**The Arithmetic Unit**

* Recall:
  + Full adder (FA)
    - Adds 2 bits & carry-in bit → produces sum bit & carry-out bit
  + Ripple-carry adder
    - Chain of FA’s linked by carry bits
    - Adding 2 n-bit numbers → n FA’s
  + Take 2’s complement of 2nd operand to perform subtraction with an adder
    - i.e. XOR bits & +1
* **Unsigned multiplication**
* Note: **product = multiplicand x multiplier**
* Array multiplier
  + Each cell = FA + AND gate
  + Cell input:
    - Bit of multiplicand
    - Bit of multiplier
    - Bit of incoming partial product
    - Carry-in bit
  + Cell output:
    - Bit of outgoing partial product
    - Carry-out bit
  + Row input:
    - Bit of multiplier
    - Multiplicand
    - Incoming PP (PP0 = all 0’s)
  + Row output:
    - Outgoing PP = incoming PP + multiplicand (if multiplier bit = 1)
* Sequential multiplier
  + Uses a single n-bit adder
  + M holds multiplicand
  + Q holds multiplier
  + Flip-flop C holds carry-out bit
  + C, A & Q (concatenated) hold partial product
  + A begins with all 0’s
  + Every cycle:
    - If LSB of Q (multiplier bit) = 1, A ← [M] + [A] (to produce next partial product)
    - Shift CAQ right one bit – eliminates previous multiplier bit, LSB of Q becomes next multiplier bit
  + AQ holds product
* **Signed multiplication**
* Negative multiplicand \* positive multiplier
  + Every time multiplicand is added to the partial product, its sign bit must be extended to the MSB of the product (sign extension)
* Positive multiplicand \* negative multiplier
  + Take 2’s-complement of both operands, then treat as previous case
* Booth’s Algorithm
  + Given signed 2’s-complement n-bit operands, generates 2n-bit product
  + The # of operations can be reduced by recoding the multiplier as a difference b/t 2 #’s
    - E.g. (30)10 = (0011110)2
    - → (32)10 – (2)10 = (0100000)2 – (0000010)2
    - The multiplier can be represented as 0 +1 0 0 0 -1 0 (Booth recoding)
  + If the recoded multiplier has fewer 1’s than the original, the # of operations is reduced
    - Worst case – # of operations x2
    - Best case − # of operations = 1
    - Advantageous when multiplier contains blocks of 1’s
  + When scanning the multiplier:
    - If bit n & bit n-1 = 01 → add multiplicand at position n
      * i.e. recoded bit = +1
    - If bit n & bit n-1 = 10 → add negative of multiplicand at position n
      * i.e. recoded bit = -1
    - If bit n = bit n-1 → add 0
      * i.e. recoded bit = 0
    - E.g. 0 0 1 1 1 1 0 → 0 +1 0 0 0 -1 0
      * Add –M at b1; add +M at b5
* Bit-pair recoding
  + Reduces the max # of operations by half
  + Consider the pair of bits (+1 -1) in Booth’s Algorithm:
    - Adding +M at position n & -M at position n-1 ⇔ adding +M at position n-1
    - i.e. (+1 -1) is equivalent to (0 +1)
  + E.g.
    - 1 1 1 0 1 0 ← original
    - 0 0 -1 +1 -1 0 ← Booth recoding
    - 0 -1 -2 ← bit-pair recoding
      * (0 0) ↔ 0
      * (-1 +1) ↔ -1
      * (-1 0) ↔ -2
  + Note
    - x 2 → left shift 1 bit
    - x -2 → take 2’s-complement, then left shit 1 bit
* **Division**
* Note: **quotient = dividend / divisor**
* Restoring division algorithm
  + Similar to sequential multiplier
  + Uses a (n+1)-bit adder
  + M holds divisor
  + Q holds dividend
  + A begins with all 0’s
  + Every cycle:
    - Shift AQ left one bit
    - A ← [A] – [M]
    - If sign of A is 1 (i.e. A < 0) ⇒ set LSB of Q = 0 and A ← [A] + [M] (i.e. restore A)
    - If sign of A is 0 (i.e. A > 0) ⇒ set LSB of Q = 1
  + Q contains quotient; A contains remainder
  + Sign of quotient must be determined (separately) by comparing the sign bits of dividend & divisor
* Non-restoring division algorithm
  + Every cycle:
    - Shift AQ left one bit
    - If sign of A is 0 (i.e. A > 0) ⇒ A ← [A] – [M]
    - If sign of A is 1 (i.e. A < 0) ⇒ A ← [A] + [M]
    - Set LSB of Q = !(sign bit of A)
  + Then:
    - If sign of a is 1 ⇒ A ← [A] + [M] (restore remainder)
* **Representing real numbers**
* Fixed-point representation
  + Integer & binary fraction separated by a fixed decimal point
  + Has fixed precision
  + Useful for small range of values with high precision
  + For a n-bit binary fraction:
    - Equivalent fixed-point number has the range -1 ≤ F ≤ 1 – 2-(n-1)
    - B = b0b-1b-2…b-(n-1)
    - Then fixed-point number **F = b0 \* 20 + b-1 \* 2-1 + … b-(n-1) \* 2-(n-1)**
  + E.g. 0000 0010 . 110

= 21 + 2-1 + 2-2 = 2.75

* Floating-point representation
  + Integer & binary fraction separated by a variable decimal point
  + Has variable precision
  + Useful for large range of values with low precision
  + Floating-point number **F = M \* βE**
    - M = Mantissa (signed fixed-point quantity)
    - E = exponent (signed integer quantity)
    - β = base (2)
* IEEE single-precision floating-point standard (32-bit)
  + 1 S bit – 8 E’ bits – 23 M bits
  + **F = (-1)S \* (1.M) \* 2E’ – 127**
  + E’ = E + 127 (excess-127 encoding)
    - Allows for signed representation of the exponent without sign bit
    - E.g. 2 is encoded as 129; -2 is encoded as 125
  + E.g. 0xC1280000
    - S = 1
    - E’ = (1000 0010)2 = (130)10
    - M = (010 1000 0000 0000 0000 0000)2 = (0.3125)10
    - F = (-1)1 \* 1.3125 \* 2130 – 127 = -10.5
* IEEE double-precision floating-point standard (64-bit)
  + 1 S bit – 11 E’ bits – 52 M bits
  + **F = (-1)S \* (1.M) \* 2E’ – 1023**
  + E’ = E + 1023 (excess-1023 encoding)
* Normalized number – scale the number using exponent so that M has MSB = 1
* Special values of IEEE standards
  + E’ = 0 & M = 0 → F = 0 (S determines ±)
  + E’ = 255 & M = 0 → F = ∞ (S determines ±)
  + E’ = 0 & M ≠ 0 → F = denormalized value
    - Denormalized number
      * F = ± 0.M \* 2-126 for single-precision
      * F = ± 0.M \* 2-1022 for double-precision
      * Allows for gradual underflow – provides additional precision for very small values
  + E’ = 255 & M ≠ 0 → F = NaN
    - Not-a-number – result of invalid operation, e.g. division by 0, sqrt(-1)
* Floating point addition/subtraction
  + Let F1, F2 be numbers so that E1 < E2; let F3 = result
  + Shift M1 right (E2 – E1) bits & set E3 = E2 (scale & match exponents)
  + M3 = M1 ± M2 and determine sign bit
  + Normalize result if necessary
* Floating-point multiplication
  + Let F3 = F1 \* F2
  + E3 = E1 + E2 – bias
  + M3 = M1 \* M2 and determine sign bit
  + Normalize result if necessary
* Floating-point division
  + Let F3 = F1/F2
  + E3 = E1 – E2 + bias
  + M3 = M1/M2 and determine sign bit
  + Normalize result if necessary
* Preserving accuracy in arithmetic operations
  + Extra guard bits are retained during calculation & normalization to preserve accuracy of a result