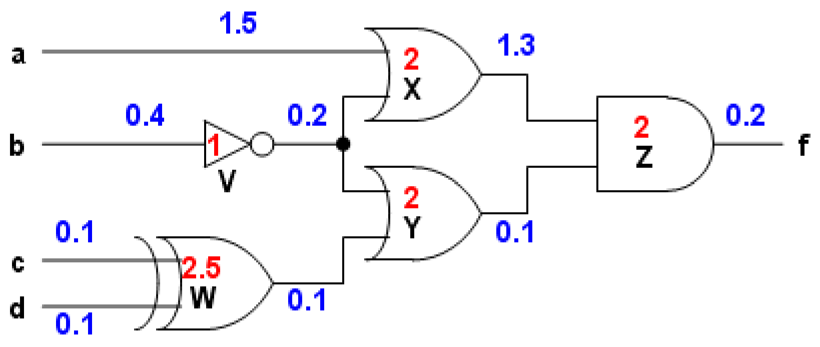
HW05 [ECE 720]

Digvijay Anand
200478940

Q1. Forward Propagation algorithm:



Solution:

Node	List of incoming edges	Delay known in each path	Path delay	Maximum delay (Node)

V	b -> V	delay(b) = 0.0;
		wire_delay(b -> V) = 0.4;
		gate_delay(V) = 1.0
			0.0 + 0.4 + 1.0 = 1.4	1.4

W	c -> W	delay(c) = 0.0;
		wire_delay(c -> W) = 0.1;
		gate_delay(W) = 2.5
			0.0 + 0.1 + 2.5 = 2.6	2.6

	d -> W	delay(d) = 0.1;
		wire_delay(c -> W) = 0.1;
		gate_delay(W) = 2.5
			0.1 + 0.1 + 2.5 = 2.7	max (2.6, 2.7) = 2.7

X	a -> X	delay(a) = 0.0;
		delay(a -> X) = 1.5;
		gate_delay(X) = 2.0
			0.0 + 1.5 + 2.0 = 3.5	3.5

	V -> X	delay(V) = 1.4;
		wire_delay(V -> X) = 0.2;
		gate_delay(X) = 2.0
			1.4 + 0.2 + 2.0 = 3.6	max (3.5, 3.6) = 3.6

Y	V -> Y	delay(V) = 1.4;
		wire_delay(V -> Y) = 0.2;
		gate_delay(X) = 2.0
			1.4 + 0.2 + 2.0 = 3.6	3.6

	W -> Y	delay(V) = 2.7;
		wire_delay(b -> V) = 0.1;
		gate_delay(V) = 2.0
			2.7 + 0.1 + 2.0 = 4.8	max (3.6, 4.8) = 4.8

Z	X -> Z	delay(X) = 3.6;
		wire_delay(X -> Z) = 1.3;
		gate_delay(Z) = 2.0
			3.6 + 1.3 + 2.0 = 6.9	6.9

	Y -> Z	delay(Y) = 4.8;
		wire_delay(Y -> Z) = 0.1;
		gate_delay(Z) = 2.0
			4.8 + 0.1 + 2.0 = 6.9	max (6.9, 6.9) = 6.9

f	**Z -> *f***	delay(Z) = 6.9;
		wire_delay(Z -> *f*) = 0.2;
		op_delay(f) = 0.0
			6.9 + 0.2 + 2.0 = 7.1	**7.1**

Critical path through back-tracing the Forward propagation algorithm:

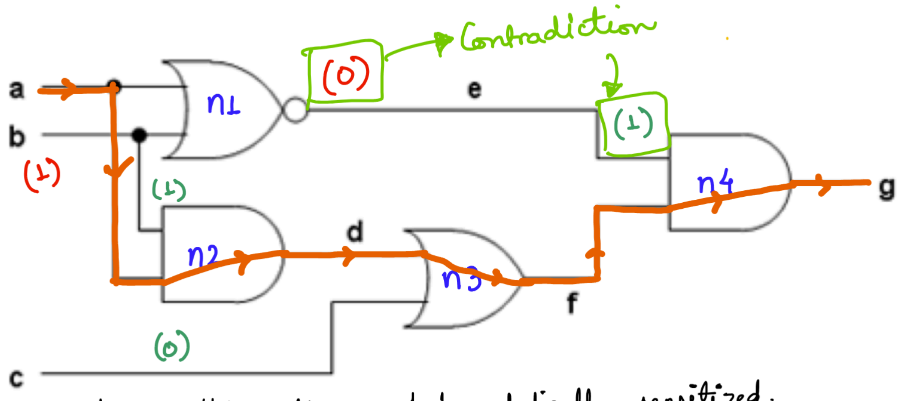
path - 1**:**	<Z, f> - <Y, Z> - <W, Y> - <d, W>	Critical path: **d -> W -> Y -> Z -> f**;
path - 2**:**	<Z, f> - <X, Z> - <V, X> - <b, V>	Critical path: **b -> V -> X -> Z -> f**;

maximum delay: **7.1**

Q2. Static Sensitization:

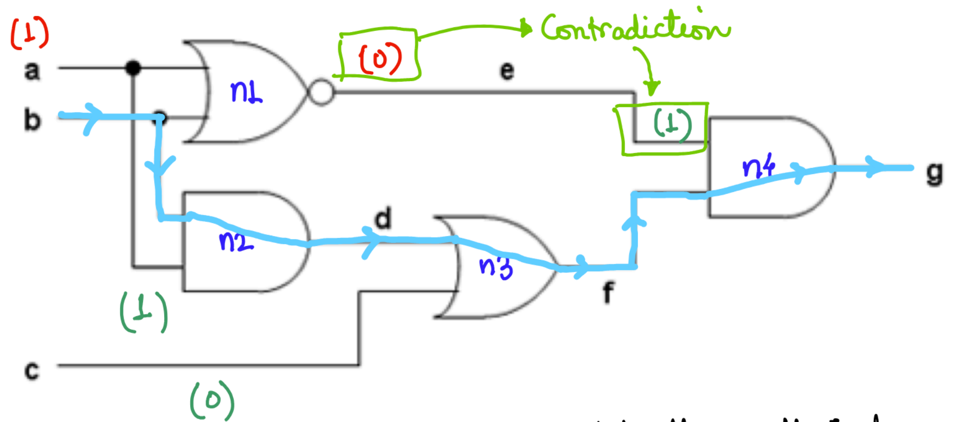
Solution:

False Path 1: a -> n2 -> d -> n3 -> f -> n4 -> g



Hence, this path cannot be statically sensitized.

False Path 2: b -> n2 -> d -> n3 -> f -> n4 -> g



Hence, this path cannot be statically synthesized.

Q3. Rocket-Chip Simulation Flow

Solution:

For this question, I modified the parameters within each .c file and Makefile. I have reported the set constraints below:

	EMULATOR - RTL			SPIKE - ISS		
Algorithm	parameter	avg. cycles/s	avg. CPI	parameter	avg. cycles/s	avg. CPI
Fibonacci	N = 20, MAXCYCLES = 500000	26054.34	1.12	N = 26 MAXCYCLES = 500000	330024.74	0.999
Gregory-Leibniz	# iterations = 1500, MAXCYCLES = 550000	32036.82	24.158	# iterations = 500000 MAXCYCLES = 550000	352828.71	0.999

Comment:

- In the generated *.emulator.out files, we can see the message Passed after a successful run. I used this to make sure that the design run is successful.
- However, during spike simulation, there was no such message in *.spike.out, so my metric for the choice of above parameters are to have the run in tens of seconds as instructed.

Explanation:

The discrepancies are included in the following:

a. Fibonacci is a recursive algorithm, which results in a higher clock cycle count compared to the iterative Gregory-Leibniz algorithm. Each individual integer operation in Fibonacci is relatively inexpensive, leading to an average CPI of around 1.12, which is reasonable.
b. On the other hand, the Gregory-Leibniz algorithm has a significantly higher average CPI of 24.158 due to the many ALU operations, including floating-point multiplication/division, which typically require 15–25 clock cycles on most modern RISC-V chips. This higher CPI is expected given the nature of the algorithm.
c. When it comes to average clock cycles per second, Gregory-Leibniz performs better due to its iterative structure, which allows for more efficient memory predictions. In contrast, Fibonacci's recursive nature results in performance degradation.
d. Since Spike simulation focuses on instruction emulation rather than detailed hardware behavior, there is minimal performance difference between Fibonacci and Gregory-Leibniz in terms of average cycles per second or CPI.