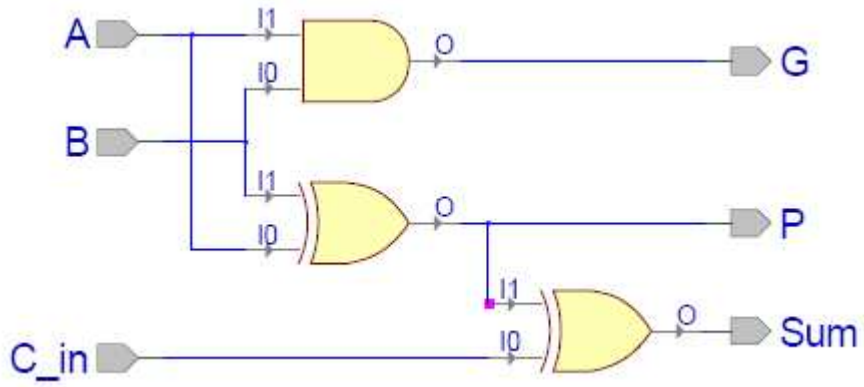
CSE 370 – Winter 2008

Homework 5 – Solutions

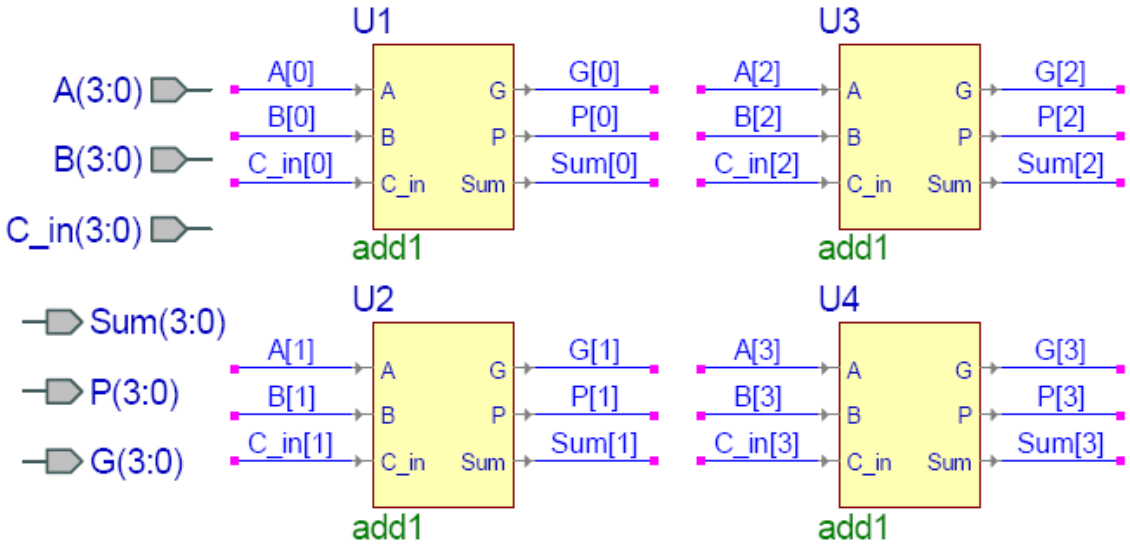
---

1) Carry Look-Ahead Adder (CLA)

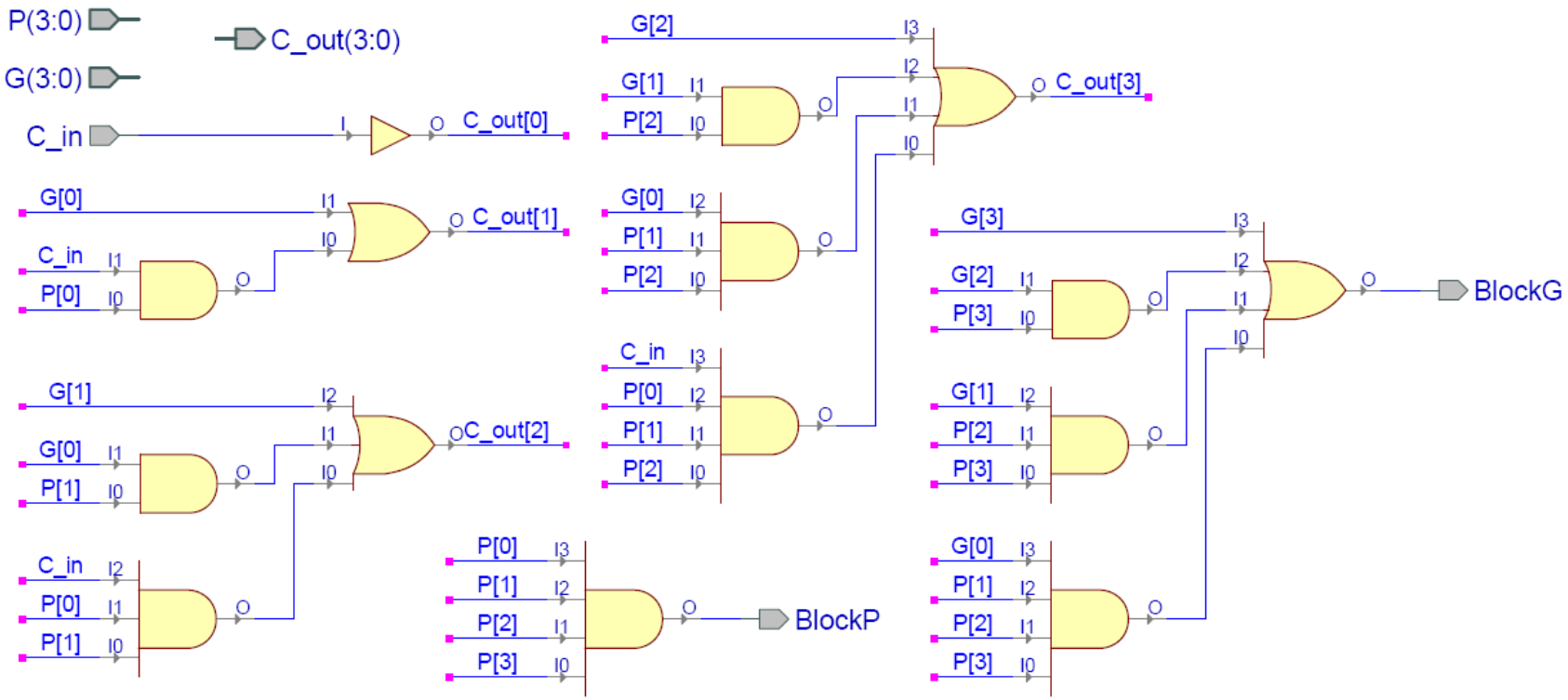
a) **add1**



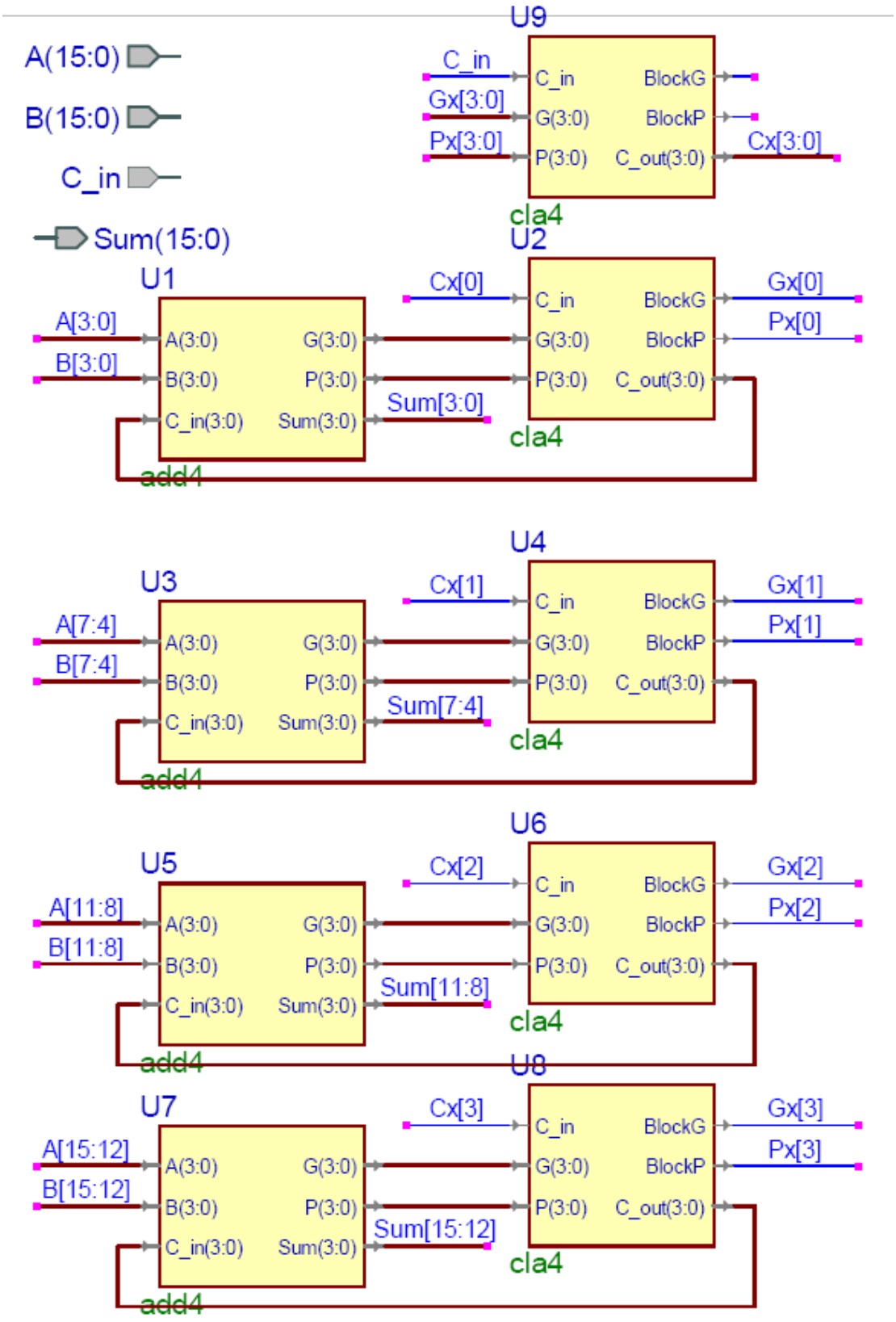
b) **add4**



c) **cla4**



d) **cla16**


A(15:0)
B(15:0)
C_in
Sum(15:0)
U9
C_in
Gx[3:0]
Px[3:0]
C_in
G(3:0)
P(3:0)
BlockG
BlockP
C_out(3:0)
Cx[3:0]
cla4
U1
A[3:0]
B[3:0]
A(3:0)
B(3:0)
C_in(3:0)
G(3:0)
P(3:0)
Sum(3:0)
add4
U2
Cx[0]
Sum[3:0]
Gx[0]
Px[0]
cla4
U3
A[7:4]
B[7:4]
Cx[1]
Sum[7:4]
Gx[1]
Px[1]
cla4
U4
U5
A[11:8]
B[11:8]
Cx[2]
Sum[11:8]
Gx[2]
Px[2]
cla4
U6
U7
A[15:12]
B[15:12]
Cx[3]
Sum[15:12]
Gx[3]
Px[3]
cla4
U8

e)
Gate Count: 118 gates
add1 : 3 gates
add4 : 4*Add1 = 12 gates
cla4 : 14 gates
cla16: (4*Add4) + (5*Cla4) = 48 + 70 = **118 gates**

Delay:
Look at the schematic for **add16** above. The A and B inputs first go through **add4** blocks**.** In the **add4**, they produce the P and G signals. This takes 1 gate delay. They also produce a sum; however the carry input to the gate producing the sum hasn't arrived yet. So we do not consider the sum now.

The P and G signals produced by any **add4** block, say U1, goes into a **cla4** U2. In the **cla4**, it takes 2 gate delays to produce a blockP and a blockG. The **cla4** also produces carry output bits, however the carry in required to compute the cout hasn't arrived yet. So we do not consider the cout now.

The blockP's and blockG's from the **cla4's** U2, U4, U6 and U8 are available at the same time. These are fed to the **cla4** - U9. In U9, the Carry in is available and hence it computes the cout bits. In the worst case, it takes 2 gate delays to do this. The blockP and blockG output of U9 is available at this time (5 gate delays for BlockP and G signals).

The carry bits produced by U9 are fed back into **cla4's** U2, U4, U6 and U8. Only now are the carry in bits available to these four blocks. Hence now we consider the cout of these blocks. It takes 2 gate delays to produce the cout bits in the worst case.

These cout bits act as the carry in bits for the add4 blocks U1, U3, U5 and U7. Only now have its carry in bits arrived. So now we can consider the sum. From the time the carry in bit arrives it takes only 1 gate delay to compute the sum output. The other input to the XOR gate was computed long before and hence we do not worry about the gate it traverses through.

Total: 1+2+2+2+1 = 8 gate delays.

f) Yes.
A **cla64** would simply have 4 **cla16's** and use 1 **cla4** that takes the blockG and blockP signals from the **cla16's** to generate the carry inputs for each of the **cla16's**.

Gate Count: (4*cla16) + cla4 = 486

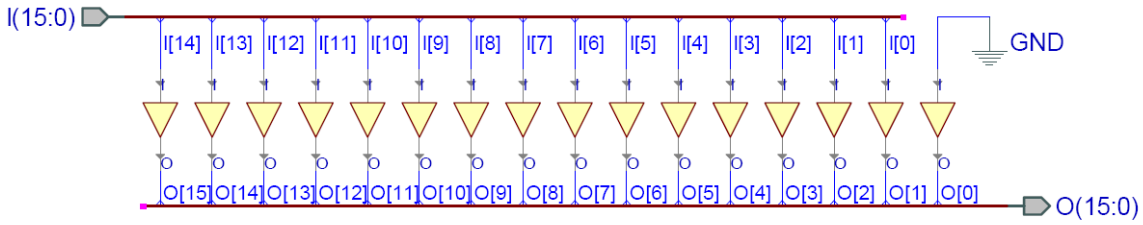
Delay: 5 gate delays for blockP and blockG of **cla16** (see above). 2 gate delays in the cla4 to produce the carry in for the cla16's. Once the cla16 has its carry input it takes only 5 gate delays to produce the sum as its bit-wise P &G has already been computed.

Total: 5+2+5 =12

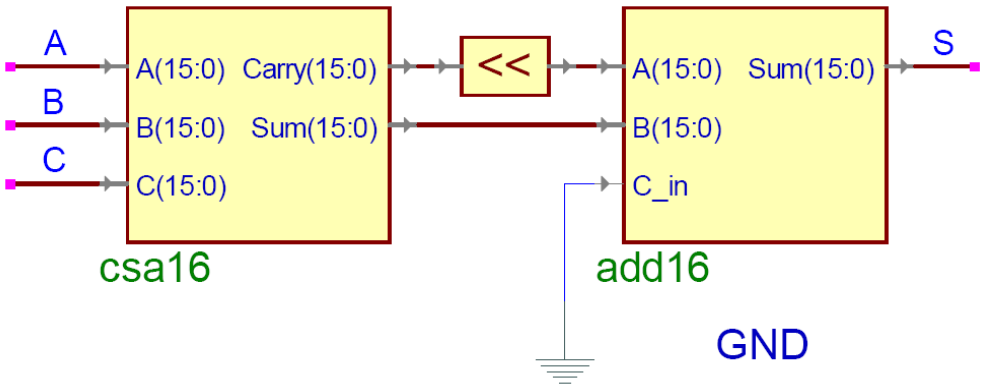
2)
a) The first stage of the carry save adder (CSA_1st) consists of 16 full adders (FA). Call the inputs of the ith FA A^i, B^i, $C_{in}{}^i$ and the outputs Sum^i and $C_{out}{}^i$. Then the CSA_1st component makes the following (very straightforward) connections to the 16 FAs:

A[i] ⇔ A^i
B[i] ⇔ B^i
Sum[i] ⇔ Sum^i
Carry[i] ⇔ $C_{out}{}^i$

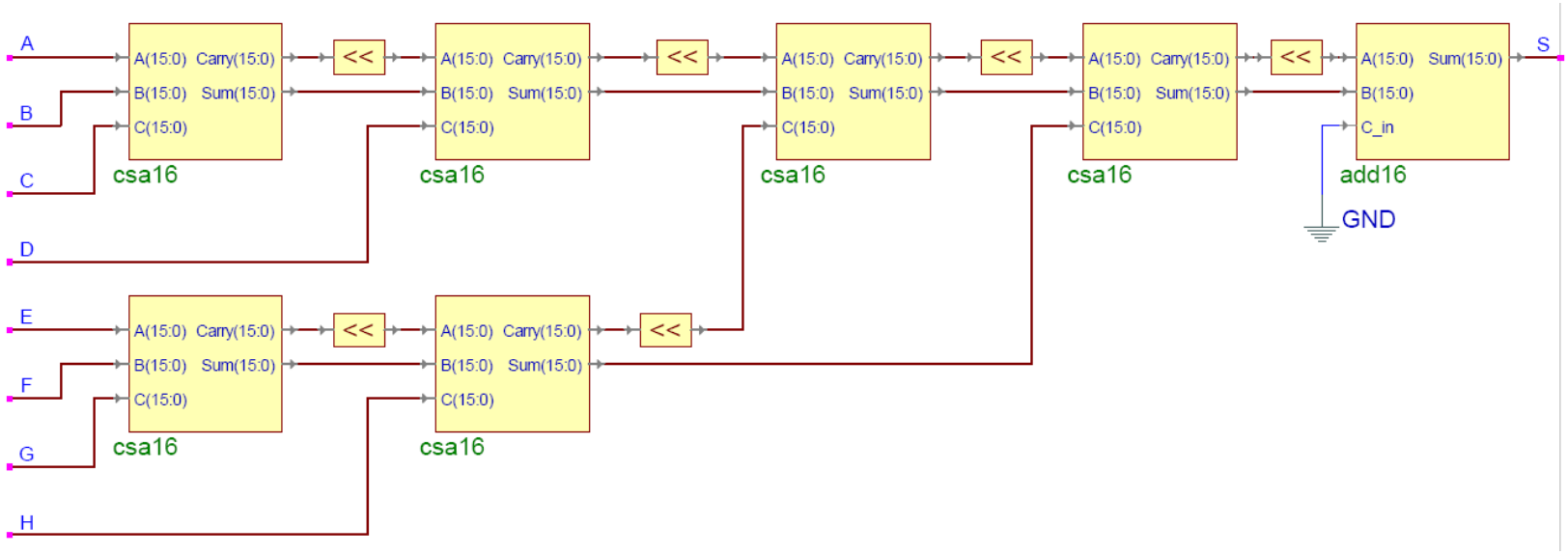
The following answers make use of a component [<<] which denotes a left shift by one bit. Normally one would accomplish this by setting O[15:1] = I[14:0], O[0] = GND.



b)



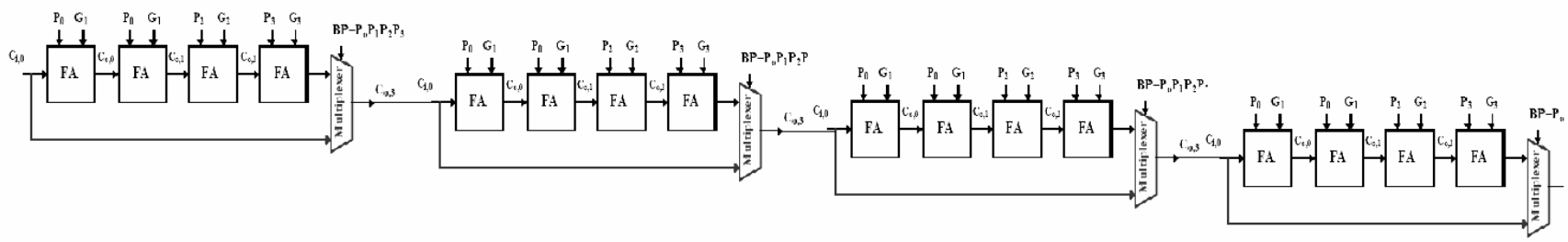
c)


A
B
C
D
E
F
G
H
A(15:0) Carry(15:0)
B(15:0) Sum(15:0)
C(15:0)
csa16
<<
A(15:0) Sum(15:0)
B(15:0)
C_in
add16
GND
S

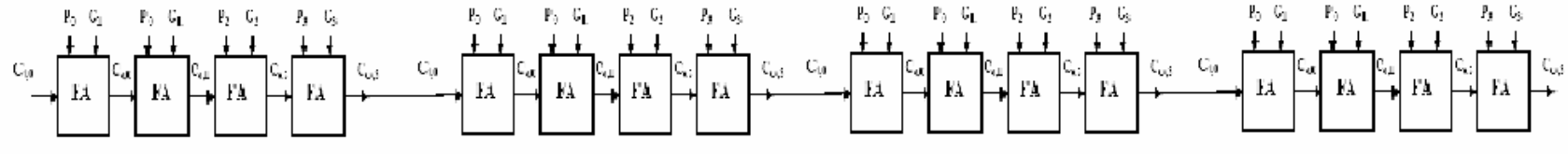
3)

a) **makepg**: add1 block with just P and G outputs. Or you can reuse add1 just leaving Sum output unconnected.
b) **make4pg:** same concept as above, applied to add4.
c) **S = P XOR C_{in} ; Co = G + PC_{in}**
d) 16 bit adder: This architecture is called a carry skip or a carry bypass adder.

Simply chain together 4 of the ones in the question



e) Why this architecture is faster:
Consider you did not use the multiplexers. Then you get a standard ripple carry. It would look like this:



The worst case for the above circuit is when all the full adders propagate. In that case we would have the following delay:
To get all P and G signals: 1 gate delay (see add1 block)
Once you have P, G and C_{in}, it takes 1 gate delay for sum output and 2 gate delays to get C_{out}. Remember; $C_{out} = G + PC_{in}$.
The first full adder hence takes 3 gate delays to provide its C_{out}. Ever subsequent full-adder takes 2 gate delays. Hence the final full adder will produce its sum bit: 3 + 2*(14) +1 = 32 gate delays later.

Compare this to the 8 that the CLA takes. Now let us consider the carry skip architecture.

Carry skip: Clearly, the all propagate case which is the worst case for the ripple carry is not the worst case for the Carry skip adder. Because, if all propagates were true, you would simply have 3 mux delays plus (3*2 + 1)gate delays to get the final sum bit. Assuming that the mux delay is comparable to a gate delay that would give us a total of 10 gate delays for the all propagate case.

f) The worst case would however be different. In the worst case,

You have a bit 0 generating, bits 1 -15 propagating. In this case the first stage does not skip the carry. However stages 2 and 3 skip it leading into the final stage where the carry ripples to produce a sum bit. Hence you have,

1 gate delays for the C_{out} from bit 0. Then 3*2 = 6 gate delays for the FA's of bits 1-3. Then 3 gate delays to go through the 3 multiplexers. And finally (3*2 +1); 7 gate delays to get the final sum bit.

This gives a total of 17 gate delays.

If more than one stage had a generate that was true, then you would get parallel computation of sum bits in more than one stage and hence would definitely be faster than the above case.

Compared to 32 gate delays for the ripple carry, carry-skip is twice as fast. Plus you only need very little additional logic to implement it. The CLA is twice as fast as the Carry skip. But it needs much more logic. Hence, there is a tradeoff between logical complexity and the speed you can achieve.

g) Input sequence to trigger this worst case:

As it was suggested above, you want bit-0 to generate, bit 1-15 to propagate. This might happen in the following configuration:

A: 0000000000000001 C_{in}= 1 or 0
B: 1111111111111111

There are several other input combinations that can show the above behavior.