High Speed CMOS Circuit Design

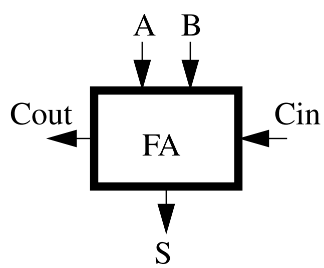
Lecture 6: Adders

(c) 1998 David Harris

1.0 Full Adders

A common adder building block is a full adder, also known as a 3:2 carry-save adder (CSA) because it takes three inputs and produces two outputs. The full adder takes three inputs, A, B, and Cin, and adds them to produce a two bit result, Cin and Sum. It can be viewed as a unary to binary converter.

FIGURE 1. Full Adder

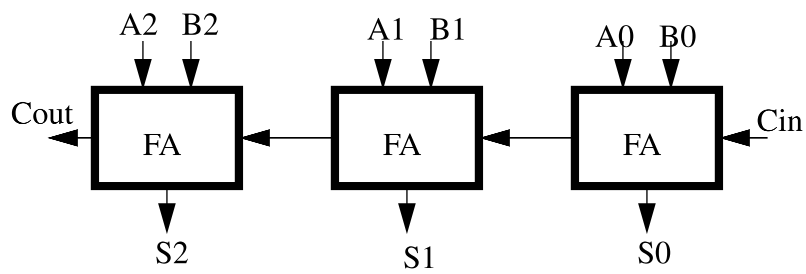


Static full adders are often built with a symmetric CMOS gate (Fig 8.6 of Weste) or with transmission gates (shown incorrectly in fig 8.12 of Weste). The symmetric design is compact and fast from Cin to Cout, but slower from any input to Sum. Hence it is a good choice when the carry path is most critical. Dynamic full adders are usually built with dual rail sum and carry gates.

2.0 Ripple Carry Adders

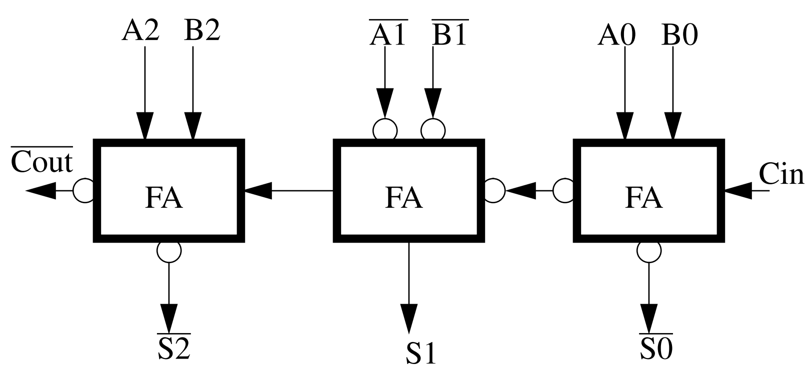
There are many ways to build an N-bit adder that sums two N-bit inputs A and B plus perhaps a carry Cin. The simplest scheme is to just cascade a number of ripple carry adders, as shown in Figure 2.

FIGURE 2. 3-bit Ripple Carry Adder



The full adders used in ripple carry adders should be optimized for fast Cin->Cout paths. One trick is to build an inverting carry chain that can use a single gate from Cin to Cout, rather than a non-inverting chain which requires two gates. This usually is slightly faster when static logic is used. Domino adders need the inversion anyway, so the trick is less relevant.

FIGURE 3. Inverting Ripple Carry Adder



The delay through a 16 bit ripple carry adder is:

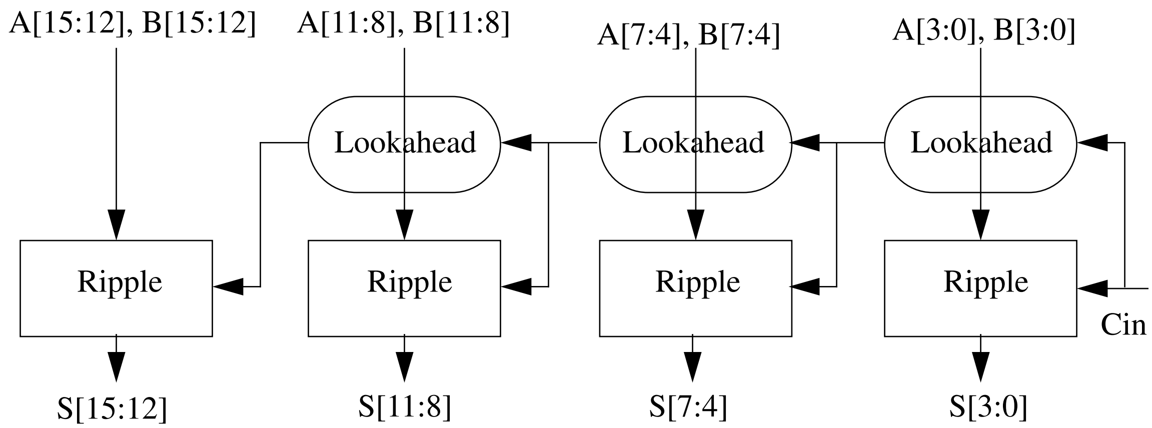
$$t = 16t_{CSA} \tag{EQ 1}$$

3.0 Carry Lookahead Adders

Even a domino ripple carry adder is far too slow for most long adders, since an N bit ripple carry adder takes N full adder delays. The delay can be reduced by quickly computing the carry through several bits using one complicated gate instead of a cascade of several full adders.

A 16 bit carry lookahead adder is shown in Figure 4. It is built from four 4-bit blocks. Each block contains a four-bit ripple carry adder and a lookahead circuit. The lookahead circuits quickly send the carry to the most significant bits.

FIGURE 4. Carry Lookahead Adder



To understand the lookahead circuitry, we need to define "Propagate" and "Generate" signals. Propagate means that the carry out of a block will be true if the carry in is true; the block propagates carries from the input to the output. Generate means that the carry out of a block will always be true; the block generates a carry out.

For a block consisting of a single bit, a carry will always be produced if both inputs A and B are true. Thus, generate signal G = A*B. A carry will be passed from input to output if either A or B is true. Thus propagate signal P = A+B. Propagate could also be written A xor B, but xor is more complex, so a simple OR is usually used.

For a block consisting of many bits, generate and propagate signals can be derived from the smaller block (or single bit) generates and propagates. For example, consider the G4 and P4 signals for a 4-bit block:

$$P4 = P_3P_2P_1P_0 \tag{EQ 2}$$

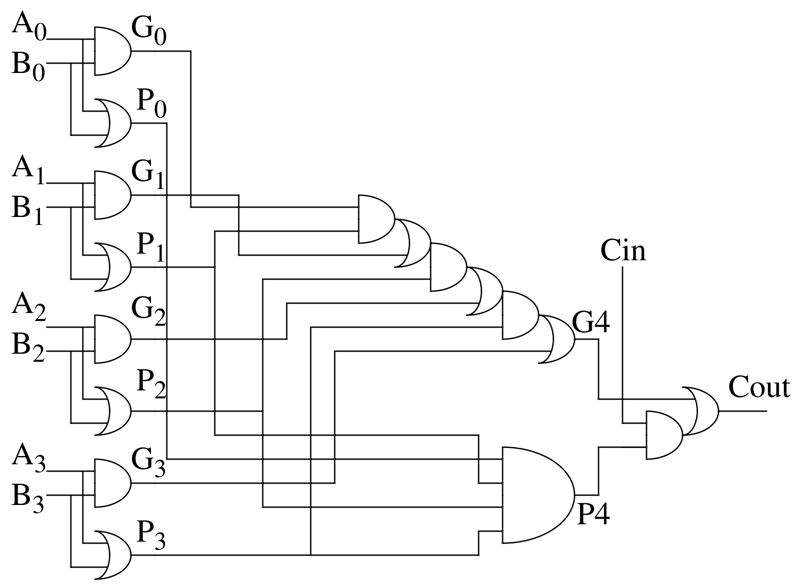
$$G4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0)) \tag{EQ 3}$$

The carry out of a block is true either if the block generates a carry or if it propagates a carry and the carry in was true:

$$Cout = G + P*Cin \tag{EQ 4}$$

A 4-bit lookahead block can be constructed using generate and propagate logic, as shown in Figure 5. Four single bit generate and propagate gates are needed. A four bit generate and propagate gate is also needed. Finally, the carry out is computed.

FIGURE 5. 4-bit lookahead circuit



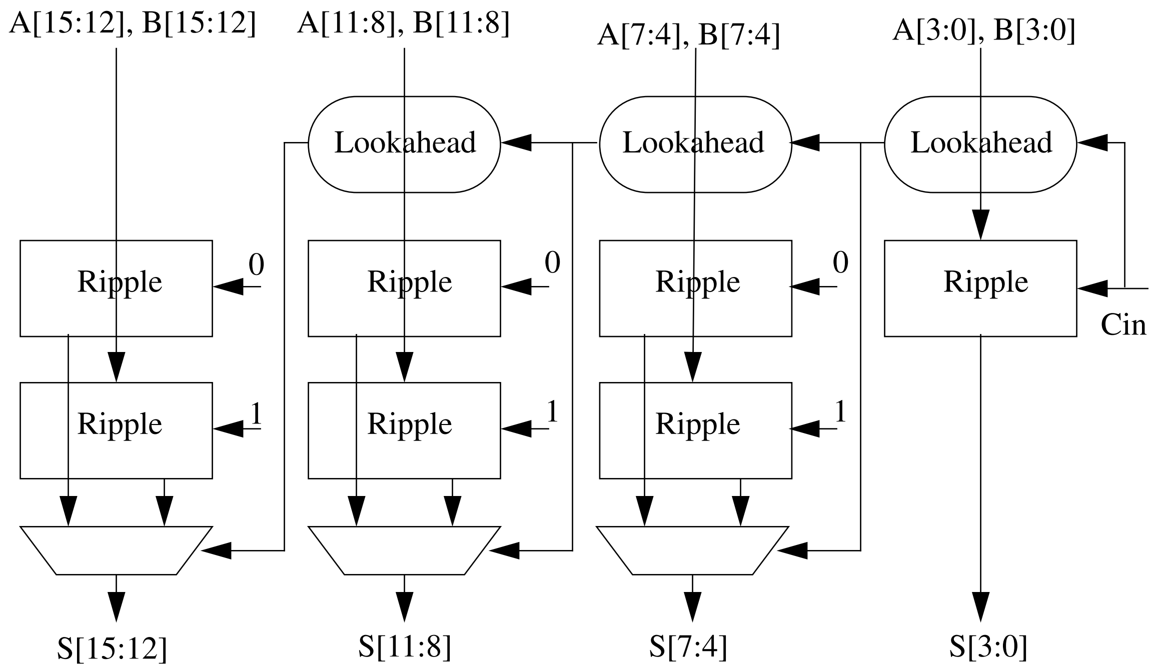
The total delay through the 16 bit adder is thus the delay to compute the bitwise and 4-bit generate and propagate signals, the time for the carry to ripple through three AND/OR gates, plus the time for the final four bit ripple carry adder to run:

$$t = t_{GP} + t_{GP4} + 3t_{AND/OR} + 4t_{CSA} \quad \text{(EQ 5)}$$

4.0 Carry Select Adders

The carry lookahead adder is much faster than rippling through N CSAs for N above about 4. However, it still involves ripple carry through a smaller block for the final sum computation. The critical path involves finding the carry in to the last block, then rippling this carry through a short adder. The usual logic design trick of precomputing answers applies here: two short adders can be used to speculatively calculate the sum assuming the carry in is a 0 or a 1. Then the actual carry in can trigger a mux which selects the appropriate sum, as shown in Figure 6.

FIGURE 6. Carry Select Adder



This technique is called a carry select adder. The critical path now replaces a short ripple with a single mux:

$$t = t_{GP} + t_{GP4} + 3t_{AND/OR} + t_{MUX} \qquad \text{(EQ 6)}$$

5.0 Logarithmic Adders

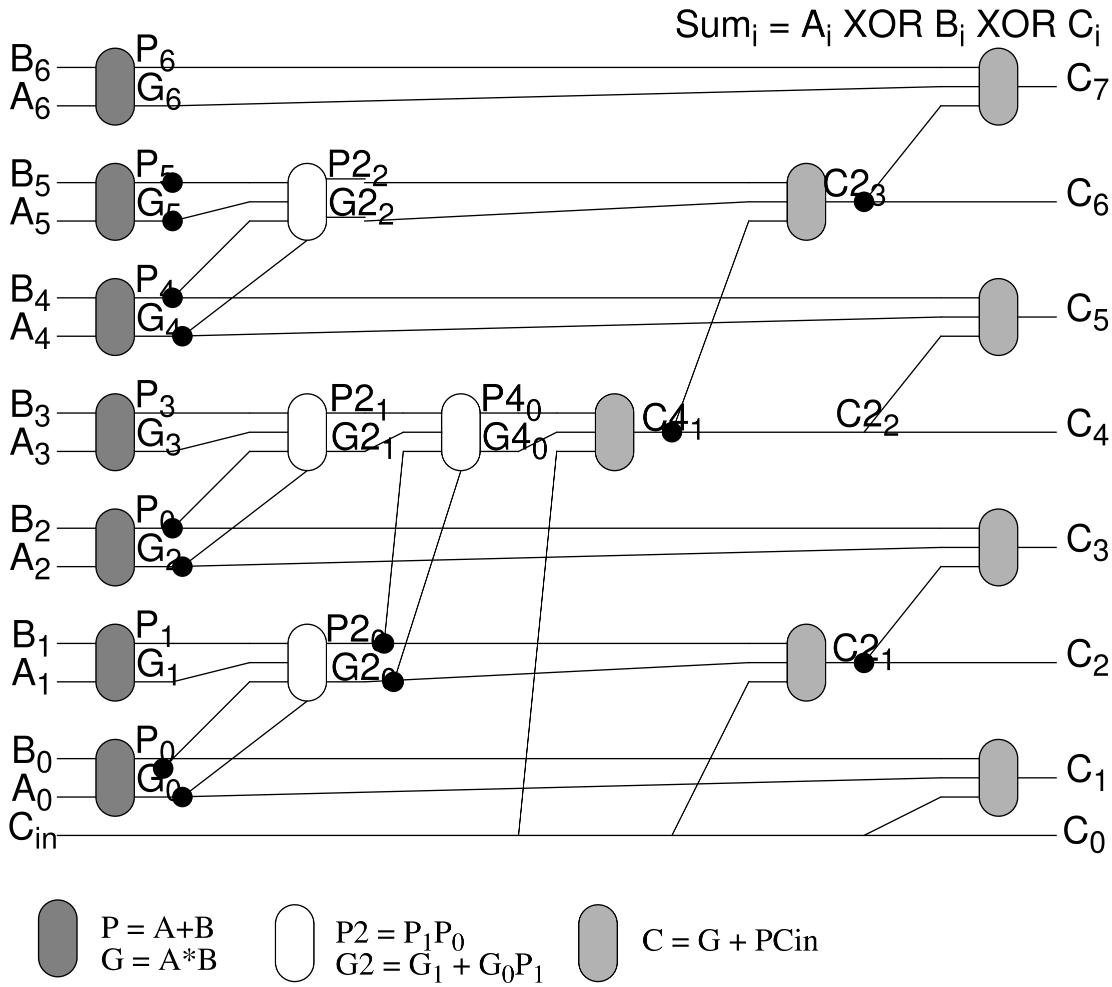
For adders of 4-16 bits, the carry selection architecture is very good. For wider adders, the time required to ripple through the carry lookahead circuitry is still linear in the number of bits and may become unacceptably large; for example, a good 64 bit lookahead adder takes about 11 FO4 delays. It can be reduced by doing a lookahead across lookahead blocks, and even a lookahead across lookaheads across lookaheads, etc. By building such recursive lookahead tree, the add time can drop from a linear to logarithmic function of the number of bits. Thus, such adders are known as “logarithmic” or “tree” adders.

The logarithmic lookahead can be done as follows to compute the generate, propagate, and carry signals for an N-bit adder:

1. Compute single bit $G_i = A_iB_i$, $P_i = A_i + B_i$ $(0 <= i < N)$
2. Compute $G2_i = G_{2i+1} + G_{2i}P_{2i+1;}$ $P2i = P_{2i+1}P_{2i}$ $(0 <= i < N/2)$
3. Compute $G4_i = G2_{2i+1} + G2_{2i}P2_{2i+1;}$ $P4_i = P2_{2i+1}P2_{2i}$ $(0 <= i < N/4)$
4. ...(continue up binary tree to find all generates & propagates)
5. ...(work down tree to find carry ins)
6. $C4_{2i+1} = G4_{2i} + C8_iP4_{2i;}$ $C4_{2i} = C8_i$

7. $C2_{2i+1} = G2_{2i} + C4P2_{2i;}\ C2_{2i} = C4_i$
8. $Cin_{2i+1} = G_{2i} + C2_iP_{2i;}\ Cin_{2i} = C2_i$
9. $Sum_i = A_i$ xor B_i xor C_i

FIGURE 7. 8-bit logarithmic adder



The example combines two bits per stage, corresponding to a logarithm of base 2. With larger fan-in gates, especially feasible for domino designs, base 4 may be used to reduce the number of levels in the logarithmic tree.

In this design, the delay involves both rising up the tree to compute P and G for progressively larger blocks, then descending the tree to compute C into each bit. Logarithmic adders can be combined with carry selection techniques so that some or all of the descent can be skipped to make the adder faster.

Notice that all of the logic to compute carries is non-inverting. This means that the carries could be computed with single-rail domino. Unfortunately, the sum logic uses an XOR, which requires both true and complementary versions of the carries. Therefore, the adder must either switch to static logic at the end to implement the non-monotonic function, or use dual-rail domino throughout. Switching to static means that there must later be a conversion back to domino. Since clock skew must be budgeted at the conversion, this can degrade cycle time. Thus, as transistors become more plentiful, fully dual-rail adders become more popular.

Good 64-bit logarithmic adders can operate in about 7 FO4 delays.

6.0 Example: 64 bit Adder Design

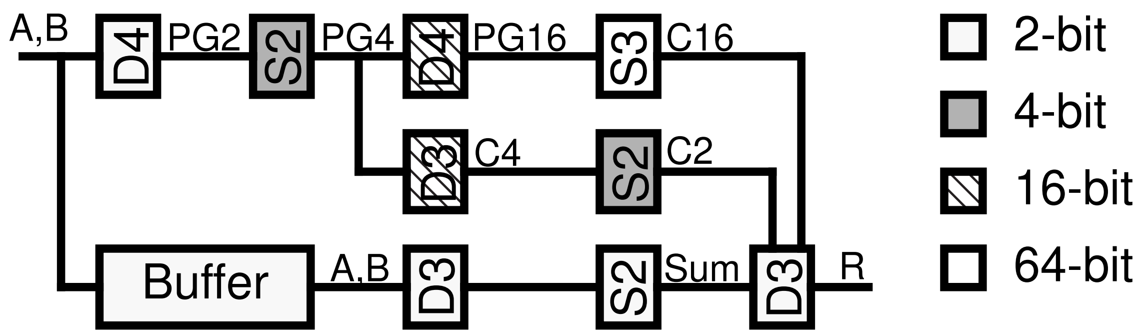
To illustrate some of the issues in a large-scale design, let us look at a complete 64 bit adder. The adder is built entirely from dual-rail domino. It employs two levels of carry selection to minimize delay. It is limited to only perform addition and does not accept a carry in; in contrast, most adders used in processors must optionally invert an input and add a carry to perform subtraction. Simulated in the HP14 0.6 micron process with estimated long wire loads, the adder has a latency of 6.4 FO4 delays.

The adder architecture is illustrated in Figure 8. The adder is divided into 2, 4, 16, and 64 bit blocks. The gates in the critical path are labeled S or D for static or dynamic. The number following the letter indicates the number of series transistors. This is a rough metric of the complexity and delay of the gate.

In each of the 32 two-bit blocks, the 2-bit propagate and generate signals are computed from A and B. In each of 16 four-bit blocks, the 2-bit propagate and generate signals are combined into 4-bit propagates and generates. In the sixteen-bit blocks, the P4 and G4 signals are further combined into P16 and G16 signals. Finally, in the top-level 64-bit block, the carries into each 16 bit block are computed. They are driven back to muxes in each 2-bit block to select the appropriate result.

To do this, the adder must have calculated sums for each 16-bit block assuming the carry in was 0 and 1. This calculation itself is critical, so it is done with a smaller 16 bit logarithmic adder that shares the 4 bit P/Gs with the 64 bit adder. In the middle fork of the picture, the 4 bit carries are computed from the 4-bit P and G signals. Then 2 bit carries are also produced. The 2-bit carries do an initial level of carry selection so that only 2-bit ripple carry adders, shown in the bottom form of the figure, are needed for speculative sum computation. The 16 bit and 2 bit carry selection is merged into a single large mux because both carry signals are critical. Notice how the inputs are buffered before the speculative sum generation to avoid loading the critical path.

FIGURE 8. Adder architecture



The contents of each block are specified in detail below:

2-bit Logic

2-bit Propagates & Generates

- $P2 = P_0P_1 = (A_0+B_0)(A_1+B_1)$
- $G2 = G_1+G_0P_1 = A_1B_1 + A_0B_0(A_1+B_1)$

Speculative Sums to bit b assuming carry in c: $sumc_b$

- $sum0_0 = A_0$ xor B_0
- $sum1_0 = \sim(A_0$ xor $B_0)$
- $sum0_1 = A_1$ xor B_1 xor (A_0B_0)
- $sum1_1 = A_1$ xor B_1 xor $(A_0 + B_0)$

Final Result

- $R_0 = cin16 \ ?\ (cin21 \ ?\ sum1_0 : sum0_0) : (cin20 \ ?\ sum1_0 : sum0_0)$
- $R_1 = cin16 \ ?\ (cin21 \ ?\ sum1_1 : sum0_1) : (cin20 \ ?\ sum1_1 : sum0_1)$

4-bit Logic

Carry in to 2 bit block b assuming cin to 16 bit block is c: $cin2c_b$

- $cin20_0 = cin40$
- $cin21_0 = cin41$
- $cin20_1 = g2_0 + p2_0cin40$
- $cin11_1 = g2_0 + p2_0cin41$

4-bit Propagates and Generates

- $g4 = g2_1 + p2_1g2_0$

- $p4 = p2_0p2_1$

16-bit Logic

Carry in to 4 bit block b assuming cin to 16 bit block is c: $cin4c_b$

- $cin40_0 = 0$
- $cin41_0 = 1$
- $cin40_1 = g4_0$
- $cin41_1 = g4_0 + p4_0$
- $cin40_2 = g4_1 + p4_1(g4_0)$
- $cin41_2 = g4_1 + p4_1(g4_0 + p4_0)$
- $cin40_3 = g4_2 + p4_2(g4_1 + p4_1(g4_0))$
- $cin41_3 = g4_2 + p4_2(g4_1 + p4_1(g4_0 + p4_0))$

16-bit Propagates and Generates

- $g16 = g4_3 + p4_3(g4_2 + p4_2(g4_1 + p4_1g4_0))$
- $p16 = p2_0p2_1p2_2p2_3$

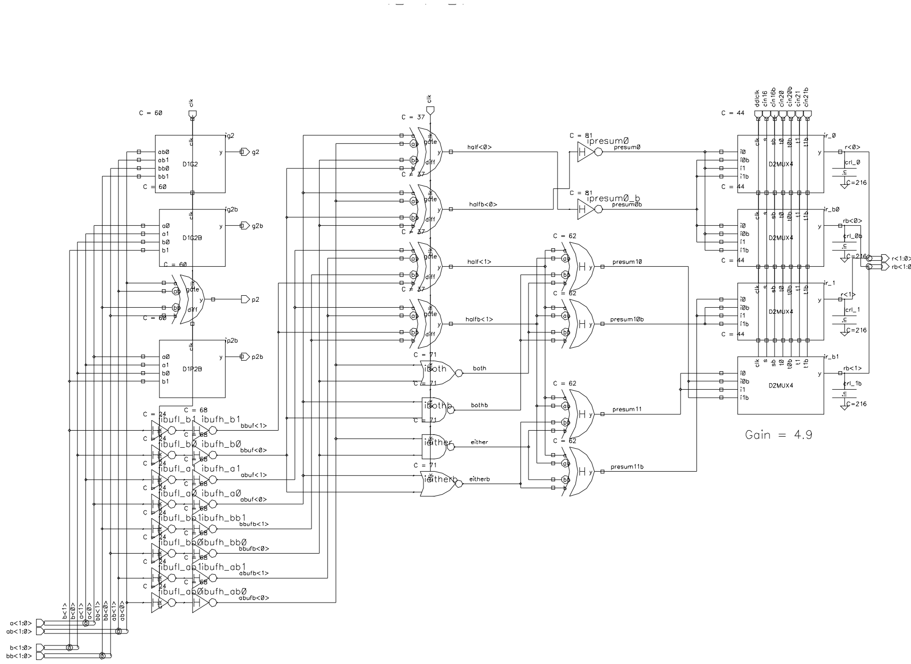
64-bit Logic

Carry in to 16 bit block b assuming cin to 16 bit block is c: $cin16_b$

- $cin16_0 = 0$
- $cin16_1 = g16_0$
- $cin16_2 = g16_1 + p16_1(g16_0)$
- $cin16_3 = g16_2 + p16_2(g16_1 + p16_1(g16_0))$

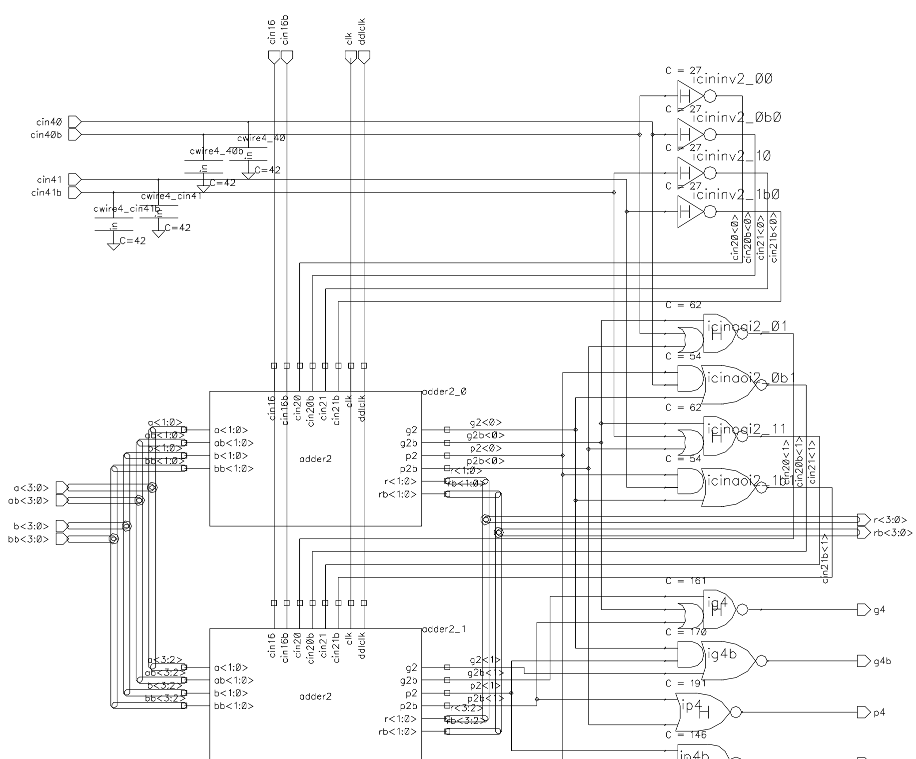
Complete schematics of the blocks are also shown in the following figures. The capacitors represent lumped capacitance of long wires. Large gates are necessary in the 16 and 64 bit blocks to drive the wires and the heavy loads attached.

FIGURE 9. 2-bit block schematic



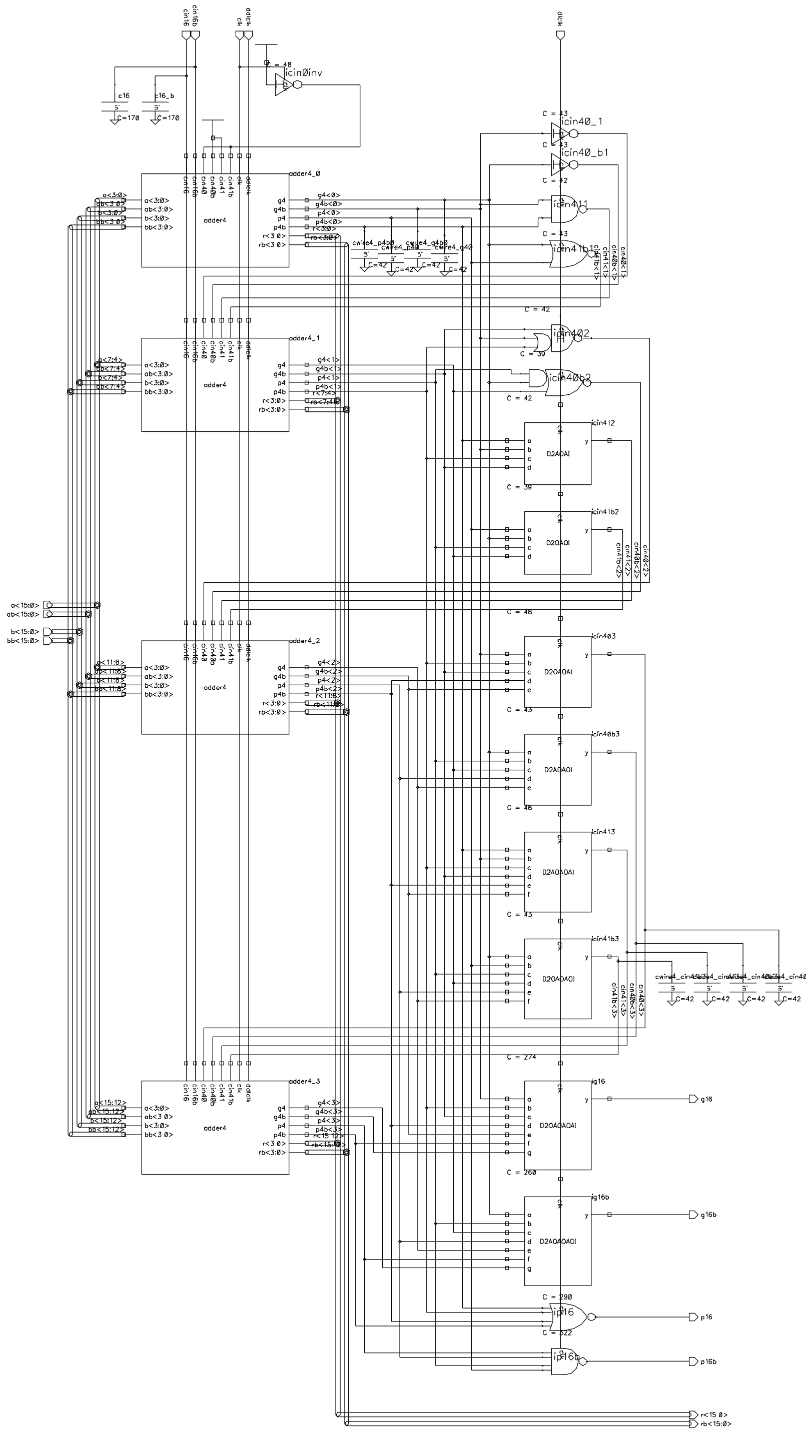
The 2-bit blocks contain the 2-bit P,G logic, the result selection multiplexors, and the buffered 2-bit sum computation logic.

FIGURE 10. 4-bit block schematic



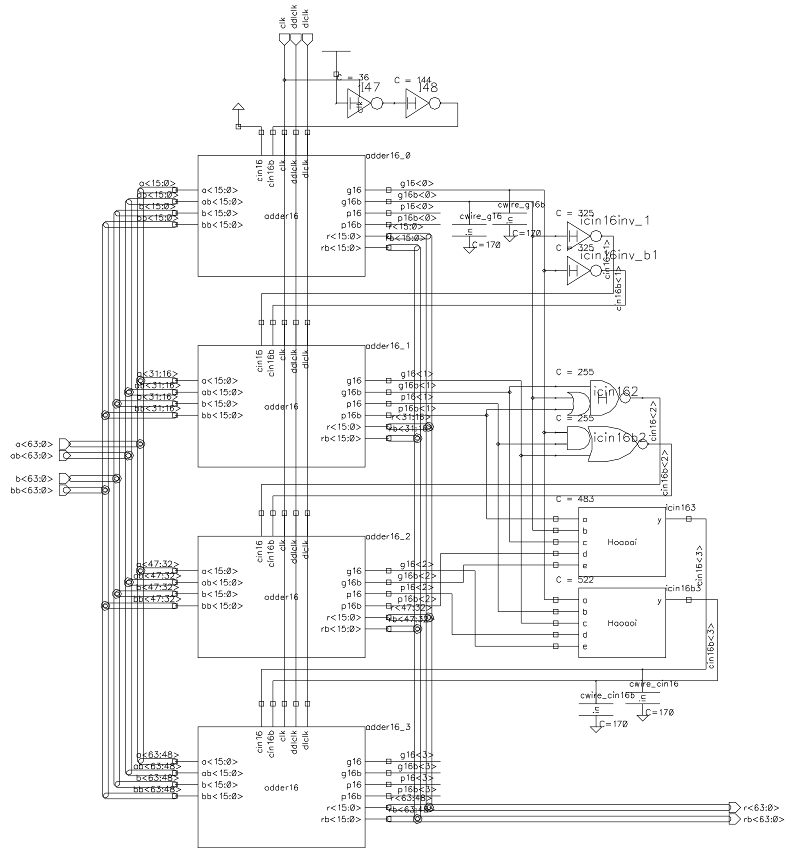
The 4-bit blocks contain 2 2-bit blocks, 4-bit P,G logic, and logic to compute the carries into each 2-bit block.

FIGURE 11. 16-bit block schematic



The 16-bit block contains 4 4-bit blocks, the 16-bit P/G logic, and gates to compute carries in to each 4 bit block.

FIGURE 12. 64-bit block schematic



Finally, the 64-bit block, representing the entire adder, contains 4 16-bit blocks and logic to compute the carry in to each 16 bit block.

7.0 Ling Adders

A clever designer noticed another way to write the adder equations which slightly reduces the critical path in the carry chain. Adders are such a specialized circuit that such savings is significant; the technique is now known as the "Ling" adder.

The Ling adder is based on an observation about the 4-bit generate and propagate signals. The conventional signals are defined below, repeated from earlier equations:

$$P4 = P_3P_2P_1P_0 \qquad \text{(EQ 7)}$$

$$G4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0)) \qquad \text{(EQ 8)}$$

The equations could be rewritten in terms of A and B inputs:

$$P4 = (A_3+B_3)(A_2+B_2)(A_1+B_1)(A_0+B_0) \qquad \text{(EQ 9)}$$

$$G4 = A_3B_3 + (A_3 + B_3)(A_2B_2 + (A_2 + B_2)(A_1B_1 + (A_1 + B_1)A_0B_0)) \quad \text{(EQ 10)}$$

This G4 equation is too complicated to efficiently implement in a single domino gate because it would require too many series transistors. However, if we introduce a “Pseudo-generate” signal H4 such that G4 = H4*P3, we find that H4 is easier to implement:

$$H4 = G_3 + G_2 + G_1P_2 + G_0P_1P_2 = A_3B_3 + (A_2B_2 + (A_2 + B_2)(A_1B_1 + (A_1 + B_1)A_0B_0)) \quad \text{(EQ 11)}$$

Indeed, H4 can be built from a domino gate with 4 series transistors.

As long as the AND of H4 and P3 occurs before G4 is actually used, the rest of the carry chain can be built using H4 in place of G4. Therefore, the four bit propagates and pseudo-generates can be computed in a single complex stage of domino logic, rather than in two gates as given in the previous section. It turns out that the gating with P3 can be cleverly woven into a non-critical path so the Ling adder may be slightly faster than a regular logarithmic adder.

Naffziger describes an excellent implementation of a 64-bit Ling adder used on HP PA-RISC chips in ISSCC96. The paper is unfortunately terse, but a Verilog model with the complete logic equations may help explain the adder operation (see the Ling Adder handout). The adder runs in 7 FO4 delays and is remarkably compact.

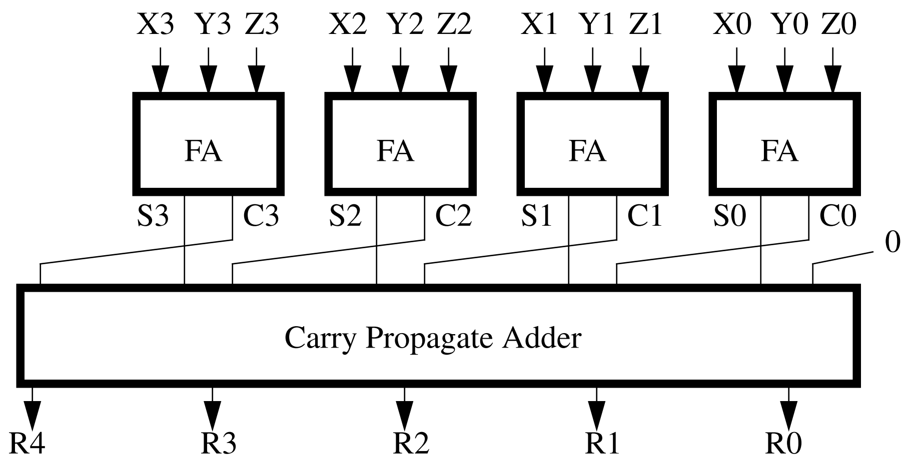
The adder implementation uses another trick of “pseudo-complements” to simplify design. Normally, dual-rail _h and _l signals are true complements of each other. This means that different gates must be designed for the _h and _l paths using DeMorgan’s law. Remarkably, most logic in an adder can be designed by using the same gates for _h and _l. Although the _h and _l signals are no longer true complements of each other, when they get consumed at the end of carry generation, the correct carries can be computed. The mathematics justifying this is nicely explained by Wang et. al in JSSC Feb. 1997. As a result, only half as many types of gates must be designed and laid out.

8.0 Multiple Input Adders

Sometimes it is necessary to add more than 2 N-bit numbers. This can be done by using CSAs to add the numbers and produce a sum and a carry output. Then a regular adder, using any of the architectures above, can add the sums and carries. The regular adder is often called a carry-propagate adder (CPA) to distinguish it from a CSA.

For example, consider adding three 4-bit numbers X, Y, and Z. The addition can be done with a 3:2 CSA and a CPA:

FIGURE 13. 3-input Adder



Notice how the carries out of the CSAs are shifted by one column before driving the CPA because the carry is one place more significant than the sum.

A similar approach is used in multipliers, which must sum large numbers of partial products.

9.0 Conclusions

We've explored a wide variety of adder architectures. The best architecture depends on the application. Very short, non-critical adders can be implemented with little effort or area using the ripple carry approach. Moderate length adders (4-16 bits) are often most efficiently implemented with a look-ahead approach, often combined with carry selection. Longer adders become too slow unless a logarithmic approach is used; therefore, high performance 64 bit processors use some form of logarithmic adder, usually with carry selection. The Ling adder is an interesting form that uses a clever logic optimization to simplify the carry path; it is a good choice for wide adders.