# *Computer Architecture: Fundamentals, Tradeoffs, Challenges*

# *Chapter 10: Fixed Point Arithmetic*

***Yale Patt***

***The University of Texas at Austin***

***Austin, Texas***

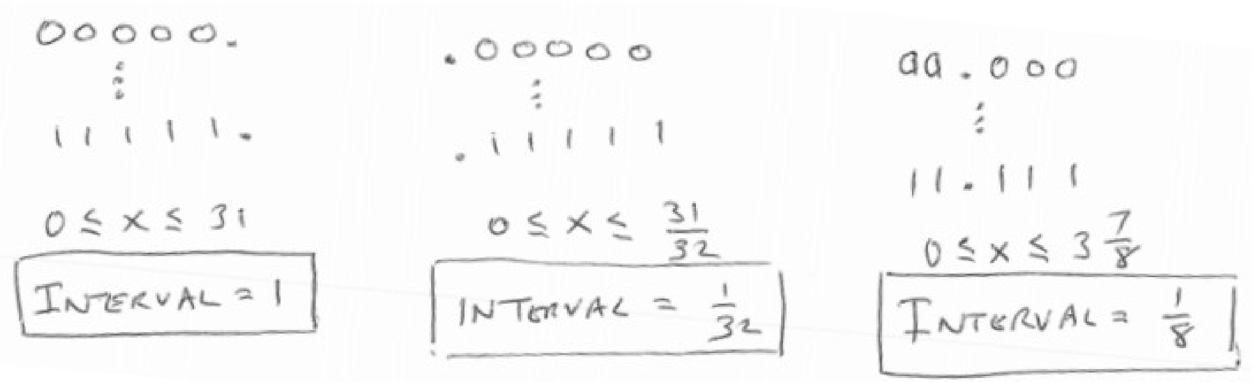
***Spring, 2023***

# *Outline*

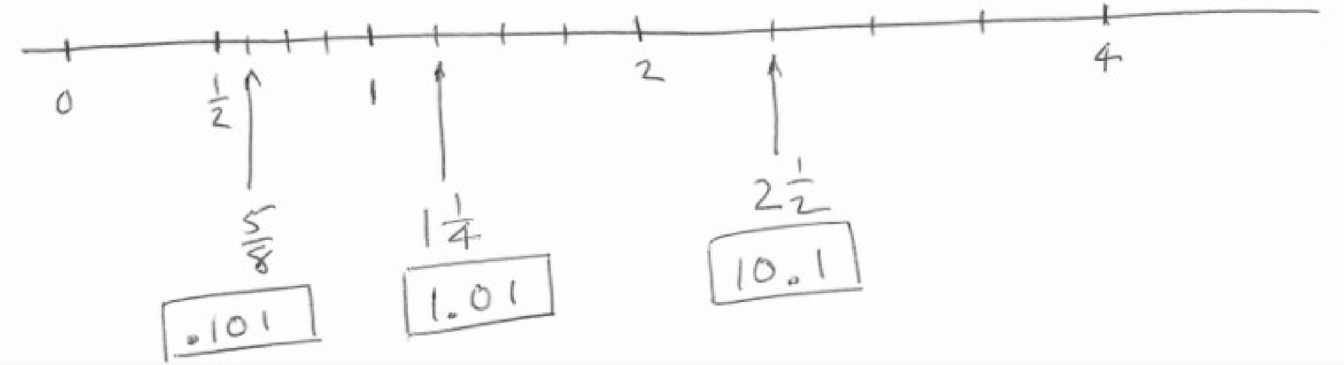
- ***The Binary Point (fixed point vs floating point)***
- ***Several Choices***
- ***2’s complement, 1’s complement, Sign-magnitude***
- ***Long Integers***
- ***Addition***
  - ***ripple carry, look ahead carry, Kogge Stone)***
  - ***Interesting anecdote: the P4 fireball***
- ***BCD Arithmetic***
- ***Multiplication***
  - ***Shift and Add, Booth’s Algorithm***
- ***Residue Arithmetic***

# *The Binary Point (fixed pt. vs. floating pt.)*
# *Where do we put the binary point?*

- ***Fixed  Point (one place, fixed for that design)***
  - ***Interval remains the same for the entire real line***




- ***Floating Point (varies from binade to binade)***
  - ***Interval changes along the real line***



# *Several choices*

- ***2's complement***
- ***1's complement***
- ***Signed magnitude***
- ***Long Integers***
  - ***When you wish to retain the structure of 2's complement***
  - ***But you need a lot more bits***
- ***BCD***
  - ***Arbitrarily large precision***
- ***Residue Numbers***
  - ***Compute intensive, low I/O  (But…)***

# *2's complement, 1's complement, Signed-magnitude*

- ***Why each?***
  - ***2's complement (Easy for the computer, representations track represented!)***
  - ***1's complement (Seymour Cray's misguided decision)***
  - ***Signed-magnitude (Easy for humans, bad for designing logic to implement)***
- ***Example (A 4-bit word length)***

| | 2's comp | 1's comp | Signed-mag |
|---|---|---|---|
| 0000 | 0 | 0 | 0 |
| 0001 | 1 | 1 | 1 |
| 0010 | 2 | 2 | 2 |
| 0011 | 3 | 3 | 3 |
| 0100 | 4 | 4 | 4 |
| 0101 | 5 | 5 | 5 |
| 0110 | 6 | 6 | 6 |
| 0111 | 7 | 7 | 7 |
| 1000 | -8 | -7 | 0 |
| 1001 | -7 | -6 | -1 |
| 1010 | -6 | -5 | -2 |
| 1011 | -5 | -4 | -3 |
| 1100 | -4 | -3 | -4 |
| 1101 | -3 | -2 | -5 |
| 1110 | -2 | -1 | -6 |
| 1111 | -1 | 0 | -7 |

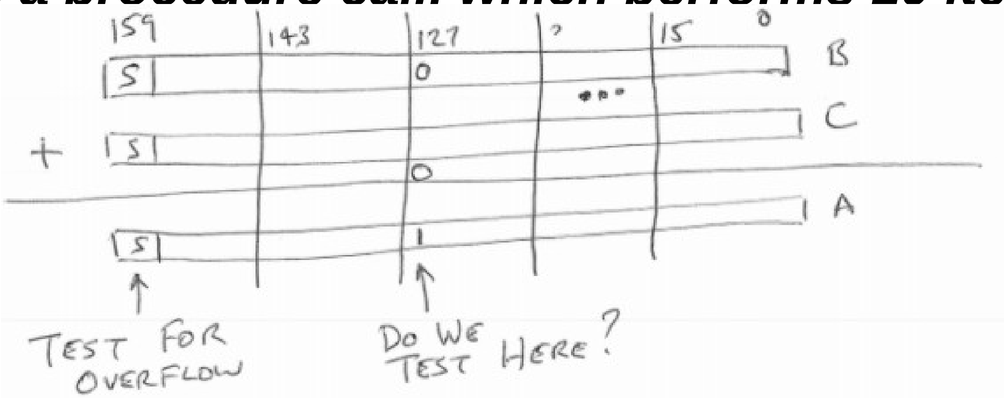
# *Observations*

- ***With 2's complement***
  - ***why can Carry bit go in the trash?***

- ***With 1's complement***
  - ***Is there a problem?***
  - ***how do we fix it***

# Long Integers

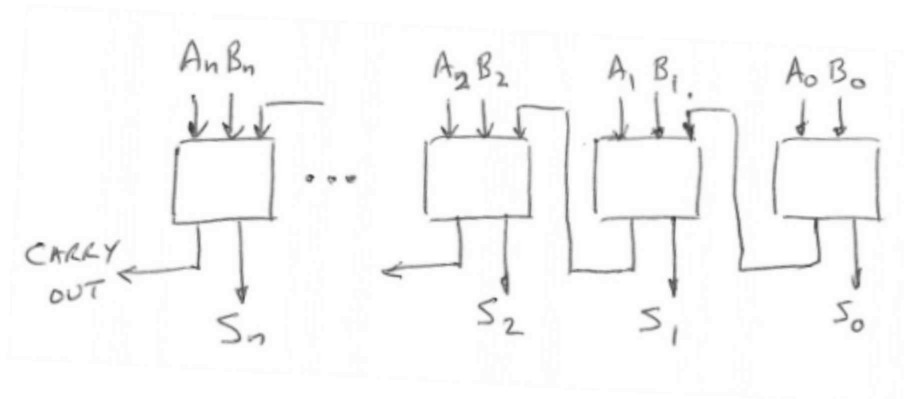
- When the number of bits in 2's complement is not enough
  - If word length is 16 bits, but you want 160 bit integer data type
- Then you need an instruction requiring that Data Type
  - ADDR, for example. (R for ridiculous!)
- Consider ADDR A,B,C, where A,B,C are 160 bit integers:
  - Requires a procedure call, which performs 10 iterations



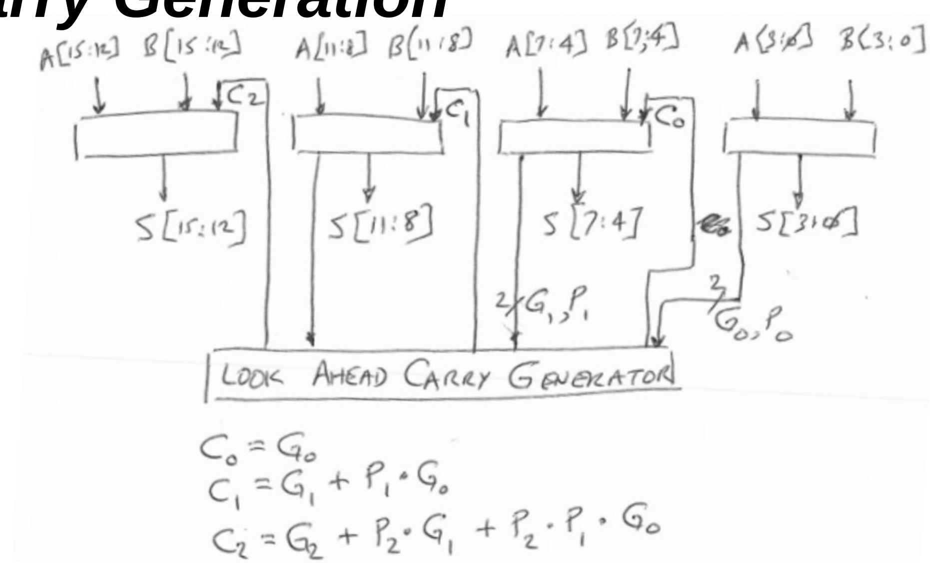

- Note: test for overflow only in the last iteration.
- ADDC (add with carry a very important opcode)

# Addition

- ***Ripple carry***

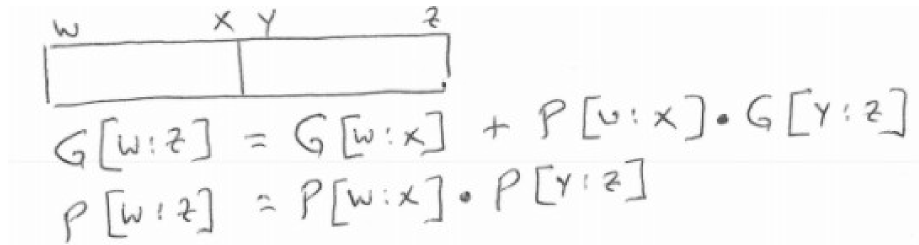



- ***Look ahead Carry Generation***

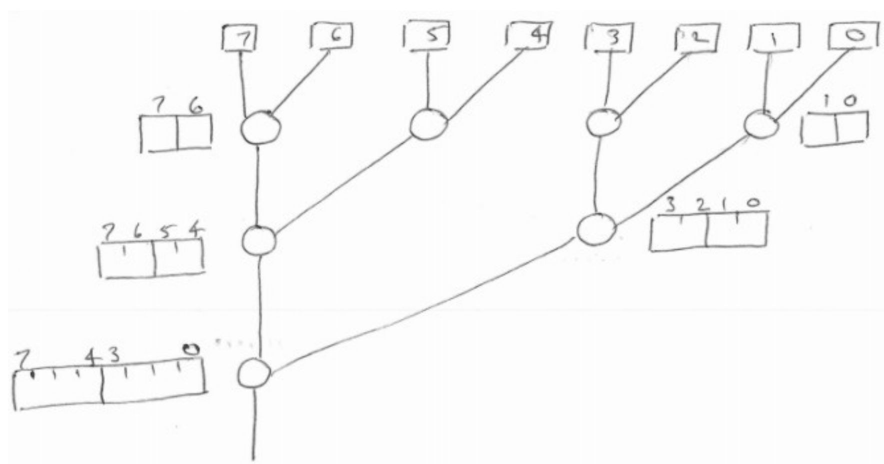


# *Addition (continued): The Kogge-Stone Adder*

- ***The needed values can be generated by a tree!***
  - ***Brilliant insight: reduces time from O(n) to O(log n).***
- ***The basic piece***




- ***The binary tree***



# *Addition (continued):Intel's P4 Fireball*

- ***The code to compute Z = A+B+C+D+E***

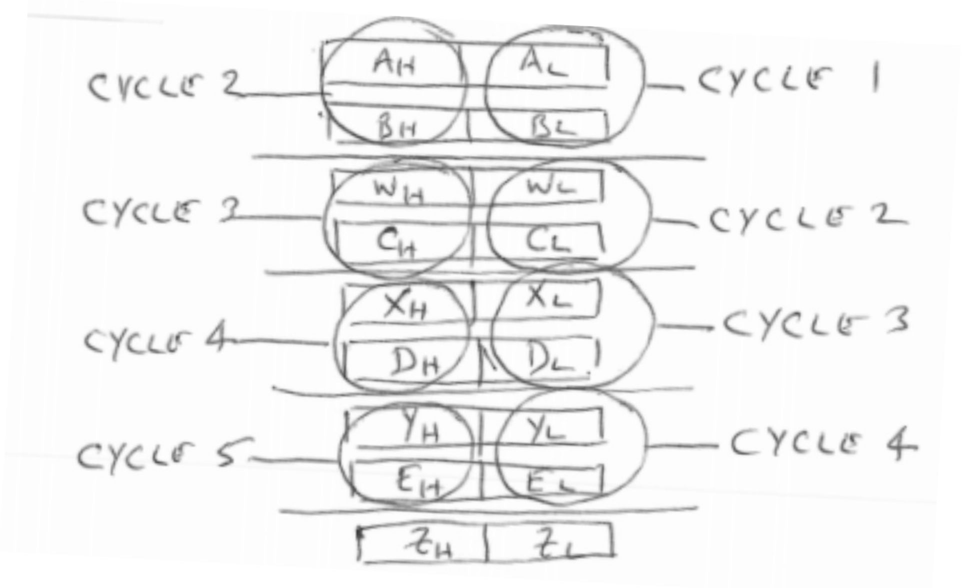
  ***W=A+B***

  ***X=W+C***

  ***Y=X+D***

  ***Z=Y=E***

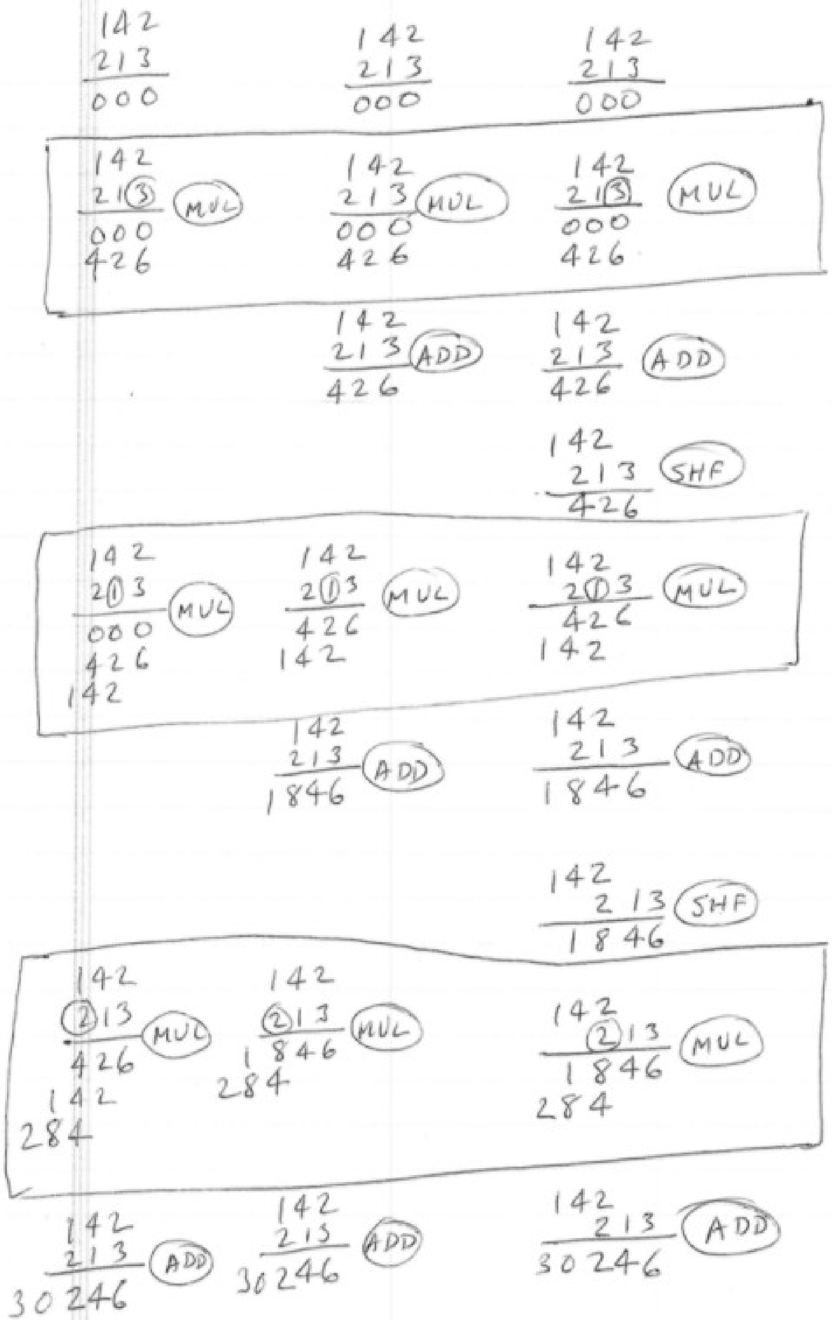
- ***Operands have too many bits, cycle time is too long***
  - ***Cut number of bits in half, e.g., A becomes A_high and A_low***
  - ***Perform 2 ADDs, each clock cycle, on half-width operands***
  - ***The result: 5 adds, rather than 4, BUT with much smaller cycle***



# *BCD Arithmetic*

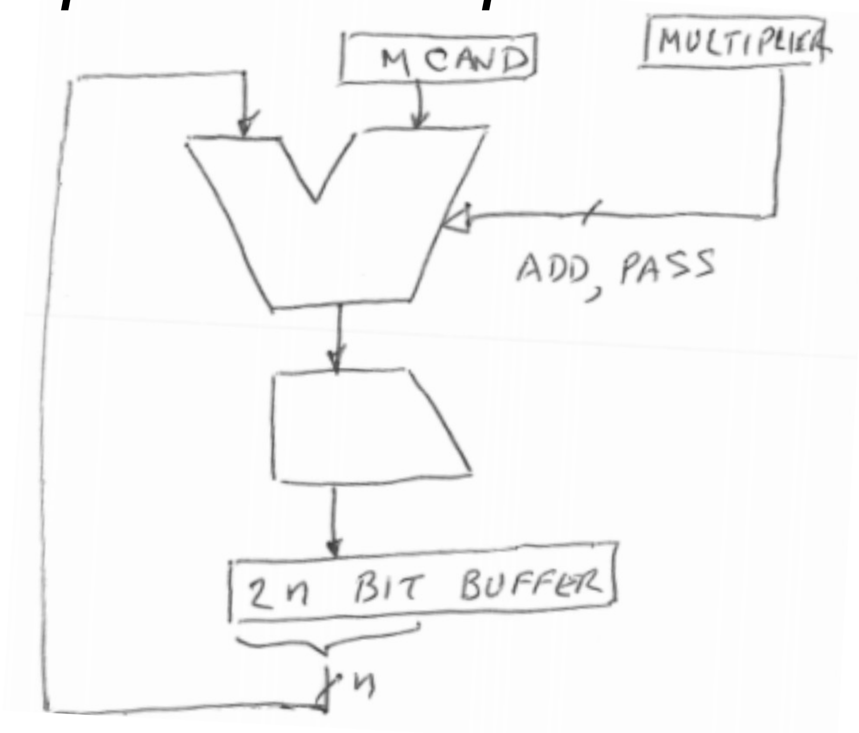
- ***BCD Each decimal digit represented by 4 bits***
- ***Memory location requires address and size***
- ***Addition with a standard 2's complement ALU***
  - ***Although we could design a special BCD Adder***
- ***The process (using a standard 2's complement ALU***
  - ***Step 1: Add x6666…6 to one of the operands. (Why?)***
  - ***Step 2: Add result to the other operand***
  - ***Step 3: Correct by subtracting 6 where necessary (When?)***
- ***An example: Add BCD numbers 283, 598***
  - ***283: 0010 1000 0011, 598: 0101 1001 1000***
  - ***Step 1: With standard ALU, 283 + 666 = 8E9***
  - ***Step 2: With standard ALU, 8E9 + 598 = E81***
  - ***Step 3: Since high digit did not generate a carry, subtract 6 from it i.e, E81 – 600 = 881, the correct answer!***

# Multiplication (let's start with decimal)



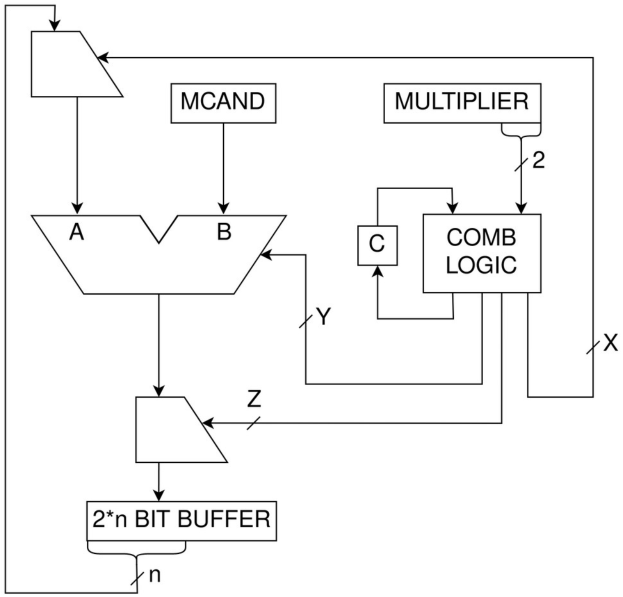
# Multiplication

- ***A sequence of shifts and adds, one bit each iteration***
  - ***Initially load the multiplier, the multiplicand, and 0 in the Buffer***
  - ***The multiplier is a shift register that right shifts one bit per cycle***
  - ***The 2n bit buffer gets the result of the multiplication***
  - ***Iterations stop when the multiplier contains all 0's.***



# Multiplication (continued)

- **Booth's Algorithm (my variation, to better explain it)**
  - *Initially load the multiplier, multiplicand, and 0 in the Buffer*
  - *The multiplier is in a shift register that right shifts two bits per cycle*
  - *The 2n bit Buffer gets the result of the multiplication*
  - *Iterations stop when the multiplier contains all zeroes*
  - *Control of the two shifters and ALU from the low two bits of the multiplier and the "c" bit, which is produced by a prior iteration*




| Bit_1 | Bit_0 | C | X | Y | Z | C' |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | SHF0 | PassA | SHF2 | 0 |
| 0 | 0 | 1 | SHF0 | ADD | SHF2 | 0 |
| 0 | 1 | 0 | SHF0 | ADD | SHF2 | 0 |
| 0 | 1 | 1 | SHF1 | ADD | SHF1 | 0 |
| 1 | 0 | 0 | SHF1 | ADD | SHF1 | 0 |
| 1 | 0 | 1 | SHF0 | SUB | SHF2 | 1 |
| 1 | 1 | 0 | SHF0 | SUB | SHF2 | 1 |
| 1 | 1 | 1 | SHF0 | PassA | SHF2 | 1 |

# ***Booth's Algorithm (first a simple example)***

- ***We want to multiply 22 by 9***
  - ***22 is 00010110, 9 is 00001001***
  - ***00010110 is the MCAND, 000001001 is the Multiplier***
- ***We partition the multiplier bits into 2-bit pieces: 00 00 10 01***
- ***Right-most bits  = 01, which is 1 times 4^0***
  - ***Add (1 times 4^0) times MCAND = 22***
  - ***Add this to the Buffer (which initially contained 0)***
  - ***Then we shift the multiplier right two bits, yielding 00 00 00 10***
  - ***And, we shift the buffer right two bits, effectively multiplying the MCAND by 4***
  - ***The MCAND is now effectively 88***
- ***Right-most bits of the multiplier are = 10, which is 2.***
  - ***Shift the MCAND one bit to the right, thereby multiplying MCAND by 2 (i.e., 176) and add it to the Buffer (176 + 22 =198)***
  - ***Then we again shift right the multiplier two bits, yielding 00 00***
- ***Since there are no more non-zero bits in the multiplier, we are done!***
  - ***The buffer contains the product of 22 times 9, i.e. 198.***

# *Booth's Algorithm (A more interesting example)*

- ***We want to multiply 22 x 14; MCAND = 00010110, Multiplier = 00001110***
- ***We partition our multiplier bits into 2-bit pieces: 00 00 11 10***
- ***Right-most bits = 10, which is 2***
  - ***Shift the MCAND one bit to the left, thereby multiplying MCAND by 2 (i.e., 44), add it to the Buffer (44), then shift right the Buffer 2 bits***
  - ***Then we shift right the multiplier two bits, yielding 00 00 11***
- ***Right-most bits are 11, which is 3. Important to note that 3 = 4 -1.***
  - ***Subtract 1 times MCAND from the Buffer and add 1 to the next iteration of the multiplier, yielding 00 01***
  - ***Net result: We have subtracted 4 times MCAND from the running sum***
  - ***As before, we right shift the contents of the Buffer two bits***
  - ***Then we shift right the multiplier two bits, yielding 00 01***
- ***Right-most bits (now) = 01, which is 1.***
  - ***Add 1 times MCAND to the Buffer.***
  - ***Net result: We have added 16 times MCAND to the running sum.***
  - ***Then we right shift the Buffer two bits.***
  - ***Then we shift right the multiplier two bits, yielding 00, and we are done.***
- ***Final result: (16 -4 +2) times MCAND = (14) times MCAND.***

# *Residue Arithmetic (an entertaining digression)*

- ***When?***
  - ***Inputs, outputs relatively small integers***
  - ***Intermediate results could be very large***
  - ***Internally compute-intensive***
  - ***Very little I/O***
- ***How?***
  - ***Step 1: transform to the residue number domain SLOW***
    - ***a, b -→ f(a), f(b)***
  - ***Step 2: Perform the operation in the residue domain. FAST***
    - ***f© ← f(a) * f(b)***
  - ***Step 3: Perform the inverse transformation SLOW***
    - ***c ← f©***
- ***Note: Does this remind you of anything you have studied in some other course?***

# *Residue Arithmetic (continued)*

- ***The detail:***
  - ***Pick a set of moduli p1, p2, ..pn  that are relatively prime***
  - ***Represent each value X as x1,x2,..xn, where xi = X mod pi***

    ***The Chinese Remainder Theorem (from the first century AD) states that each integer between 0 and (product p1,p2,…pn) -1 are uniquely represented.***

  - ***Sum (X,Y), Product (X,Y) can be computed by n simpler elements, all working concurrently, with no interaction between them, yielding a result very fast.***

- ***An example: Add, Multiply the two numbers, 19 and 24***
  - ***Using the moduli p1 = 7, p2 = 8, p3 = 9, 19 is 5,3,1***
  - ***Adding 5,3,1 to 3,0,6, we get 1,3,7, which is 43.***
  - ***Multiplying 5,3,1 to 3,0,6, we get 1,0,6, which is 456.***

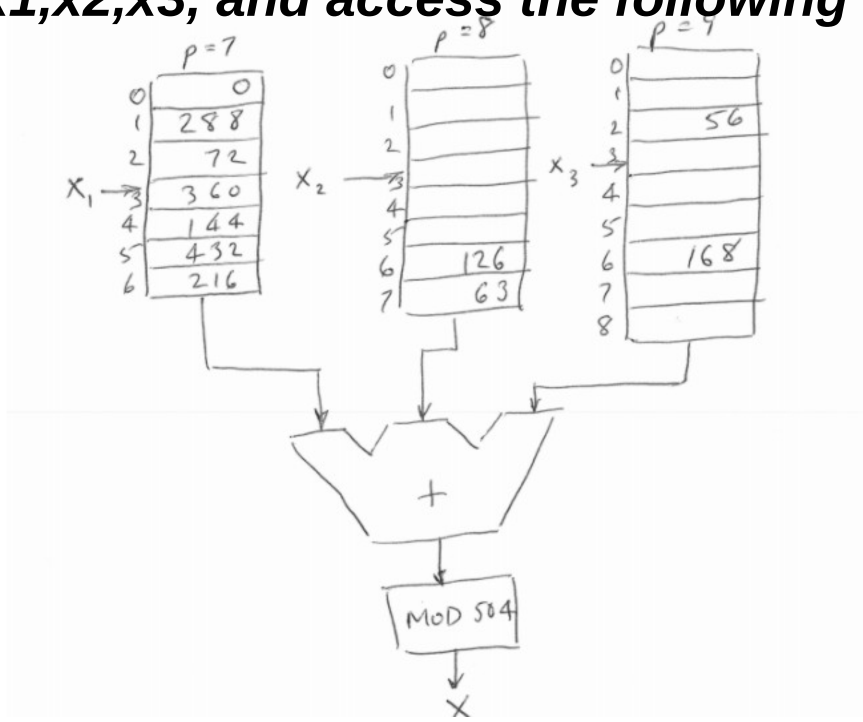
# *Residue Arithmetic (Two observations)*

- ***Why does it work?***
  - *Consider the multiplication of A and B*
  - *A * B = (m * p + a) * (n * p +b),*
    *where a is A mod p, b is B mod p.*
  - *Thus A * B = p * (m * n * p + a*n + b*m) + a*b,*
  - *From which, (A * B) mod p = a * b,*
  - *Completely independent of the other moduli.*

- ***Then why is not used?***
  - *Transformations are expensive*
  - *Comparisons are unwieldly  (e.g., How to determine if A>B.*

# Residue Arithmetic (The Inverse Transformation)

- *We multiplied 19 times 24, and got the result: 1,0,6*
  - *We know X is defined by 1 for x1, 0 for x2, and 6 for x3*
  - *It would be nice to put it into a more familiar form (e.g., 456)*
  - *We know 1,0,6 is 1,0,0 + 0,0,0 + 0,0,6. How do we know that?*
  - *We know 1,0,0 must be a multiple of 72; How do we know that?*
  - *…and 0,0,0, a multiple of 63, and 0,0,6 a multiple of 56.*
  - *So we build three tables with the entries corresponding to the values of x1,x2,x3, and access the following data path:*



***Merci***