Laboratorij za načrtovanje integriranih vezij

Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

Preizkušanje elektronskih vezij

Logična simulacija in simulacija napak
Logic and fault simulation

Overview

- Introduction
- Simulation Models
- Logic Simulation
- Fault Simulation
- Concluding remarks

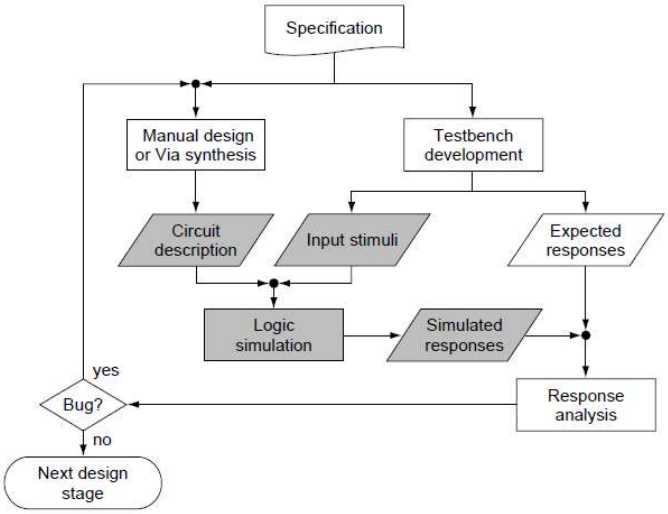
Introduction

- Set of techniques used in digital circuit verification, test development, design debug and diagnosis.
- Simulation is the process of predicting the behaviour of the circuit design before the circuit is fabricated.
- In digital circuits, simulation has two purposes:
  - To verify whether the design meets its functional specification and contains any design errors. This process is referred to as **logic simulation** or **fault-free simulation.** This process is called also **verification.**
  - In test development proces, **fault simulation** is used to simulate the faulty circuit.
  - Faulty circuit is simulated with a set of test patterns that help to locate/diagnose any manufacturing defects.
  - Fault simulation is also important component of ATPG programs.

Logic Simulation for Design Verification

- To manage growing design complexity, logic simulation is performed at each design stage.
  - Behavioral or electronic system level (ESL, C/C++, SystemC)
  - Register-transfer level (RTL, HDL code)
  - Gate-level (Gate-level netlist)
  - Transistor-level (SPICE models for switch- and transistor-level design)

Logic Simulation for Design Verification



Fault Simulation for Test and Diagnosis

Once again to remember:

- **Logic simulation** is intended for identifying design errors (by designers or CAD tools) to be caught prior physical implementation.
- **Fault simulation** is concerned with the behaviour of fabricated circuit as a consequence of fabrication imperfection.
- Manufacturing defects may cause the circuits to behave differently from the expected behaviour.
- Fault simulation assumes that the design is functionally correct.

Fault Simulation for Test and Diagnosis

- Fault simulation rates the effectiveness of a set of test patterns in detecting manufacturing defects.
- The quality of a test set is expressed in terms of **fault coverage (FC)**: the percentage of detected modeled faults that causes the design to exhibit observable erroneous responses.
- Fault simulation is one of the crucial components in ATPG:
  - The designer employs a fault simulator to evaluate the FC on a set of test patterns.
  - Fault simulation allows to compress the test set (**test compaction**).
  - Fault simulation assists in **fault diagnosis**.

Simulation Models

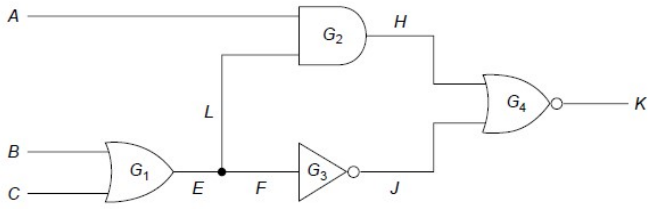
- **Gate-level circuit simulation models** for combinational and sequential circuits.
- Gate-level circuit description contains sufficient circuit structure information to capture the effects of many realistic manufacturing defects.
- The abstraction level of gate-level models is high enough to permit development of efficient simulation techniques.

Gate-Level Network

- Gate-level network is described as the interconnections of logic gates which are circuit elements that realize Boolean operations or expressions.
- The available gates range from the standard gates (AND, OR, NOT, NAND, NOR) to complex gates such as XOR and XNOR.

Gate-Level Network

- Example circuit is composed of OR (G_1), AND (G_2), NOT (G_3) and NOR (G_4) gates.

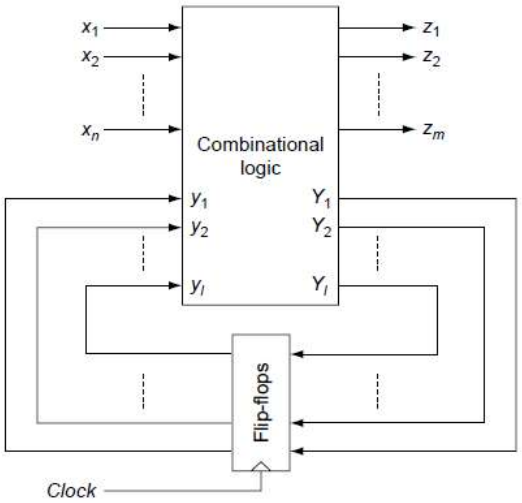


$$K = (A \cdot E + E')'$$
$$= (A + E')'$$
$$= A' \cdot (B + C)$$

Sequential Circuit

- Most of logic designs are **sequential circuits**.
- Their outputs depend on both the current and past input values.
- Sequential circuits are divided into 2 categories: synchronous and asynchronous.
- The synchronous sequential circuit is composed of two parts: **combinational logic** and **flip-flops (FF)** synchronized by a common clock signal.

Huffman model of sequential circuit

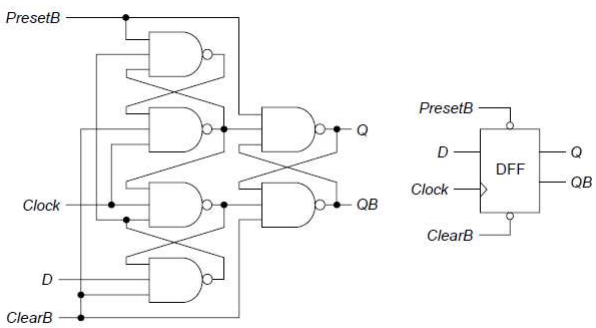


Huffman model of sequential circuit

- The inputs to the combinational logic consist of the **primary inputs** (**PI**s) x_1, x_2, … x_n and the FF outputs y_1, y_2, … y_l called **pseudo primary inputs** (**PPI**s).
- The outputs are comprised of the **primary outputs** (**PO**s) z_1, z_2, …. z_m and FF inputs Y_1, Y_2, … Y_l called **pseudo primary outputs** (**PPO**s) of the combinational logic.
- We assume that all FF are edge triggered – the states of all the FFs are updated according to the PPO values and FF characteristic function.

Description of a FF

- FF can be modeled as a functional block or as the interconnections of logic gates.
- Example of positive-edge triggered D FF.



Logic Symbols

- The basis for most digital systems is the two-valued Boolean algebra.
- A variable can assume only 2 values: true/false, represented by the two sysmbols: 1/0.
- Physical representation depends of the logic family or technology used.
- In addition to 1/0, logic simulators often include *u* (unknown, uncertain certain behavior) and Z (high-impedance, tri-state logic) symbols.
- Also, additional symbols may be used (based on value and strength).

Unknown State *u*

- By associating *u* with signals, we mean that the signal is either {1} or {0}, but we are not sure which one is the actual value; *u* = {0,1}
- Logic with (1,0,*u*) is called **ternary logic**.
- Example of Boolean operation with symbols (0,1,*u*)

$$\begin{aligned} 0 \cdot u &= \{0\} \cdot \{0, 1\} \\ &= \{0 \cdot 0, 0 \cdot 1\} \\ &= \{0, 0\} \\ &= \{0\} \\ &= 0 \end{aligned}$$

Basic Boolean Operations for Ternary Logic

- Input/output relationship for the three basic Boolean operations using ternary logic:

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*

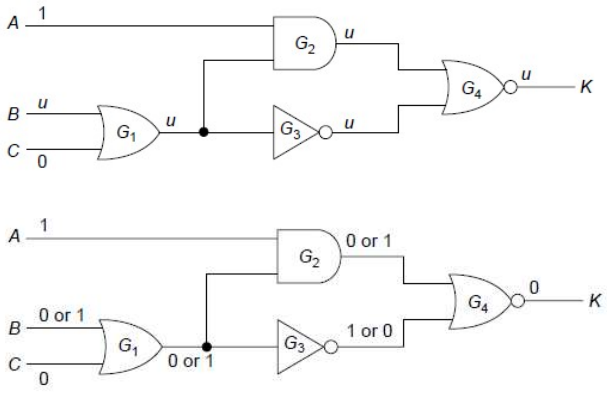
NOT	0	1	*u*
	1	0	*u*

0 is controlling value for AND gate

1 is controlling value for OR gate

Simulation with Ternary Logic

- Simulation based on ternary logic may not be accurate (information loss)
- Example:



Resolving Information Loss with Ternary Logic

- To resolve the problem of information loss, we have to introduce u_i‘
- Rules associated with each u_i

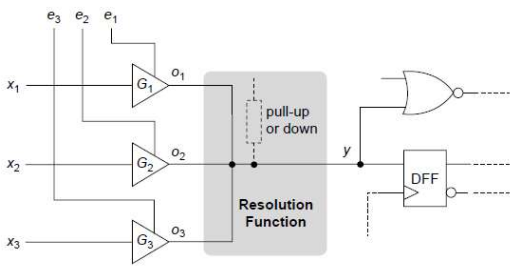
$$\begin{aligned} \text{NOT}(u_i) &= u_i' \\ \text{NOT}(u_i') &= u_i \\ u_i \cdot u_i' &= 0 \\ u_i + u_i' &= 1 \end{aligned}$$

- Check again the previous example.
- Output of G3 will be *u*‘ and K = 0.
- Each FF should have unique unknown symbol u_i

High-impedance State Z

- Tristate gates permit several gates to time-share a common wire (bus).
- A signal is in the Z state if it is connected neither to V_{dd} nor ground.
- Example: three bus drivers (G_1, G_2, G_3) drive the bus wire y

$$o_i = \begin{cases} x_i & \text{if } e_i = 1 \\ Z & \text{if } e_i = 0 \end{cases}$$



Logic Element Evaluation

- Logic element (or gate) evaluation is the process of computing the output of a logic element based on its current input and state values.
- The choice of evaluation technique depends on the types and models of the logic elements.
- Most used methods are:
  - Truth tables
  - Input scanning
  - Input counting
  - Parallel gate evaluation

Truth Tables

- This is the most straightforward way to evaluate logic elements.
- *n*-input combinational logic element requires 2^n-entry truth table to store the output value with respect to all possible input combinations.
- In practice, the truth table is stored in the array of size 2^n
- Truth-table based logic element evaluation is fast, but their usage is limited because the required memory grows exponentially with respect to *n*
- But, for example, for 9-valued logic system, 4 bits are required to code 9 symbols:
  - For 5-input element, an array size of $2^{4x5} = 2^{20}$ is required to store 9^5=19.683 truth table entries.

Input Scanning

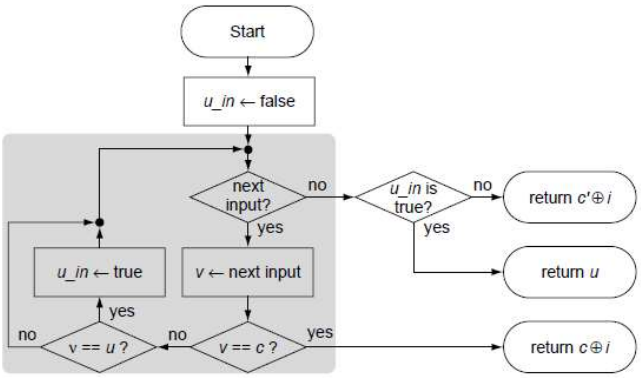
- Outputs of (N)AND and N(OR) gates can be determined if any of their inputs has a controlling value *c*.
- Simulators scans through the input and checks for the controlling value, *c*.
- In addition to *c*, inversion value, i, is required.

1. If any of the inputs is the controlling value, the gate output is $c \oplus i$.
2. Otherwise, if any of the inputs is u, the gate output is u.
3. Otherwise, the gate output is $c' \oplus i$.

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

Input Scanning

- The input scanning algorith flow:

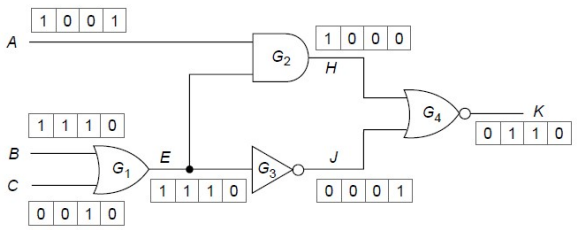


Input Counting

- Input counting algorithm mantains for each gate the number of controlling and unknown inputs.
- During logic simulation, two counts (the number of controlling and the number of unknown inputs) are updated if the value of any gate input changes.
- The same rules as those for input scanning are applied to determine the output value.

Parallel Gate Evaluation

- The goal is to speed-up logic simulation.
- Modern computers process data in the unit of a word, typically 32- or 64-bits wide -> mutiple copies of the signal can be stored in a single word and processed in the same time.
- This is referred as **parallel simulation** or **bitwise parallel simulation**.



Parallel Gate Evaluation

- Parallel simulation is more complicated for multi-valued logic -> for ternary logic, two bits are needed to code three simbols.
- For example:

$$v_0 = (00)$$
$$v_1 = (11)$$
$$v_u = (01)$$

- For each signal, two bits are located.
- Evaluation for 2-input AND gate is:

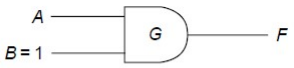
$$C_1 = \text{AND}(A_1, B_1)$$
$$C_2 = \text{AND}(A_2, B_2)$$

Timing Models

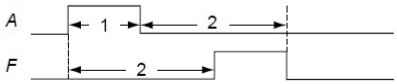
- Delay is associated to all electrical components, including logic gates and interconnection wires.
- We will analyze:
  - Transport Delay
  - Inertial Delay
  - Wire Delay
  - Functional Element Delay Model

Transport Delay

- Transport delay refers to the time duration it takes for the effect of gate input changes to appear at gate outputs.
- **Nominal delay** model specfies the same delay for the output rising and falling transition (also called **transition-independent delay** model).

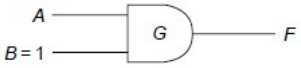


(a) Nominal delay
$d_N = 2$ ns

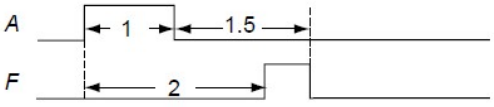


Transport Delay

- For cases, where rising and falling times are different, **rise/fall delay** model is used.

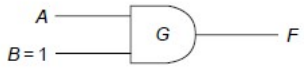


(b) Rise/fall delay
$d_r = 2$ ns
$d_f = 1.5$ ns

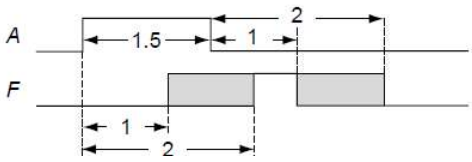


Transport Delay

- If the gate transport delay cannot be uniquely determined (due to process variations), **min-max delay** is used.
- We can combine min-max and rise/fall delay models to represent more complicated delay behaviours.

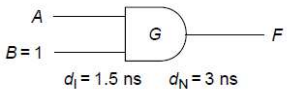


(c) Min–Max delay
$d_{min} = 1$ ns
$d_{max} = 2$ ns



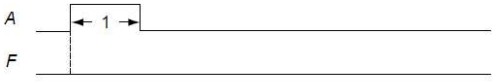
Inertial Delay

- Inertial delay is defined as the minimum input pulse duration necessary for the output to switch states.
- Pulses, shorter than the inertial delay, cannot pass through the circuit element.
- Example:

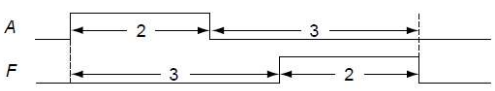


$d_I = 1.5$ ns $d_N = 3$ ns

(a) Pulse duration less than d_I

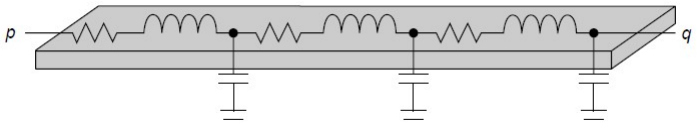


(b) Pulse duration longer than d_I



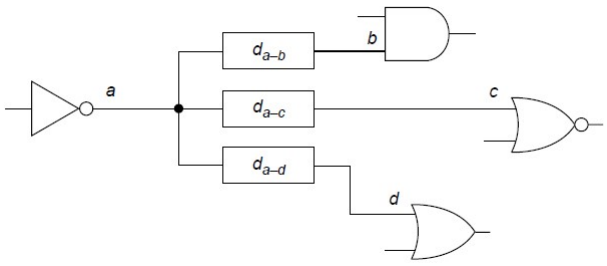
Wire Delay

- Wires are 3-D structures that are inherently resistive and capacitive.
- It takes finite time, called **propagation delay**, for a signal to travel from point *p* to point *q*.



Wire Delay

- In general, wire delay are specified for each connected gate output and gate input pair since physical distances (-> propagation delays) between the driver and receiver gates vary.
- To model the wire delays, delay elements $d_{d\text{-}r}$ are inserted into the fanout branches.



Functional Element Delay Model

- Functional elements, such as FF, have more complicated behaviour than simple logic gates and require more sophisticated timing models.
- D FF 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	↓	1	0	↑	↓	1.6	1.8	Asynchronous preset
X	X	1	↓	1	↓	↑	1.8	1.6	Asynchronous clear
1	↑	1	1	0	↑	↓	2	3	Q: 0 → 1
0	↑	1	1	1	↓	↑	3	2	Q: 1 → 0

Note: X indicates "don't care."

FF Timing model contains also setup/hold times

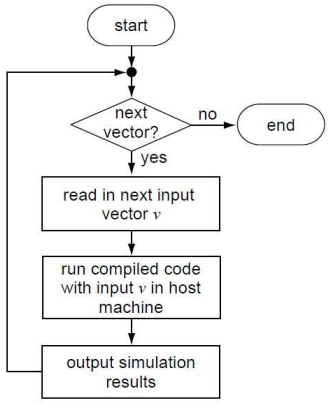
Logic Simulation

- Gate-level logic simulation methodologies:
  - Compiled-code simulation
  - Event-driven simulation
  - Hardware emulation and acceleration approaches

Compiled-Code Simulation

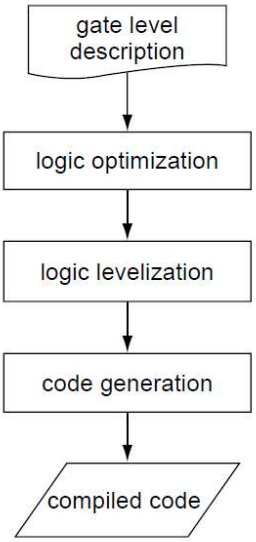
- Idea: to translate the logic network into a series of machine instructions that model the functions of the individual gates and interconnections between them.

Simulation flow:



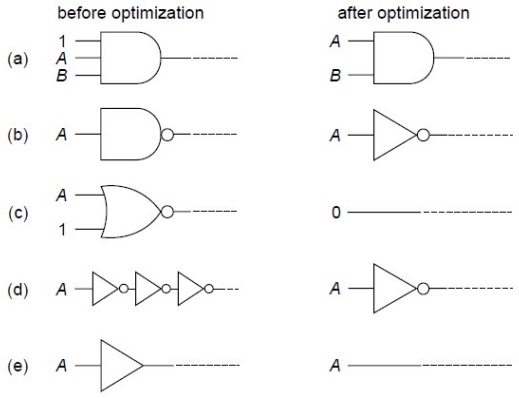
Code generation

Code generation flow:



Logic Optimization

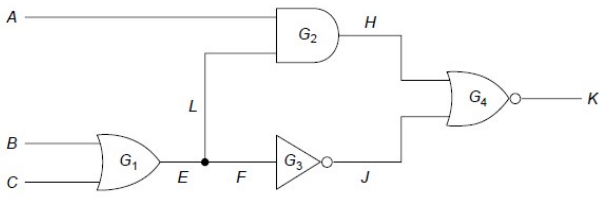
- The purpose of logic optimization is to enhance the simulation efficiency (program size and execution time).
- Typical optimization steps:



Logic Levelization

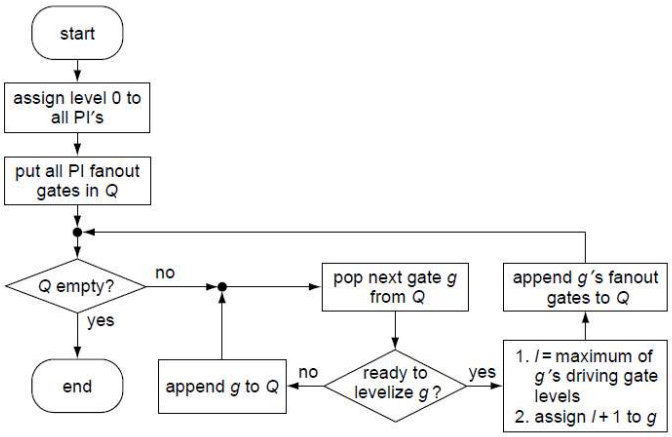
- Logic gates must be evaluated in an order such that a gate will not be evaluated until all its driving gates have been evaluated.
- For most networks, more than one evaluation order exists:

$G_1 \rightarrow G_2 \rightarrow G_3 \rightarrow G_4$ or $G_1 \rightarrow G_3 \rightarrow G_2 \rightarrow G_4$



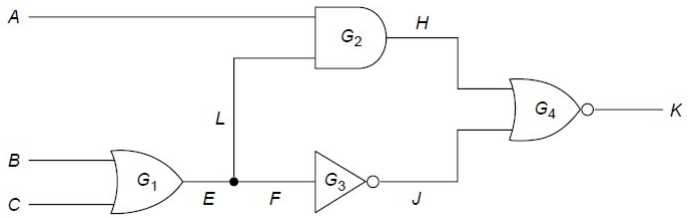
Logic Levelization

- Logic levelization algorithm:



Logic Levelization

- Example:



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	< >

Code Generation

- Different code generation techniques are used:
  - **High-level programming language source code**
    Network is described in C/C++, SystemC, ..)
  - **Native machine code**
    Target machine code is generated directly without the need for compilation (high simulation efficiency)
  - **Interpreted code**
    Target machine is a software emulator – during simulation, instructions are interpreted and executed one at a time.

Code Generation

- Pseudo code for the example circuit.
- Each statement is replaced with the corresponding language constructs or machine instruction.

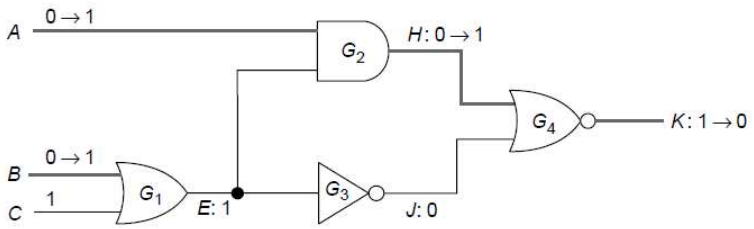
```
while(true) do
  read(A, B, C);
  E←OR(B, C);
  H←AND(A, E);
  J←NOT(E);
  K←NOR(H, J);
end
```

Code Generation

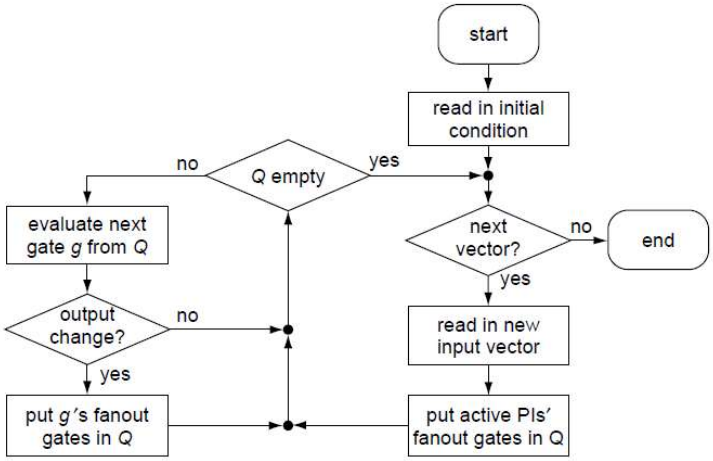
- Most effective if binary logic is used -> machine instructions are readily available for Boolean operations (AND, OR, NOT).
- The main limitation is timing modeling - gate and wire delays cannot be handled -> it fails to detect timing problems such as glitches and race conditions.
- The low efficiency of compiled-code simulation is because the entire network is evaluated for each input vector.
  - In general up to 10 % of input values change values between consecutive vectors.

Event-Driven Simulation

- High simulation efficiency by performing gate evaluations only when necessary.
- In event-driven simulation, the switching of a signal value is called the **event**. Event-driven simulators monitors the occurances of events to determine which gates to evaluate.
- Example:

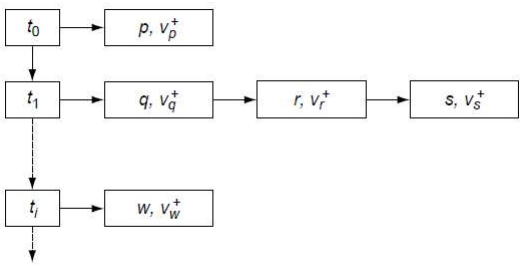


Event-Driven Simulation: Algorithm



Nominal Delay Event-Driven Simulation

- It is imporant also when to evaluate logic gate.
- The scheduler is implemented as priority que.
- The vertical list is an ordered list that stores the time stamps when events occur.
- Attached to each stamp is a horizontal list of events that occur at time t_i



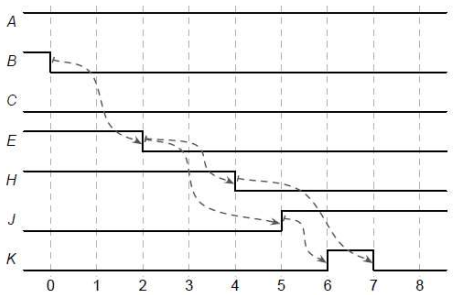
Compiled Code vs. Event-Driven Simulation

- Compiled code is suitabe for cycle-based simulation (circuit behaviour at the end of each clock cycle is important) and zero-delay simulation can be used.
- Suitable for bitwise parallel simulation.

- Event-driven supports general delays and can detect hazards.
- Ideal for circuits with low activity.
- Also very useful for circuit debugging.

Hazards

- Different delays along reconvergent signal paths.
- Transient pulses or glitches, called hazards may occur.
- Example:
  - Input vector ABC = 110 -> 100
  - Static hazard 0-hazard at node K.



Types of Hazards

- Hazards are: static and dynamic.
- Static hazards are static 1-hazard and static 0-hazard.

static 1-hazard static 0-hazard

- Dynamic hazards are dynamic 1-hazard and dynamic 0-hazard.

dynamic 1-hazard dynamic 0-hazard

Fault simulation

- More challenging as logic simulation.
- The behaviour of the circuit containing all the modeled faults must be simulated.
- The complexity of the fault simulation is $O(pn^2)$
  - p = number of test patterns, n = number of logic gates
- This is infeasible and various fault simulation techniques have been developed.

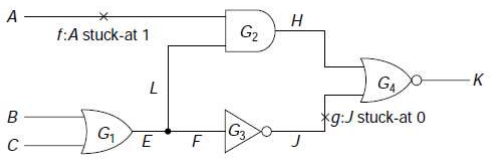
Terminology:

- Test vectors: term used for logic simulation (human written)
- Test patterns: term used for fault simulation (machine generated)

Serial Fault simulation

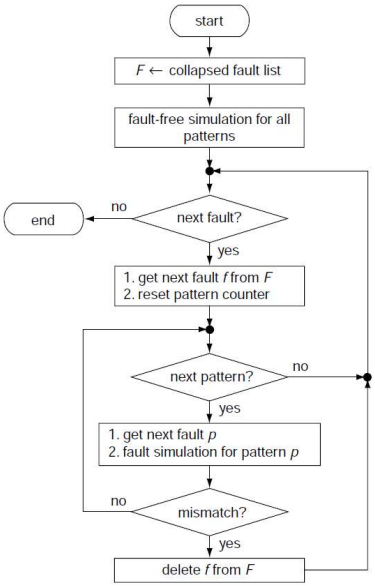
- The simplest fault simulation technique.
- It consists of
  - fault-free and
  - faulty circuit simulations.
- The fault-free responses are stored to determine later whether a test pattern can detect a fault or not.
- Serial fault simulation simulates faults one at a time.
- Presence of the fault is done with a **fault injection**.
- The faulty circuit is then simulated with a given test pattern to derive the faulty response.
- The process repeats until all faults in the fault list are simulated (**fault dropping**).

Serial Fault simulation: Example



	Input			Internal					Output		
Pattern No.	A	B	C	E	F	L	J	H	K_{good}	K_f	K_g
P_1	0	1	0	1	1	1	0	0	1	0	1
P_2	0	0	1	1	1	1	0	0	1	0	1
P_3	1	0	0	0	0	0	1	0	0	0	1

Serial Fault simulation: Algorithm



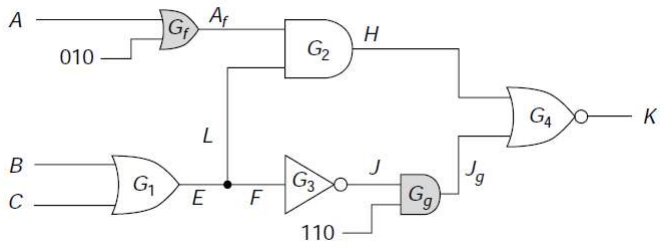
Parallel Fault Simulation

- Similar to parallel logic simulation, fault simulation takes advantage of bitwise parallelism.
- Two parallelism are possible
  - parallelism in faults (parallel fault simulation)
  - parallelism in patterns (parallel pattern fault simulation)
- In *w*-bit wide data words, *w-1* bits are used for faulty circuits, 1 bit is used for fault-free circuit.
- *w-1* faulty circuits can be processed in parallel, resulting in a speedup factor of *w-1* compared to serial fault simulation.

Parallel Fault Simulation: Example

		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**

Parallel Fault Simulation: Injecting a Fault



- Parallel fault simulation (FS) is applicable to unit or zero delay models only.
- Parallel FS is best used for simulating at the beginning of test generation, when a large number of faults are detected by each pattern.

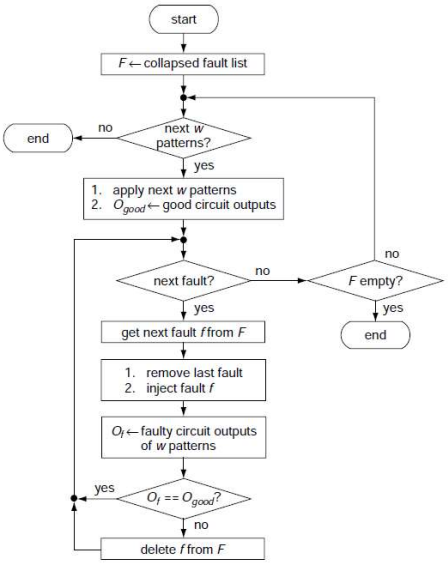
Parallel-Pattern Fault Simulation

- First, logic simulations on the fault-free circuit are performed.
- Then faults are simulated on these w test patterns
- For each fault, simulation results are compared with the correct responses to determine if the fault is detected.
- The process repeats until all faults in the fault list are simulated.

Parallel-Pattern Fault Simulation: 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**

Parallel-Pattern Fault Simulation: Algorithm

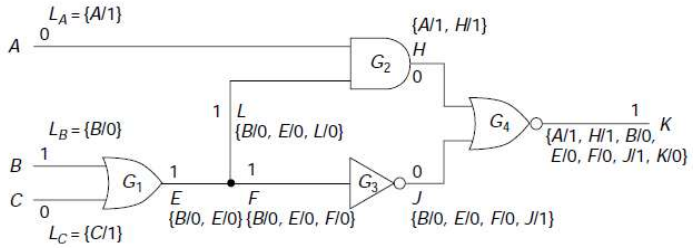


Deductive Path Simulation

- It is based on logic reasoning.
- For a given test pattern *t*, deductive simulation identifies the faults that can be detected.
- It is very fast since only fault-free simulation are performed.
- Fault list (L_x) is associated with a signal line *x*.
- L_x is a set of faults that causes *x* to differ from its fault-free value.

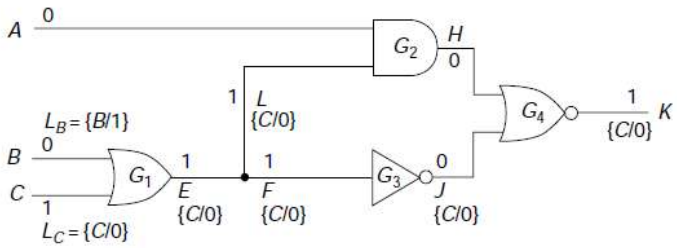
Deductive Fault Simulation

- Example, test pattern *P1 ABC* = 010
  L_A = (*A/1*), L_B = (*B/0*), L_C = (*C/1*).
- Based on logic reasoning, fault list at gate output is created.
- This process is called **fault list propagation**.



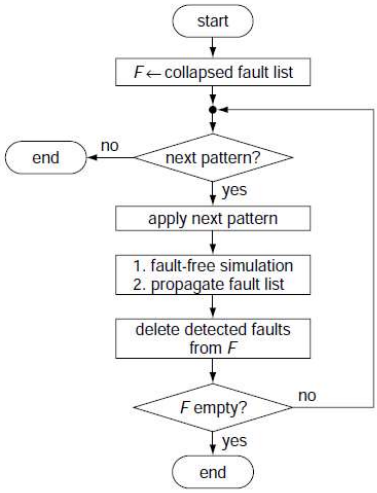
Deductive Fault Simulation

- Example for the next test pattern *P2 ABC* = 001
- Faults, detected by test pattern *P1,* are dropped and not taken into account.



Deductive Fault Simulation

- Algorithm

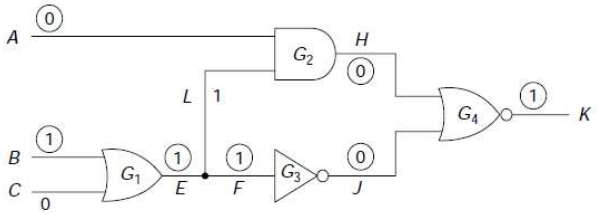


Critical Path Tracing

- Alternative to fault simulation
- Given a test pattern *t*, net *x* has a **critical value** *v* if and only if (iff) the x stuck-at *v'* is detected by *t*.
- A net that has a critical value is a **critical net**.
- **Critical path** is a path that consists of nets with critical values.
- Tracing the critical path from PI to PO gives a list of critical nets and hence a list of detected faults.

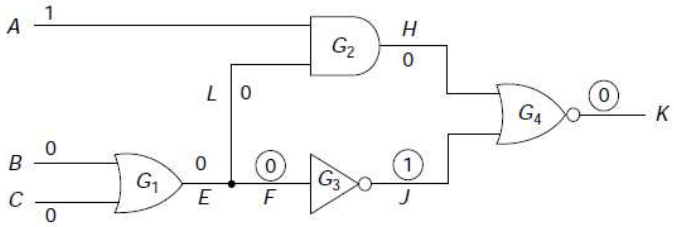
Critical Path Tracing

- Example: Consider the test pattern *ABC* = 010
- All critical values at nets *K*, *H*, *J*, *F*, *E*, *B*, *A* are circled.
- Flipping any of them would change the PO *K*.
- Net *L* is not critical since changing *L* would not change PO.
- Seven critical nets are identified and their associated faults *A/1, B/0, E/0, F/0, H/1, J/1, K/0* are detected



Critical Path Tracing

- Special attention is required when fanout branches reconverge.
- Net *E* is not critical since changing its value will propagate through gates *G2* and *G3*, but this change will not be further propagated through gate *G4*.



Other methods for fault simulation

- Concurrent fault simulation
- Differential fault simulation
- Fault sampling
- Statistical fault analysis
- And several others ...

Conclusion

- Logic simulation
  - Checks whether the design will behave as predicted before its physical implementation.
  - Event-driven simulation technique is most widely used today.
- Fault simulation
  - We have information in advance how effective is the given test pattern set in detecting faults.
- Logic and fault simulation programs are part of every commercial tool for circuit design !!