Notes for Today’s Lecture

Design of Circuits

Large streams from little fountains flow,
Tall oaks from little acorns grow.
    : David Everett, 1791. (written for a seven-year-old)

Motivations

• Not just for beauty.
• delay for each gate
  – few nanoseconds per gate
  – but want many millions of instructions per second
  – naive 32-bit adder would have circuit-delay ≈ 100 gates
  – processing an add-instruction requires more circuitry than
    just the basic adder
  – (do the math)
• space for each gate
  – errors per square inch of silicon
  – if two gates are connected but there are many other gates
    ‘between’ them, propagation-delay along (long) wires

• constraints on structure of gates:
  – fan-in
  – fan-out

Example: Test if (32-Bit) Word is Zero

• Motivations: e.g., for instruction BZ: Branch if zero
  – can reduce most other branches to this
• straightforward (naive) approach: textbook’s Figure 13.12
  (page 713)
  – sequentially OR inputs
• better scheme: textbook’s Figure 13.13 (page 714)
  – smaller number of levels
  – (same number of gates)
this scheme is better even if gates have fan-in greater than 2

• how getting this better scheme?
  • divide and conquer

**Model for Divide-and-Conquer: Adder**

• task: given two 32-bit numbers, produce their sum (and perhaps a carry-bit)

• most naive approach: sequence of one-bit adders (using Figure 13.10, page 709)
  • called a “ripple-carry adder”
  • circuit-delay of 96 gates

• next most naive approach: OK, trying some ‘dividing and conquering’, divide bits in halves, high-order and low-order; try to add them separately (Figure 13.15, page 717)
  • need a carry from the low-order bits to the high-order bits
  • this scheme is the same as ripple-carry!

• trick for better scheme: compute two sums, one in case carry-bit is 1 and the other in case carry-bit is 0; when the carry arrives, use it to select the correct result.
  • adders called “carry-lookahead” or “carry-select” use such a strategy
  • computing two sums actually doesn’t hurt much!

**Details of the Scheme:**

• $n$-bit adder, for $n$ a power of 2

• (constructed inductively/recursively)

• input-numbers $x$ and $y$ to be added: bits $x_1, x_2, \ldots, x_n$ — high-order to low-order — and $y_1, y_2, \ldots, y_n$

• two sets of outputs
  • in case the carry-in that this $n$-adder receives is 0, sum $s$ in bits $s_1, s_2, \ldots, s_n$ (high-order to low-order) and this sum’s carry-out in a bit $g$.
  • in case the carry-in that this $n$-adder receives is 1, sum $t$ in bits $t_1, t_2, \ldots, t_n$ (high-order to low-order) and this case’s sum’s carry-out in a bit $p$.

• (diagram)
The Construction:

1. Basis: \( n = 1 \)
   - one-bit inputs \( x \) and \( y \), four one-bit outputs \( s \) and \( g \), \( t \) and \( p \)
   - determine formulas for the outputs as follows:
     - in case the carry-in is 0:
       * sum of 0 and 0 is 0
       * sum of 0 and 1 is 1
       * sum of 1 and 0 is 1
       * sum of 1 and 1 is 0 with carry-out 1
     So:
     - this case’s sum \( s = \overline{x} \overline{y} \) OR \( xy \)
     - this case’s carry-out \( g = xy \)
     - in case the carry-in is 1:
       * sum of 0 and 0 plus carry-in 1 is 1
       * sum of 0 and 1 plus carry-in 1 is 0 with carry-out 1
       * sum of 1 and 0 plus carry-in 1 is 0 with carry-out 1
       * sum of 1 and 1 plus carry-in 1 is 1 with carry-out 1
     So:
     - this case’s sum \( t = \overline{x} \overline{y} \) OR \( xy \)
     - this case’s carry-out \( p = x \) OR \( y \)

   - Figure 13.16, page 718
   - no carry-in input: remember that carry-in will be used after addition to select between \( s \) (and \( g \)) or \( t \) (and \( p \)).

2. Induction: build \( 2n \)-adder from two \( n \)-adders
   - Figure 13.17, page 719
   - input-bits \( x_1, x_2, \ldots, x_{2n} \) and \( y_1, y_2, \ldots, y_{2n} \)
   - output-bits \( g, s_1, s_2, \ldots, s_{2n} \) and \( p, t_1, t_2, \ldots, t_{2n} \)

   (a) first give \( x_1, x_2, \ldots, x_n \) and \( y_1, y_2, \ldots, y_n \) to one \( n \)-adder (on the left),
   \( x_{n+1}, x_{n+2}, \ldots, x_{2n} \) and \( y_{n+1}, y_{n+2}, \ldots, y_{2n} \) to the other \( n \)-adder (on the right)

   (b) left adder returns \( g^L, s_1^L, s_2^L, \ldots, s_n^L \) and \( p^L, t_1^L, t_2^L, \ldots, t_n^L \);
      right adder returns \( g^R, s_1^R, s_2^R, \ldots, s_n^R \) and \( p^R, t_1^R, t_2^R, \ldots, t_n^R \)

   (c) we need \( g, s_1, s_2, \ldots, s_{2n} \) and \( p, t_1, t_2, \ldots, t_{2n} \)
   - case 0: to get \( g, s_1, s_2, \ldots, s_{2n} \), suppose the carry-in to the \( 2n \)-adder that we’re building is 0.
     - Then the lowest-order sum-bit \( s_{2n} \) which we need to compute is the sum of \( x_{2n} \) plus \( y_{2n} \) when the carry-in on the far right is 0
* coincidentally, \( s_n^R = x_{2n} \text{ PLUS } y_{2n} \) when the carry-in on the far right is 0
* so \( s_{2n} = s_n^R \)
  - similarly \( s_{2n-1} = s_{n-1}^R, s_{2n-2} = s_{n-2}^R, \ldots, s_{n+1} = s_1^R \)
  - next, the sum-bit \( s_n \) which we need to compute is \( x_n \text{ PLUS } y_n \) when the carry-in on the far right is 0
* but \( s_n^L = x_n \text{ PLUS } y_n \) when the carry-in in the middle is 0.
* fortunately, the carry-in in the middle is the carry-out in the middle. In this case when the carry-in on the far right is 0, the carry-out in the middle is \( g^R \).
* putting the preceding two points together: when \( g_R \) is 0, \( s_n^L \) provides the value that we need for \( s_n \)
* similar analysis shows that when \( g_R \) is 1, \( s_n = t_n^L \)
* so \( s_n = \overline{g^R} s_n^L \text{ OR } g^R t_n^L \)
  - similarly \( s_i = \overline{g^R} s_i^L \text{ OR } g^R t_i^L \) for other \( i \in \{1..n\} \),
  and \( g = \overline{g^R} g^L \text{ OR } g^R g^L \) (this can be reduced)
  - case 1: to get \( p, t_1, t_2, \ldots, t_{2n} \), suppose the carry-in to the \( 2n \)-adder that we’re building is 1. Then analysis as in the preceding case yields:
    - \( t_i = t_i^L \) for \( i \in \{(n+1) .. 2n\} \)
    - \( t_i = \overline{g^R} s_i^L \text{ OR } g^R t_i^L \) for \( i \in \{1..n\} \)
    - \( p = \overline{g^R} g^L \text{ OR } g^R g^L \)
  - for example of circuitry (for \( t_i \) with \( i \in \{1..n\} \)): Figure 13.18, page 720

This Circuit’s Circuit-Delay:

* recurrence-relation: \( D(1) = 3, D(2n) = D(n) + 3 \)
* solution: \( D(n) = 3(1 + \log_2 n) \)
* e.g. \( D(32) = 18 \), which is less than the other adder’s delay of 96
(The number of gates is larger, but only by a factor \( O(\log n) \), which is tolerable considering the better performance.

Class Exercises

1. With some technologies for circuits, instead of \text{AND}, \text{OR}-, and \text{NOT}-gates, only \text{NAND}-gates were used. Explain how to construct this ‘carry-select’-type adder using only \text{NAND}-gates. (To start, explain how to construct a \text{NOT}-gate using only \text{NAND}-gates.)

2. Considering the motivating issues — circuit-delay, propagation-delay, fan-in, fan-out — this ‘carry-select’-type scheme for the adder actually has a significant flaw; what is this flaw?