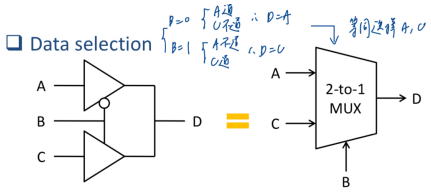
Digital System Design and Lab: HW3

Lo Chun, Chou
R13922136

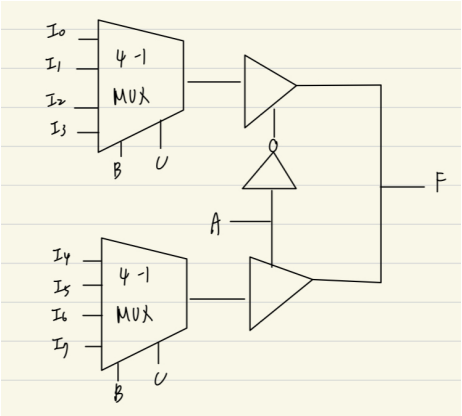
May 16, 2025

1

By lecture slide LEC-09 p.10-11, we knew that using two three-state buffers with one inverter could do data selection, and is equivalent to a 2-to-1 MUX:

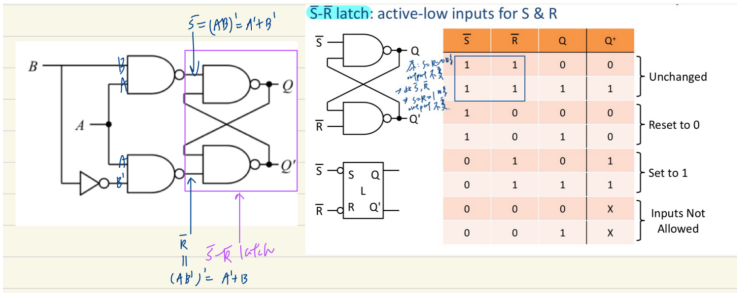


So, we can use the two 4-to-1 MUXs and this setting to implement the 8-to-1 MUX as follows:



2

First, observe that part of the given circuit is a $\bar{S} - \bar{R}$ latch:



with the inputs:

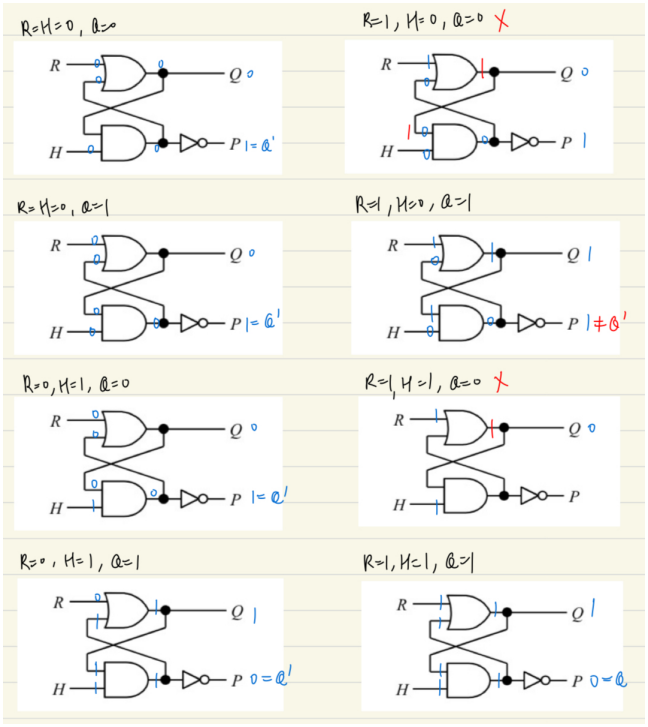
$$\bar{S} = (AB)' = A' + B'$$
$$\bar{R} = (AB')' = A' + B$$

Thus, we can form the truth table of this latch by checking the values of $\bar{S}$, $\bar{R}$ and Q in the table above:

A	B	$\bar{S}$	$\bar{R}$	Q	Q^+
0	0	1	1	0	0
0	0	1	1	1	1
0	1	1	1	0	0
0	1	1	1	1	1
1	0	1	0	0	0
1	0	1	0	1	0
1	1	0	1	0	1
1	1	0	1	1	1

3

The following is the different cases of the latch:



(1)

From the above cases, we can see that when $R = 1$ and $H = 0$, $P = 1 \neq Q' = 0$. Therefore, we should not let:

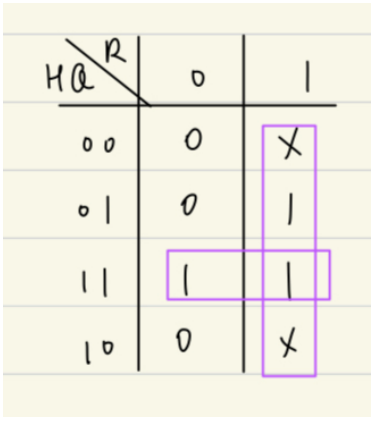
$$R = 1 \quad \text{and} \quad H = 0$$

(2)

The next-state table is shown below:

R	H	Q	Q+
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	X
1	0	1	1
1	1	0	X
1	1	1	1

And we can construct the K-map as follows:



Which would give us the characteristic equation:

$$Q^{+} = R + H \cdot Q$$

4

(1)

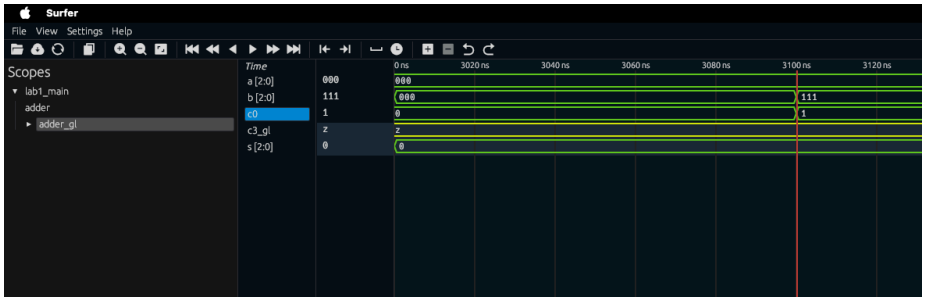
```
// carry-lookahead adder, gate level modeling
// Do not modify the input/output of module
module cla_gl(
    output C3,       // carry output
    output[2:0] S,   // sum
    input[2:0] A, B, // operands
    input C0         // carry input
    );

    // TODO:: Implement gate-level CLA

    // aim: First calculate the generate and propagate for each bit
    wire G0, G1, G2, P0, P1, P2;        // G is the generate, P is the propagate, both 3 bits
    // Note: cannot use: assign G0 = A[0] & B[0]; since we are required to use gate level moduling
    // Check LAB-01.pdf p.33 for the formula
    // G_i = A_i & B_i
    AND g_0(G0, A[0], B[0]);
    AND g_1(G1, A[1], B[1]);
    AND g_2(G2, A[2], B[2]);
    // P_i = A_i + B_i
    OR or_0(P0, A[0], B[0]);
    OR or_1(P1, A[1], B[1]);
    OR or_2(P2, A[2], B[2]);

    // aim: Then calculate the carry for each bit
    wire Cin_0, Cin_1, Cin_2, Cout_2;           // store the precomputed carries
    assign Cin_0 = C0;                          // C0 is the initial carry input

    // explain: the following equations are using notations in LAB-01.pdf p.33
    // intxy represents the y-th intermediate term in equation of C_x
    // subaim: C_1 = G_0 + P_0 & C_0
    wire p0c0;
    AND int11(p0c0, P0, Cin_0);
    OR c_1(Cin_1, G0, p0c0);

    // subaim: C_2 = G_1 + P_1 & C_1
    wire g0p1, c0p0p1;
    AND int21(g0p1, G0, P1);
    AND int22(c0p0p1, p0c0, P1);
    // note: we add 0 since we only need to OR 3 terms, but we need to choose from OR and OR4
    OR4 c_2(Cin_2, G1, g0p1, c0p0p1, 1'b0);

    // subaim: C_3 = G_2 + P_2 & C_2
    wire g1p2, g0p1p2, c0p0p1p2;
    AND int31(g1p2, G1, P2);
    AND int32(g0p1p2, g0p1, P2);
    AND int33(c0p0p1p2, c0p0p1, P2);
    OR4 c_3(Cout_2, G2, g1p2, g0p1p2, c0p0p1p2);

    // aim: Calculate the result of each full adder
    // note: the carries are stored in Di (dummy), since we've already precomputed them, and these Di would not be used later
    wire D0, D1, D2;
    FA fa_0(D0, S[0], A[0], B[0], Cin_0);
    FA fa_1(D1, S[1], A[1], B[1], Cin_1);
    FA fa_2(D2, S[2], A[2], B[2], Cin_2);
endmodule
```

(2)

```
//  ripple-carry adder, gate level modeling
//  Do not modify the input/output of module
module rca_gl(
    output C3,       // carry output (C3 is the output of the MSB adder)
    output[2:0] S,   // sum (S is the 3-bit sum, where S[2] is the MSB and S[0] is the LSB)
    input[2:0] A, B, // operands
    input C0         // carry input
    );


    // TODO:: Implement gate-level RCA

    wire c0, c1, c2, c3;
    assign c0 = C0;
    assign C3 = c3;

    // First consider the least significant bit (LSB)
    // We add the inputs A[0], B[0] along with the C_in (C0)
    // then we get the sum of this bit S[0] and the carry C1 to the next adder
    FA fa_0(c1, S[0], A[0], B[0], c0);
    // middle bit
    FA fa_1(c2, S[1], A[1], B[1], c1);
    // most significant bit (MSB)
    FA fa_2(c3, S[2], A[2], B[2], c2);

endmodule
```

5

I use Surfer instead of GTKWave to present the waveform:



6

We can find the maximum delay and one of the transition from the attached terminal output screenshots.

(1)

```
The maximum delay is 23 ticks on transition 000+000+0 --> 000+111+1
lab1.v:70: $finish called at 3276800000 (1ps)
3276800 / 000 / 111 / 1 / 1000 / 1111
```

(2)

```
The maximum delay is 20 ticks on transition 000+000+0 --> 000+011+1
lab1.v:70: $finish called at 3276800000 (1ps)
3276800 / 000 / 011 / 1 / 0100 / z111
```

7

First, since we are assuming a n-bit carry lookahead, we must compute $C_1, \ldots, C_n$. From the implementation details at p.33 in LAB-01.pdf, we can generalize the equation of C_n as follows:

$$
\begin{aligned}
C_1 &= G_0 + C_0 \cdot P_0 \\
C_2 &= G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \\
C_3 &= G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \\
&\vdots \\
C_n &= \underline{G_{n-1}} + \underline{G_{n-2} \cdot P_{n-1}} + \cdots + \underline{G_0 \cdot P_1 \cdot \cdots \cdot P_{n-1}} + C_0 \cdot P_0 \cdot \cdots \cdot P_{n-1}
\end{aligned}
$$

And C_n can be rewritten as:

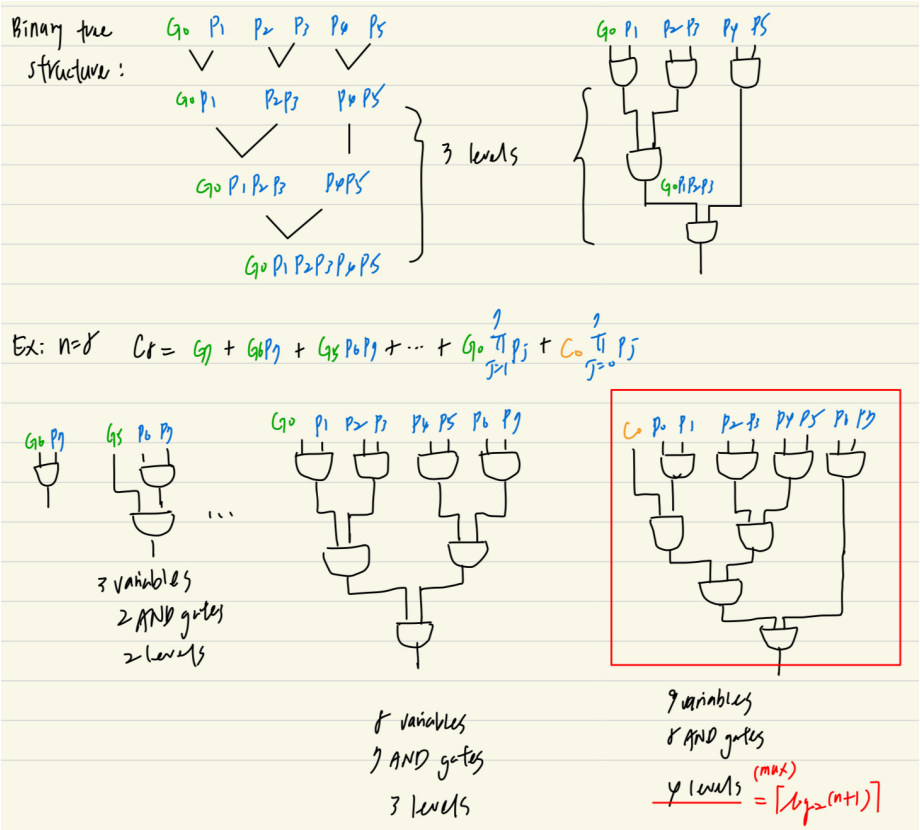
$$
C_n = \underline{\sum_{i=0}^{n-1} \left(G_i \cdot \prod_{j=i+1}^{n-1} P_j \right)} + C_0 \cdot \prod_{j=0}^{n-1} P_j
$$

Observe the equation for C_3, we can see that since we are only allowed to use 2-input gates, for each term, we need:

- G_2: 0 AND gate
- $G_1 \cdot P_2$: 1 AND gate
- $G_0 \cdot P_1 \cdot P_2$: 2 AND gates
- $C_0 \cdot P_0 \cdot P_1 \cdot P_2$: 3 AND gates

And we need 3 OR gates to sum up all the terms.

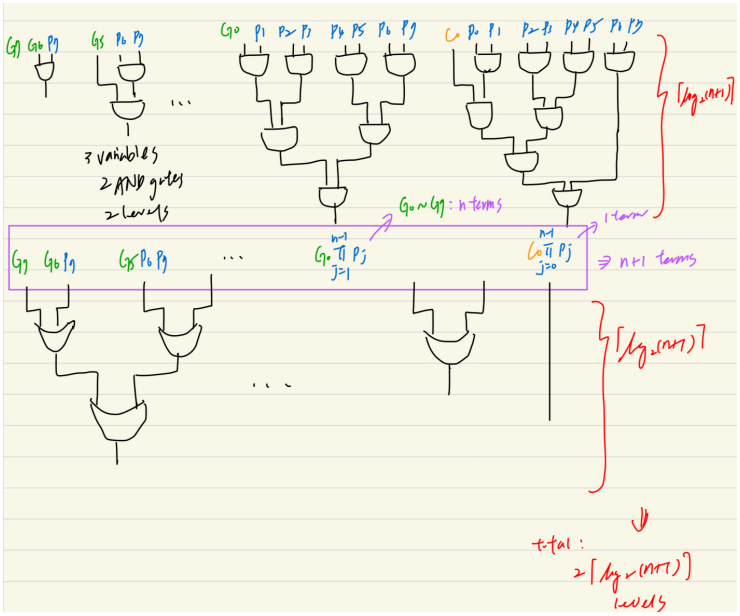
Consider the structure of a more complicated example:



We can see that for $n = 8$, for the term that needs the most levels, it needs 4 levels of AND gates, and this happens at $C_0 \cdot \prod_{j=0}^{7} P_j$.

Generalizing this to the n-bit case, the maximal depth happens at $C_0 \cdot \prod_{j=0}^{n-1} P_j$, and this requires $\lceil \log_2(n+1) \rceil$ levels of AND gates.

The next step is to OR these resulting $n+1$ terms together, and this is again a binary tree structure, and the depth is also $\lceil \log_2(n+1) \rceil$, the process is similar and shown below:



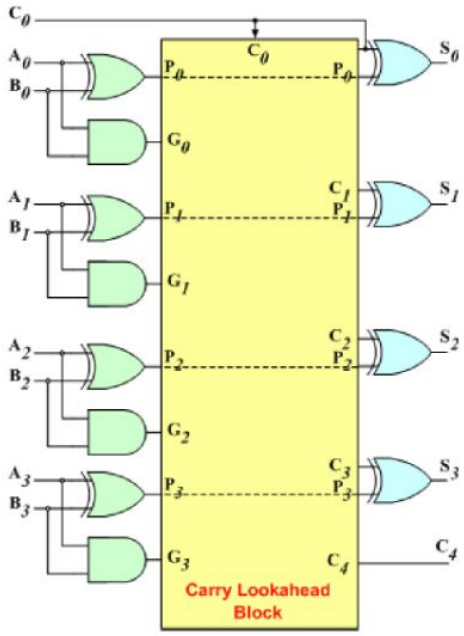
Last, we did not count the process of generating G_i, P_i as a level, so the total depth is:

$$\lceil \log_2(n+1) \rceil + \lceil \log_2(n+1) \rceil + 1 = 2\lceil \log_2(n+1) \rceil + 1$$

But until now, we have not considered the sum, which has the equation:

$$S_i = A_i \oplus B_i \oplus C_i$$

Thus, for each bit, we need 2 XOR gates ($A_i \oplus B_i$, $(A_i \oplus B_i) \oplus C_i$). Observe the following figure, we can see that we only need to compute S_3 by $C_3 \oplus P_3$, so we only need up to C_{n-1}:



Therefore, the sum depth is C_{n-1} (which needs $2\lceil \log_2(n) \rceil + 1$ levels) plus the two XOR levels, which is $2\lceil \log_2(n) \rceil + 3$.

From the same above graph, the number of levels would depend on whether C_4 or S_3 is having the maximal depth, and in our generalized case, it would be:

$$\max\{2\lceil \log_2(n+1) \rceil + 1, 2\lceil \log_2(n) \rceil + 3\} \qquad \square$$