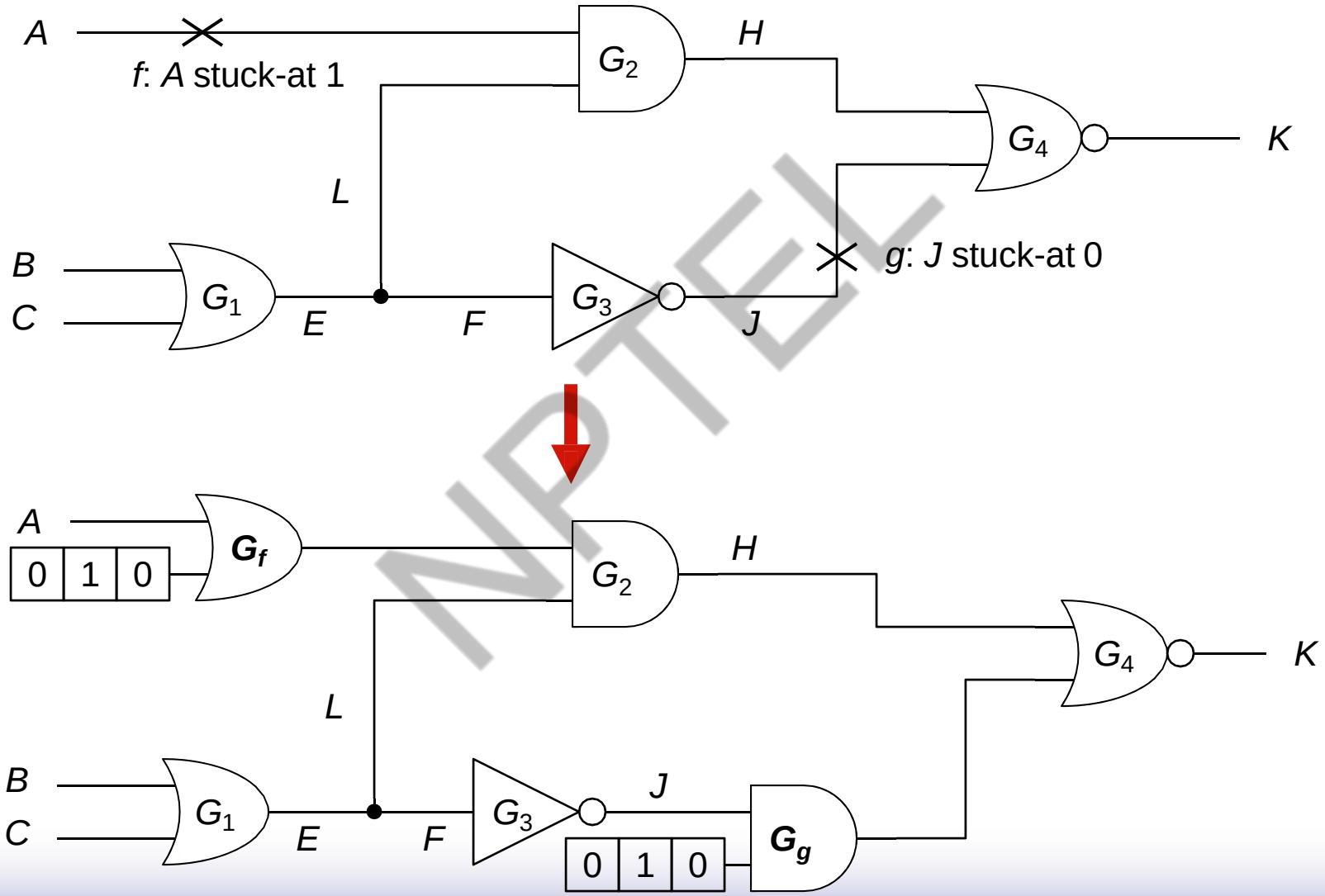
Parallel Fault Simulation

- Exploit the inherent parallelism of bitwise operations
- Parallel fault simulation
  - Parallel in faults
- Parallel pattern fault simulation
  - Parallel in patterns

Parallel Fault Simulation

- Assumption
  - Use binary logic: one bit is enough to store logic signal
  - Use *w*-bit wide data word
- Parallel simulation
  - *w*-1 bit for faulty circuits
  - 1 bit for fault-free circuit
- Process faulty and fault-free circuit in parallel using bitwise logic operations

Fault Injection



Example

Pat #		Input				Internal						Output
		A	A_f	***B***	***C***	***E***	***F***	***L***	***J***	J_g	***H***	***K***
P_1	FF	0	0	1	0	1	1	1	0	0	0	1
	f	**0**	**1**	1	0	1	1	1	0	0	**1**	**0**
	g	0	0	1	0	1	1	1	0	0	0	1
P_2	FF	0	0	0	1	1	1	1	0	0	0	1
	f	0	**1**	0	1	1	1	1	0	0	**1**	**0**
	g	0	0	0	1	1	1	1	0	0	0	1
P_3	FF	1	1	0	0	0	0	0	1	1	0	0
	f	1	1	0	0	0	0	0	1	1	0	0
	g	1	1	0	0	0	0	0	1	**0**	0	**1**

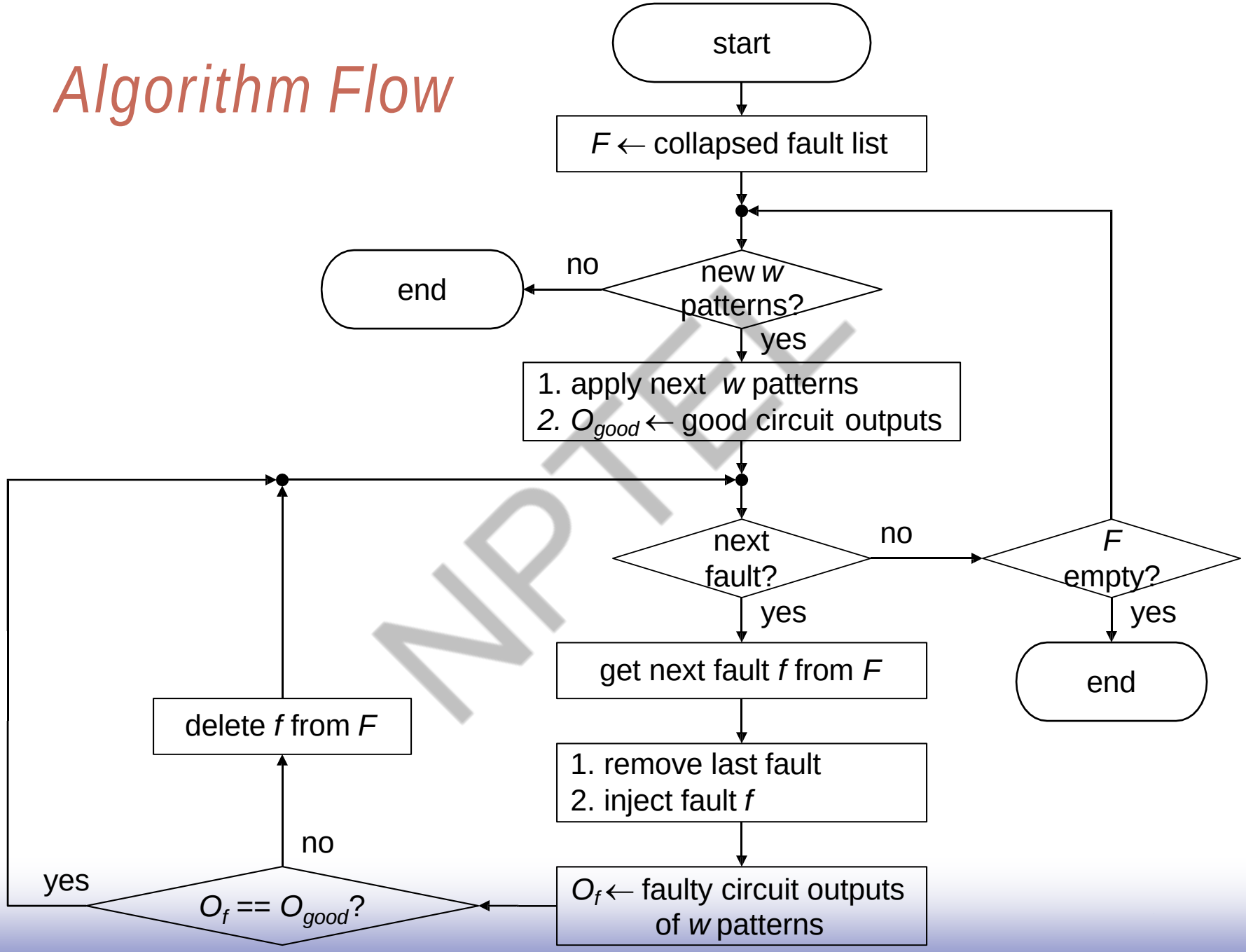
Pro and Con

- Advantages
  - A large number of faults are detected by each pattern when simulating the beginning of test sequence
- Disadvantages
  - Only applicable to the unit or zero delay models
  - Faults cannot be dropped unless all (*w*-1) faults are detected

Parallel Pattern Fault Simulation

- Parallel pattern single fault propagation (PPSFP)
- Parallel pattern
  - With a *w*-bit data width, *w* test patterns are packed into a word and simulated for the fault-free or faulty circuit
- Single fault
  - First, fault-free simulation
  - Next, for each fault, fault injection and faulty circuit simulation

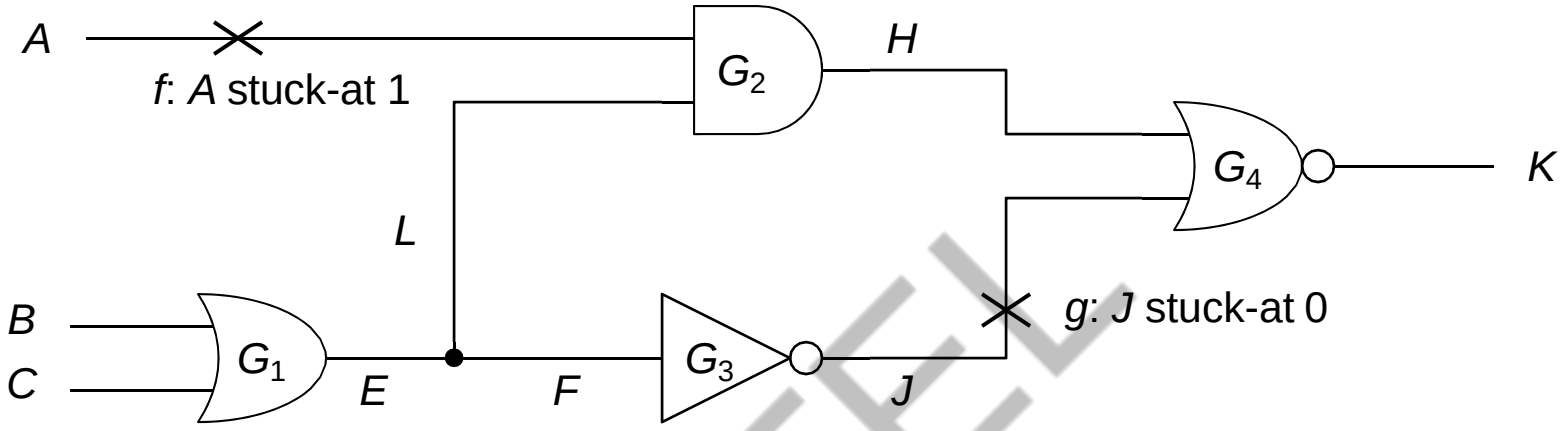
Algorithm Flow



Logic and Fault Simulation (contd.)

Lecture 15

Example



		Input			Internal					Output
		A	***B***	***C***	***E***	***F***	***L***	***J***	***H***	***K***
Fault Free	P_1	0	1	0	1	1	1	0	0	1
	P_2	0	0	1	1	1	1	0	0	1
	P_3	1	0	0	0	0	0	1	0	0
f	P_1	**1**	1	0	1	1	1	0	**1**	**0**
	P_2	**1**	0	1	1	1	1	0	**1**	**0**
	P_3	1	0	0	0	0	0	1	0	0
g	P_1	0	1	0	1	1	1	0	0	1
	P_2	0	0	1	1	1	1	0	0	1
	P_3	1	0	0	0	0	0	**0**	0	**1**

Pro and Con

- Advantages
  - Fault is dropped as soon as detected
  - Best for simulating test patterns that come later, where fault dropping rate per pattern is lower
- Disadvantages
  - Not suitable for sequential circuits

Deductive Fault Simulation

- Based on logic reasoning rather than simulation
- Fault list attached with signal *x* denoted as L_x
  - Set of faults causing *x* to differ from its fault-free value
- Fault list propagation
  - Derive the fault list of a gate output from those of the gate inputs based on logic reasoning

Fault List Propagation Rules

c : controlling value
i : inversion value
I : set of gate inputs
z : gate output
S : inputs holding controlling value

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

- All gate inputs hold non-controlling value

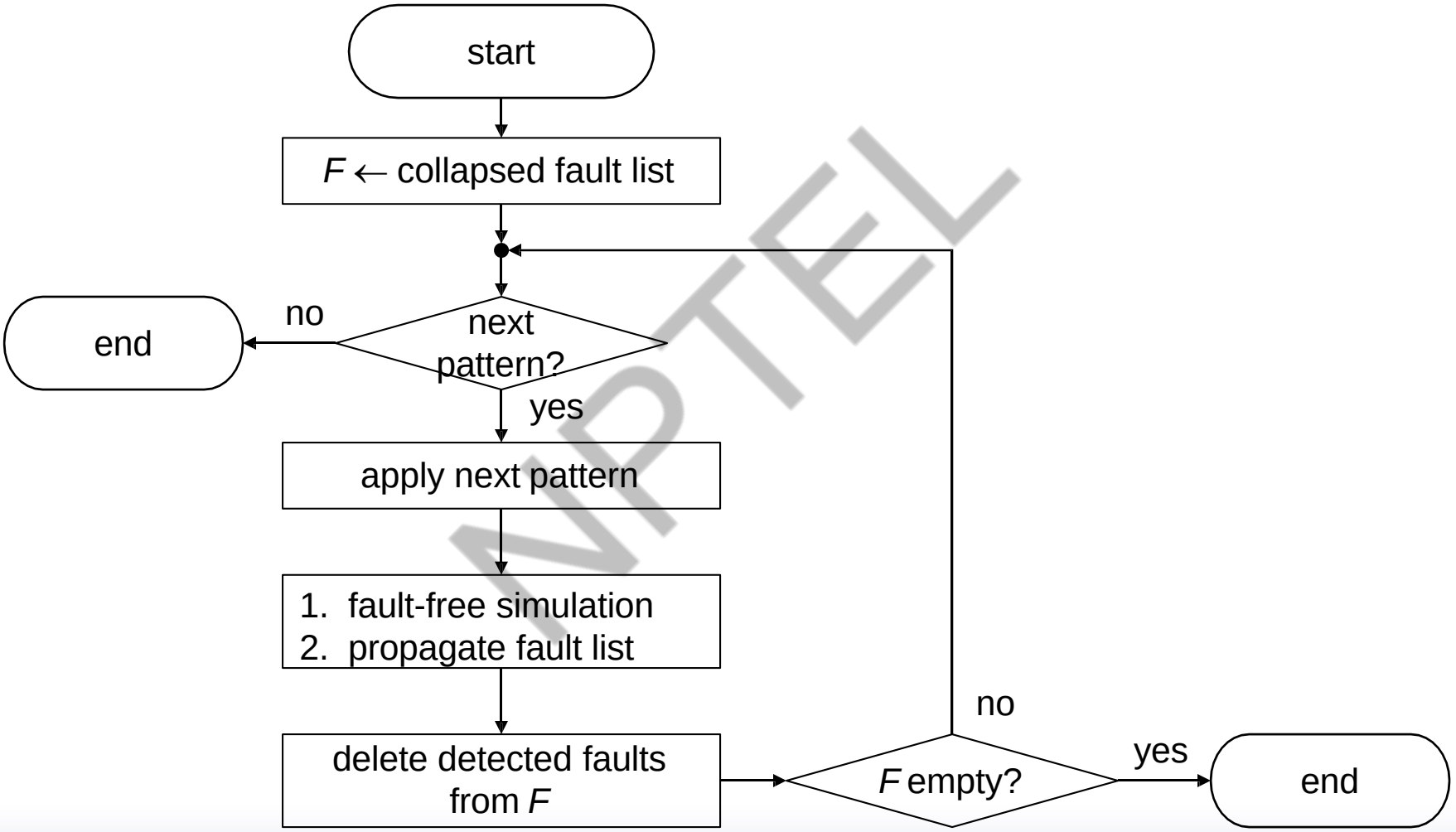
$$L_z = \left(\bigcup_{j \in I} L_j \right) \cup \{ z/(c \oplus i) \}$$

- At least one input holds controlling value

$$L_z = [(\bigcap_{j \in S} L_j) - (\bigcup_{j \in I-S} L_j)] \cup \{ z/c \oplus i' \}$$

Z stuck-at c XOR i

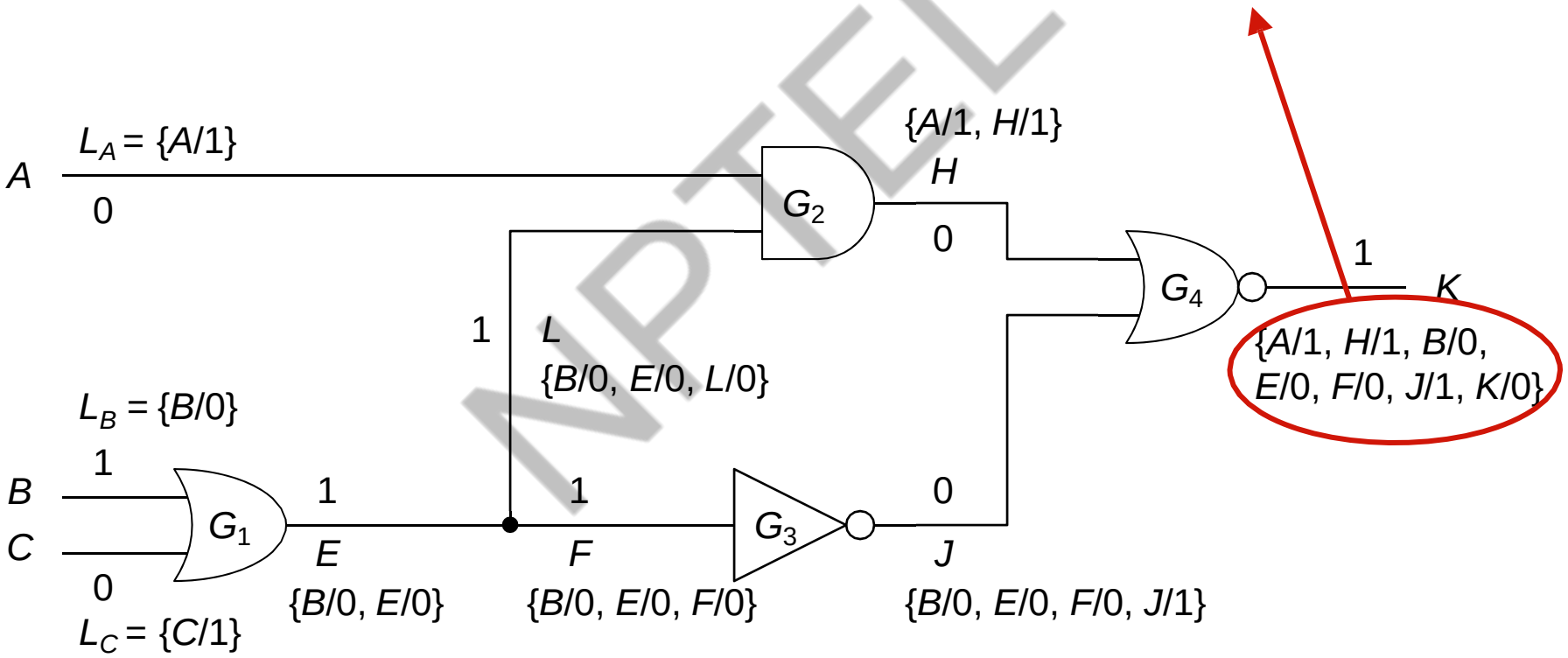
Algorithm Flow



Example

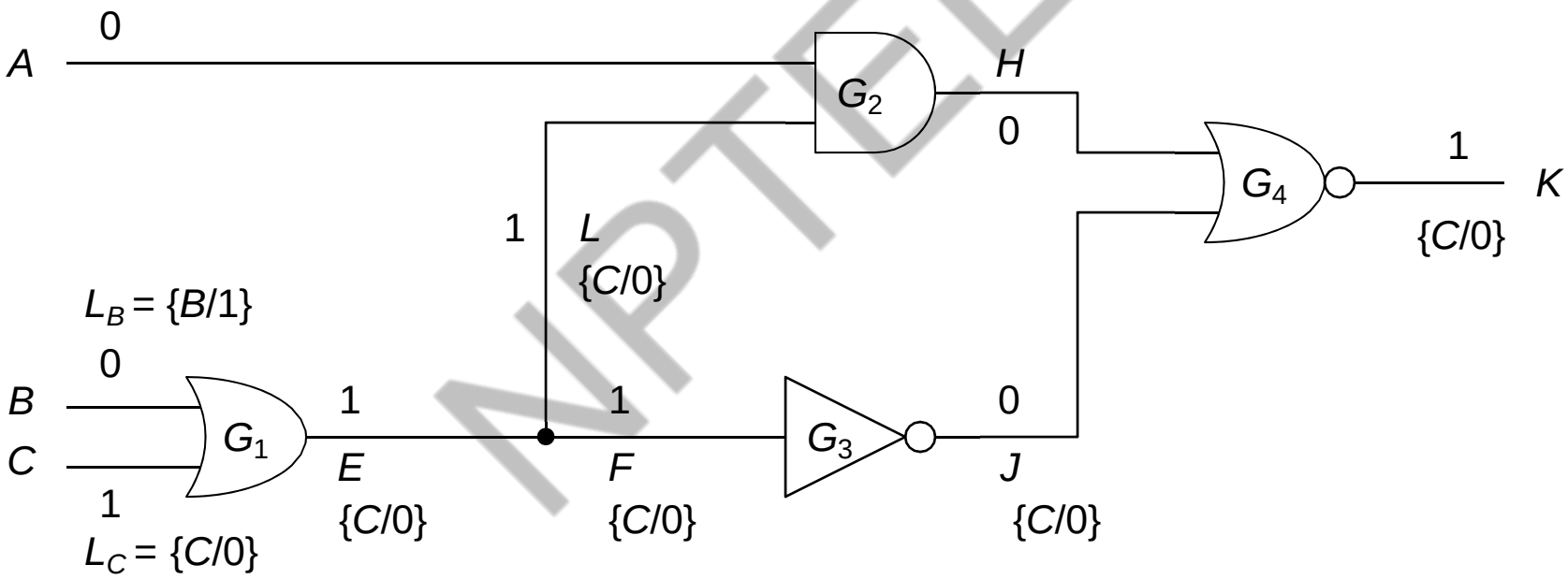
D P_1

$$L_K = L_H \cup L_J \cup \{K/0\}$$ by first Eq.



Example (cont'd)

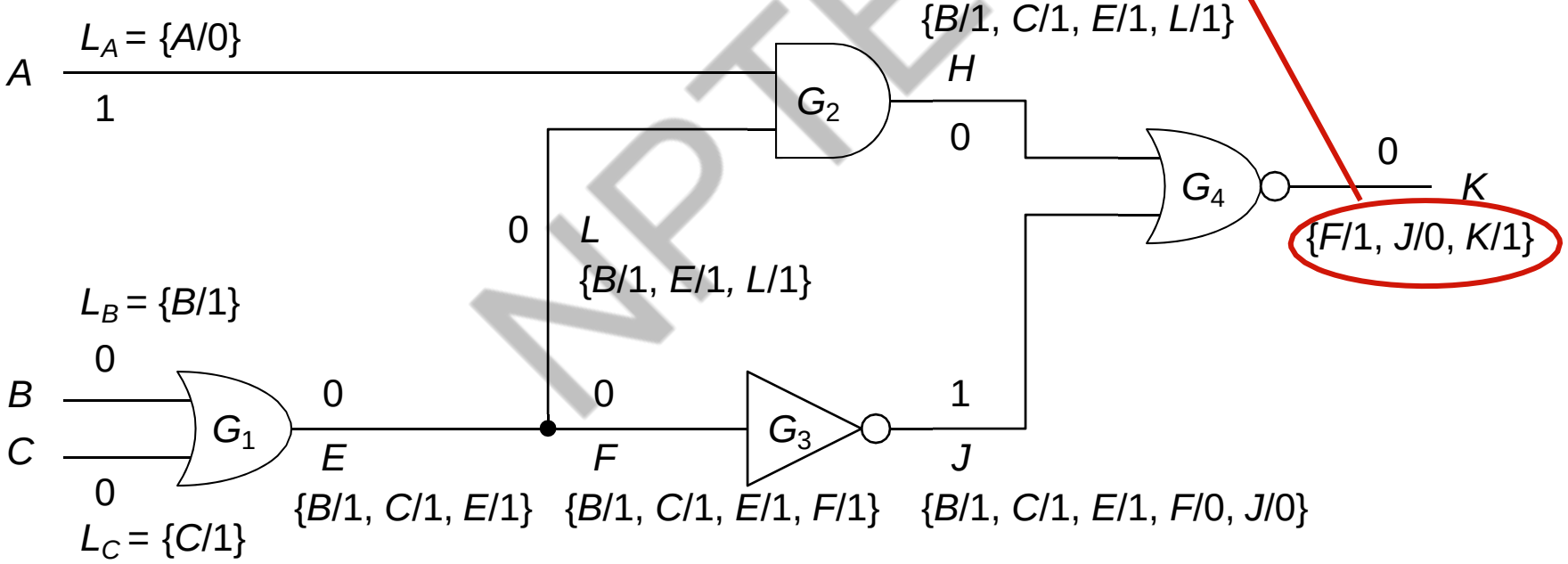
- P_2



Example (cont'd)

- P_3

$$L_K = (L_J - L_H) \cup \{K/1\}$$ by second Eq.



Pro and Con

- Advantages
  - Very efficient
  - Simulate all faults in one pass
- Disadvantages
  - Not easy to handle unknowns
  - Only for zero-delay timing model
  - Potential memory management problem

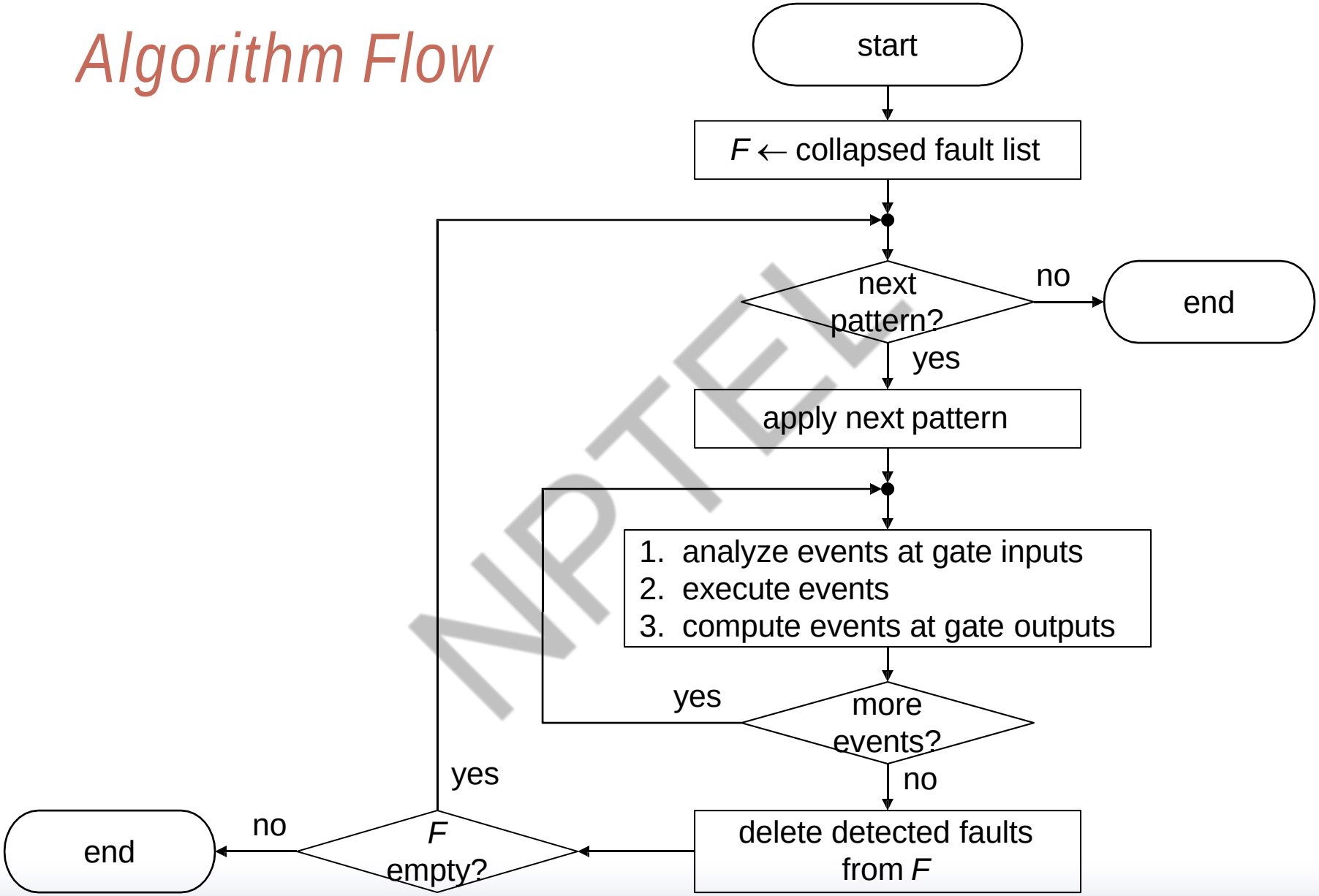
Concurrent Fault Simulation

- Simulate only differential parts of whole circuit
- Event-driven simulation with fault-free and faulty circuits simulated altogether
- Concurrent fault list for each gate
  - Consist of a set of bad gates
    - Fault index & associated gate I/O values
  - Initially only contains local faults
  - Fault propagate from previous stage

Good Event and Bad Event

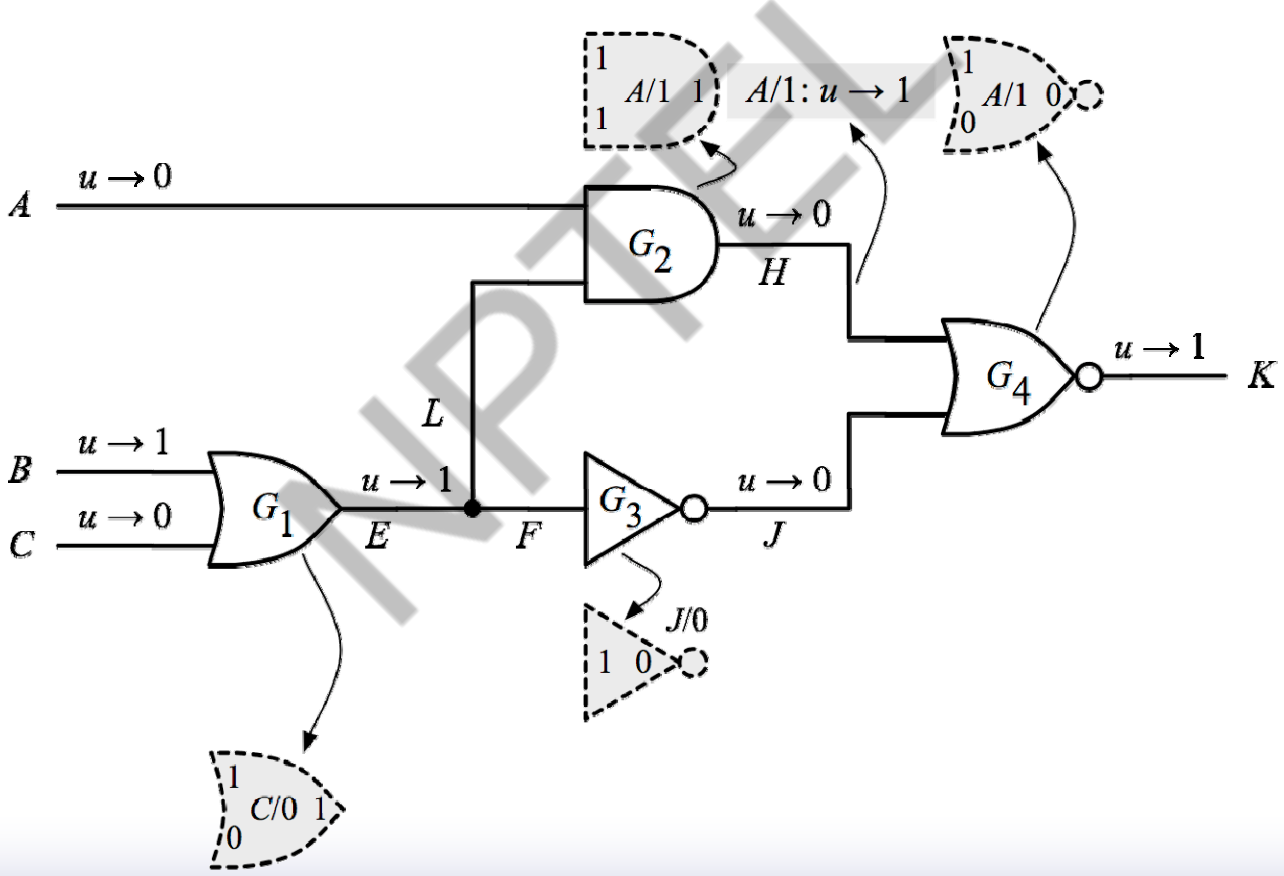
- Good event
  - Events that happen in good circuit
  - Affect both good gates and bad gates
- Bad event
  - Events that occur in the faulty circuit of corresponding fault
  - Affect only bad gates
- Diverge
  - Addition of new bad gates
- Converge
  - Removal of bad gates whose I/O signals are the same as corresponding good gates

Algorithm Flow



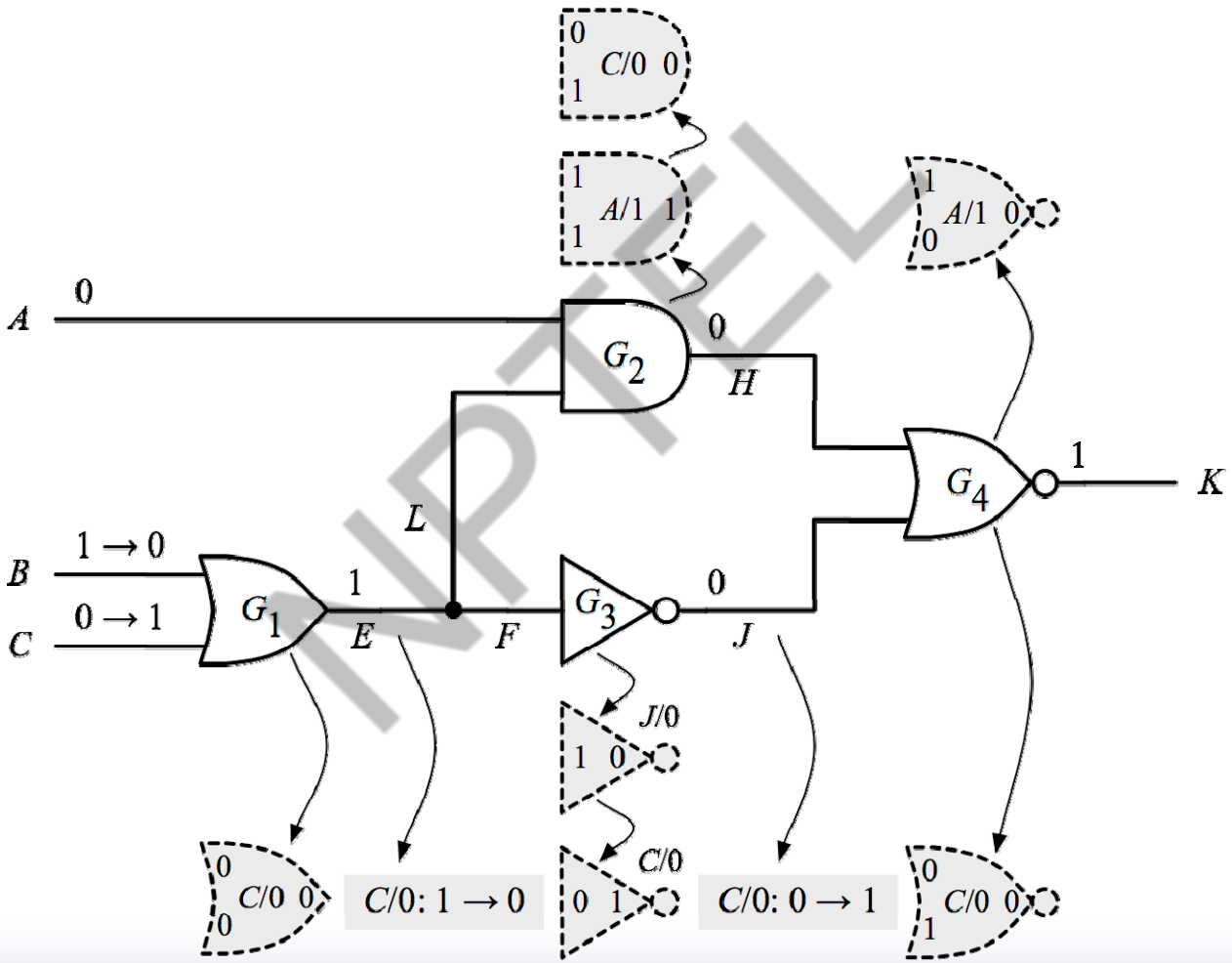
Example

- D P_1



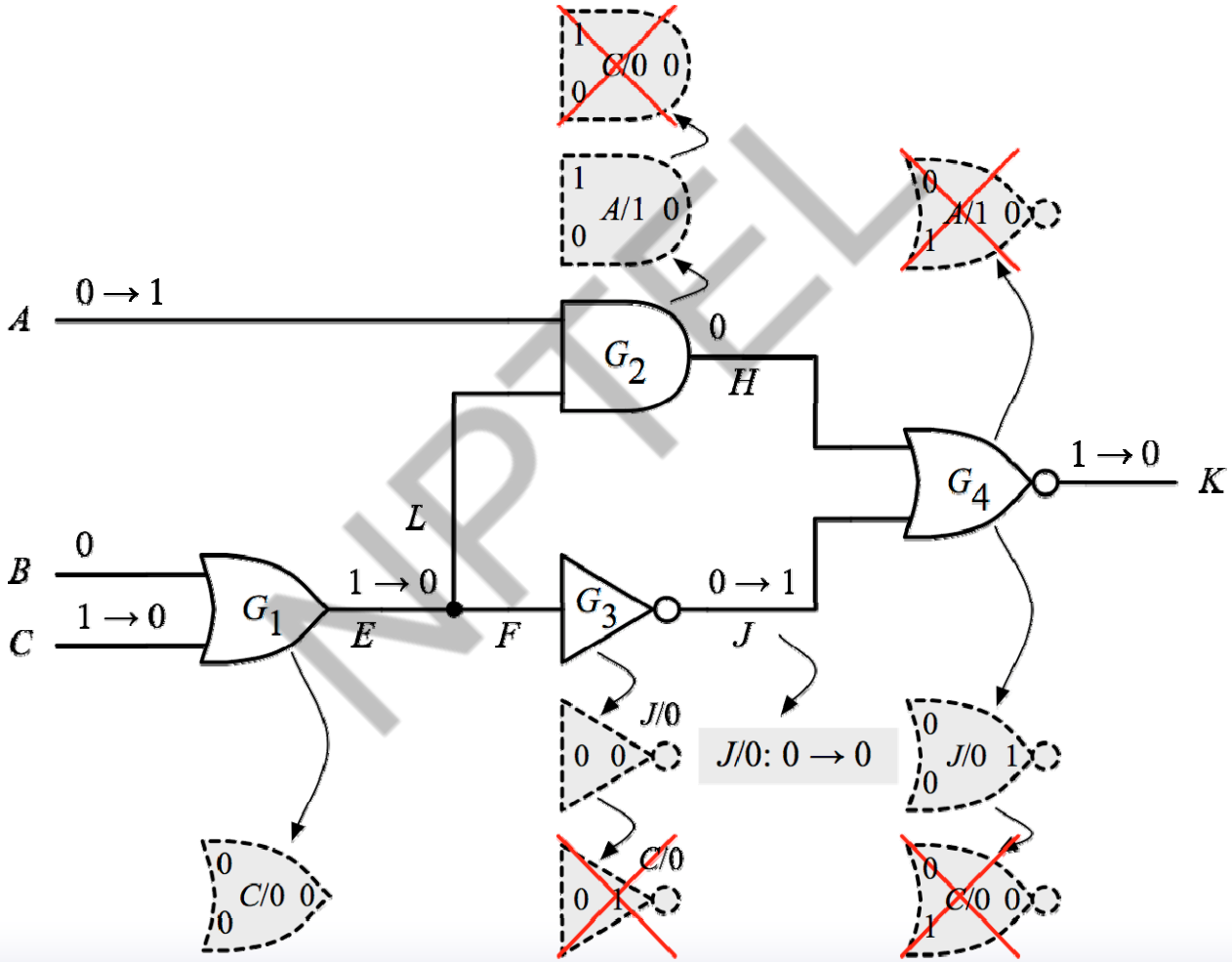
Example (cont'd)

- P_2



Example (cont'd)

- D P_3



Pro and Con

- Advantages
  - Efficient
- Disadvantages
  - Potential memory problem
    - Size of the concurrent fault list changes at run time

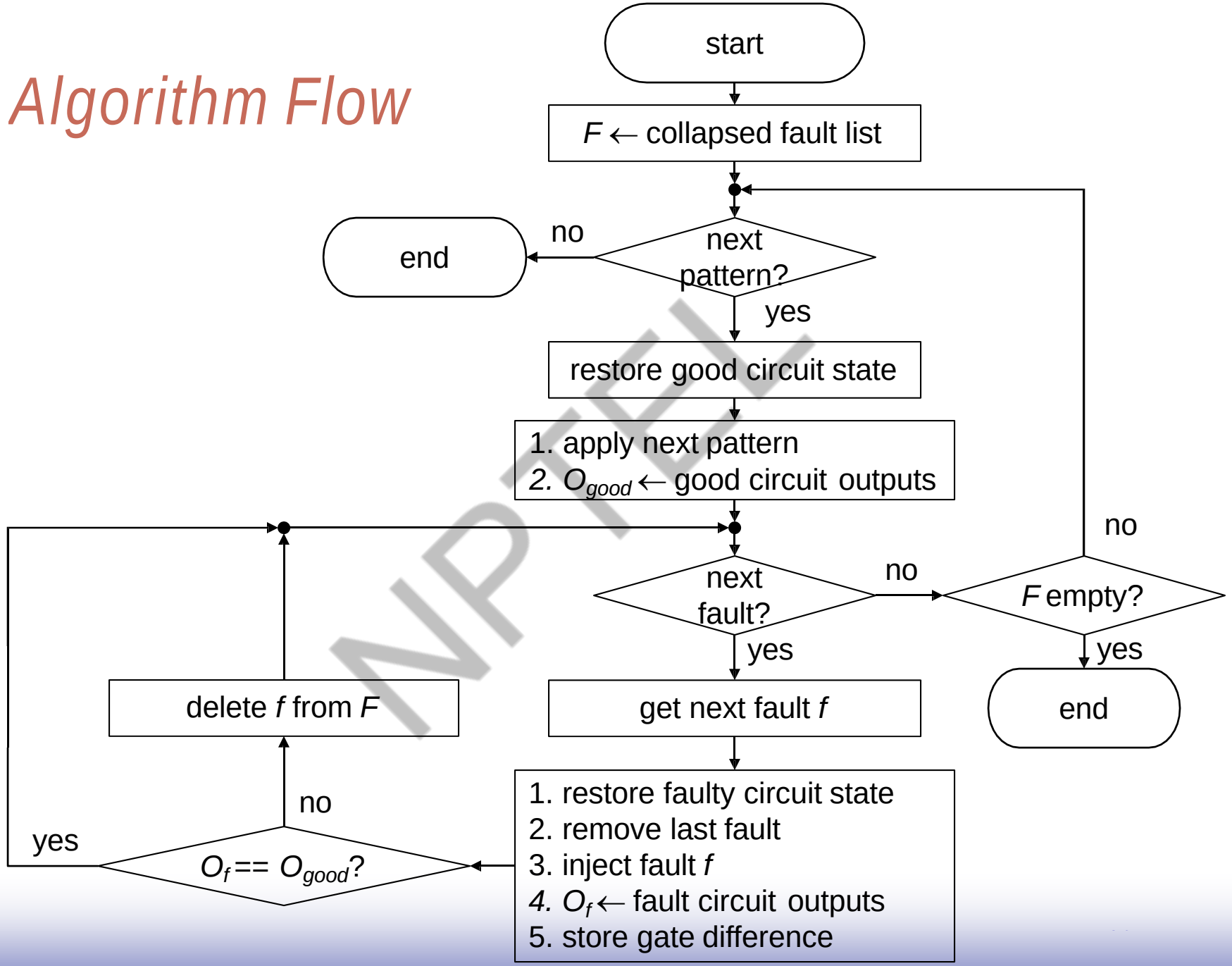
Differential Fault Simulation

- [Cheng 1989]
- Combines the merits of two techniques
  - Concurrent fault simulation
  - PPSFP
- Idea
  - Simulate in turn every fault circuit
  - Track only difference between faulty circuit and last simulated one
  - Inject differences as events
  - Easily implemented by event-driven simulator

Simulation Sequence

	P_1	P_2	…	P_i	P_{i+1}	…	P_n
Good	G_1	G_2	…	G_i	G_{i+1}	…	G_n
f_1	$F_{1,1}$	$F_{1,2}$	…	$F_{1,i}$	$F_{1,i+1}$	…	$F_{1,n}$
f_2	$F_{2,1}$	$F_{2,2}$	…	$F_{2,i}$	$F_{2,i+1}$	…	$F_{2,n}$
.	.	.	…	.	.	…	.
f_k	$F_{k,1}$	$F_{k,2}$	…	$F_{k,i}$	$F_{k,i+1}$	…	$F_{k,n}$
f_{k+1}	$F_{k+1,1}$	$F_{k+1,2}$	…	$F_{k+1,i}$	$F_{k+1,i+1}$	…	$F_{k+1,n}$
.	.	.	…	.	.	…	.
f_m	$F_{m,1}$	$F_{m,2}$	…	$F_{m,i}$	$F_{m,i+1}$	…	$F_{m,n}$

Algorithm Flow



Pro and Con

- Advantages
  - Suitable for sequential fault simulation
- Disadvantages
  - Order of events caused by faulty sites is NOT the same as the order of the timing of their occurrence

Logic and Fault Simulation (contd.)

Lecture 16

Fault Detection

- Hard detected fault
  - Outputs of fault-free and faulty circuit are different
    - 1/0 or 0/1
    - No unknowns, no *Z*
- Potentially detected fault
  - Whether the fault is detected is unclear
  - Example: stuck-at-0 on enable signal of tri-state buffer

Fault Detection (cont'd)

- Oscillation faults
  - Cause circuit to oscillate
  - Impossible to predict faulty circuit outputs
- Hyperactive faults
  - Catastrophic fault effect
    - Fault simulation is time and memory consuming
  - Example: stuck-at fault on clock
  - Usually counted as detected
    - Save fault simulation time

Comparison of Fault Simulation Techniques (1)

- Speed
  - Serial fault simulation: slowest
  - Parallel fault simulation: O(n^3), n: num of gates
  - Deductive fault simulation: O(n^2)
  - Concurrent fault is faster than deductive fault simulation
  - Differential fault simulation: even faster than concurrent fault simulation and PPSFP
- Memory usage
  - Serial fault simulation, parallel fault simulation: no problem
  - Deductive fault simulation: dynamic allocate memory and hard to predict size
  - Concurrent fault simulation: more severe than deductive fault simulation
  - Differential fault simulation: less memory problem than concurrent fault simulation

Comparison of Fault Simulation Techniques (2)

- Multi-valued fault simulation to handle unknown (*X*) and/or high-impedance (*Z*)
  - Serial fault simulation, concurrent fault simulation, differential fault simulation: easy to handle
  - Parallel fault simulation: difficult
- Delay and functional modeling capability
  - Serial fault simulation: no problem
  - Parallel fault simulation, deductive fault simulation: not capable
  - Concurrent fault simulation: capable
  - Differential fault simulation: capable

Comparison of Fault Simulation Techniques (3)

- Sequential circuit
  - Serial fault simulation, parallel fault simulation, concurrent fault simulation, differential fault simulation: no problem
  - PPSFP: difficult
  - Deductive fault simulation: difficult due to many unknowns

Comparison of Fault Simulation Techniques (4)

- PPSFP and concurrent fault simulation are popular for combinational (full-scan) circuits
- Differential fault simulation and concurrent fault simulation is popular for sequential circuits
- Multiple-pass fault simulation
  - Prevent memory explosion problem
- Distributed fault simulation
  - Reduce fault simulation time

Summary

- Fault simulation is very important for
  - ATPG
  - Diagnosis
  - Fault grading
- Popular techniques
  - Serial, Parallel, Deductive, Concurrent, Differential
- Requirements for fault simulation
  - Fast speed, efficient memory usage, modeling functional blocks, sequential circuits

Logic and Fault Simulation

- Introduction
- Simulation models
- Logic simulation
- Fault simulation
- **Concluding remarks**

Conclusions

- Logic and fault simulations, two fundamental subjects in testing, are presented
- Into the nanometer age, advanced techniques are required to address new issues
  - High performance
  - High capacity
  - New fault models