Problem 1: Variable-Block Carry-Skip Adder

The carry-skip adder is a pretty good circuit. However, upon closer inspection, you notice that if all the skip blocks are of the same size, the latter blocks will finish switching quickly and then sit idle for a while waiting for the carry signal to pass through all the bypass multiplexers. For example, in the diagram of a 32-bit carry-skip adder below, the carry-out for bits 4-7 will be ready at the same time as the carry-out for bits 0-3. This second block will sit around doing nothing while MUX1 does its job.

To speed up the circuit, we could vary the size of the skip block. Intuitively, we should then be able to reduce the size of the first skip block and make each subsequent block increasingly larger. Because the critical path includes the last skip block, we must also start to taper down the size of each block as we approach the end. To obtain the optimal size of all the skip blocks, you realize that some really smart guy has already done all the mathematical derivations…which means that you don’t have to do it yourself. After talking to this really smart guy, you know that the optimal configuration for a 32-bit adder is (under the assumption that \( t_{MUX} = 2t_{prop} \)):

Estimate the worst-case delay for the simple 32-bit carry-skip adder in the first diagram and then estimate the amount of delay improvement with this new variable-block scheme. Assume that the setup (creation of propagate and generate signals) takes \( t_{setup} \), each bit of carry propagation takes \( t_{prop} \) (i.e., a skip block of \( m \) bits has a delay of \( m \times t_{prop} \)), a MUX has a delay of \( t_{MUX} \), and the sum generation has a delay of \( t_{sum} \). Leave your answers in terms of \( t_{setup} \), \( t_{prop} \), \( t_{MUX} \), and \( t_{sum} \).
Problem 2: Short Adders

In this question we are going to compare the speed of different types of adders.

a) Calculate the worst case delay of an 8-bit Ripple carry adder consisting of the full adder blocks shown below. You can use $t_p$ for the AND and OR functions and $2t_p$ for the XOR gates. Express your answer in terms of $t_p$.

b) For the second part of this question you are to implement an 8-bit Carry-Look Ahead adder as in the diagram below. For n-input AND/OR gates use the $t_{p,n} = 0.25*n^2*t_p$, similarly for the XOR gate $t_{XOR,n} = 0.5*n^2*t_p$. Find the worst-case delay of this adder again in terms of $t_p$.

c) Repeat the same calculations for 32-bit adders. **Hint:** Implement the Carry-Look Ahead Adder as a block CLA of 4-bit block-length.

Problem 3: Pipelined Multipliers

An array multiplier consists of rows of adders, each producing partial sums that are subsequently fed to the next adder row. In this problem, we consider the effects of pipelining such a multiplier by inserting registers between the adder rows.

a) Redraw Figure 11.30 (textbook, pg. 590) inserting word-level pipeline registers as required to achieve maximal throughput for the 4x4 multiplier. **Hint #1:** you must use additional registers to keep the input bits synchronized to the appropriate partial sums. **Hint #2:** just use little filled black rectangles to indicate registers and assume all registers are clocked using the same clock.

b) Repeat part (a) for a carry-save, as opposed to ripple-carry, architecture.

c) For each of the two multiplier architectures, compare the critical path, throughput, and latency of the pipelined and non-pipelined versions.

d) Which architecture is better suited to pipelining, and how does the choice of a vector-merging adder affect this decision?
Problem 4: Modified Booth Recoding

Start with a NxN array multiplier and notice that the number of partial products required is N. This implies N-1 additions, and thus, N-1 rows in the array. Modified Booth Recoding (MBR) is a technique for halving the number of partial products produced during a multiplication. This is nice because fewer partial products means fewer additions, ultimately resulting in a faster multiplication.

a) Two important number system principles are required to understand how MBR works. First, the base of the number system is called the radix. Decimal is radix-10, binary is radix-2, hexadecimal is radix-16, and so on. MBR uses a radix-4 number system. Since two binary bits can represent four numbers, we can take an ordinary binary number and split it into two bit ‘groups’ to form a radix-4 number:

Ordinary radix-2 (binary) number:
\[ \[0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0\]_2 = 0*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 58 \]
Radix-4 number:
\[ \[00 \ 11 \ 10 \ 10\]_4 = 0*4^3 + 3*4^2 + 2*4^1 + 2*4^0 = 58 \]

Note that in binary, the 8 bits mean that we will have 8 partial products. In radix-4, we have only four ‘bits’, hence half the partial products. When multiplying X*Y, two steps are taken before the multiplication is performed. First, we recode Y using radix-4. Second, we calculate the four possible unshifted partial products: 0*X, 1*X, 2*X, and 3*X. The radix-4 ‘bit’ tells us which of these partial products to select and how far to shift it (ie. how many zeros to append to the end). Demonstrate how this works by multiplying 94*121 using this technique.

b) Note that the biggest problem with this radix-4 multiplication is the partial product generation. 0*X, 1*X, and 2*X are easily generated using AND gates and a shifter. However, 3*X must be generated by adding 1*X + 2*X. This addition is in the critical path of the multiplier, so we would like to remove it. We do this by getting rid of all the 3*X partial products in the radix-4 calculation. Essentially, we need to remove radix-4 ‘bits’ that have the value \[3\]_4 or, equivalently, \[11\]_2.

Consider a number system in which each ‘bit’ position can hold three values: {-1, 0, 1}. This is called a redundant number system because there is more than one way to represent the same number. Numbers in this format can be treated in the same way as ordinary binary numbers, eg:

Ordinary binary number:
\[ \[0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0\]_2 = 0*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 32 + 16 + 8 + 2 = 58 \]
Reducant number system:
\[ \[0 \ 1 \ 0 \ -1 \ 1 \ 0 \ 1 \ 0\]_2 = 0*2^7 + 1*2^6 + 0*2^5 + -1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 64 – 16 + 8 + 2 = 58 \]

Note that all ordinary binary numbers are also included in this redundant number system, as well as a whole bunch more numbers that contain –1 ‘bits’.

Convert the following redundant numbers into standard binary numbers and then into radix-4 numbers: \[0 \ 0 \ 1 \ 0 \ 0 \ -1 \ 0 \ 0\]_2, \[0 \ 1 \ 0 \ -1 \ 0 \ 1 \ 0 \ 1\]_2, \[0 \ 1 \ 0 \ 1 \ 1 \ -1 \ 0 \ 1\]_2. Note that standard binary sequences of the form: \{0, some ones\} can be converted to redundant sequences of the form: \{1, some zeros, -1\}. By replacing a string of 1’s with 0’s, we can eliminate the possibility of two one’s in a group, thus eliminating the 3*X partial product!

c) MBR basically searches for strings of one’s in the binary number, converts them into an equivalent redundant number representation, treats the result in radix-4, then does the multiplication. This can be easily accomplished by using the look-up table in Table 1. Since we are using radix-4, i = {0, 2, 4, 6, 8, …}. Also, Y_\text{rec} = 0.

Now for X*Y, the partial products 0*X, 1*X, 2*X, -2*X, -1*X, -0*X are generated, Y is recoded according to the table, and then the multiplication is performed. Recode Y = 121 into radix-4 bits of \{-2, -1, 0, 1, 2\} according to the table. Now perform the multiplication 94*121.
d) Now let’s generate those partial products. A straightforward generation can be made using three signals: **negate** (1: negate X, 0: no change), **shift** (1: shift left by one, 0: no change), and **zero** (1: force to zero, 0: no change). Design a circuit that implements these three signals using standard gates (AND, OR, INVERTER, XOR, etc.).

e) So what does all this gain us? We’ve traded a 3*X partial product for –1*X and – 2*X. Recall that negation in two’s complement requires us to negate all the bits, then add 1. How can we add these one’s in without making an entirely new adder in the critical path? **Hint:** Try to find “holes” in the multiplication (ie. low order bits that are known to be zero and can be replaced with our negate signal).

f) Design a circuit that uses the three signals in Part d to generate –2*X, –1*X, 0*X, 1*X, and 2*X. Bear in mind that the negation does not need to add one because that will be taken care of using the method in Part e.

g) Congratulations! You’ve created all the primary building blocks of a Booth recoded multiplier. Lastly, what additional improvement can be made to make this one of the fastest multipliers available?

---

**Table 1: Modified Booth recoding look-up table**

<table>
<thead>
<tr>
<th>( [Y_{p-1}, Y_0, Y_{-1}] )</th>
<th>Unshifted partial product</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>+0X</td>
</tr>
<tr>
<td>001</td>
<td>–1X</td>
</tr>
<tr>
<td>010</td>
<td>+1X</td>
</tr>
<tr>
<td>011</td>
<td>+2X</td>
</tr>
<tr>
<td>100</td>
<td>-2X</td>
</tr>
<tr>
<td>101</td>
<td>-1X</td>
</tr>
<tr>
<td>110</td>
<td>-1X</td>
</tr>
<tr>
<td>111</td>
<td>-0X</td>
</tr>
</tbody>
</table>