Two-Level CLA for 4-bit Adder

- Individual carry equations
  - $C_i = g_i + p_i C_{i-1}$
- Fully expanded (infinite hardware) CLA equations
  - $C_1 = g_0 + p_0 C_0$
  - $C_2 = g_1 + p_1 g_0 + p_1 p_0 C_0$
  - $C_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$
  - $C_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$
- Hierarchical CLA equations
  - **First level**: expand $C_2$ using $C_1$ and $C_4$ using $C_3$
    - $C_2 = g_1 + p_1 (g_0 + p_0 C_0) = (g_1 + p_1 g_0) + (p_1 p_0) C_0 = G_1-0 + P_1-0 C_0$
    - $C_4 = g_3 + p_3 (g_2 + p_2 C_2) = (g_3 + p_3 g_2) + (p_3 p_2) C_2 = G_3-2 + P_3-2 C_2$
  - **Second level**: expand $C_4$ using expanded $C_2$
    - $C_4 = G_3-2 + P_3-2 (G_1-0 + P_1-0 C_0) = (G_3-2 + P_3-2 G_1-0) + (P_3-2 P_1-0) C_0 = G_3-0 + P_3-0 C_0$

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: $14$ & $5$ (?!)

Latency?
- 2 for “outer” CLA, 4 for “interior” B
  - G/P go “up”, C go “down”
  - Total: $9$ (1 for GP, 2 for S)
  - Infinite: $5$

- 2L: bigger and slower?!?

Two-Level CLA for 16-bit Adder

- 4 G/P inputs per level
- Hardware?
  - First level: 1485 * 4 blocks
  - Second level: 1485 * 1 block
  - Total: $7085$
  - Infinite: 152817
- 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: $d_1$

- Signals ready after 1 gate delay
  - $C_0$
  - Individual G/P

CLA Tree Signal Timing: $d_3$

- What is ready after 3 gate delays?
  - First level group G/P

CLA Tree Signal Timing: $d_5$

- And after 5 gate delays?
  - Outer level "interior" carries
    - $C_0$, $C_4$, $C_{12}$, $C_{16}$

CLA Tree Signal Timing: $d_7$

- And after 7 gate delays?
  - First level "interior" carries
    - $C_0$, $C_4$, $C_1$
    - $C_5$, $C_{10}$, $C_7$
    - $C_{15}$, $C_{16}$, $C_{17}$
    - 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**

- [material only available in class]

**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 ^ B_i ^ C_i`

**Shifts**

- Shift: move all bits in a direction (left or right)
  - Denoted by `<<` (left shift) and `>>` (right shift) in C/C++/Java
  - Left shift example: `001010 << 2 = ?`
  - 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`
    - This compiler optimization is called **strength reduction**
      - Easier to shift than 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

A Simple (Left) Shifter

- The simplest 16-bit shifter: can only shift left by 1
  - Implement using wires
- Slightly more complicated: can shift left by 1 or 0
  - Implement using wires and a multiplexer (mux16_2to1)

Barrel Shifter

- [material only available in class]

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)
    - srl 110011, 2 = 001100
    - sra 110011, 2 = 111100
  - Caveat: 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

![Shift Register Diagram](image)

Decimal Multiplication

- Remember decimal multiplication from 3rd grade?
  
  ```
  43 // multiplicand
  * 12 // multiplier
  + 430
  516 // product
  ```

  - 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` // multiplicand
- `* 12 = 00000001100` // multiplier
- `0 = 00000000000`
- `0 = 00000000000`
- `172 = 00010101100`
- `+ 344 = 00101011000`
- `516 = 01000000100` // product

- Same thing except
  - There are more individual steps (smaller base)
  - But each step is simpler
  - Multiply multiplicand by least significant multiplier digit
    - 0 or 1 → no actual multiplication, just add multiplicand or not
  - Add to total: we know how to do that
  - Shift multiplicand left, multiplier right by one digit: **shift registers**

Simple 16-bit 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
**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

**Another Inefficiency**
- [material only available in class]

**Even Better 16-bit Multiplier**
- [material only available in class]
Multiplying Negative Numbers

- Just works...
  - As long as right shifts are arithmetic and not logical
  - Try it out for yourself now
- 0101 * 0111 = ?