Subtraction: Addition’s Tricky Pal

- Sign/magnitude subtraction is mental reverse addition
  - Two's complement subtraction is addition
- How to subtract using an adder?
  - sub A, B = 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 = A\^B\^C

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)
  - Example: `srl 110011, 2 → result is 001100`
  - Example: `sra 110011, 2 → result is 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

Base10 Multiplication

- Remember base 10 multiplication from 3rd grade?
  - 43 // multiplicand
  - * 12 // multiplier
  - 86
  - + 430 // 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 = 0000001100 // 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 bit
      - 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 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: 0110 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: 0110 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

Booth’s Algorithm

- Notice the following equality (Booth did)
  - \(2^1 + 2^3 + 2^5 + \ldots + 2^k = 2^{k+1} - 2^0\)
  - Example: 1111 = 1000 - 0001
  - We can exploit this to create a faster multiplier

- How?
  - Sequence of N 1s in the multiplier yields sequence of N additions
  - Replace with one addition and one subtraction

Booth In Action

- For each multiplier bit, also examine bit to its right
  - **00:** middle of a run of 0s, do nothing
  - **10:** beginning of a run of 1s, subtract multiplicand
  - **11:** middle of a run of 1s, do nothing
  - **01:** end of a run of 1s, add multiplicand

\[
\begin{align*}
43 &= 000001011011_2 \\
12 &= 000000001110_2 \\
0 &= 000000000000_2 \\
+ &= 000000000000_2 \\
-172 &= 11101010100_2 \\
+ &= 000000000000_2 \\
688 &= 01010110000_2 \\
512 &= 010000000000_2 \\
\end{align*}
\]

ICQ: so why is Booth better?
Booth Hardware

- Control algorithm: repeat 16 times
  - Multiplier LSBs == 10? Subtract multiplicand from product
  - Multiplier LSBs == 01? Add multiplicand to product
  - Shift product/multiplier right by 1 (not by 2!)

Booth in Summary

- Performance/efficiency
  - Good for sequences of 3 or more 1s
    - Replaces 3 (or more) adds with 1 add and 1 subtract
  - Doesn’t matter for sequences of 2 1s
    - Replaces 2 adds with 1 add and 1 subtract (add = subtract)
    - Actually bad for singleton 1s
    - Replaces 1 add with 1 add and 1 subtract

- Bottom line
  - Worst case multiplier (101010) requires N/2 adds + N/2 subs
  - What is the worst case multiplier for straight multiplication?
  - How is this better than normal multiplication?

Modified Booth’s Algorithm

- What if we detect singleton 1s and do the right thing?

- Examine multiplier bits in groups of 2s plus a helper bit on the right (as opposed to 1 bit plus helper bit on right)
  - Means we’ll need to shift product/multiplier by 2 (not 1)
  - 000: middle of run of 0s, do nothing
  - 100: beginning of run of 1s, subtract multiplicand<<1 (M*2)
    - Why M*2 instead of M?
  - 010: singleton 1, add multiplicand
  - 110: beginning of run of 1s, subtract multiplicand
  - 001: end of run of 1s, add multiplicand
  - 101: end of run of 1s, beginning of another, subtract multiplicand
    - Why is this? –2^{J+1} + 2^{J} = –2^{J}
  - 011: end of a run of 1s, add multiplicand<<1 (M*2)
  - 111: middle of run of 1s, do nothing

Modified Booth Hardware

- Control algorithm: repeat 8 times (not 16!)
  - Based on 3b groups, add/subtract shifted/unshifted multiplicand
  - Shift product/multiplier right by 2

Modified Booth In Action

\[
\begin{align*}
43 &= 00000101011 \\
\times 12 &= 00000000010 \\
0 &= 00000000000 \\
-172 &= 11101011000 \\
+688 &= 01010110000 \\
516 &= 01000000100
\end{align*}
\]

Another Multiplier: Multiple Adders

- Can multiply by N bits at a time by using N adders
  - Doesn’t help: 4X fewer iterations, each one 4X longer (4^9=36)
Carry Save Addition (CSA)

- Carry save addition (CSA): \( d(N \text{ adds}) < N \times d(1 \text{ add}) \)
  - Enabling observation: unconventional view of full adder
    - 3 inputs \((A, B, C)\) → 2 outputs \((S, C_{out})\)
    - But what if we are adding three numbers \((A+B+D)\)?
  - One option: back-to-back conventional adders
    - Add \(A + B = \text{temp}\)
    - Add \(\text{temp} + D = \text{Sum}\)
  - Better option: instead of rippling carries in first addition \((A+B)\), feed the \(D\) bits in as the carry bits (treat \(D\) bits as \(C\) bits)
    - Assume \(A+B+D = \text{temp2}\)
    - Then do traditional addition (not CSA) of \(\text{temp2}\) and \(C\) bits generated during addition of \(A+B+D\)

- 2 conventional adders
  - \(g / d(16) = 9\)
  - \(d = 11\)
- \(k\) conventional adders
  - \(d = [k \times d(16)]\)
  - \(d = 9k\)
- CSA + conventional adder
  - \(d = [d(\text{CSA}) + d(\text{add})]\)
  - \(d = 2k + 9\)

Carry Save Multiplier

- 4-bit at a time multiplier using 3 CSA + 1 normal adder
  - Actually helps: 4X fewer iterations, each only \((2+2+2+9=15)\)

Wallace Tree Multiplier (based on CSA)