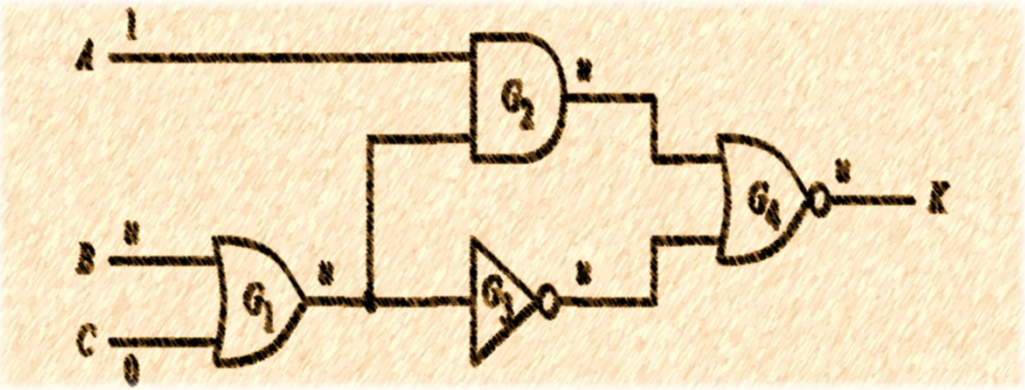
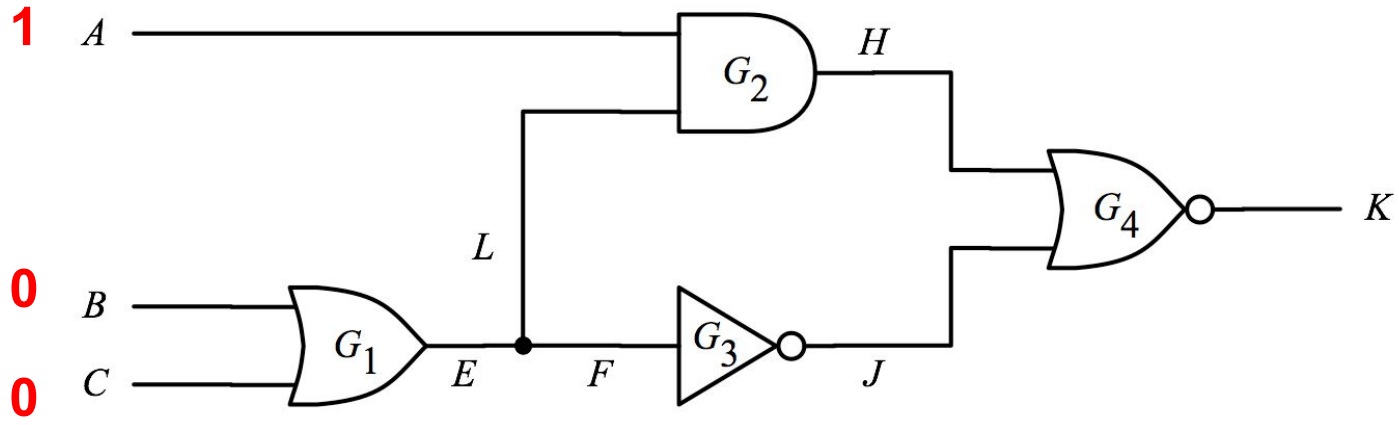
Logic Simulation

- Introduction
- Simulation Models
- **Logic Simulation Techniques**
  - **Compiled-code simulation**
    - **Logic Optimization**
    - **Logic Levelization**
    - **Code Generation**
  - Event-driven simulation
  - Parallel Simulation
- Issues of Logic Simulations
- Conclusions



Compiled-Code Simulation

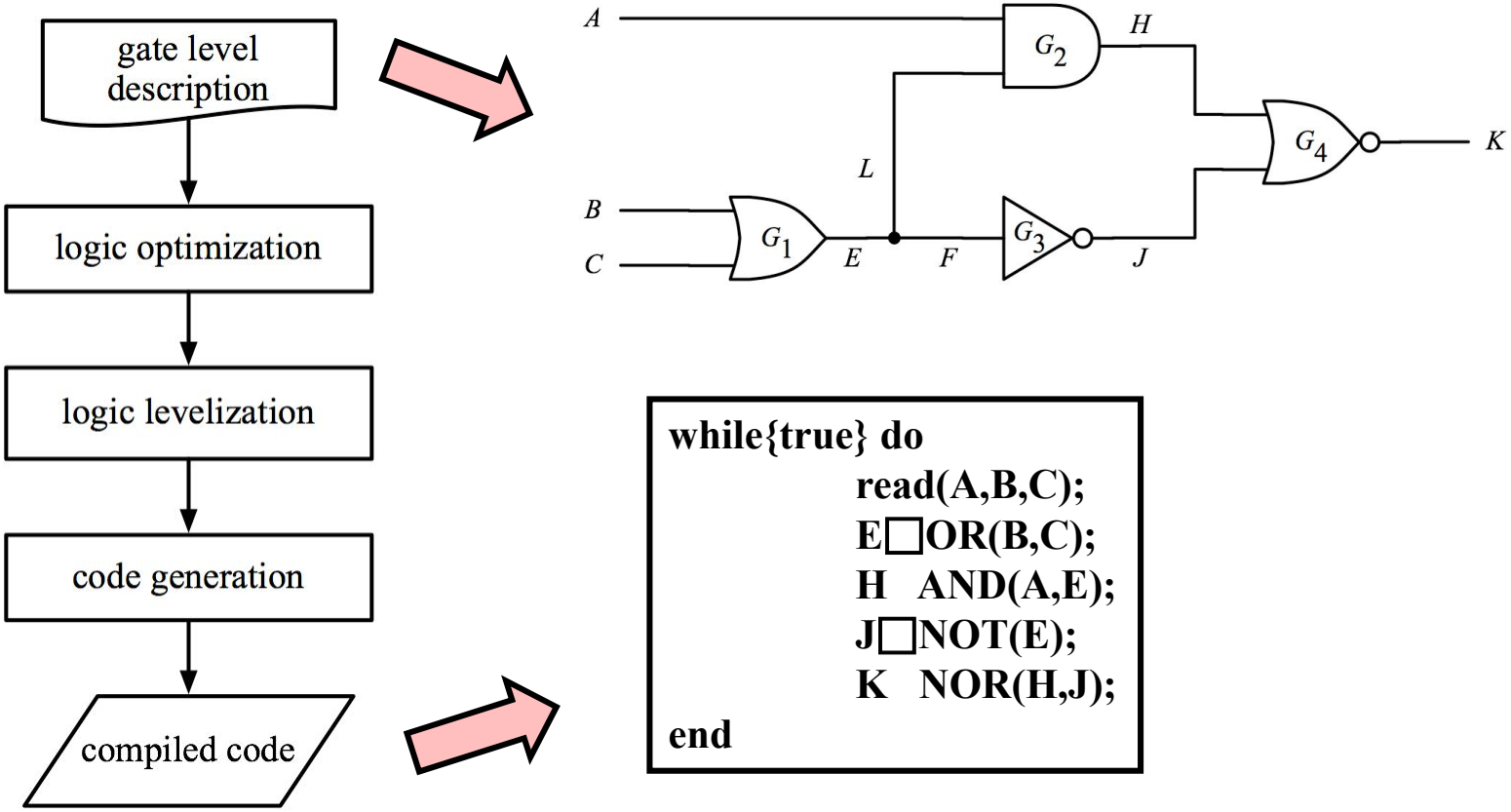
- **Translate circuit into sequence of codes**
  - **Execute codes = run logic simulation**




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

E=OR(0,0)=0
H=AND(1,0)=0
J=NOT(0)=1
K=NOR(0,1)=0

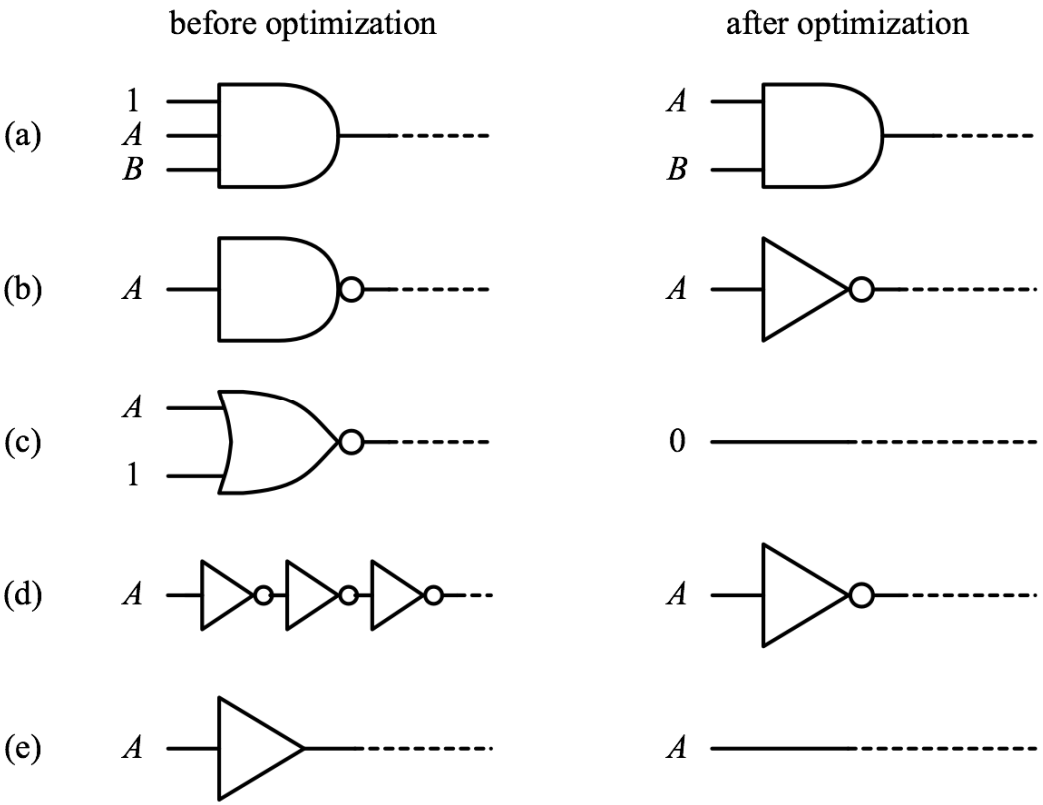
How to Compile Code?

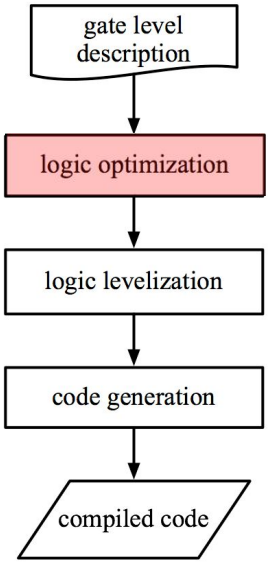


(WWW Fig. 3.12)

Logic Optimization

- **Simplify logic before generating codes**
- **Shorten code length and simulation time**
- **Example** (WWW Fig. 3.13)

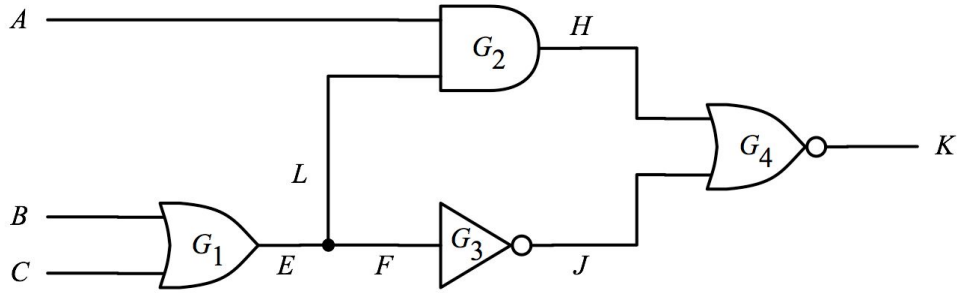


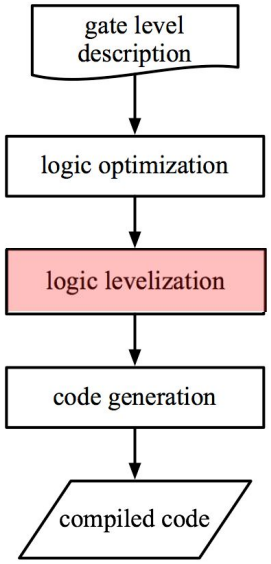


(b) Code generation flow

Logic Levelization

- *Levelization*: order gate in sequence such that
  - a gate won't be evaluated until
  - all its driving gates have been evaluated





(b) Code generation flow

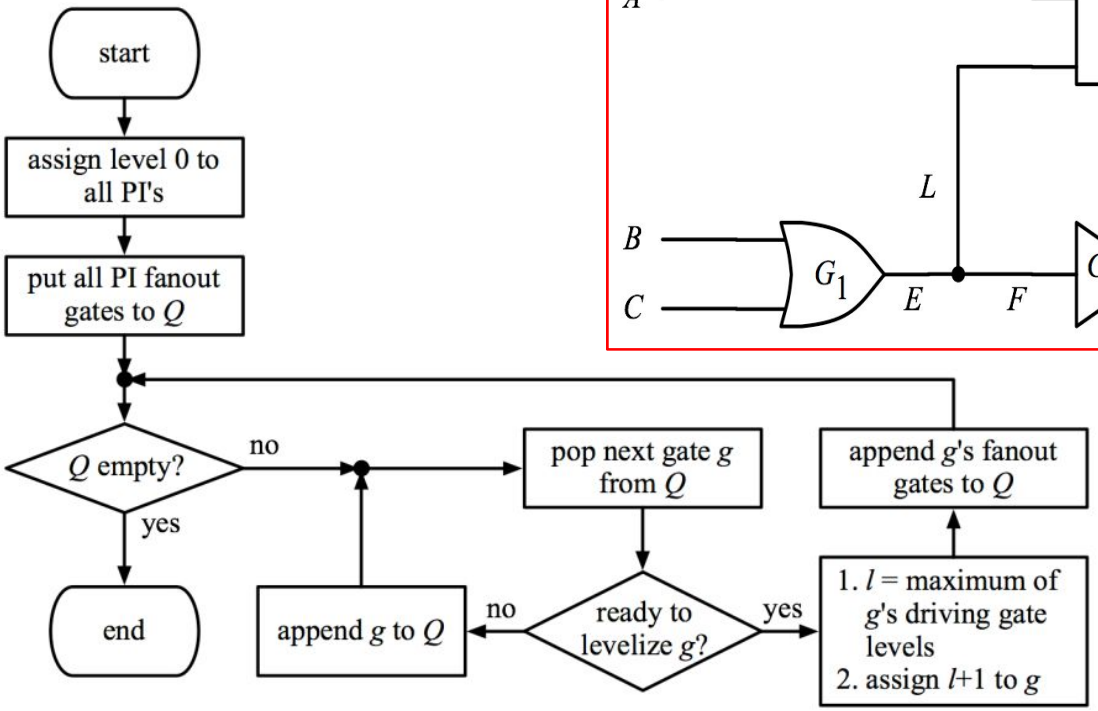
correct

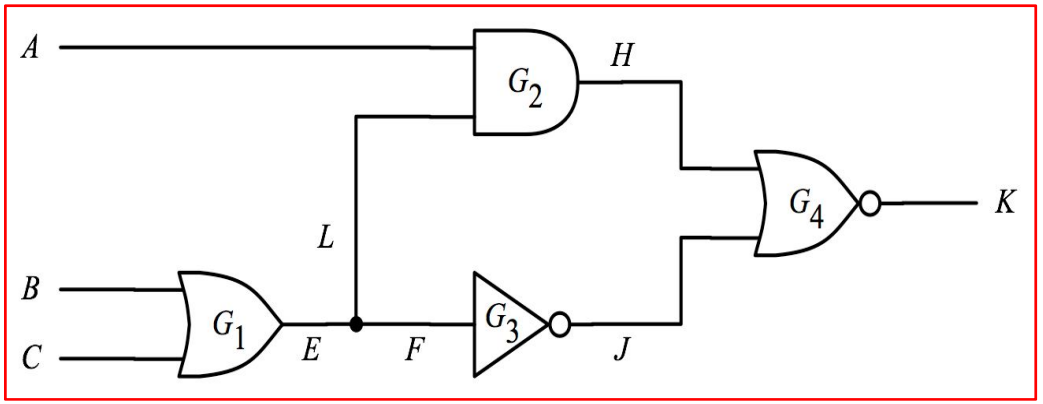
```
while{true} do
        read(A,B,C);
        E□OR(B,C);
        H  AND(A,E);
        J□NOT(E);
        K  NOR(H,J);
end
```

wrong

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

Levelization Ensures Correct Order



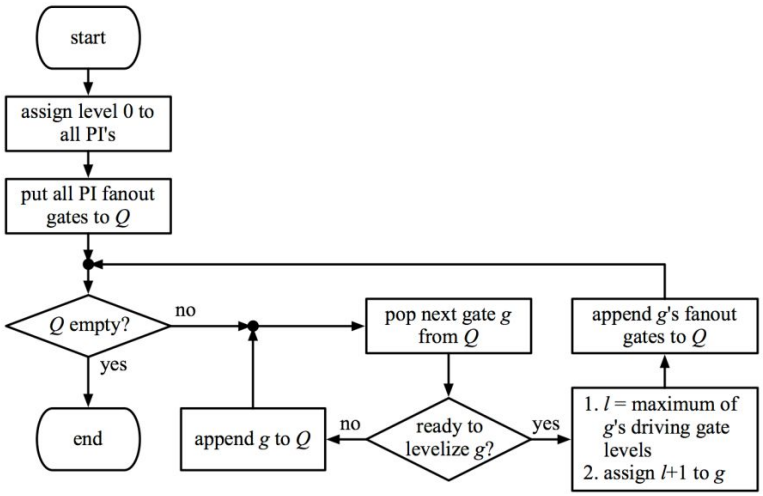


(WWW Fig 3.17)

g = gate
l = level
Q = queue (FIFO)

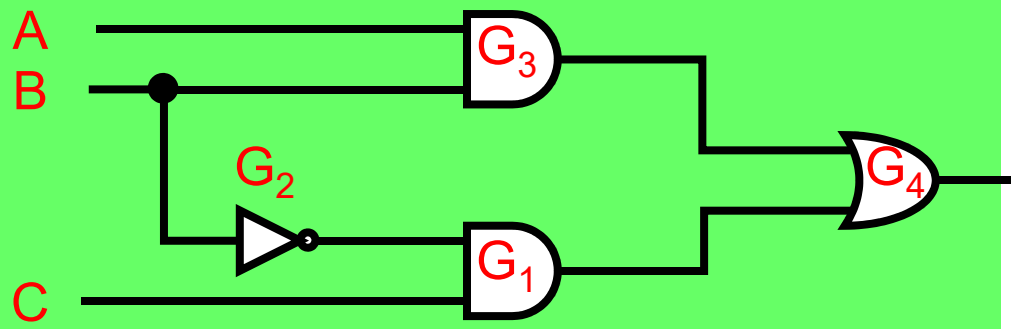
step	A	B	C	G_1	G_2	G_3	G_4	Q <front, back>
0	0	0	0					<G_2, G_1>
1	0	0	0					<G_1, G_2> put G_2 back
2	0	0	0	1				<G_2, G_2, G_3>
3	0	0	0	1	2			<G_2, G_3, G_4>
4	0	0	0	1	2			<G_3, G_4, G_4> why? FFT
5	0	0	0	1	2	2		<G_4, G_4, G_4>
6, 7, 8	0	0	0	1	2	2	3	< >

Quiz



Q: Please levelize this circuit

A:



step	A	B	C	G_1	G_2	G_3	G_4	Q <front, back>
0	0	0	0					$<G_1, G_2, G_3>$
1								
2								
3								
4								
5								
6								

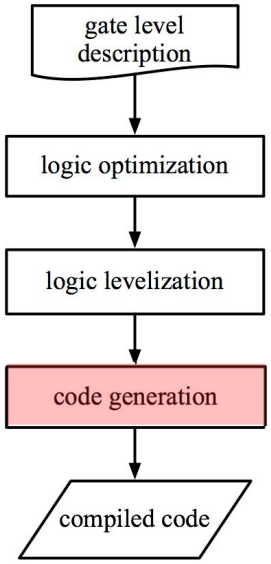
Code Generation

① **High-level code (like C)**
- ☺ **Portable, easy debug**
- ☹ **Need compilation every time circuit changed**

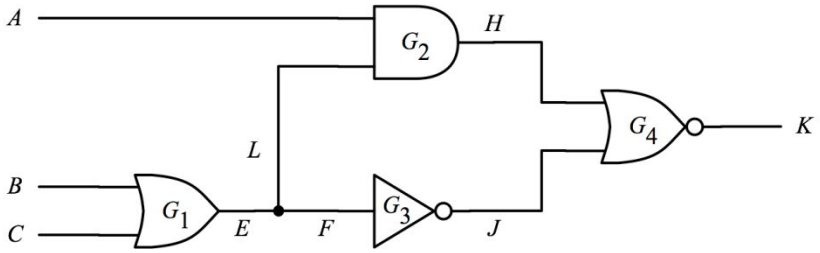
② **Machine code**
- ☺ **Fast to run**
- ☹ **Not portable, hard to debug**

③ **Interpreted code**

(at run time, codes are interpreted and executed)
- ☺ **Portable, easy debug**
- ☹ **Slower than machine code**



(b) Code generation flow




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

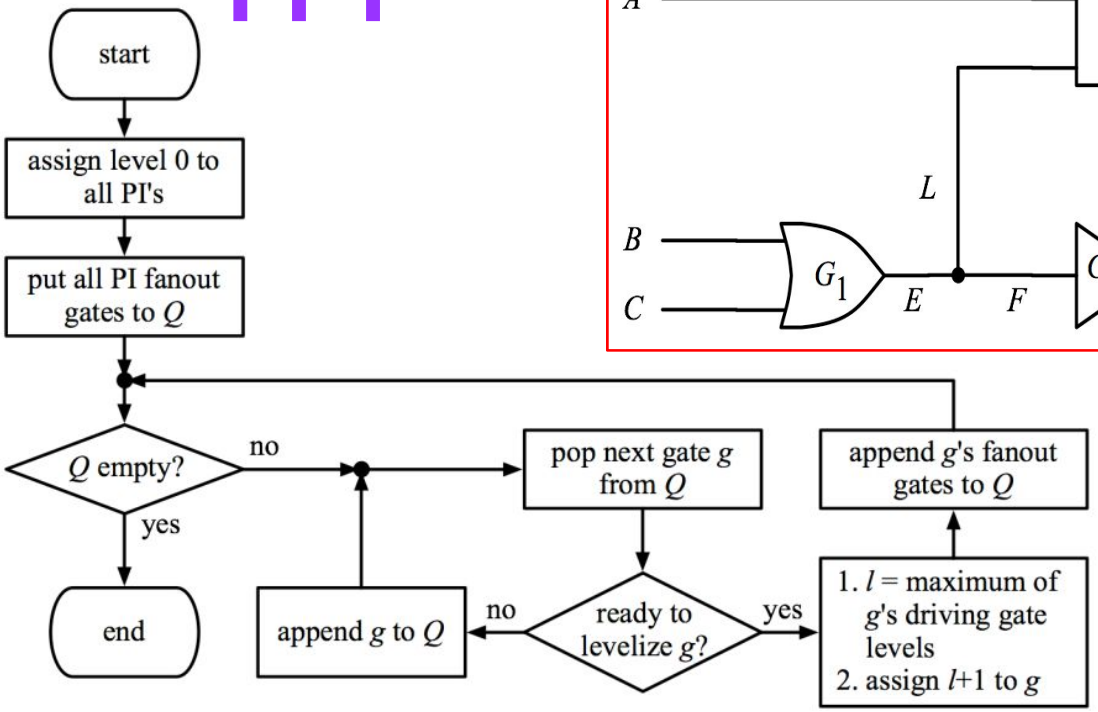
Summary

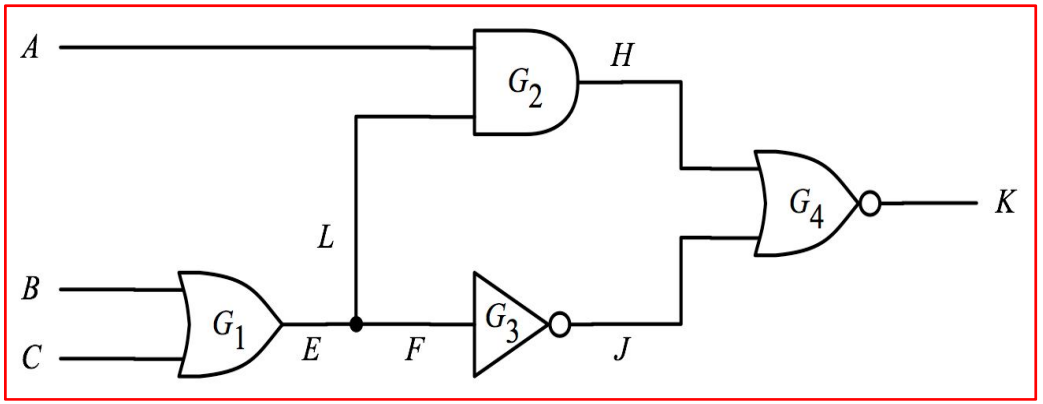
- Compiled-code simulation: convert gates into codes for evaluation
  - Optimization: simplifies logic
  - Levelization: sort gates in order (*i.e.* topological sort of graph)
  - Code generated: 1.high-level, 2.machine, 3.interpreted

☺ Pros
  - Simple to implement
  - Can speed-up by parallelism
    - see parallel simulation

☹ Cons
  - Only *cycle-based accuracy*, no timing (zero gate delay)
  - Need to evaluate whole circuit even only small portion changed
    - see event-driven simulation

FFT





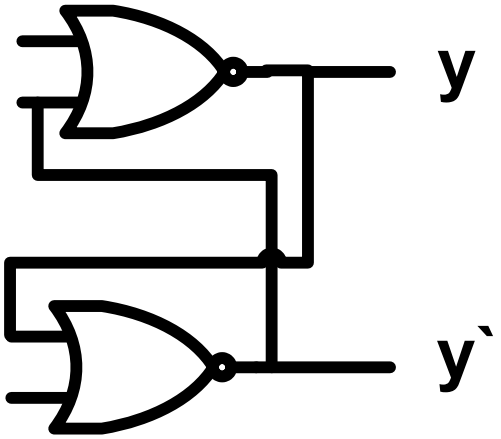
(WWW Fig 3.17)

g = gate
l = level
Q = queue (FIFO)

step	A	B	C	G_1	G_2	G_3	G_4	Q <front, back>
0	0	0	0					$<G_2, G_1>$
1	0	0	0					$<G_1, G_2>$
2	0	0	0	1				$<G_2, G_2, G_3>$ why G_2 again?
3	0	0	0	1	2			$<G_2, G_3, G_4>$
4	0	0	0	1	2			$<G_3, G_4, G_4>$ why G_4 again?
5	0	0	0	1	2	2		$<G_4, G_4, G_4>$
6, 7, 8	0	0	0	1	2	2	3	< >

FFT

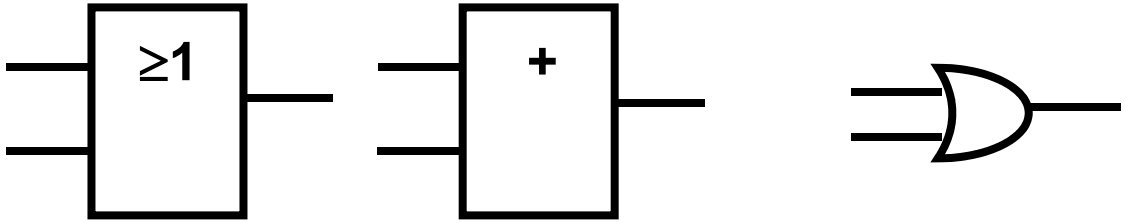
- Q:How to levelize SR latch?
  - with feedback



Appendix: Logic Symbols

- IEEE logic symbols: rectangular shape v.s. distinctive shape
- AND

- Or



- inverter