Lab Document #3 Report Required: Yes
Building Gate-Level Circuits Due In: 1 week

Learning Objectives:

- Create a 32-bit ripple-carry adder using JSim gate-level components.
- Modify the adder so it can perform either addition or subtraction.

For our third lab assignment, we now leave the world of device-level simulations and try our hand at gate-level simulations. No more resistors, capacitors, and transistors. JSim will now provide us with all the logic gates and devices that we need.

To use JSim's gate-level simulation capabilities, your netlist must include the file stdcell.jsim or else your circuit will not run. And use the gate-level simulation button, not the old device-level one.

There are many gates available in JSim. For those of you who demand information normally reserved for hardcore computer engineers, the propagation and contamination delays (in nanoseconds) and the rise and fall times (in nanoseconds / pico-Farads) in the list below have been taken from a 0.18 micron CMOS process measured at room temperature. Load is in pico-Farads, size is in square microns.

Netlist	*Function*	t_{CD}	t_{PD}	t_R	t_F	load	size
`Xid z constant0`	*Z = 0*	–	–	–	–	–	0
`Xid z constant1`	*Z = 1*	–	–	–	–	–	0
`Xid a z inverter`	*Z = ~A*	.005	.02	2.3	1.2	.007	10
`Xid a z inverter_2`		.009	.02	1.1	.6	.013	13
`Xid a z inverter_4`		.009	.02	.56	.3	.027	20
`Xid a z inverter_8`		.02	.11	.28	.15	.009	56
`Xid a z buffer`	*Z = A*	.02	.08	2.2	1.2	.003	13
`Xid a z buffer_2`		.02	.07	1.1	.6	.005	17
`Xid a z buffer_4`		.02	.07	.56	.3	.01	30
`Xid a z buffer_8`		.02	.07	.28	.15	.02	43
`Xid e a z tristate`	*Z = A* when *e=1* else *Z* not driven	.03	.15	2.3	1.3	.004	23
`Xid e a z tristate_2`		.03	.13	1.1	.6	.006	30
`Xid e a z tristate_4`		.02	.12	.6	.3	.011	40
`Xid e a z tristate_8`		.02	.11	.3	.17	.02	56
`Xid a b z and2`	*Z = AB*	.03	.12	4.5	2.3	.002	13
`Xid a b c z and3`	*Z = ABC*	.03	.15	4.5	2.6	.002	17
`Xid a b c d z and4`	*Z = ABCD*	.03	.16	4.5	2.5	.002	20
`Xid a b z nand2`	*Z = ~(AB)*	.01	.03	4.5	2.8	.004	10
`Xid a b c z nand3`	*Z = ~(ABC)*	.01	.05	4.2	3.0	.005	13
`Xid a b c d z nand4`	*Z = ~(ABCD)*	.01	.07	4.4	3.5	.005	17

`Xid a b z or2`	$Z = A+B$	.03	.15	4.5	2.5	.002	13
`Xid a b c z or3`	$Z = A+B+C$	.04	.21	4.5	2.5	.003	17
`Xid a b c d z or4`	$Z = A+B+C+D$	.06	.29	4.5	2.6	.003	20
`Xid a b z nor2`	$Z = \sim(A+B)$	.01	.05	6.7	2.4	.004	10
`Xid a b c z nor3`	$Z = \sim(A+B+C)$	.02	.08	8.5	2.4	.005	13
`Xid a b c d z nor4`	$Z = \sim(A+B+C+D)$	.02	.12	9.5	2.4	.005	20
`Xid a b z xor2`	$Z = A\hat{}B$	.03	.14	4.5	2.5	.006	27
`Xid a b z xnor2`	$Z = \sim(A\hat{}B)$	.03	.14	4.5	2.5	.006	27
`Xid a b c z aoi21`	$Z = \sim((AB)+C)$	.02	.07	6.8	2.7	.005	13
`Xid a b c z oai21`	$Z = \sim((A+B)C)$	.02	.07	6.7	2.7	.005	17
`Xid s d0 d1 z mux2`	$Z = D0$ when $S = 0$ $Z = D1$ when $S = 1$	.02	.12	4.5	2.5	.005	27
`Xid s0 s1 d0 d1 d2 d3 z mux4`	$Z = D0$ when $S_{10} = 00$ $Z = D1$ when $S_{10} = 01$ $Z = D2$ when $S_{10} = 10$ $Z = D3$ when $S_{10} = 11$	.04	.19	4.5	2.5	.006	66
`Xid d clk q dreg`	$D \rightarrow Q$ on $CLK\uparrow$ *tsetup = .15, thold = 0*	.03	.19	4.3	2.5	.002	56

You will note that the order of inputs in these devices are from least to most significant. If something doesn't seem to be working correctly, you may have used the wrong order of nodes. And, to make matters worse, the order of bits used for the W-voltage source (see below) is most to least significant. Don't get confused!

Generating 32-bit data for tests can be tedious using 1-output voltage sources and piece-wise linear specifications. JSim includes a “W” voltage source that generates digital waveforms for many nodes (e.g., a bus) at once:

Wid nodes… nrz(vlow, vhigh, tperiod, tdelay, trise, tfall) data…

If N nodes are specified, think of them as an N-bit value where the node names are listed most-significant bit first. The “W” source will set those nodes to a sequence of data values using the data specified at the end of the “W” statement. At each step of the sequence, the N low-order bits of each data value will be used to generate the appropriate voltage for each of the N nodes. The voltage and timing of the signals is given by the nrz parameters:

vlow = voltage used for a logic low value (usually 0)
vhigh = voltage used for a logic high value (usually 5)
tperiod = interval (in seconds) at which values will be changed
tdelay = initial delay (in seconds) where nodes are assigned the value 0 and not the data values
trise = rise time (in seconds) for low-to-high transitions
tfall = fall time (in seconds) for high-to-low transitions

Note that the times are specified in seconds, so don’t forget to specify the “n” scale factor when

entering times!

Sample code below, remember to include 8clocks, nominal, and stdcell!

```
Wtest data[7:0] nrz(0, 5, 10ns, 10ns, 0.1ns, 0.1ns)
+ 5 -1 123 0x18 0b10101010
.tran 60ns
.plot data[7:0]
```

As you can see, data values can be in decimal, hex ("0x" prefix), octal ("0" prefix) or binary ("0b" prefix). The last data value is maintained until the end of the analysis if necessary. In a gate-level analysis, JSim plots a set of nodes as a single multi-bit data value instead of multiple plots.

If you zoomed in on one of the transition times, you would see that the values actually turn to an invalid signal for 0.1ns while making the transition between valid logic levels. This particular "green bar of death" is okay, since it represents the transition through the "forbidden zone", which cannot be avoided. If the green bar of death is too long (more than 500ns, maybe?) then there might be something wrong with your circuit.

In gate-level simulations, JSim also has additional control statements that you can use:

.connect node1 node2 node3 ...

The .connect statement merges all the nodes specified into a single node. Be wary when typing something like:

```
.connect a[5:0] b[5:0]
```

as this will connect ALL 12 nodes together as a single node. If your intention is to connect corresponding bits (a5 and b5, a4 and b4, etc.) you may wish to consider using a subcircuit instead:

```
.subckt knex a b
.connect a b
.ends
```

which brings us to the .subckt statement, which allows you to define a "blueprint" for a device, then instantiate that device as many times as needed.

.subckt DeviceName node1 node2 node3 ...
 # subcircuit definition for DeviceName here
.ends

The blueprint (and therefore its instantiations) is a self-contained world: It cannot see anything outside (including `vdd` and `clk[8:1]`) except the special `0` (ground) node. A node inside a blueprint can have the same name as a node outside or in another blueprint, since they can't "see" each other. Inside a subcircuit, however, you can instantiate any JSim gate and any subcircuit you have already defined.

Instantiating a subcircuit is similar to instantiating any of JSim's gates (which are also subcircuits). To instantiate the knex subcircuit once:

```
Xconn someNode anotherNode knex
```

Now why bother with a connector subcircuit if you can simply type multiple `.connect` statements instead? Well, you can also instantiate subcircuits with a single line:

```
Xconn a[5:0] b[5:0] knex
```

JSim first determines how many instantiations (in this case, 6) you need based on the number of nodes provided, then proceeds to assign the nodes to each instantiation. In this case, a5 is assigned as the first node to the first instantiation, a4 to the next, a3 to the one after... and a0 to the last. Then, b5 is assigned as the second node to the first instantiation... etc. In other words, the above line can be rewritten as:

```
Xconn#0 a5 b5 knex
Xconn#1 a4 b4 knex
Xconn#2 a3 b3 knex
Xconn#3 a2 b2 knex
Xconn#4 a1 b1 knex
Xconn#5 a0 b0 knex
```

You can also repeat a node in a multi-instantiation line:

```
Xtest switchNode#4 dataZero[3:0] dataOne[3:0] out[0:3] mux2
```

which results in:

```
Xtest#0 switchNode dataZero3 dataOne3 out0 mux2
Xtest#1 switchNode dataZero2 dataOne2 out1 mux2
Xtest#2 switchNode dataZero1 dataOne1 out2 mux2
Xtest#3 switchNode dataZero0 dataOne0 out3 mux2
```

Notice that the order of the “array indices” in a multi-node declaration is important here.

I. Building the Adder

Demo: Subcircuit for a half-adder. Also, how to instantiate and test.

Discussion: Full-adder truth table

1. Draw a gate-level schematic for your full-adder module.

2. Generate a subcircuit definition for your full-adder. *Must be optimized for BOTH speed AND cost. Refer to lecture slides on “bubbles” for a hint.*

3. Generate a subcircuit definition for your 32-bit ripple-carry adder.

Inputs:
A: 32-bit signed integer
B: 32-bit signed integer
Cin: Initial carry-in bit (see note below)

Outputs:
F: 32-bit signed integer equal to A+B

You will be penalized if your definition uses 32 lines to instantiate the 32 full-adders needed. There is a way to do it with just TWO lines.
Note: Your adder may have to take in an initial carry-in bit (Cin) instead of always defaulting it to 0. If you modify your subcircuit to do this, then there is a way to *instantiate your 32 full-adders with just ONE line.*
Hint: Cin should be renamed such that it has a number in its name, preferably a 0.

II. Building the Adder/Subtractor

Discuss: Where part I does F=A+B, part II has the option do F=A–B

Guide questions (class participation):

- How are negative numbers represented in 2's-complement? Given X, can you write –X as a function based on this representation?
- Does the adder work only for unsigned numbers? Does the adder function correctly even if A, B, and F are signed integers (assume answers are within the 32-bit range)?
- Is there another way to write A–B, especially since we already have an adder at this point?
- Given OP, how do you tell the circuit what function it should perform, and yet use only ONE adder subcircuit instantiation?

1. Generate a subcircuit definition for a 32-bit adder/subtractor. Include test code (see note below). *You are allowed to instantiate ONLY ONE 32-bit adder inside the definition. The rest of your circuitry will have to somehow make use of the adder and still be capable of subtraction if necessary.*

Inputs:
A: 32-bit signed integer
B: 32-bit signed integer
OP: A bit that determines what operation to perform
0 = add
1 = subtract

Outputs:
F: 32-bit signed integer equal to
A+B if OP == 0
A–B if OP == 1

Note: You may want to perform these tests to check if your adder/subtractor works:

```
0x00000000 + 0x00000000 = ?
0x55555555 + 0x00000000 = ?
0x00000000 + 0x55555555 = ?
0x55555555 + 0x55555555 = ?
0xAAAAAAAA + 0x00000000 = ?
0x00000000 + 0xAAAAAAAA = ?
```

```
0xAAAAAAAA + 0xAAAAAAAA = ?
0x00000000 - 0x00000000 = ?
0x00000000 - 0xFFFFFFFF = ?
0x00000001 + 0xFFFFFFFF = ?
0xFFFFFFFF + 0x00000001 = ?
0xFFFFFFFF - 0xFFFFFFFF = ?
```