University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2003 J. Kubiatowicz
Homework/Lab #3

Homework 3 / Lab 3
Exercises Due Wednesday 3/05/03 in Lecture
Lab Due Thursday 3/06/03 at midnight via submit
CS152 Computer Architecture and Engineering

PLEASE put the TIME or TA NAME of the DISCUSSION section that you attend as well as your NAME, STUDENT ID, and CLASS ACCOUNT. Homeworks will be handed back in discussion.

Homework Policy: Homework assignments are due in class. No late homeworks will be accepted. There will be a short quiz in lecture the day the assignment is due; the quiz will be based on the homework. Study groups are encouraged, but what you turn in must be your own work.

Lab Policy: Labs are due at midnight on Thursdays via submit program. No late labs will be accepted.

As decided in class, the penalty for cheating on homework or labs is no credit for the full assignment.

Homework 3:

Please do the following problems from P&H: 4.13, 4.18, 4.23, 4.24, 4.37, 4.44, 4.46, 4.51, 4.52, 4.53, 4.54

Lab 3:

The objective of this assignment is to get acquainted with the Xilinx ECS Schematic editor, and ModelSim for logic and timing simulation. You will be using the Windows-2000 workstations in 119 Cory. Make sure to start early, since there is a fairly steep learning curve with the tool set.

You should start by reading Ben Liao's tutorial on the Xilinx tool set. This tutorial is available off the handouts page:

`http://inst.eecs.berkeley.edu/~cs152/handouts/Xilinx_tutorial.html`

Note: It is important to go through this tutorial before attempting this lab.

You will be using the logic elements introduced in class to implement a 32-bit adder. You are allowed to use the following cells:

nand2	nand3	nand4
nor2	nor3	nor4
xor2	xnor2	dff

As part of this lab, we want you to keep an on-line notebook as discussed in lecture. For each problem, create a separate entry in the notebook. Be sure to follow these steps:

1. Open an editor and keep it running while you work. This editor should only be used to maintain your online notebook.

2. When starting a new entry, type "date" at a unix prompt. Copy and paste it into your notebook.

3. Write down your goal for that entry.

4. Make note of accomplishments, bugs, or insights you may develop as you work.

5. Copy the output of waveforms (HDL Bencher output, for instance) at key points in development.

6. At the end of a session, type "date" at a unix prompt. Copy and paste it into your notebook.

Problem 1: Timing Characteristics

This problem will give you an opportunity to design a few simple circuits and measure their timing characteristics using Xilinx ECS schematic editor and Modelsim. Although you can do behavioral simulation to test functionality, you should make sure to use post-place-and-route simulation for actual timing. Make sure to turn in your Schematics and Waveform outputs in your lab report.

1.a: First measure the timing delays of an XOR gate and of a two-input NAND gate (Low to High and High to Low, load-dependent delays) using post-place-and-route timings. When calculating the load-dependent delay, assume that the input capacitance for a NAND gate is 50pf; you can use NAND gates with the two inputs tied together as capacitors. Take several delay data points to try to compute the load-dependent delay.

1.b: Use the schematic editor, create a circuit that implements the XOR function, using five NAND gates (ie. $a \oplus b = \overline{\overline{a \cdot \overline{b \cdot b}} \cdot \overline{\overline{a \cdot a} \cdot b}}$). Simulate it with post-place-and-route delays using Modelsim and measure the timing delays (Low to High and High to Low) from input A to output Q with input B set to 0. These are the "Internal Delays" of the XOR gate from input A to output Q.

1.c: Predict the internal delays for your XOR gate based on the numbers that we gave for NAND gates in Lecture 3 (this lecture is online, as are all of the others). Explain your calculations and any discrepancies between calculated and observed values, if any. What is the delay intercept (β) and slope (m) for this XOR gate that you have constructed using NAND gates? How does it compare with the XOR gate from the library?

1.d: Create a simple *divide by 2 circuit* using a D flipflop (dff) and an inverter. This is done by simply connecting the inverter between the Q output and the D input of the flipflop. Is the flip-flop rising or falling-edge triggered? Run the circuit using a symmetric square wave clock at frequencies of 1MHz, 10 MHz and 100 MHz. Explain your observations. Find the fastest clock rate you can run this circuit at. What happens when you try to run it faster?

Problem 2: A 4-bit CLA Adder

In this problem, you will design a 4-bit *Carry Lookahead Adder* (CLA). Refer to P&H (pp 241-249).

When adding two bit vectors A[N:0] and B[N:0], you can determine if there is a carry in for the (i + 1)th term by the following equation:

$g_i = a_i \cdot b_i$
$p_i = a_i + b_i$
$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_1 g_0 + p_i p_{i-1} \ldots p_1 p_0 c_0$

Notice that this equation allows you to determine the carry bit of any term from only the input vectors and the original carry in, c_0. An obvious way to implement the carry lookahead unit is to compute the 4 carries separately using the above equation.

Another way is to build up the p's and g's in steps. We know that $c_1 = g_0 + c_0 p_0$. Similarly, $c_2 = G_{01} + P_{01} c_0$ where G_{01} is the carry generated out of the block consisting of the first two bits. P_{01} is the carry propagated through this block. (Note that G_{01} is $g_1 + p_1 g_0$ and P_{01} is simply $p_1 p_0$.)

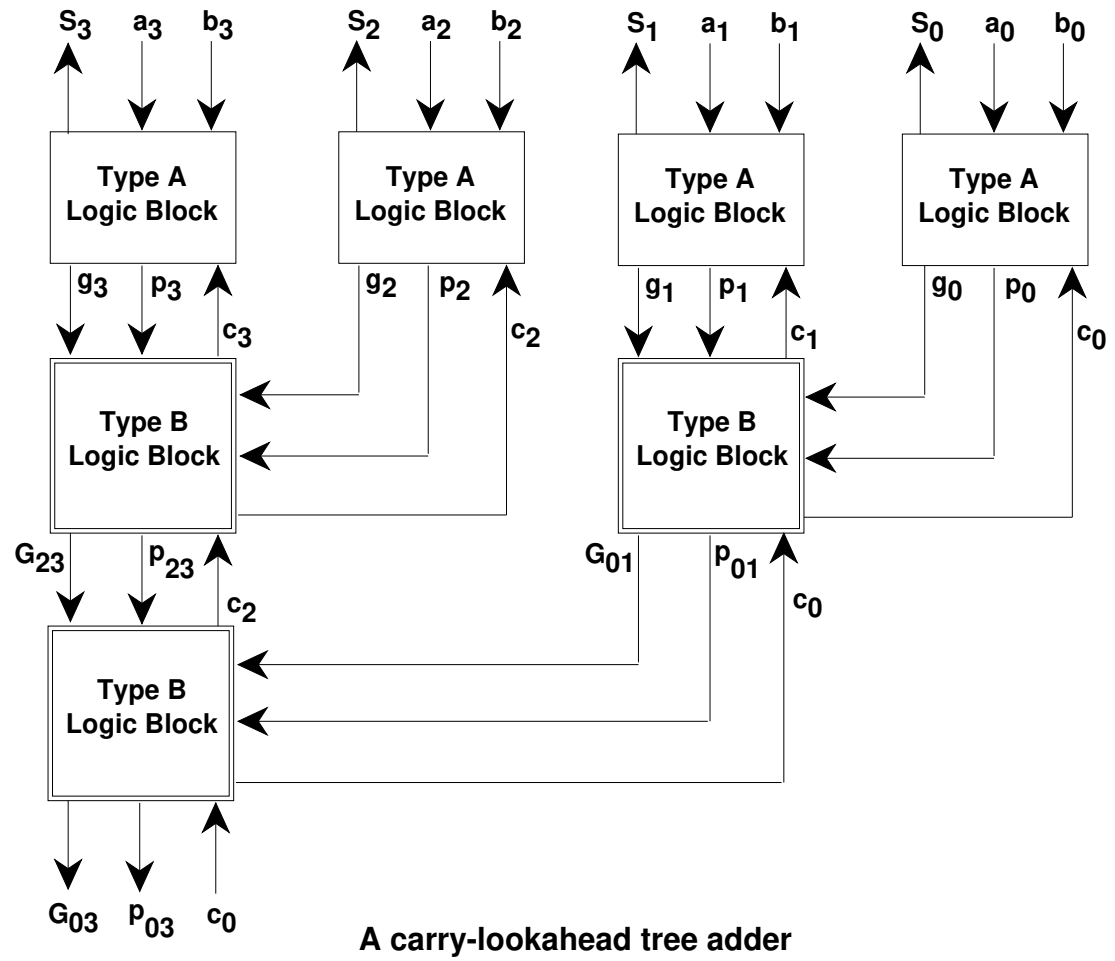
In general, for any j with $i < j, j + 1 < k$, we have the following recursive relations:

$$c_{k+1} = G_{ik} + P_{ik} c_i$$
$$G_{ik} = G_{j+1,k} + P_{j+1,k} G_{ij}$$
$$P_{ik} = P_{ij} P_{j+1,k}$$

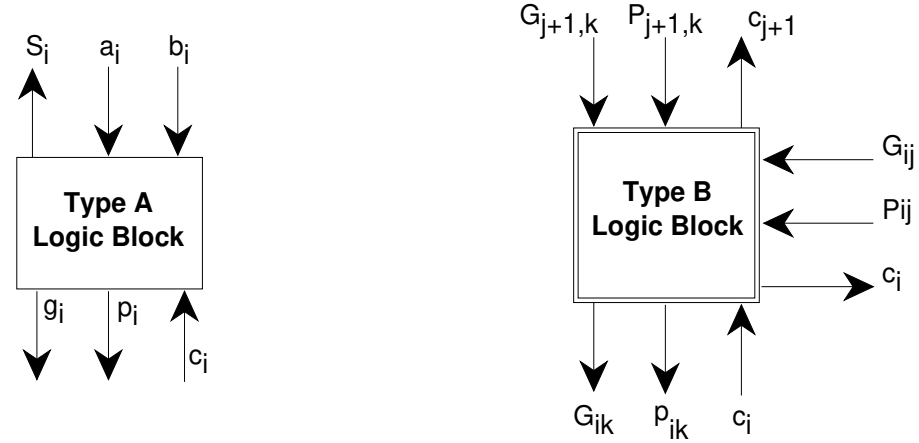
With these equations, you can implement a *carry-lookahead tree* as diagrammed in Figure 1 (This figure is shown on the next page).

2.a: Choose one of the above methods to implement the 4-bit CLA. You should structure your design into an easily understood hierarchy (eg. making a symbol for a block in the carry-lookahead tree). Also, make a symbol for the 4-bit CLA. Turn in schematic for the interior of this block and also show your top-level schematic symbol. Turn appropriate groups of signals into buses at the top level.

2.b: Build a test-bench to make sure that this is functional. Test it in behavioral mode. Then, characterize the delay of your CLA. You don't need to compute load-dependent delays. Just find the critical path (slowest path) for your 4-bit adder.



A carry-lookahead tree adder



$S_i = a_i \text{ xor } b_i \text{ xor } c_i$

$p_i = a_i + b_i$

$g_i = a_i \,\&\, b_i$

$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij}$

$P_{ik} = P_{ij}P_{j+1,k}$

$c_{j+1} = G_{ij} + P_{ij}c_i$

Figure 1: Four-bit carry-lookahead tree

Problem 3: A 32-bit Fast Adder

Construct a 32-bit carry-lookahead adder out of 4-bit carry-lookahead adders. Build this in three tiers, i.e. construct a 16-bit carry-lookahead adder from four 4-bit carry-lookahead adders and build a 32-bit carry-lookahead adder from two 16-bit carry-lookahead adders.

Your completed adder should take a *Cin* and two 32-bit inputs *A[31:0]* and *B[31:0]* and output a *Cout* and a 31-bit result *S[31:0]* (which is the sum of the inputs).

3.a: Turn in a schematic for your carry-lookahead adder.

3.b: Simulate your adder with a set of judiciously chosen test vectors in a test bench. Explain your criteria for choosing numbers. Turn in a schematic of your adder and a log of your simulation. Also, identify the critical path for your design and provide delay estimates (using post-place-and-route Modelsim).

3.c: Build a top-level Verilog module to take your adder through its paces. This top-level module should include your adder as a structural module, and should apply pseudo-random 32-bit numbers to the adder and check that the result is correct, and print out results in the following form:

```
1: number1 + number2 = result + carry
2: number3 + number4 = result + carry
3: number5 + number6 = result + carry
4: number7 + number8 = result + carry
                    ⋮
```

If there is ever a failure, you should flag it, say like this:

```
>>> ERROR <<<
>>> 5: number1 + number2 => result + carry
>>>                    Should be: realresult+carry
```

You should be able to run this for a long time, but try a small number of results first. Turn in a trace from a 100-number loop. If you find errors, fix them. Note that for this functional test, you can run in behavioral mode (without mappping). Turn in a list of your code and a trace of your output.

Hint: here you will want to use Verilog system functions. There is the $random function to get random values and the $display function to print results. There is an online Verilog manual under the handouts section. Look at chapter 17 (17.1 talks about $display and 17.9 talks about $random).

Problem 4: Online Notebook

Using your online notebook, figure out how long (to the nearest 15 minutes) you spent on each problem? Turn in your notebook. If possible, try to print two columns per page and use a smaller font to save paper.