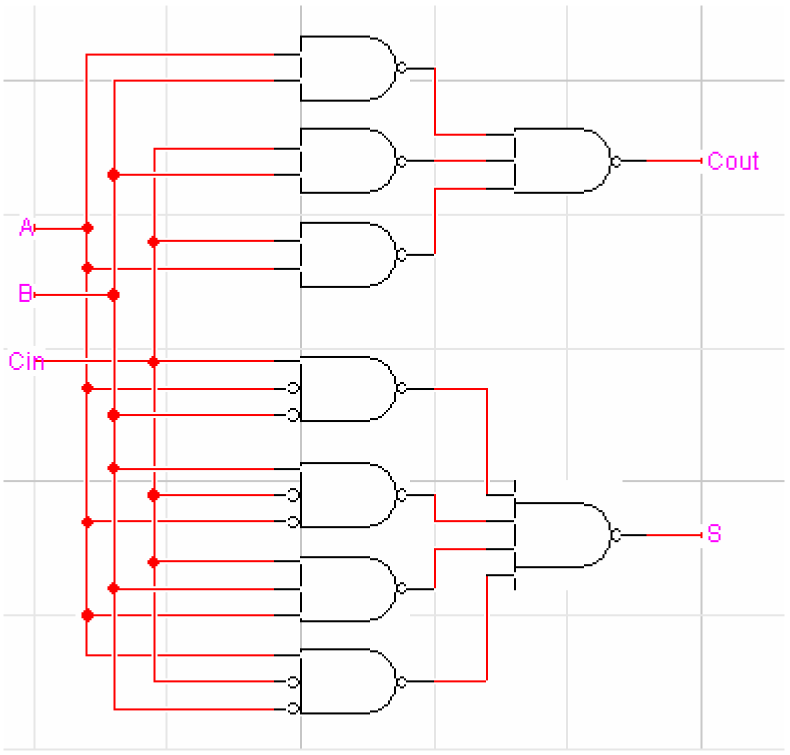
# Tutorial 5

## Part 1
## (From Tutorial 4)

**Truth Table for Full Adder:**

| $C_{in}$ | A | B | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Implementation of Full Adder:**



**Testing Full Adder:**

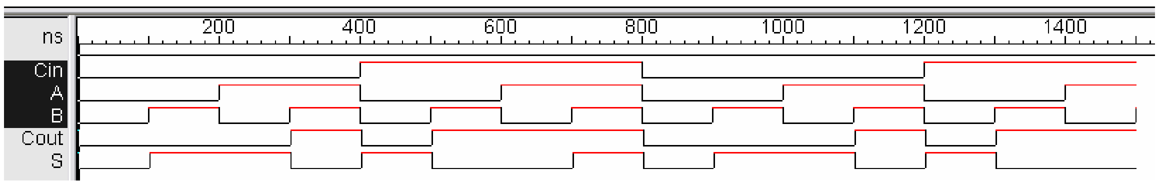
Test using a timing file (tab delimited, with column labeled $T (time) $D (delay) and $I (input) - make sure to use the same names for inputs as the labels you used on the wires in your circuit:

| $T | $D | $I Cin | $I A | $I B |
|---|---|---|---|---|
| 0 | 100 | 0 | 0 | 0 |
| | 100 | 0 | 0 | 1 |
| | 100 | 0 | 1 | 0 |
| | 100 | 0 | 1 | 1 |
| | 100 | 1 | 0 | 0 |
| | 100 | 1 | 0 | 1 |
| | 100 | 1 | 1 | 0 |
| | 100 | 1 | 1 | 1 |

Simulation->Import Timing...,

Fles of type: All Files

Navigate to your timing file.



Now, look at Cout and S

- they are the same as Cout (0 0 0 1 0 1 1 1) and S (0 1 1 0 1 0 0 1) from the original truth table
- so the circuit works properly.

**Part 2**
**From Tutorial 4...**
**Karnaugh Maps for functions with four inputs**

**Given a function:**

$$F = \sum \{ 0, 2, 3, 5, 7, 8, 10 \}$$

**Create a truth table:**

| A | B | C | D | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

**Karnaugh maps and minimizations:**

Sum of Products:

|  |  | **CD** |  |  |  |
|---|---|---|---|---|---|
|  |  | 00 | 01 | 11 | 10 |
|  | 00 | 1 | 0 | 1 | 1 |
| **AB** | 01 | 0 | 1 | 1 | 0 |
|  | 11 | 0 | 0 | 0 | 0 |
|  | 10 | 1 | 0 | 0 | 1 |

$$F = \overline{B}\,\overline{D} + \overline{A}\,B\,D + \overline{A}\,C\,D$$

Product of Sums:

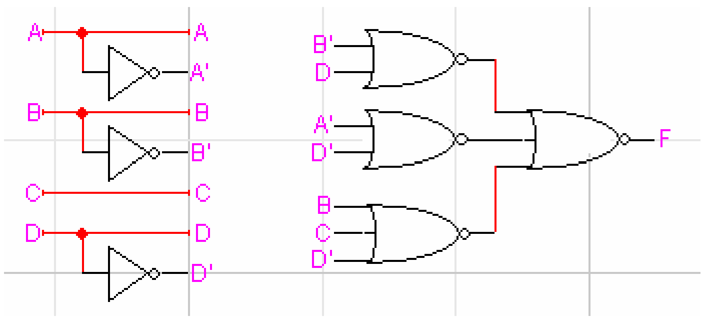
$$\begin{array}{c|c|cccc} & & \multicolumn{4}{c}{\textbf{CD}} \\ & & 00 & 01 & 11 & 10 \\ \hline & 00 & 1 & 0 & 1 & 1 \\ \textbf{AB} & 01 & 0 & 1 & 1 & 0 \\ & 11 & 0 & 0 & 0 & 0 \\ & 10 & 1 & 0 & 0 & 1 \end{array}$$

$$F = (\overline{B} + D)(\overline{A} + \overline{D})(B + C + \overline{D})$$

## Part 3
## Implementing the Function from Part 2

### A) NOR-NOR Logic:

Notice how labeling the wires simplifies the circuit:

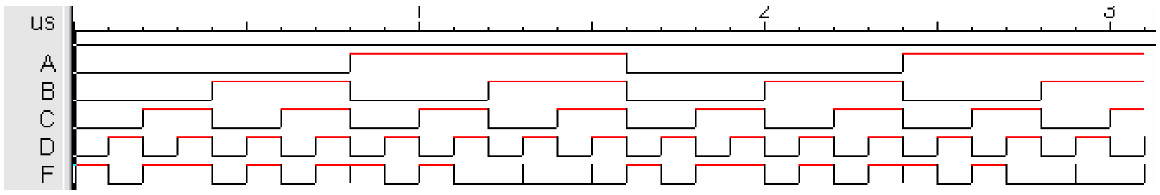


Create a timing file:

| $T | $D | $I A | $I B | $I C | $I D |
|---|---|---|---|---|---|
| 0 | 100 | 0 | 0 | 0 | 0 |
| | 100 | 0 | 0 | 0 | 1 |
| | 100 | 0 | 0 | 1 | 0 |
| | 100 | 0 | 0 | 1 | 1 |
| | 100 | 0 | 1 | 0 | 0 |
| | 100 | 0 | 1 | 0 | 1 |
| | 100 | 0 | 1 | 1 | 0 |
| | 100 | 0 | 1 | 1 | 1 |
| | 100 | 1 | 0 | 0 | 0 |
| | 100 | 1 | 0 | 0 | 1 |
| | 100 | 1 | 0 | 1 | 0 |
| | 100 | 1 | 0 | 1 | 1 |
| | 100 | 1 | 1 | 0 | 0 |
| | 100 | 1 | 1 | 0 | 1 |

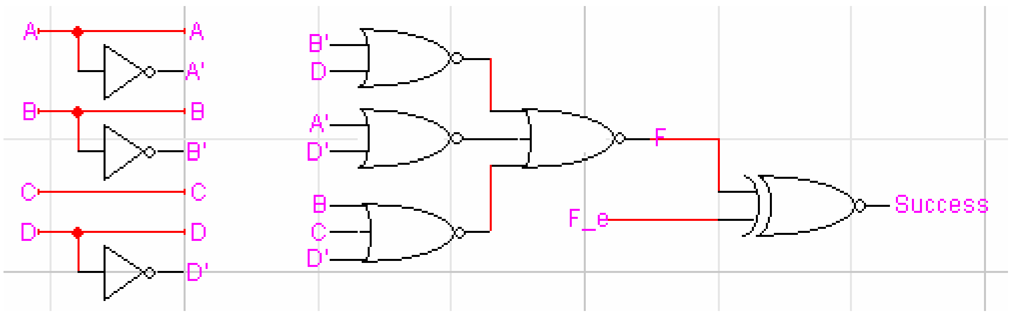
| | | | | |
|---|---|---|---|---|
| 100 | 1 | 1 | 1 | 0 |
| 100 | 1 | 1 | 1 | 1 |
| 100 | 0 | 0 | 0 | 0 |
| 100 | 0 | 0 | 0 | 1 |
| 100 | 0 | 0 | 1 | 0 |
| 100 | 0 | 0 | 1 | 1 |
| 100 | 0 | 1 | 0 | 0 |
| 100 | 0 | 1 | 0 | 1 |
| 100 | 0 | 1 | 1 | 0 |
| 100 | 0 | 1 | 1 | 1 |
| 100 | 1 | 0 | 0 | 0 |
| 100 | 1 | 0 | 0 | 1 |
| 100 | 1 | 0 | 1 | 0 |
| 100 | 1 | 0 | 1 | 1 |
| 100 | 1 | 1 | 0 | 0 |
| 100 | 1 | 1 | 0 | 1 |
| 100 | 1 | 1 | 1 | 0 |
| 100 | 1 | 1 | 1 | 1 |

Import the timing file, this is the result:



We could directly check F against the truth table - but it is hard to be sure it is correct since there are now 16 cases to check.

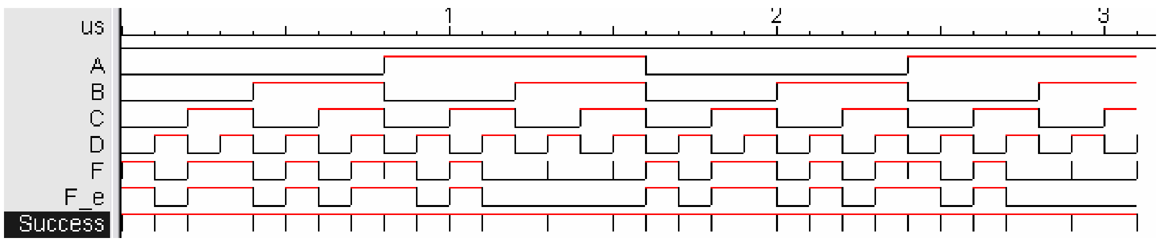
An easier way is to create a circuit that can test F against the expected value of F, F_e:



Add a column to the timing file, F_e, which gives the expected value of F:

| $T | $D | $I A | $I B | $I C | $I D | $I F_e |
|---|---|---|---|---|---|---|
| 0 | 100 | 0 | 0 | 0 | 0 | 1 |
| | 100 | 0 | 0 | 0 | 1 | 0 |
| | 100 | 0 | 0 | 1 | 0 | 1 |
| | 100 | 0 | 0 | 1 | 1 | 1 |
| | 100 | 0 | 1 | 0 | 0 | 0 |
| | 100 | 0 | 1 | 0 | 1 | 1 |
| | 100 | 0 | 1 | 1 | 0 | 0 |
| | 100 | 0 | 1 | 1 | 1 | 1 |
| | 100 | 1 | 0 | 0 | 0 | 1 |
| | 100 | 1 | 0 | 0 | 1 | 0 |
| | 100 | 1 | 0 | 1 | 0 | 1 |
| | 100 | 1 | 0 | 1 | 1 | 0 |
| | 100 | 1 | 1 | 0 | 0 | 0 |
| | 100 | 1 | 1 | 0 | 1 | 0 |
| | 100 | 1 | 1 | 1 | 0 | 0 |
| | 100 | 1 | 1 | 1 | 1 | 0 |
| | 100 | 0 | 0 | 0 | 0 | 1 |
| | 100 | 0 | 0 | 0 | 1 | 0 |
| | 100 | 0 | 0 | 1 | 0 | 1 |
| | 100 | 0 | 0 | 1 | 1 | 1 |
| | 100 | 0 | 1 | 0 | 0 | 0 |
| | 100 | 0 | 1 | 0 | 1 | 1 |
| | 100 | 0 | 1 | 1 | 0 | 0 |
| | 100 | 0 | 1 | 1 | 1 | 1 |
| | 100 | 1 | 0 | 0 | 0 | 1 |
| | 100 | 1 | 0 | 0 | 1 | 0 |
| | 100 | 1 | 0 | 1 | 0 | 1 |
| | 100 | 1 | 0 | 1 | 1 | 0 |
| | 100 | 1 | 1 | 0 | 0 | 0 |
| | 100 | 1 | 1 | 0 | 1 | 0 |
| | 100 | 1 | 1 | 1 | 0 | 0 |
| | 100 | 1 | 1 | 1 | 1 | 0 |

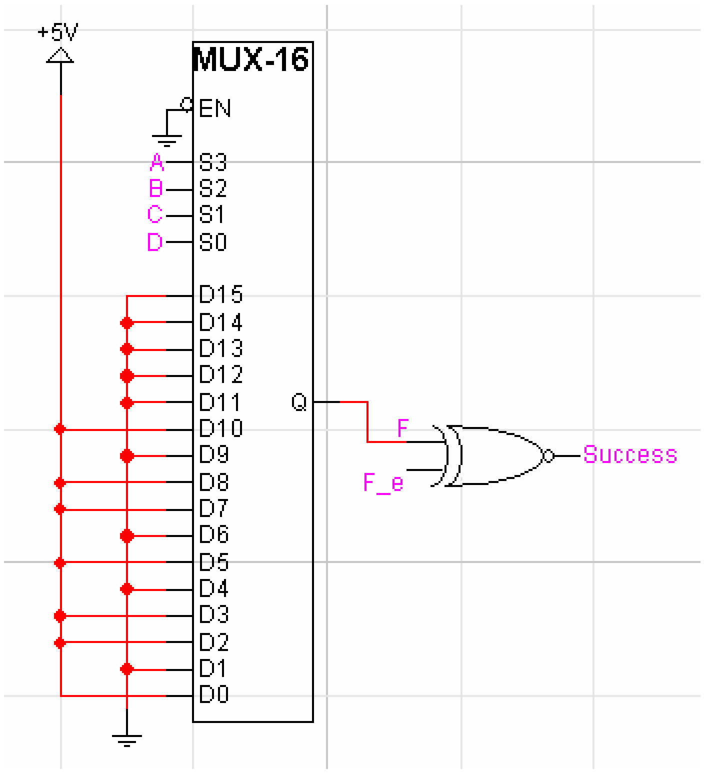
Now, run the circuit with the new timing file.  This is the result:



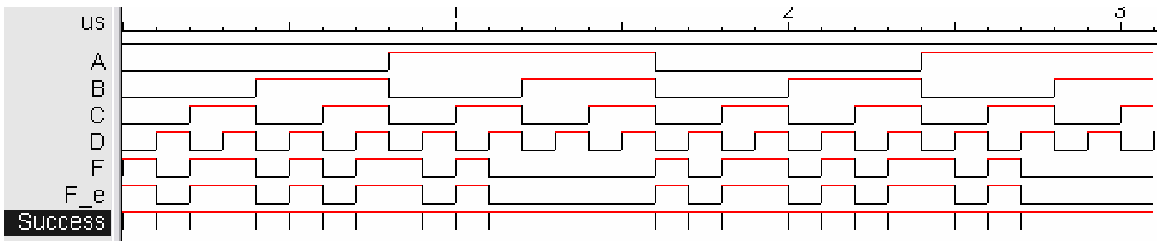
By looking at the Success line, we can see that the circuit always works.

**B)  MUX-16:**

Can implement any circuit with *n* inputs using a multiplexer with $2^n$ lines, with each line connected to 0 or 1 according to the truth table.  So, for this circuit, we will need an MUX-16.  Since logicworks doesn't have one, you can make one using two MUX-8's (you don't need to encapsulate it, but so as not to give everything away I've encapsulated one for the purposes of the tutorial):



Here's the resulting timing file: it works!



## C) MUX-8:

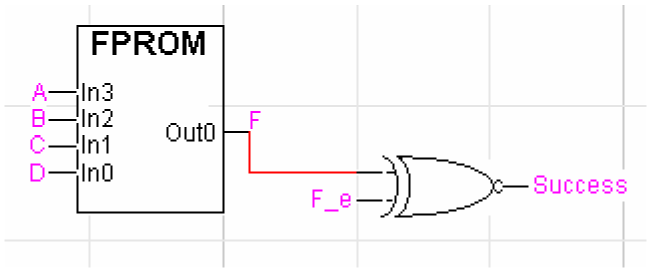
Note that the truth table can be grouped into twos, as follows:

| *A* | *B* | *C* | *D* | *F* |  |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | D' |
| 0 | 0 | 0 | 1 | 0 |  |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |  |
| 0 | 1 | 0 | 0 | 0 | D |
| 0 | 1 | 0 | 1 | 1 |  |
| 0 | 1 | 1 | 0 | 0 | D |
| 0 | 1 | 1 | 1 | 1 |  |
| 1 | 0 | 0 | 0 | 1 | D' |
| 1 | 0 | 0 | 1 | 0 |  |
| 1 | 0 | 1 | 0 | 1 | D' |
| 1 | 0 | 1 | 1 | 0 |  |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |  |
| 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |  |

So, could use an MUX-8 with A, B, and C as the select lines, and then connect 0, 1, D, or D' for each combination of A, B, and C.

## D) PROM:

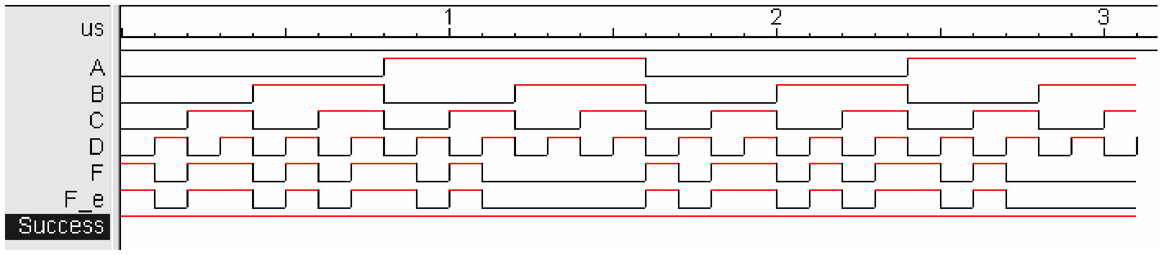
Use the PROM wizard to create a PROM with 4 address lines, and 1 bit output:



Here's the contents of the PROM:

```
1 0 1 1
0 1 0 1
1 0 1 0
0 0 0 0
```

And here's the resulting timing file:



It works!