## **Computer Sciences Department**

## **University of Wisconsin-Madison**

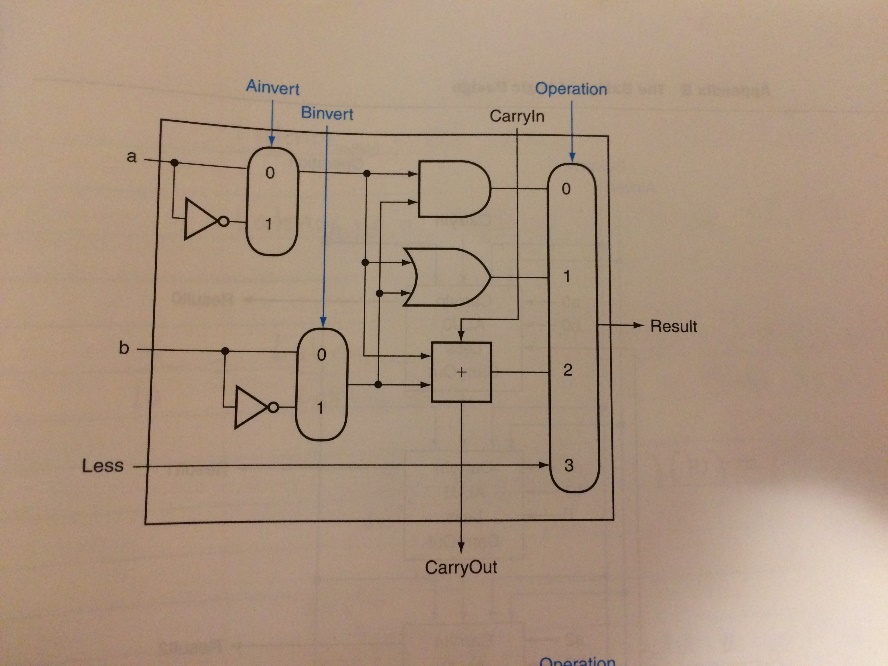
## **CS/ECE 552 – Introduction to Computer Architecture**

## **In-Class Exercise (02/06)**

**Answers to all questions should be uploaded on Canvas.**

1. [2 points] From the textbook:

1. [1 point] (Check Yourself B-5) Suppose you wanted to add the operation NOT (*a* AND *b*), called NAND. How could you change the ALU (from page B-33, Figure B.5.10) to support it?



1. No change. You can calculate NAND quickly using the current ALU since !(a\*b) = !a + !b and we already have NOT a, NOT b, and OR.
2. You must expand the big multiplexor to add another input, and then add new logic to calculate NAND.

（1） Use Demorgon’s Law

1. [1 point] (Check Yourself B-6) Using the simple estimate of hardware speed above (see page B-46) with gate delays, what is the relative performance of a ripple carry 8-bit add versus a 64-bit add using carry-lookahead logic?
2. A 64-bit carry-lookahead adder is three times faster: 8-bit adds are 16 gate delays and 64-bit adds are 7 gate delays.
3. They are about the same speed, since 64-bit adds need more levels of logic in the 16-bit adder.
4. 8-bit adds are faster than 64 bits, even with carry lookahead.

(1)

2. [3 points] The following MIPS assembly code performs 64-bit unsigned integer addition: *X* + *Y* = *Z*, where [*$s0*] = *X*[31:0] and [*$s1*] = *X*[63:32]; [*$s2*] = *Y*[31:0] and [*$s3*] = *Y*[63:32]; [*$t0*] = *Z*[31:0] and [*$t1*] = *Z*[63:32].

addu $t0, $s0, $s2

sltu $t7, $t0, $s0

addu $t1, $s1, $s3

addu $t1, $t1, $t7

Modify the assembly code to perform 64-bit unsigned integer subtraction: *X* – *Y* = *Z*.

subu $t0, $s0, $s2

sgtu $t7, $t0, $s0

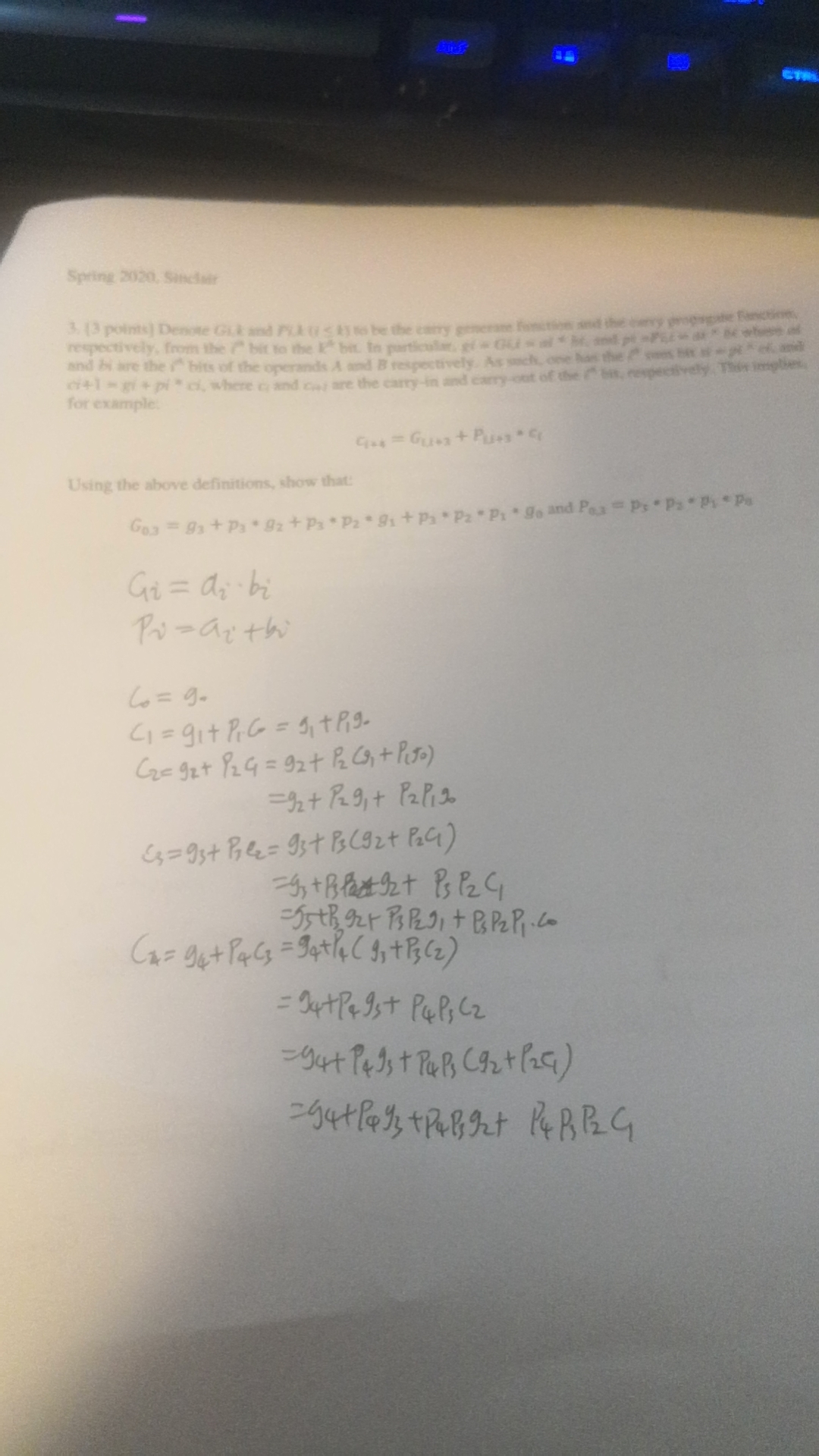
subu $t1, $s1, $s3

subu $t1, $t1, $t7

3. [3 points] Denote *Gi,k* and *Pi,k* (*i* ≤ *k*) to be the carry generate function and the carry propagate function, respectively, from the *ith* bit to the *kth* bit. In particular, *gi* = *Gi,i* = *ai* \* *bi*, and *pi* =*Pi,i* = *ai* ^ *bi* where *ai* and *bi* are the *ith* bits of the operands *A* and *B* respectively. As such, one has the *ith* sum bit *si* = *pi* ^ *ci*, and *ci+*1 = *gi* + *pi* \* *ci*, where *ci* and *ci+1* are the carry-in and carry-out of the *ith* bit, respectively. This implies, for example:

Using the above definitions, show that:

and



4. [2 points] (Challenge) Consider a 32-bit ripple carry adder (RCA) like the one in the textbook (e.g., similar to Figure B.5.7). To speed up the computations discussed in 1b above, your friend Ron suggests creating 2 copies of each 1-bit RCA, RCAi-0 and RCAi-1, which are identical except RCAi-0 always takes a carry-in of 0 and RCAi-1 always takes a carry-in of 1 (as shown in the picture below). What additional logic is needed beyond this to make your new carry adder (NCA for short) work? What advantages does it provide over RCAs?

A close up of a screen

Description automatically generated

The missing logic is a 2-to-1 mux.

The advantage is it is faster than a CLA when adding numbers with lots of bit (say, 128bits or more) since the delay becomes delay of a full adder which is constant and the time to propagate forward from the lowest bit.