Another Option

• Is the piece-wise faster adder as fast as we can go?
  • No!

• Another approach to using additional resources
  • Instead of redundantly computing sums assuming different carries, use redundancy to compute carries more quickly
  • This approach is called **carry lookahead addition (CLA)**
Review: Carry Lookahead Addition (CLA)

• Let’s look at the carry function
  • \( C_{16} = CO_{15} = A_{15}B_{15} + A_{15}C_{15} + B_{15}C_{15} = (A_{15}B_{15}) + (A_{15} + B_{15})C_{15} \)

• Very important insights into CLA:
  • \((A_{15}B_{15}) \) generates a carry regardless of \( C_{15} \) → rename to \( g_{15} \)
  • \((A_{15} + B_{15}) \) propagates \( C_{15} \) → rename to \( p_{15} \)

• \( C_{16} = g_{15} + p_{15}C_{15} \)
• \( C_{16} = g_{15} + p_{15}(g_{14} + p_{14}C_{14}) \)
• \( C_{16} = g_{15} + p_{15}g_{14} + p_{15}p_{14}(g_{13} + p_{13}C_{13}) \)
• \( C_{16} = g_{15} + p_{15}g_{14} + \ldots + p_{15}p_{14}\ldots p_{2}p_{1}g_{0} + p_{15}p_{14}\ldots p_{2}p_{1}p_{0}p_{0} \)
  • Important note: can compute \( C_{16} \) in 2 levels of logic!
• Similar functions for \( C_{15} (=CO_{14}) \), etc.
  • In general: \( C_{i} = g_{i-1} + p_{i-1}C_{i-1} \)
Infinite Carry Lookahead

- Previous slide’s CLA functions assume “infinite” hardware
  - Performance? Critical path is $d(S_{15}) = ?$
    - $d(p_{14}, g_{14}) + d(c_{15} \text{ given } p_{14}, g_{14}) + d(S_{15} \text{ given } c_{15}) = 1 + 2 + 2 = 5$ !
  - Constant delay, i.e., not a function of $N$
  - But not very practical in terms of hardware
  - Assume $2N$ gates to compute $p_i$ and $g_i$ initially (ICQ: why $2N$?)
  - Computation of a single $C_N$ needs the following hardware:
    - $N$ AND gates + 1 OR gate, and largest gates have $N+1$ inputs
  - Computation of all $C_N \ldots C_1$ needs:
    - $N*(N+1)/2$ AND gates + $N$ OR gates, max $N+1$ inputs
  - Not too bad if $N=16$: 152 gates, max input 17
  - Pretty bad if $N=64$: 2144 gates, max input 65
    - Big circuits are slow and high input gates are slow
Motivation for Multi-Level Carry Lookahead

- Let’s look at what we have so far (the two extremes)
  - **Ripple carry**
    + Few small gates: no additional gates used to speed up addition
    - Laid in series: 2N latency
  - **Infinite CLA**
    - Many big gates: N*(N+3)/2 additional gates, max N+1 inputs
    + Laid in parallel: constant latency of 5 gate delays
  - We’d like something in between
    - Reasonable number of small gates
    - Sublinear (doesn’t have to be constant) latency
  - **Multi-level CLA**
    - Exploits hierarchy to achieve good compromise between the two extremes
Two-Level CLA for 4-bit Adder

- Individual carry equations
  - \( C_1 = g_0 + p_0C_0, \) \( C_2 = g_1 + p_1C_1, \) \( C_3 = g_2 + p_2C_2, \) \( C_4 = g_3 + p_3C_3 \)

- Fully expanded (infinite hardware) CLA equations
  - \( C_1 = g_0 + p_0C_0 \)
  - \( C_2 = g_1 + p_1g_0 + p_1p_0C_0 \)
  - \( C_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0C_0 \)
  - \( C_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0C_0 \)

- Hierarchical CLA equations
  - **First level**: expand \( C_2 \) using \( C_1 \) and \( C_4 \) using \( C_3 \)
    - \( C_2 = g_1 + p_1(g_0 + p_0C_0) = (g_1 + p_1g_0) + (p_1p_0)C_0 = G_{1-0} + P_{1-0}C_0 \)
    - \( C_4 = g_3 + p_3(g_2 + p_2C_2) = (g_3 + p_3g_2) + (p_3p_2)C_2 = G_{3-2} + P_{3-2}C_2 \)
  - **Second level**: expand \( C_4 \) using expanded \( C_2 \)
    - \( C_4 = G_{3-2} + P_{3-2}(G_{1-0} + P_{1-0}C_0) = (G_{3-2} + P_{3-2}G_{1-0}) + (P_{3-2}P_{1-0})C_0 \)
    - \( C_4 = G_{3-0} + P_{3-0}C_0 \)
Two-Level (2L) CLA for 4-bit Adder

• Hardware?
  • First level: block is infinite CLA for N=2
    • 5 gates per block, max # gate inputs (MNGI)=3
    • 2 of these “blocks”
  • Second level: 1 of these “blocks”
  • Total: 15 gates & 3 MNGI
    • Infinite CLA: 14 & 5 (?!)

• Latency?
  • Total: 9 (ICQ: why?)
    • Infinite CLA: 5
  • 2L: bigger and slower??!
    • ICQ: what happened?
Two-Level CLA for 16-bit Adder

- 4 G/P inputs per level

- Hardware?
  - First level: 14&5 * 4 blocks
  - Second level: 14&5 * 1 block
  - Total: **70&5**
    - Infinite: 152&17

- Latency?
  - Total: **9 (1 + 2 + 2 + 2 + 2)**
  - Infinite: 5

- That’s more like it!
  - CLA for a 64-bit adder?
A Closer Look at CLA Delay

- CLA block has “individual” G/P inputs
  - Uses them to perform **two** calculations
  - Group G/P on way up tree
  - Group interior carries on way down tree
    - Given group carry-in from level above
- Group carry-in for outer level \((C_0)\) ready at 0
- Outer level G/P, interior carries in parallel
CLA Tree Signal Timing: d1

- Signals ready after 1 gate delay
  - $C_0$
  - Individual G/P
CLA Tree Signal Timing: d3

- What is ready after 3 gate delays?
  - First level group G/P
CLA Tree Signal Timing: d5

- And after 5 gate delays?
  - Outer level “interior” carries
    - $C_4$, $C_8$, $C_{12}$, $C_{16}$
CLA Tree Signal Timing: d7

• And after 7 gate delays?
  • First level “interior” carries
    • $C_1$, $C_2$, $C_3$
    • $C_5$, $C_6$, $C_7$
    • $C_9$, $C_{10}$, $C_{11}$
    • $C_{13}$, $C_{14}$, $C_{15}$
    • Essentially, all remaining carries

• $S_i$ ready 2 gate delays after $C_i$
  • All sum bits ready after 9 delays!
Subtraction: Addition’s Tricky Pal

- Sign/magnitude subtraction is mental reverse addition
  - Two’s complement subtraction is addition
- How to subtract using an adder?
  - \( \text{sub } A, B = \text{add } A, -B \)
  - Negate \( B \) before adding (fast negation trick: \(-B = B' + 1\))
- Isn’t a subtraction then a negation and two additions?
  + No, an adder can implement \( A+B+1 \) by setting the carry-in to 1
  + Clever, huh?
A 16-bit ALU

- Build an ALU with functions: add/sub, and, or, not, xor
  - All of these already in CLA adder/subtractor
  - add A B, sub A B (done already)
  - not B is needed for subtraction
  - and A, B are first level Gs
  - or A, B are first level Ps
  - xor A, B?
  - \( S_i = A_i \oplus B_i \oplus C_i \)
This Unit: Arithmetic and ALU Design

- Integer Arithmetic and ALU
  - Binary number representations
  - Addition and subtraction
  - The integer ALU
  - Shifting and rotating
  - Multiplication
  - Division

- Floating Point Arithmetic
  - Binary number representations
  - FP arithmetic
  - Accuracy
Shifts

- Shift: move all bits in a direction (left or right)
  - Denoted by `<<` (left shift) and `>>` (right shift) in C/C++/Java
- **ICQ**: Left shift example: 001010 `<<` 2 = ?
- **ICQ**: Right shift example: 001010 `>>` 2 = ?
- Shifts are useful for
  - Bit manipulation: extracting and setting individual bits in words
  - Multiplication and division by powers of 2
    - A * 4 = A `<<` 2
    - A / 8 = A `>>` 3
    - A * 5 = (A `<<` 2) + A
  - Compilers use this optimization, called **strength reduction**
    - Easier to shift than it is to multiply (in general)
Rotations

- Rotations are slightly different than shifts
  - 1101 rotated 2 to the right = ?
- Rotations are generally less useful than shifts
  - But their implementation is natural if a shifter is there
  - MIPS has only shifts
Barrel Shifter

- What about shifting left by any amount from 0 to 15?
  - Cycle input through "left-shift-by-1" up to 15 times?
    - Complicated, variable latency
  - 16 consecutive "left-shift-by-1-or-0" circuits?
    - Fixed latency, but would take too long
- **Barrel shifter**: four "shift-left-by-X-or-0" circuits ($X = 1,2,4,8$)
Right Shifts and Rotations

- Right shifts and rotations also have barrel implementations
  - But are a little different

- Right shifts
  - Can be **logical** (shift in 0s) or **arithmetic** (shift in copies of MSB)
    - $\text{srl} \ 110011,2 \rightarrow \text{result is} \ 001100$
    - $\text{sra} \ 110011,2 \rightarrow \text{result is} \ 111100$
  - Caveat: $\text{sra}$ is not equal to division by 2 of negative numbers
  - Why might we want both types of right shifts?

- Rotations
  - Mux in wires of upper/lower bits
Shift Registers

- **Shift register**: shift in place by constant quantity
  - Sometimes that’s a useful thing
Base10 Multiplication

• Remember base 10 multiplication from 3rd grade?

\[
\begin{array}{c}
43 \quad // \text{ multiplicand} \\
* 12 \quad // \text{ multiplier} \\
\hline
86 \\
+ 430 \\
\hline
516 \quad // \text{ product}
\end{array}
\]

• Start with running total 0, repeat steps until no multiplier digits
  • Multiply multiplicand by least significant multiplier digit
  • Add to total
  • Shift multiplicand one digit to the left (multiply by 10)
  • Shift multiplier one digit to the right (divide by 10)

• Product of N-digit and M-digit numbers potentially has N+M digits
Binary Multiplication

\[ 43 = 00000101011 \quad // \text{multiplicand} \]
\[ \times 12 = 00000001100 \quad // \text{multiplier} \]
\[ 0 = 00000000000 \]
\[ 0 = 00000000000 \]
\[ 172 = 00010101100 \]
\[ + 344 = 00101011000 \]
\[ 516 = 01000000100 \quad // \text{product} \]

• Same thing except ...
  – There are more individual steps (smaller base)
  + But each step is simpler
• Multiply multiplicand by least significant multiplier bit
  • 0 or 1 \( \rightarrow \) no actual multiplication, just add multiplicand or not
• Add to total: we know how to do that
• Shift multiplicand left, multiplier right by one bit: shift registers
Simple 16x16=32bit Multiplier Circuit

- **Control algorithm**: repeat 16 times
  - If LSB(multiplier) == 1, then add multiplicand to product
  - Shift multiplicand left by 1
  - Shift multiplier right by 1

4b example: 0101 x 0110
Inefficiencies with Simple Circuit

- Notice
  - 32-bit addition, but 16 multiplicand bits are always 0
  - And 0-bits are always moving
  - Solution? Instead of shifting multiplicand left, shift product right
Better 16-bit Multiplier

- **Control algorithm**: repeat 16 times
  - LSB(multiplier) == 1 ? Add multiplicand to upper half of product
  - Shift multiplier right by 1
  - Shift product right by 1

4b example: 0101 x 0110
Another Inefficiency

- Notice one more inefficiency
  - What is initially the lower half of product gets thrown out
    - As useless lower half of product is shifted right, so is multiplier
  - Solution: use lower half of product as multiplier
Even Better 16-bit Multiplier

- **Control algorithm**: repeat 16 times
  - LSB(multiplier) == 1 ? Add multiplicand to upper half of product
  - Shift product right by 1

4b example: 0101 x 0110
Multiplying Negative Numbers

- If multiplicand is negative, our algorithm still works
  - As long as right shifts are arithmetic and not logical
  - Try 1111*0101
- If multiplier is negative, the algorithm breaks

- Two solutions
  1) Negate multiplier, then negate product
  2) Booth’s algorithm