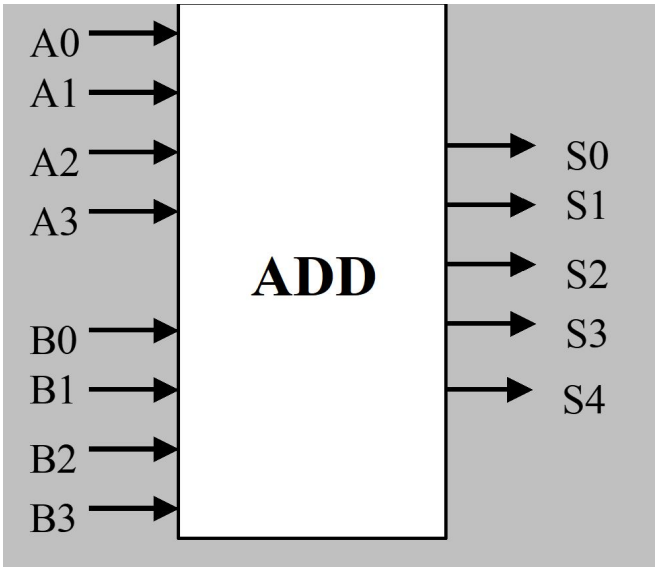
ISTD 50.002 Computation Structures

Lab #2 Adder

Modified by: Kenny Choo, Natalie Agus, Oka Kurniawan 2020.

Introduction

Your mission this week is to design and test a CMOS circuit that performs addition on two unsigned 4-bit numbers, producing a 5-bit result:



Submission

When you have completed and tested your design, use JSim to send your circuit to the online assignment system using the process we described at the end of Lab #1. The checkoff file for Lab #2 (*lab2checkoff.jsim*) checks that your circuit has the right functionality; the online system will give you 5 points for checking off your lab using this file.

> ***Note**: Our ability to provide automated checkoffs is predicated on trusting that you will use the checkoff and library files as given. Since these files are included in your submission, we will be checking to see if these files have been used as intended.* ***Submissions that include modified checkoff or library files will be regarded as a serious breach of our trust and will be dealt with accordingly**.*

Due Date

Refer to the course handout for due date.

Simple 4-bit Ripple-Carry Adder

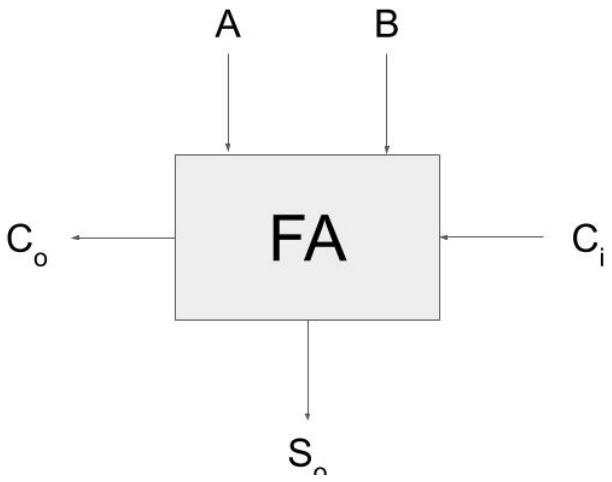
Let's start a simple 1-bit **full-adder** module before proceeding to create a 4-bit Ripple-Carry adder. Later we will discuss higher performance adder architectures you can use in the implementation of the Beta (the computer central processing unit we will be designing in later labs).

Full Adder

The full adder module has 3 inputs (A, B, and C_i) and 2 outputs (S and C_o). The logic equations and truth tables for S and C_o are shown below.

Logic Equations: $S = A \oplus B \oplus C_{in}$ $C_o = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$

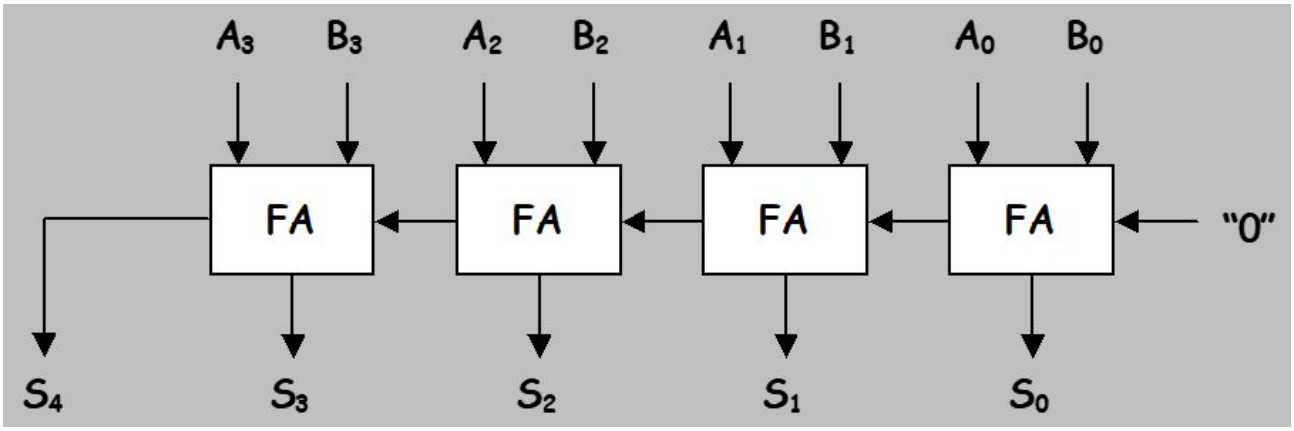
C_i	A	B	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



S is typically implemented using two cascaded 2-input XOR gates. One can use three 2-input NANDs and one 3-input NAND to implement C_o (remember that by De Morgan's Law, two cascaded NANDs are logically equivalent to a cascade of AND/OR).

Cascade Full Adders to get the Ripple-Carry Adder

The module performs the addition of two one-bit inputs (A and B) incorporating the carry in from the previous stage (C_i). The result appears on the S output and a carry (C_o) is generated for the next stage. A possible schematic for the 4-bit adder is shown below:

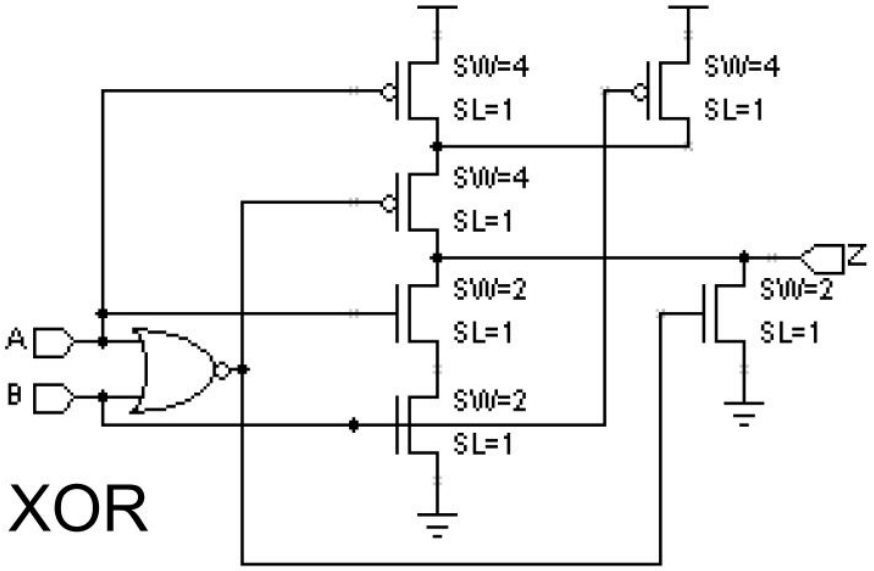


4-bit Ripple-Carry Adder

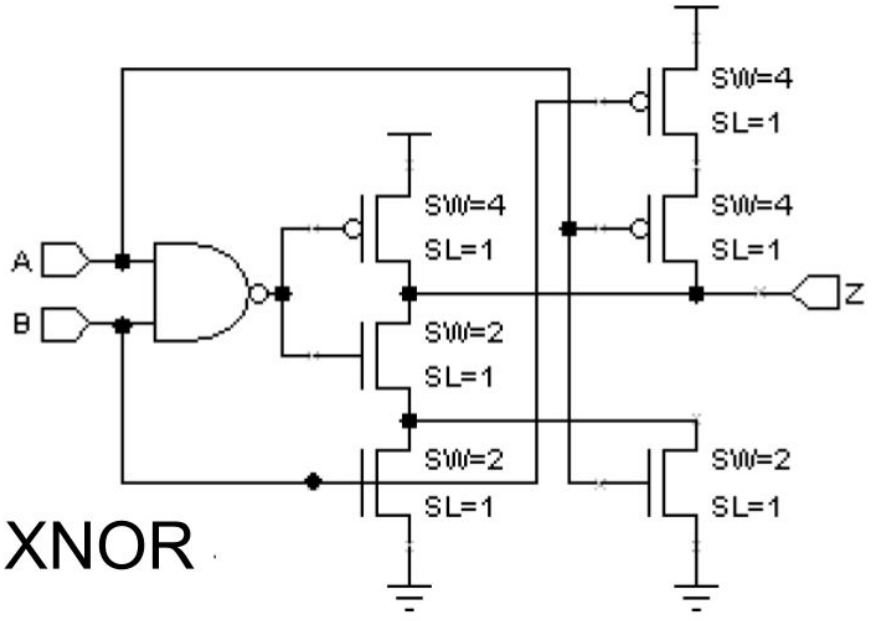
Building Logic Gates

Since logic gates are used to implement the logic for the full adder, a good place to start is to 1) **build** your own gate library (e.g., inverter, 2-input NAND, 2-input NOR, 2-input XOR), 2) **test** them individually, and then 3) use them to **implement** your design. *It's much easier to debug your circuit module-by-module rather than as one big lump*. XOR/XNOR can be challenging gates to design; here's ***one suggestion*** for how they might be implemented:

XOR gate schematic



XNOR gate schematic



Testing your Circuit

You can use voltage sources with either a pulse or piece-wise linear waveforms to generate test signals for your circuit (see Lab #1 for details). Another source of test waveforms is the file "/50002/8clocks.jsim" which can be included in your netlist. It provides eight different square waves (50% duty cycle) with different periods:

clk1	period = 10ns
clk2	period = 20ns
clk3	period = 40ns
clk4	period = 80ns
clk5	period = 160ns
clk6	period = 320ns
clk7	period = 640ns
clk8	period = 1280ns

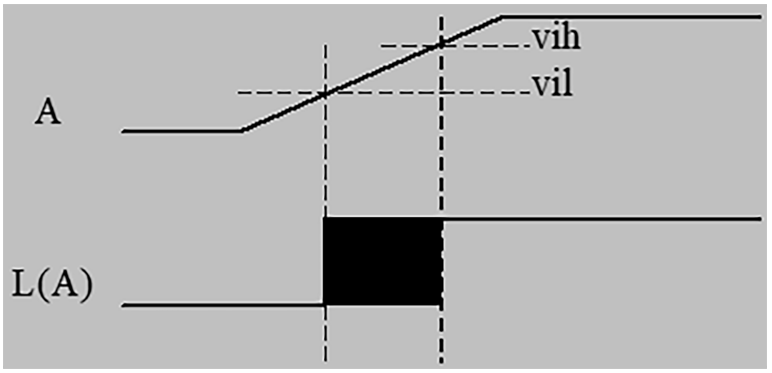
For example, to completely test all possible input combinations for a 2-input gate, you could connect clk1 and clk2 to the two inputs and simulate for 20ns.

Interpreting analogue signal levels using JSim

Interpreting analog signal levels as logic values can be tedious. JSim will do it for you automatically if you ask to plot "L(a)" instead of just "a". The logic-high and logic-thresholds are determined by the "vih" and "vil" options:

```
.options vih=2.6 vil=0.6
```

Initial values are specified in "/50002/nominal.jsim", but you can respecify them in your own netlist. Voltages between vil and vih are displayed as a filled-in rectangle to indicate that the logic value cannot be determined. For example:



You can also ask for the values of a set of signals to be displayed as a bus, e.g., "L(a3,a2,a1,a0)". The signals should be listed, most-significant bit first. A bus waveform is displayed as a

filled-rectangle if any of the component signals has an invalid logic level or as a hexadecimal value otherwise.

In the following plot the four signals a3, a2, a1 and a0 are interpreted as a 4-bit integer where the high-order bit (a3) is making a 1→0 transition.

That is, the value of L was initially `1111 (0xF)`, then the highest bit makes a transition to zero, and the value of L ultimately become `0111 (0x7)`.

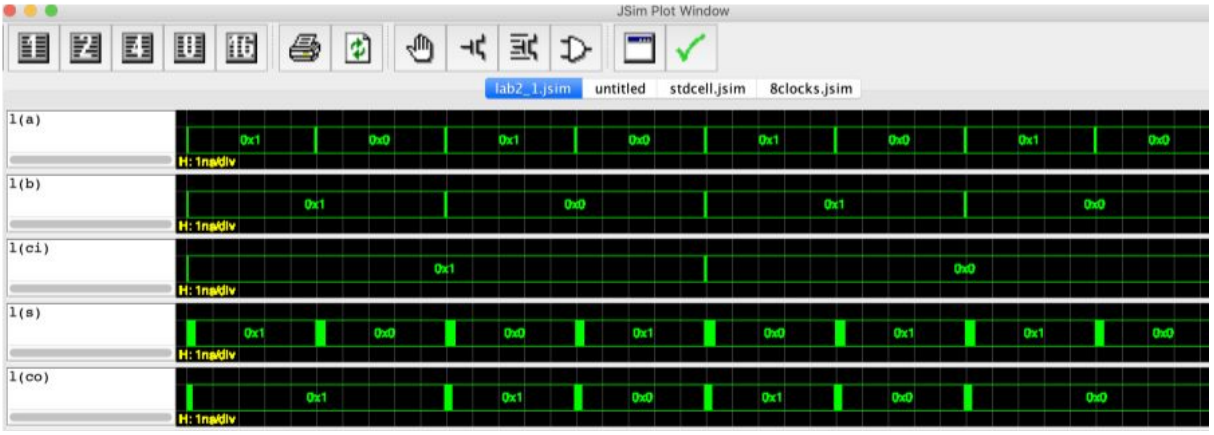
The filled-in rectangle represents the period of time during which a3 transitions from V_{IH} to V_{IL}, rendering them invalid denoted in the shaded rectangle:

Below is the screenshot of the JSIM plot that you will see after you complete your code and would like to test the answer. You can use the plot to check if your code implements the functional specification of a 4-bit adder correctly. Each row is a plot of the values of the terminals over time. For simplicity, they're ***represented*** in **hex** instead of in binary.

So for example, the value of `a(1-bit)` is periodically changed between 1 and 0 (but when represented as hex it becomes 0x1 and 0x0 respectively).

Note that a doesn't necessarily have to be strictly 4 bits in length to the actual circuit your texting. The plot is merely a digital value representation of the signal in ***hex****.*

The value of `b` and `ci` is initially `0` as well. The expected output when `a=1, b=1`, and `ci = 1` is `s=1` and `c0=1`. The plot confirms that this is true since the values of `s` and `c0` are shown to indeed be 1 given that combination of `a=1, b=1,` and `ci=1.`



Suggested Design Tasks to Complete the Lab

Step 1

Draw a ***gate-level schematic*** for the full-adder module. XOR gates can be used to implement the S output; two levels of NAND gates are handy for implementing C_o as a sum of products.

Step 2

Create a MOSFET circuit for each of the logic gates you used in Step 1.

Step 3

Enter .subckt definitions in your netlist for each of the logic gates. Use JSim to test each logic gate with all possible combinations of inputs. Debugging your gate designs one-by-one will be much easier than trying to debug them as part of the adder circuit. Here's a sample netlist for testing a 2-input NAND gate called nand2:

```
.include "/50002/nominal.jsim"
.include "/50002/8clocks.jsim"

.subckt nand2 a b z
… internals of nand2 circuit here
.ends

Xtest clk1 clk2 z nand2
.tran 20ns
.plot clk1
.plot clk2
.plot z
```

Step 4

Enter a .subckt definition for the full-adder, building it out of the gates you designed and tested above. Use JSim to test your design with all 8 possible combinations of the three inputs. At this point you probably want to switch to using "Fast Transient Analysis" to do the simulations as it is much faster than "Device-level Simulation".

Step 5

Enter the netlist for the 4-bit adder and test the circuit using input waveforms supplied by lab2checkoff.jsim. Note that the checkoff circuitry expects your 4-bit adder to have exactly the terminals shown below – the inside circuitry is up to you, but the ".subckt ADDER4…" line in your netlist should match exactly the one shown below.

```
.include "/50002/nominal.jsim"
.include "/50002/lab2checkoff.jsim"

… subckt definitions of your logic gates

.subckt FA a b ci s co
… full-adder internals here
.ends

.subckt ADDER4 a3 a2 a1 a0 b3 b2 b1 b0 s4 s3 s2 s1 s0
* remember the node named "0" is the ground node
* nodes c0 through c3 are internal to the ADDER module
Xbit0 a0 b0 0 s0 c0 FA
Xbit1 a1 b1 c0 s1 c1 FA
Xbit2 a2 b2 c1 s2 c2 FA
Xbit3 a3 b3 c2 s3 s4 FA
.ends
```

lab2checkoff.jsim contains the necessary circuitry to generate the appropriate input waveforms to test your adder. It includes a .tran statement to run the simulation for the appropriate length of time and a few .plot statements showing the input and output waveforms for your circuit.

When debugging your circuits, you can plot additional waveforms by adding .plot statements to the end of your netlist. For example, to plot the carry-out signal from the first full adder, you could say

```
.plot Xtest.c0
```

where Xtest is the name lab2checkoff.jsim gave to the ADDER4 device it created and c0 is the name of the internal node that connects the carry-out of the low-order FA to the carry-in of the next FA.