CS201 W'00 Lecture Notes
Lecture 6

Faster Addition

Here are a (slightly different) set of equations for a one-bit full adder in bit position i:

$$C_{i+1} = B_iC_i + A_iC_i + A_iB_i$$

$$S_i = A_i\overline{B_i}\overline{C_i} + \overline{A_i}B_i\overline{C_i} + \overline{A_i}\overline{B_i}C_i + A_iB_iC_i$$

If we form a **ripple-carry** adder for n-bit numbers, by connecting the carries together in the obvious way, the worst-case **delay** is $2n$ gate delays.

We can do better using **"Infinite" Hardware**

If we view n-bit adder as a combinational circuit (with $2n$ inputs and $n+1$ outputs), we know we can find a two-level sum-of-products representation with **constant delay** of two gates.

Idea is to expand out the definitions of the C_i. For example,

$$C_2 = B_1C_1 + A_1C_1 + A_1B_1$$

But similarly

$$C_1 = B_0C_0 + A_0C_0 + A_0B_0$$

Substituting for C_1 in the first equation, we get

$$\begin{aligned} C_2 \;=\; & A_1A_0B_0 + A_1A_0C_0 + A_1B_0C_0 + \\ & B_1A_0B_0 + B_1A_0C_0 + B_1B_0C_0 + A_1B_1 \end{aligned}$$

We can keep doing this all the way up to C_n. Unfortunately the **size** of the corresponding hardware grows very fast with n.

Carry Lookahead Adders

What about a compromise: something faster than ripple-carry but not using as much hardware?

First, let's refactor our equations:

$$\begin{aligned} C_{i+1} &= B_iC_i + A_iC_i + A_iB_i \\ &= A_iB_i + (A_i + B_i)C_i \end{aligned}$$

Now if we expand C_2, we see some repeated patterns:

$$C_2 = A_1B_1 + (A_1 + B_1)(A_0B_0 + (A_0 + B_0)C_0)$$

Let's define

generate $G_i = A_iB_i$
propagate $P_i = A_i + B_i$

Using them, see

$$C_{i+1} = G_i + P_iC_i$$

$$\begin{aligned} C_1 &= G_0 + P_0C_0 \\ C_2 &= G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\ C_4 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}$$

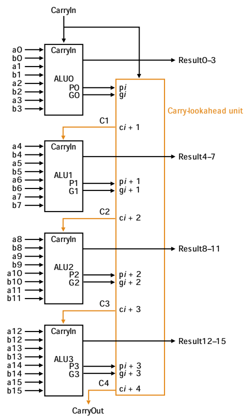
Delay has gone up by just one gate level; hardware is smaller but still grows quickly with n.

Two-Level Carry Lookahead

To keep hardware size down, we go to a second level of abstraction: build an n-bit adder using p n/p-bit adders.

Suppose we build a four-bit adder using the generate-propagate scheme. Then we can build a 16-bit adder by combining four four-bit adders.

Key idea: it is possible to compute overall generate and propagate for each second-level adder **in parallel**, without waiting on the carry bits of the earlier adders.



Details

Each four-bit adder generates 4 result bits and overall generate and propagate bits, which **don't** depend on the input carry.

$$
\begin{aligned}
P_n^4 &= P_{4n+3}P_{4n+2}P_{4n+1}P_{4n} \\
\\
P_0^4 &= P_3P_2P_1P_0 \\
P_1^4 &= P_7P_6P_5P_4 \\
P_2^4 &= P_{11}P_{10}P_9P_8 \\
P_3^4 &= P_{15}P_{14}P_{13}P_{12}
\end{aligned}
$$

$$
\begin{aligned}
G_n^4 &= G_{4n+3} + P_{4n+3}G_{4n+2} + P_{4n+3}P_{4n+2}G_{4n+1} \\
&\quad + P_{4n+3}P_{4n+2}P_{4n+1}G_4n \\
G_0^4 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 \\
G_1^4 &= G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 \\
G_2^4 &= G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 \\
G_3^4 &= G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12}
\end{aligned}
$$

The carry lookahead unit calculates these overall carry bits:

$$
\begin{aligned}
C_1^4 &= G_0^4 + P_0^4C_0 \\
C_2^4 &= G_1^4 + P_1^4G_0^4 + P_1^4P_0^4C_0 \\
C_3^4 &= G_2^4 + P_2^4G_1^4 + P_2^4P_1^4G_0^4 + P_2^4P_1^4P_0^4C_0 \\
C_4^4 &= G_3^4 + P_3^4G_2^4 + P_3^4P_2^4G_1^4 + P_3^4P_2^4P_1^4G_0^4 + P_3^4P_2^4P_1^4P_0^4C_0
\end{aligned}
$$

Carry Lookahead Example

Example: 0001 1010 0011 0011 + 1110 0101 1110 1011

A_i	0001	1010	0011	0011
B_i	1110	0101	1110	1011
G_i	0000	0000	0010	0011
P_i	1111	1111	1111	1011
C_i	1111	1111	1110	0011
G_i^4	0	0	1	0
P_i^4	1	1	1	0
C_i^4	1	1	1	0

and $C_{15} = C_4^4 = 1$.

Multiplication

Basic idea is to emulate manual multiplication, e.g.:

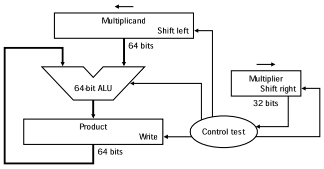
```
Multiplicand                  1000
Multiplier                  X 1001
                              ----
                              1000
                             0000
                            0000
                           1000
                           -------
Product                    1001000
```

Same in base two as in base ten! But in base two, note that formation of partial product is particularly simple:

- If multiplier bit is 0, do nothing.
- If multiplier bit is 1, just add in copy of multiplicand.

Product of (unsigned) n-bit and m-bit numbers may produce $n+m$-bit result.

Basic Circuit



Algorithm:

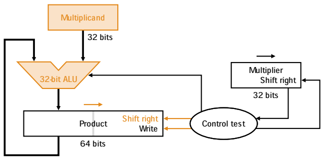
```
product = 0;
for (bit = 0; bit < 32; bit++) {
  if (multiplier & 1 == 1)
    product = product + multiplicand;
  multiplicand = multiplicand << 1;
  multiplier = multiplier >> 1;
}
```

Example:

0010 x 0011 =

Better Hardware

Observe that only half the bits in multiplicand are non-0, and so half positions in ALU are wasted. Cute trick: instead of shifting multiplicand left, shift product right!



Algorithm:

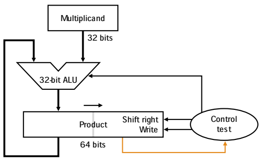
```
product = 0;
for (bit = 0; bit < 32; bit++) {
  if (multiplier & 1 == 1)
    product = (((product >> 32) + multiplicand) << 3
               | (product & 0xffffffff);
  product = product >> 1;
  multiplier = multiplier >> 1;
}
```


Better still:

The product register begins with 32 bits of wasted space but makes use of one more bit on each step.

The multiplier throws away one bit on each step and ends with 32 bits of wasted space.

So we can combine them:



Algorithm:

```
product = multiplier;
for (bit = 0; bit < 32;  bit++) {
  if (product & 1 == 1)
    product = (((product >> 32) + multiplicand) << 3
               | (product & 0xffffffff);
  product = product >> 1;
}
```


Signed Multiplication

Easiest approach is to multiply 31-bit magnitudes (ignoring sign) and then set the sign correctly when we're done:

product < 0 iff (multiplicand < 0) xor (multiplier < 0)

If we're working with signed numbers anyhow, we can use **Booth's Algorithm**, which uses both addition and subtraction steps, but (in a good case) fewer of them than in the standard algorithm.

It relies on the following trick:

Look at the binary representation of a number, e.g., $6 = 0110$.

Can get value by adding 2^i for each bit position i containing a one (numbering bit positions from LSB=0), e.g.,$2^1 + 2^2 = 6$.

Alternatively, look for the runs of consecutive 1's. (Assume an invisible 0 to the right and sign extension to the left of the word.) For each run beginning in bit position i and ending in bit position j, $i \leq j$, subtract 2^i and add 2^{j+1}.

Eg., $-2^1 + 2^3 = 6$.

(Don't forget to use arithmetic shift!)

Division

Dividend = Quotient $\times$ Divisor + Remainder

Again, hardware algorithm is based on ordinary grade-school long division; won't give details

As with multiplication, can use a single 64-bit register to hold 64-bit dividend to begin with and 32-bit quotient and 32-bit remainder to end with.

For signed division, must make sure that dividend and remainder always have the **same sign**.

For example, $-7/+2$ gives a quotient of -3 and a remainder of -1 (*not* a quotient of -4 and a remainder of +1).

Note that performing an (arithmetic) right shift does *not* produce the correct quotient for negative dividends that aren't exact powers of two. Example: -5 = 1...1011 but if try to divide by 4 by rightshifting by 2, we get 1...10 = -2 rather than -1.

SPARC multiplication and division use a special register %y to hold the upper half of the 64-bit register.

Floating Point

Represent numbers in scientific notation, but using base 2 instead of 10.

$1.xxxxxxxxx_2 \times 2^{yyyy}$

Here $xxxxxxxxx$ is the **significand** (or **mantissa**) and $yyyy$ is the **exponent**.

A **normalized** number is one that has no leading 0's.

To fit floating point numbers into fixed-sized words requires a tradeoff between size of significand (= accuracy) and size of exponent (= range).

IEEE 754 is a nearly universal standard for FP representation.

IEEE 754 Single precision:

```
|31|30 29 28 27 26 25 24 23|22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1|
|s | exponent (8 bits)     |                  significand (23 bits)                 |
```

Range is $10^{+/-38}$, precision is about 9 decimal digits.

IEEE 754 Double precision:

```
|31|30 29 28 27 26 25 24 23 22 21 20|19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1|
|s | exponent (11 bits)             |         significand (20 bits)                 |

|31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1|
|                                significand (continued -- 32 bits)                 |
```

Range is $10^{+/-308}$, precision is about 17 decimal digits.

IEEE 754 Encoding details

Sign is kept separately in leading bit.

Numbers are assumed to be normalized, so leading "1." is omitted from significand.

The value 0 is indicated by a special exponent value of 0. (Also, certain special values like "NaN" (Not-a-Number) are indicated by a special exponent value of 255.)

To preserve integer-like sorting properties, exponents are stored in **biased** notation, so that the most negative exponent is $00...00_2$ and the most positive is $11...11_2$. The bias is 127 for single precision and 1023 for double precision.

So full story is

$(-1)^{\text{sign}} \times (1 + \text{significand}) \times 2^{(\text{exponent-bias})}$

Example: $-0.75_{10} = -3/2^2_{10} = -11_2/2^2_{10} = -0.11_2 = -0.11_2 \times 2^0 = -1.1_2 \times 2^{-1}$

So sign bit is 1, significand is $1.1 - 1 = 0.1$, SP biased exponent is 126, and DP biased exponent is 1022. SP and DP bit patterns are:

```
1 01111110 10000000000000000000000

1 01111111110 10000000000000000000
00000000000000000000000000000000
```


Accuracy

Not all real numbers can be exactly represented by a floating-point number.

(This is obviously the case, since there are a countable number of floating point numbers, and an infinite number of reals!)

For example, an irrational number like π lacks a finite representation.

Less obviously, some numbers with very simple **decimal** representations, e.g., 0.1_{10}, lack a finite **binary** represention.

In addition, numbers with very large or small magnitudes may need exponents beyond the available range. Calculations producing an overlarge positive exponent are said to **overflow**; those producing an overlarge negative exponent **underflow**.

Algorithm for FP Addition

1. Compare exponents. Shift the smaller number right until its exponent matches that of the larger number, and round as necessary.

2. Add significands.

3. Normalize sum, either shifting right or left, and adjusting exponent accordingly.

3a. Check for underflow or overflow of exponent.

4. Round significand to available number of bits.

4a. If no longer normalized, go back to step 3.

Example (Base 10): $9.999_{10} \times 10^1 + 1.610_{10} \times 10^{-1}$

Assume significand is only 4 decimal digits.

1. Align second summand: $0.01610_{10} \times 10^1$, which rounds to $0.016_{10} \times 10^1$

2. Add significands to get 10.015_{10}. So sum is $10.015_{10} \times 10^1$.

3. Normalize to get $1.0015_{10} \times 10^2$.

4. Round (using "even" rule) to $1.002_{10} \times 10^2$.

Algorithm for FP Multiplication

1. Add biased exponents, subtracting bias from sum to get new biased exponent.

2. Multiply significands.

3. Normalize if necessary.

3a. Check for overflow.

4. Round to available number of bits.

4a. If no longer normalized, go back to step 3.

5. Set sign of product based on signs of operands.

Example: $-1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$

1. Add the exponents to get: 10 + (-5) = 5.

If exponents are biased, we need to subtract the bias when we're done, e.g.,

$(10 + 127) + (-5 + 127) = 259; 259 - 127 = (5 + 127)$

2. Multiply significands to get 10.212000_{10} and round; product is $10.212_{10} \times 10^5$.

3. Normalize to $1.0212_{10} \times 10^6$.

4. Round to $1.021_{10} \times 10^6$.

5. Set sign to product of operand signs: answer is $-1.021_{10} \times 10^6$.

Intermediate values and Rounding Policies

We were rather vague about how much precision we carried during internal calculations (like the multiply step 2).

In fact, IEEE 754 insists that there are at least 2 extra bits kept on the right during all internal calculations. This guarantees that the result will never be more than 50% wrong in the last (stored) unit.