Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.313 Contemporary Computer Design

Problem Set #4 Solutions Fall, 1995, Wed 11 October

1. a) $f \times f^{n-1} = f^n = m \Rightarrow f = m^{1/n}$

b) $\tau_d = n(\tau_0 + f\tau_f) = n\tau_i$

c) for $f = m^{1/n}, \tau_d = n(\tau_0 + m^{1/n}\tau_f)$

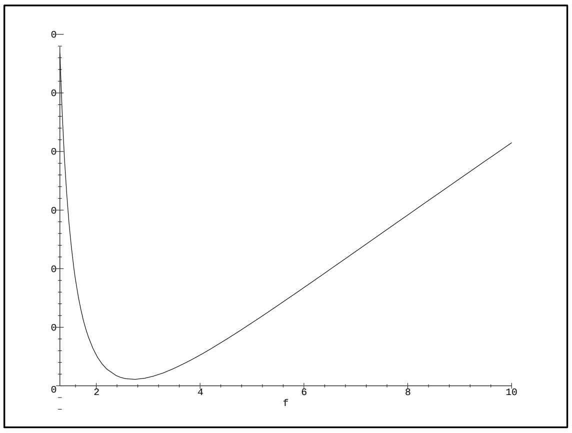
d) for $\tau_0 = 0, \tau_d = nm^{1/n}\tau_f$

$$\begin{aligned}
\frac{d}{dn}\tau_d &= \tau_f(m^{1/n} + nm^{1/n}(\ln m)(-1/n^2)) \\
&= \tau_f m^{1/n}(\frac{-1}{n}\ln m + 1) = 0 \\
(\frac{-1}{n}\ln m + 1) &= 0 \\
n &= \ln m
\end{aligned}$$

e) From a), we see that $\frac{\ln m}{\ln f} = n$, so for the optimal n, we must have $f = e$.

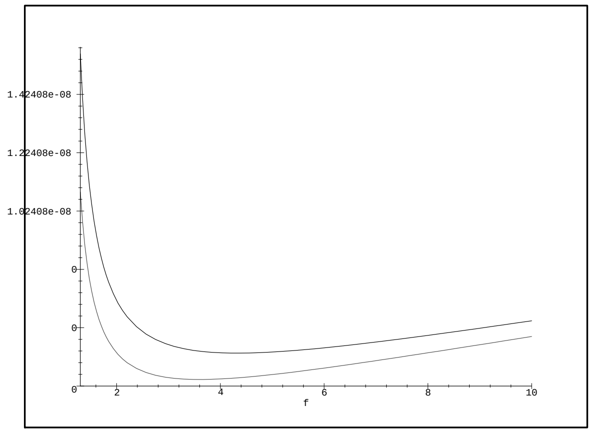
f) $\tau_d = nm^{1/n}\tau_f = nm^{1/\ln m}\tau_f = ne\tau_f = e\ln m\tau_f$

g) $\tau_d = nf\tau_f = \frac{\ln m}{\ln f}f\tau_f$ The following plot is for m=500 and $\tau_f = 200ps$



h) $\tau_0 = n\tau_f(1+f)$ and $\tau_0 = n\tau_f(2+f)$.
$\tau_0 = \frac{\ln m}{\ln f}\tau_f(1+f)$ and $\tau_0 = \frac{\ln m}{\ln f}\tau_f(2+f)$.

i) The larger intrinsic delay, shown in the upper curve, slows the system down and requires a larger fanout factor, or fewer stages.



j) The practical number must be greater than e, and probably greater than 4. Under some circumstances, a factor of 10 is acceptable. Five is a nice rule of thumb to work with, but the design hides a number of subtleties. Designs for maximum area, power, or bandwidth require different "optimal" scaleup factors. Take 6.372 for a more in-depth treatment.

k) This should be the same as your answer to j), for the reason that with a small number of subordinates, the chain of command expends too much effort in communicating results, and with a large number of subordinates, the individual manager spends too much time communicating with her underlings.

2. first b) Note that the ripple happens from left to right, unlike a ripple adder which ripples right to left.

second b) The delay is of order N, where N is the number of bits in the word.

c) The delay now is N/2 plus the delay of the multiplexer. The area has been increased by slightly more than 50%.

d) For FMSO, the "carry" is the bit which indicates we have found a more significant one already. This carry is generated if the current input bit is a 1, and it is propagated if the current bit is a 0. (It is also propagated if the current input bit is a 1, and in the circuit implementation we'll take advantage of this, but technically the carry is propagated with a 0 input.) So,

$$\begin{aligned}
P_i &= \overline{I}_i \\
G_i &= I_i \\
C_{0_i} &= C_{i_i}P_i + G_i \\
FMSO_i &= G_i\overline{f}_{i_i}
\end{aligned}$$

And using the recursion relation to find the FMSO and carry of the LSBs,

$$\begin{aligned}
C_{0_{i-1}} &= (C_{i_i}P_i + G_i)P_{i-1} + G_{i-1} \\
FMSO_{i-1} &= G_{i-1}(\overline{C_{i_i}P_i + G_i})
\end{aligned}$$

For gates with a sufficient number of inputs, the function may be computed in constant time. In reality, we have a limit of about 4 inputs for a single NOR gate, we we'll either use multiple levels of logic or use carry lookahead in conjunction with other techniques.

e) The key point to the Manchester carry chain is that you *must* have the kill and propagate signals be distinct. For the circuit shown below, the intermediate nodes are first precharged by $\overline{PC}$, then the PFETs are switched off and the chain is evaluated. The first $FMSO_i$ bit is asserted if I_i is high. This corresponds to a 1 in the MSB position. K_i is asserted if I_i is high (see the equations for K and P below) and pulls the node low. The kill signal is asserted if the bit to the left is high, meaning that the current bit could not possibly be the MSO. The propagate signal is high if the bit to the left is low, so the value on the node to the left is propagated.

The delay through this circuit is proportional to the wire delay. For small N it is effectively constant since wire delays are negligible. For larger N the delay will grow as N^2.

3. a) If we treat the line as a distributed RC line, delay goes as N^2.

b) From PS3, the optimal number of buffers is $n = \sqrt{\frac{\mathcal{RC}L^2}{T}}$ where $\mathcal{RC}$ is the time constant per unit length, L is the length of the wire, and T is the delay through a buffer. Buffers should be placed between the propagate transistors.

c)

d) This question is far harder than it appears at first blush. Calculating $3/4^{16}$ or $1/2^{16}$ merely gives you the probability that if a block of 16 bits has a carry in, it will have a carry out. Using $1/2^{16}$, it turns out to be about 0.0015%. What we really want to know is, given a 64 bit word, what is the probability that we will have a carry go through 16 bits, meaning that each bitslice has both a carry in and a carry out. We can make a Markov diagram in which we enumerate the states and transition probabilities for finding numbers of blocks in a row which have both a carry in and a carry out. Using this diagram, we can create a transition probabilities matrix for each current state and next state's transition probabilities. Exponentiating this matrix to the 64^{th} power and multiplying by a vector which gives a unity probability to the first state, we can read off the probability that the system will end up in the final state, indicating that 16 blocks in a row had a carry in and a carry out. Running this 18 by 18 matrix through Matlab, I found that the total probability is about 0.02%.

This means that if we can do 16 ripples in one cycle, a dumb method would be to take 4 cycles for an add, but a smart method could take multiple (say 4) cycles only 0.02% of the time, for an average of 1.0006 cycles.

Let's look at the problem another way. Since we know that once the carry in to a block is high, we will get at least a 16 bit ripple 0.0015% of the time, we just need to figure out how often we get a carry in to a block of 16 bits. The probability that a block will generate a carry out is the probability that it will generate a carry (1/4) plus the probability that it will propagate a carry (1/2) and has a carry-in itself. Or,

$$P_i(Co) = 1/4 + 1/2(P_{i-1}(Co)) \tag{1}$$

For an initial value of 1/4 (since we know that the first block does not have a carry in) P_i quickly tends to 1/2. We then have 47 chances to generate a carry out to feed a block of 16 bitslices ($47 = 63 - 16$). If we don't mind an overestimate and do a little double counting, we can estimate the probability that we will have a full ripple of 16 bits as $47 \times 1/2^{16} \times 1/2 = 0.036\%$. Again looking at the performance difference when taking advantage of this fact, we can take one cycle 99.96% of the time, for an average cycle time of 1.001. The difference between the exact 1.0006 and approximated 1.001 is not too bad.

The point of the exercise, however, was to show you that long ripples don't happen very often and that if you can make an asynchronous adder you don't have to wait for ripples that don't happen. Because of the probabilistic nature of the carries, this information doesn't help you to choose the size of Manchester carry chains or carry select adders or the like.