# CSE370 HW4 Solutions (Winter 2010)

1. **Verilog Calendar**

   There were many correct ways to approach this problem. My solution is just one potential one. A couple of general comments about the problem:

   - Some people used some Verilog features that we haven't talked about yet: integers, functions, etc. Try to avoid these.
   - Some people used while and for loops. These loops don't necessarily behave like they would in a programming language. In many cases loops are "unrolled" and produce N copies of the body.
   - A lot of people missed some of the error cases. The main things is to make sure that you address all the corner cases:
     1. Day must not be 0, and must be less than 32.
     2. Day must be less than 31 for all months that don't have 31 days
     3. Day must be less than 30 for February
     4. Day must be less than 29 for February if it is not a leap year
     5. Month must not be 0, 13, 14, or 15.
     6. You can't just add the leap year bit into the total because in January and February the leap day hasn't yet occurred.
   - Finally, some people got the "All Tests Passed" output even when their code had errors in it. It is important not to rely too much on the test fixture because in general they do not test every case that your circuit might encounter. Also, your circuit may have been outputting "z" or holding a value from a previous case in a register, which could cause things to mess up.

   Anyway, here is my code so that you can look at to see a different implementation from your own:

```
//-------------------------------------------------------------------------
// Title       : Calendar module
// Design      : Homework 4 Calendar (Problem 1)
// Author      : Gaetano Borriello and Craig Prince
// Company     : CSE 370
//
//-------------------------------------------------------------------------
//
// File        : cal.v
//
// Description : This module computes the index of the specified in
//               the year given the month, day of the month, and
//               whether the year is a leap year.  It also raises an
//               error flag if there is a problem with the input values.
//-------------------------------------------------------------------------
`timescale 1ns / 1ns


module cal (MONTH, DAY, LEAP_YEAR, DAY_OF_YEAR, ERROR);

input  [3:0] MONTH;           // Data input for month of the year
input  [4:0] DAY;             // Data input for day of the month
input        LEAP_YEAR;       // Data input for leap year flag
output [9:0] DAY_OF_YEAR;     // Results from calendar subsystem
output       ERROR;           // Error flag from calendar subsystem
```

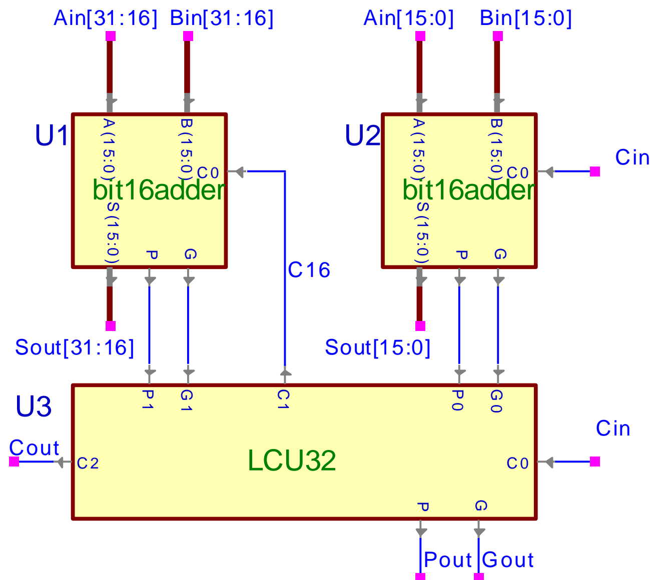
```
reg    [9:0] DAY_OF_YEAR;     // Data input
reg          ERROR;           // Data input

reg [8:0] offset;             // Day offset of the start of month
reg [4:0] daysinmonth;        // Total days in the month

always @(MONTH or DAY or LEAP_YEAR) begin
      // Get number of days in the month
      case({MONTH,LEAP_YEAR})       // Appending the month and leap year
            5'b00010, 5'b00011,           // Jan - 31 day months
            5'b00110, 5'b00111,           // Mar
            5'b01010, 5'b01011,           // May
            5'b01110, 5'b01111,           // Jul
            5'b10000, 5'b10001,           // Aug
            5'b10100, 5'b10101,           // Oct
            5'b11000, 5'b11001:           // Dec
                  daysinmonth[4:0] = 31;
            5'b00100:                     // Feb, no leap
                  daysinmonth[4:0] = 28;
            5'b00101:                     // Feb, leap
                  daysinmonth[4:0] = 29;
            5'b01000, 5'b01001,           // Apr - 30 day months
            5'b01100, 5'b01101,           // Jun
            5'b10010, 5'b10011,           // Sep
            5'b10110, 5'b10111:           // Nov
                  daysinmonth[4:0] = 30;
            default:                      // Error
                  daysinmonth[4:0] = 0;
      endcase

      // Get the month offset
      case ({MONTH,LEAP_YEAR})      // Appending the month and leap year
            5'b00010, 5'b00011:     // January
                  offset[8:0] = 0;

            5'b00100:               // February (+31)
                  offset[8:0] = 31;
            5'b00101:
                  offset[8:0] = 31;

            5'b00110:               // March (+28 | +29)
                  offset[8:0] = 59;
            5'b00111:
                  offset[8:0] = 60;

            5'b01000:               // April (+31)
                  offset[8:0] = 90;
            5'b01001:
                  offset[8:0] = 91;

            5'b01010:               // May (+30)
                  offset[8:0] = 120;
            5'b01011:
                  offset[8:0] = 121;

            5'b01100:               // June (+31)
                  offset[8:0] = 151;
            5'b01101:
                  offset[8:0] = 152;

            5'b01110:               // July (+30)
                  offset[8:0] = 181;
            5'b01111:
```

```
            offset[8:0] = 182;

        5'b10000:              // August (+31)
            offset[8:0] = 212;
        5'b10001:
            offset[8:0] = 213;

        5'b10010:              // September (+31)
            offset[8:0] = 243;
        5'b10011:
            offset[8:0] = 244;

        5'b10100:              // October (+30)
            offset[8:0] = 273;
        5'b10101:
            offset[8:0] = 274;

        5'b10110:              // November (+31)
            offset[8:0] = 304;
        5'b10111:
            offset[8:0] = 305;

        5'b11000:              // December (+30)
            offset[8:0] = 334;
        5'b11001:
            offset[8:0] = 335;
    endcase

    // Handle all the error cases and calculate the result
    if( DAY == 0 || DAY > daysinmonth )
        begin
            ERROR = 1;
            DAY_OF_YEAR  = 0;
        end
    else
        begin
            ERROR = 0;
            DAY_OF_YEAR = offset + DAY;
        end
end

endmodule
```

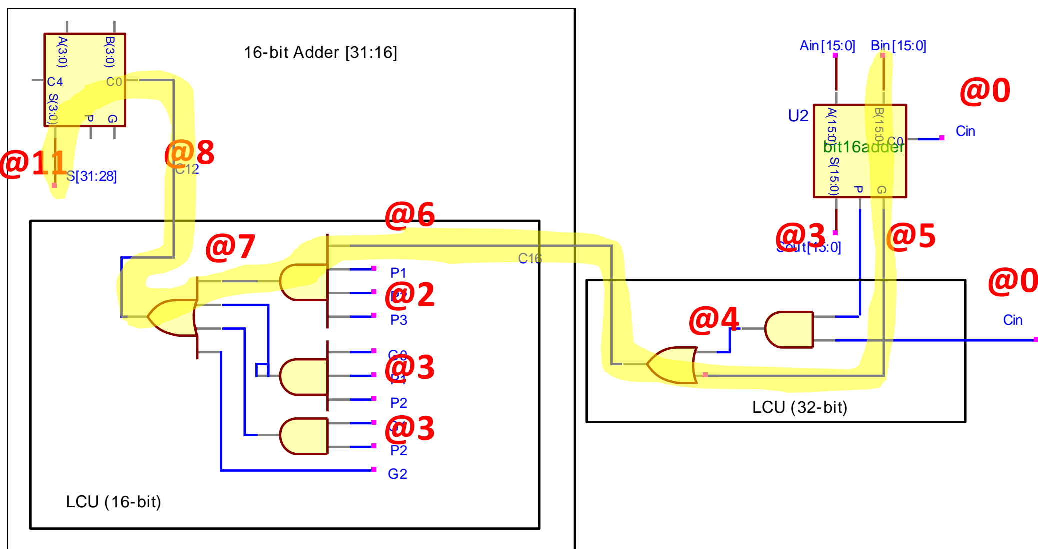
2. **CLD2e, 5.9 part a/b**

I am going to do parts a and b together because it is easier to put the delays directly on the circuits as I draw them.

For the 32-bit adder we basically need to replicate a circuit that is similar in structure to Figure 5.18 in the book, but only half as long. When we do this we get a circuit as follows:

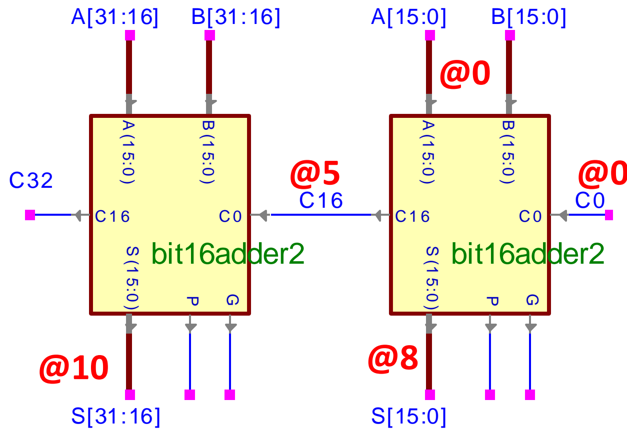



This 32-bit adder uses 2x 16-bit adders. To show the delay of the longest path through this circuit I am going to draw it showing the internals of some of the boxes, I have then highlighted the longest path, annotating the delays along it:

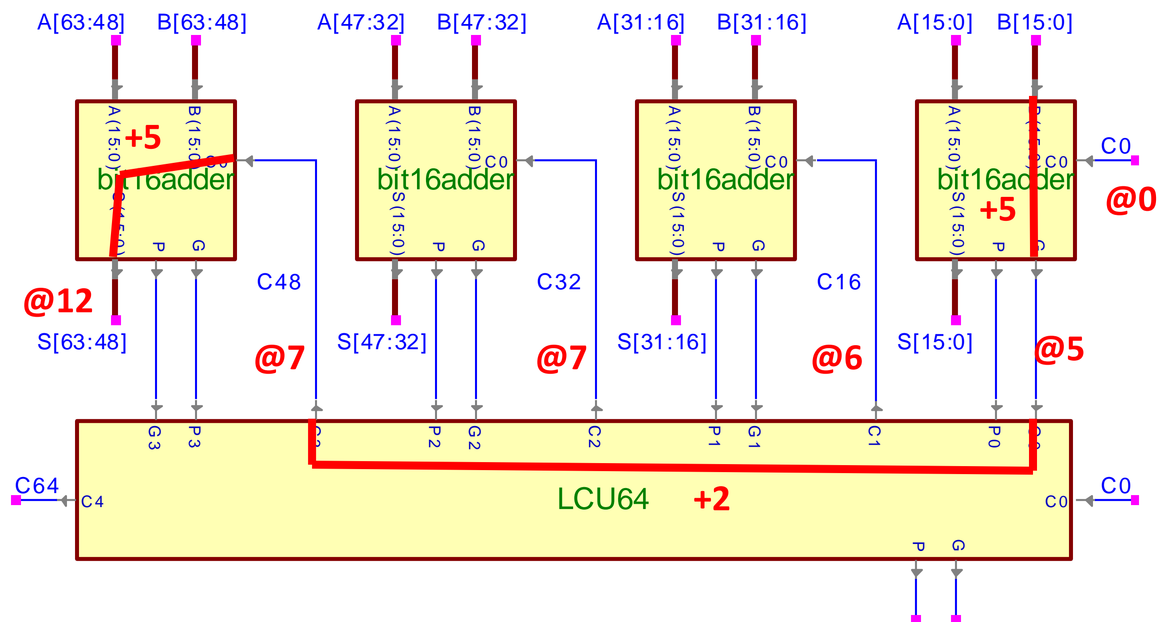



So the longest path for the 32-bit adder requires 11 gate delays.

**Note:** Some people noticed that there is actually a way to get a slightly faster 32-bit adder. If we chain two 16-bit adders together instead of using another level of Propagate/Generate logic then we can save one gate delay for a total of 10 delays. That diagram is as follows:




Now for the 64-bit adder, we get a similar circuit:




If we analyze the delay of this circuit it goes almost exactly as in the 32-bit case…however, within the top-level LCU it takes one more gate delay to generate C48 than it took to generate C16. Everything else remains the same. As a result the delay for the 64-bit adder is 12 gate delays.

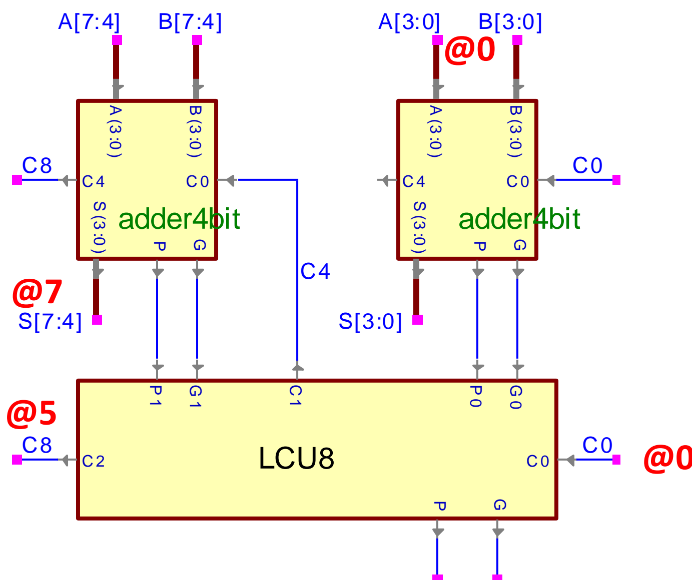
Thus, the final solution is that the 32-bit adder has a worst-case delay of 11, while the 64-bit adder has a worst case delay of 12.

**Note:** A lot of people figured out the correct delay for the carry-in to the last 16-bit adder (@7), but messed up with the delay within the last 16-bit adder. Some people said that the last stage added +8 delay which is the normal delay of a 16-bit adder. Also, some people added +3 delay (the delay of the 4-bit adder). But really the last 16-bit adder adds +5 delay.

Consider a 16-bit adder where the inputs A and B and C0 are available @0, then it is true that the output is available 8 gates later (@8). However, if the carry-in is not available until @7, then our P and G from each 4-bit adder would already have been computed and be available before C0 is available (@2 and @3, respectively). However, our LCU can't start until it has C0, which it won't get until @7. At this point, it takes +1 delay to calculate $C_4$ and +2 delay for $C_8$ and $C_{12}$. This means $C_{12}$ won't be ready until @9, then there is another +2 delay within the 4-bit adder's LCU to generate $C_{15}$ internally @11. Finally $C_{15}$, can be used to calculate the S[15] with an additional +1 delay @12! This is why we get the +5 delay. See Appendix A for another way to think about delay.

**3. CLD2e, 5.10**

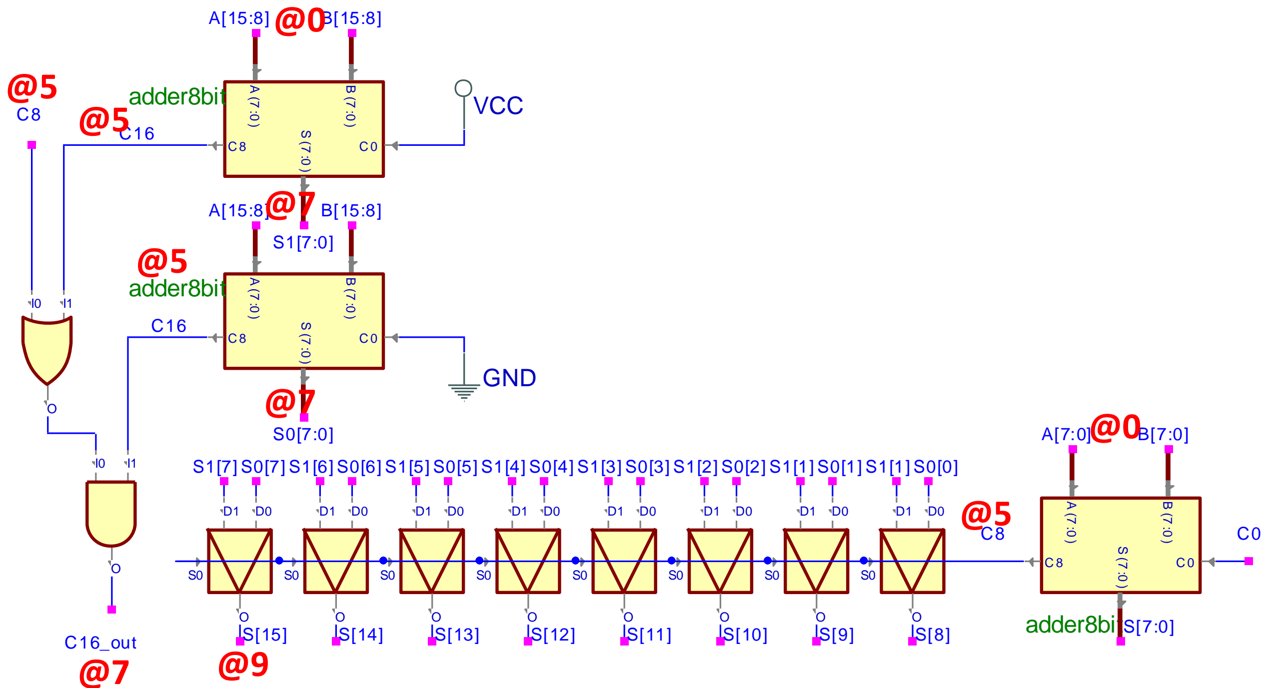
This problem asks us to compare a 16-bit carry-select adder to a 16-bit ripple-carry adder and a 16-bit carry-lookahead adder. The problem asks us to assume that we have an 8-bit carry-lookahead adder as a basic unit to build our carry-select adder. Let us first examine the delay of this 8-bit subcomponent. We have the following design for our 8-bit adder.




From this we see that when all the inputs are there @0, then the sum is generate @7 and the carry-out is generated @5.

**Note:** Many people stated that the delay for an 8-bit carry-lookahead adder would be 4 gate delays. This is technically true, but to build this circuit you would need a very large amount of logic to calculate the C8 bit in the Carry-Lookahead Unit. In fact you would need an 8-input OR gate, and (9,8,…,2)-input AND gates. In reality, these large gates would be really slow which is why we have gone with the design above.

Now that we have this 8-bit adder and know the delay of its outputs, we can figure the delay of our larger adder. The larger circuit is seen below:




From this diagram we can see that the inputs to all adders are available @0 and that means it takes 7 gate delays to get the inputs to the MUXes. The MUXes add 2 more gate delays for a total of @9. This is the worst case delay for this circuit.

Next we need to contrast this against a 16-bit carry-lookahead adder as well as a 16-bit ripple-carry adder. For the 16-bit carry-lookahead adder we can see from Figure 5.18 in the book that the delay is 8. For the ripple-carry adder if we chain 16 1-bit adders each carry out will require two gate delays for a total of 16x2=32 gate delays.

To summarize the delay for each adder is: carry-select = 9, carry-lookahead = 8, ripple-carry = 32.

## Appendix A

Here I wanted to expand a little bit more on how to deal a little more formally with estimating the delay of hierarchical components. This is mostly to give people another way to think about delay. If you find another method more clear then feel free to ignore this. Fundamentally, we are trying to find the longest path through our circuit. However, this gets complicated when we have boxes like an 8-bit or 16-bit adder.

We can't just find the longest path through the component and then use that as our delay for the component because that longest path assumes that all inputs arrive at the same time (i.e. the longest path up until reaching every input has been the same delay).

The trick instead is to consider the longest gate delay from every input to every output. So if our component has inputs A,B,C,D and outputs X,Y,Z. Then we need to find the longest delays: delay(A,X), delay(B,X), delay(C,X), delay(D,X), delay(A,Y)…delay(D,Z). With this information it is pretty easy to see that if we have the longest delays up until input A,B,C,D (call them $d_A$, $d_B$, $d_C$, $d_D$), we can easily calculate the longest path for each output X,Y,Z as follows:

$d_X (d_A,d_B,d_C,d_D)$ = MAX[ $d_A$ + delay(A,X), $d_B$ + delay(B,X), $d_C$ + delay(C,X), $d_D$ + delay(D,X) ];
$d_Y (d_A,d_B,d_C,d_D)$ = MAX[ $d_A$ + delay(A,Y), $d_B$ + delay(B,Y), $d_C$ + delay(C,Y), $d_D$ + delay(D,Y) ];
$d_Z (d_A,d_B,d_C,d_D)$ = MAX[ $d_A$ + delay(A,Z), $d_B$ + delay(B,Z), $d_C$ + delay(C,Z), $d_D$ + delay(D,Z) ];

So let's apply this to our circuits in exercise 5.9. Let's start with trying to figure out the delay behavior of the 16-bit adder (Figure 5.18). To do this we need to know the delay behavior of the 4-bit adder and the 16-bit LCU.

For a 4-bit adder, the $d_S$ = MAX[$d_{AB}$+4, $d_{C0}$+3], $d_P$ = $d_{AB}$+2, and $d_G$ = $d_{AB}$+3 (here we assume A and B always arrive at the same time so we group them into a single input and let the delay be the longest of either path). We can summarize the delay behavior as a grid:

| **4-bit adder** | $d_{AB}$ + ? | $d_{C0}$ + ? |
|---|---|---|
| $d_S$ | +4 | +3 |
| $d_P$ | +2 | N/A |
| $d_G$ | +3 | N/A |

Now we can consider the 4-bit adder as a black box and to calculate the delay for each output and all we need to know is the delay value for each input. So in the 16-bit adder, looking at the leftmost adder we see that the $d_{AB}$ is @0 and $d_{C12}$ is @5. So this means that $d_{S[15\text{-}12]}$ = MAX[ 0 + 4, 5 + 3 ] = @8! Notice that the $d_P$ = MAX[ $d_{AB}$+2 ] = 0+2 = @2 and $d_G$ = MAX[ $d_{AB}$+3 ] = 0+3 = @3, which is exactly what Figure 5.18 says.

We also need to consider the delay behavior of the LCU in the 16-bit adder. The LCU has inputs P3,P2,P1,P0,G3,G2,G1,G0, and C0. It has outputs P3-0,G3-0,C1,C2,C3,C4:

| LCU(16) | $d_{P3}$ | $d_{P2}$ | $d_{P1}$ | $d_{P0}$ | $d_{G3}$ | $d_{G2}$ | $d_{G1}$ | $d_{G0}$ | $d_{C0}$ |
|---|---|---|---|---|---|---|---|---|---|
| $d_{P3\text{-}0}$ | +1 | +1 | +1 | +1 | N/A | N/A | N/A | N/A | N/A |
| $d_{G3\text{-}0}$ | +2 | +2 | +2 | +2 | +2 | +2 | +2 | +2 | N/A |
| $d_{C1}$ | N/A | N/A | N/A | +1 | N/A | N/A | N/A | +1 | +2 |
| $d_{C2}$ | N/A | N/A | +1 | +2 | N/A | N/A | +1 | +2 | +2 |

| $d_{C3}$ | N/A | +1 | +2 | +2 | N/A | +1 | +2 | +2 | +2 |
|---|---|---|---|---|---|---|---|---|---|
| $d_{C4}$ | +1 | +2 | +2 | +2 | +1 | +2 | +2 | +2 | +2 |

If we make the simplifying assumption that all the P's arrive at the same time as do all the G's, then we can make the table smaller:

| **LCU(16)** | $d_P$ | $d_G$ | $d_{C0}$ |
|---|---|---|---|
| $d_{P3-0}$ | +1 | N/A | N/A |
| $d_{G3-0}$ | +2 | +2 | N/A |
| $d_{C1}$ | +1 | +1 | +2 |
| $d_{C2}$ | +2 | +2 | +2 |
| $d_{C3}$ | +2 | +2 | +2 |
| $d_{C4}$ | +2 | +2 | +2 |

With the table for **LCU(16)** and **4-bit adder** we can now describe the behavior for the 16-bit adder…

| 16-bit adder | $d_{AB}$ | dC0 |
|---|---|---|
| $d_S$ | MAX[ $d_S$, $d_P$+$d_{C3}$+$d_S$, $d_Q$+$d_{C3}$+$d_S$ ] = MAX[ 4, 2+2+3, 3+2+3 ] = MAX[ 4, 7, 8 ] = **+8** | MAX [ $d_{C3}$ + $d_S$ ] = MAX[ 2 + 3 ] = **+5** |
| $d_{P3-0}$ | MAX[ $d_P$ + $d_{P3-0}$ ] = MAX[ 2 + 1 ] = MAX[ 3 ]= **+3** | N/A |
| $d_{G3-0}$ | MAX[ $d_P$ + $d_{G3-0}$, $d_G$ + $d_{G3-0}$] = MAX[ 2 + 2, 3 + 2 ] = MAX[ 4, 5] = **+5** | N/A |
| $d_{C16}$ | MAX[ $d_P$ + $d_{C4}$, $d_G$ + $d_{C4}$ ] = MAX[ 2 + 2, 3 + 2 ] = MAX[ 4, 5 ] = **+5** | MAX[ $d_{C4}$ ] = MAX[ 2 ] = **+2** |

So we got each cell by following all paths from the input to the output, whenever we encountered a block, we used our lookup tables to determine the delay for passing through that block. Cleaning up the table we get:

| **16-bit adder** | $d_{AB}$ | $d_{C0}$ |
|---|---|---|
| $d_S$ | +8 | +5 |
| $d_{P3-0}$ | +3 | N/A |
| $d_{G3-0}$ | +5 | N/A |
| $d_{C16}$ | +5 | +2 |

Now from the chart is it easy to explain the behavior from Figure 5.18 and also the behavior in Exercise 5.9 a/b on the homework where component only added +5 to the delay.

When AB and C0 all arrive at the same time (@0), then we see that $d_S$ = MAX[0+8, 0+5] = @8 and $d_{C16}$ = MAX[0+5, 0+2] = @5 and $d_{P3-0}$ = MAX[0+3] = @3 and $d_{G3-0}$ = MAX[0+5] = @5. Exactly, matching the delays in Figure 5.18.

However, when $d_{AB}$ = @0 and $d_{C0}$ = @7 (as in 5.9b) we get entirely new delay behavior. Now $d_S$ = MAX[0+8, 7+5] = MAX[9,12] = @12 and $d_{C16}$ = MAX[0+5, 7+2] = MAX[5, 9] = @9, while $d_{P3-0}$ = MAX[0+3] = @3 and $d_{G3-0}$ = MAX[0+5] = @5 still. Voila!

Note that this technique will fail on logic that has feedback.