CPE300: Digital System Architecture and Design

Fall 2011

MW 17:30-18:45 CBC C316

Arithmetic Unit

10122011

http://www.egr.unlv.edu/~b1morris/cpe300/

Outline

- Recap
- Fixed Point Arithmetic
  - Addition/Subtraction
  - Multiplication
  - Divide
- Floating Point Arithmetic

Digital Number Systems

- Expanded generalization of lecture 07 topics

- Number systems have a base (radix) b
- Positional notation of an m digit base b number
  - $x = x_{m-1}x_{m-2} \ldots x_1x_0$
  - $\text{Value}(x) = \sum_{i=0}^{m-1} x_i b^i$
- Base b fraction
  - f= .f_{-1} f_{-2}… f_{-m}
  - Value is integer f_{-1} f_{-2}… f_{-m} divided by b^m
- Mixed fixed point number
  - $x_{n-1}x_{n-2} \ldots x_1x_0.x_{-1}x_{-2} \ldots x_{-m}$
  - Value of n+m digit integer
    - $x_{n-1}x_{n-2} \ldots x_1x_0x_{-1}x_{-2} \ldots x_{-m}$
  - Divided by b^m

Negative Numbers and Complements

- Given m digit base b number x
- Radix complement (b's complement)
  - $x^c = (b^m - x) \bmod b^m$
  - mod bm only has effect for x=0
    - What is radix complement of x = 0?
- Diminished radix complement ((b-1)'s complement)
  - $\hat{x}_c = b^m - 1 - x$
- The complement operations are related
  - $x^c = (\hat{x}^c + 1) \bmod b^m$
  - Given one, easy to compute other

Digitwise Computation of Diminished Radix Complement

- $\hat{x}_c = bm - 1 - x$
- $\hat{x}_c = \sum_{i=0}^{m-1}(b-1)b^i - \sum_{i=0}^{m-1}(x_i)b^i$
- $\hat{x}_c = \sum_{i=0}^{m-1}(b-1-xi)b^i$

- Diminished radix number is an m digit base b number
  - Each digit is obtained (as diminished complement) from corresponding digit in x

Scaling Complement Numbers

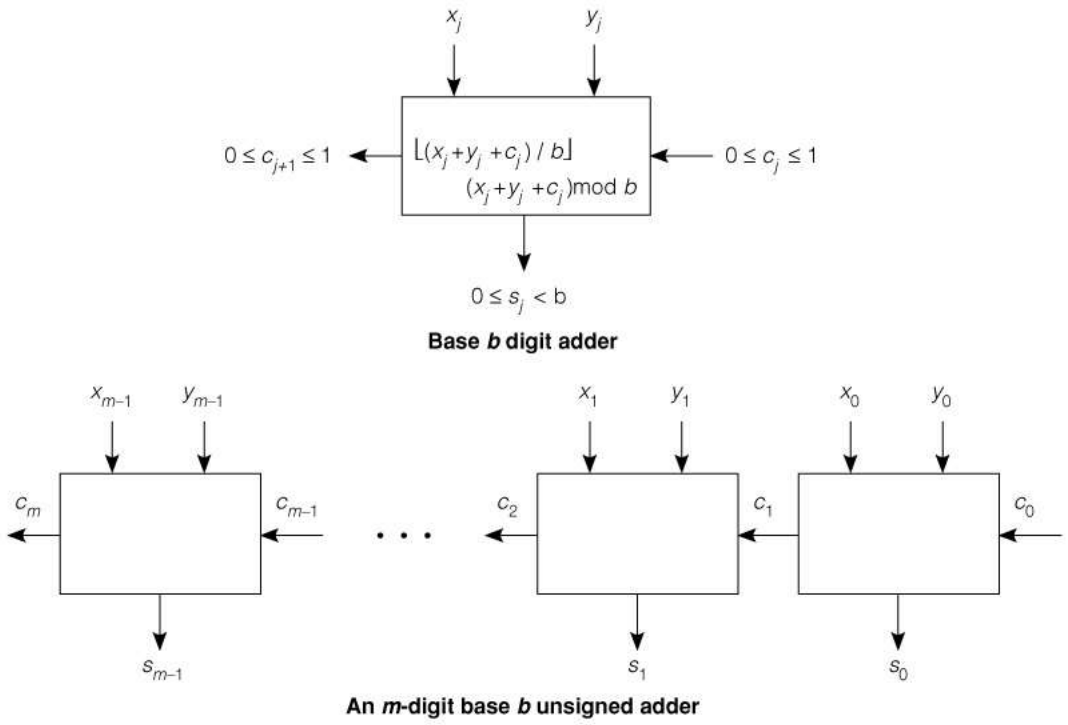
- Divide by b – move radix point left => shift right
  - Fill rule for even b
    - Zero fill when $x_{m-1} < b/2$ (positive)
    - (b-1) fill when $x_{m-1} \geq b/2$ (negative)
- Multiply by b – move radix point right => shift left
  - Overflow conditions
    - Result appears to change sign
    - Non-zero digit shifted off msb (positive)
    - None-(b-1) digit off msb (negative)

Fixed Point Addition and Subtraction

- When radix point is in the same position for both operands
  - Add/Sub acts as if numbers were integers
- Addition of signed numbers in radix complement system only needs an unsigned adder
  - Must design m digit base b unsigned adder
- Radix complement signed addition theorem
  - s = rep(x) + rep(y) = rep(x+y)
  - rep(x) := b's complement representation of x
  - Does not consider overflow

Unsigned Addition Hardware

- Perform operation on each digit of m digit base b number
- Each digit cell requires operands x_j and y_j as well as a carry in c_j
- Sum
  - $s_j = (x_j + yj + cj) \bmod b$
- Carry-out
  - $c_{j+1} = \lfloor (x_j + yj + cj)/b \rfloor$
  - All carries are less than equal to 1 regardless of b
- Works for any fixed radix point location (e.g. fractions)



Ripple-carry adder: carry propagates from low digits to msb

Unsigned Addition Example

Op1		1	2	.	0	3_4	=	6.1875_{10}
Op2	+	1	3	.	2	1_4	=	7.5625_{10}
Carry	0	1	0		1			
Sum		3	1	.	3	0_4	=	13.75

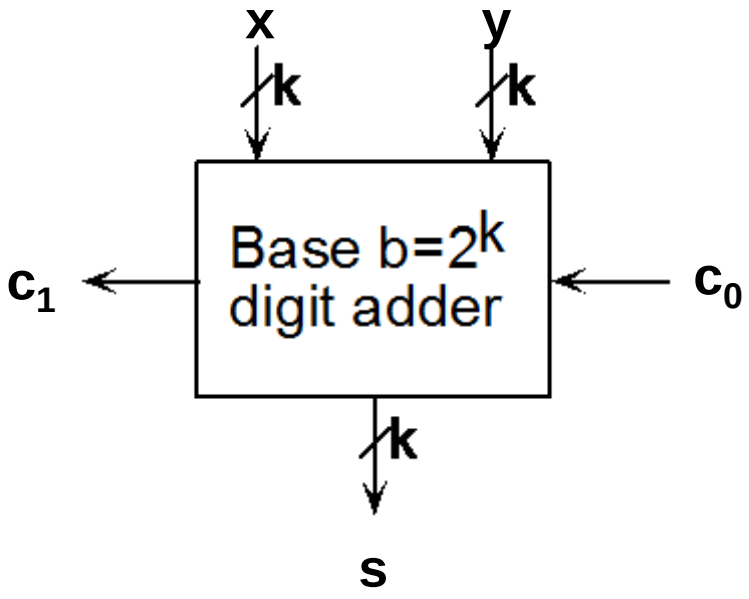
+	0	1	2	3
0	00	01	02	03
1	01	02	03	10
2	02	03	10	11
3	03	10	11	12

Base 4 addition table

- With fixed number of digits, overflow occurs on carry from leftmost digit
- Carries are 0 or 1 in all cases
- Addition is defined by a table of sum and carry for b^2 digit pairs

Adder Implementation Alternatives

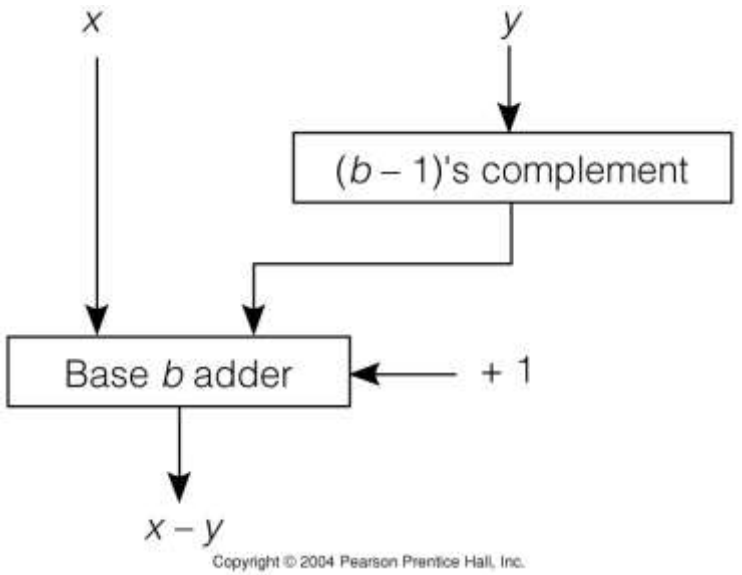
- For base $b=2^k$, each digit is equivalent to k bits
- Adder can be viewed as logic circuit with 2k+1 inputs and k+1 outputs



- Ripple carry adder
  - Choice of k affects computation delay
  - When 2 level logic is used what is max gate delay for m digit addition?
    - 2m

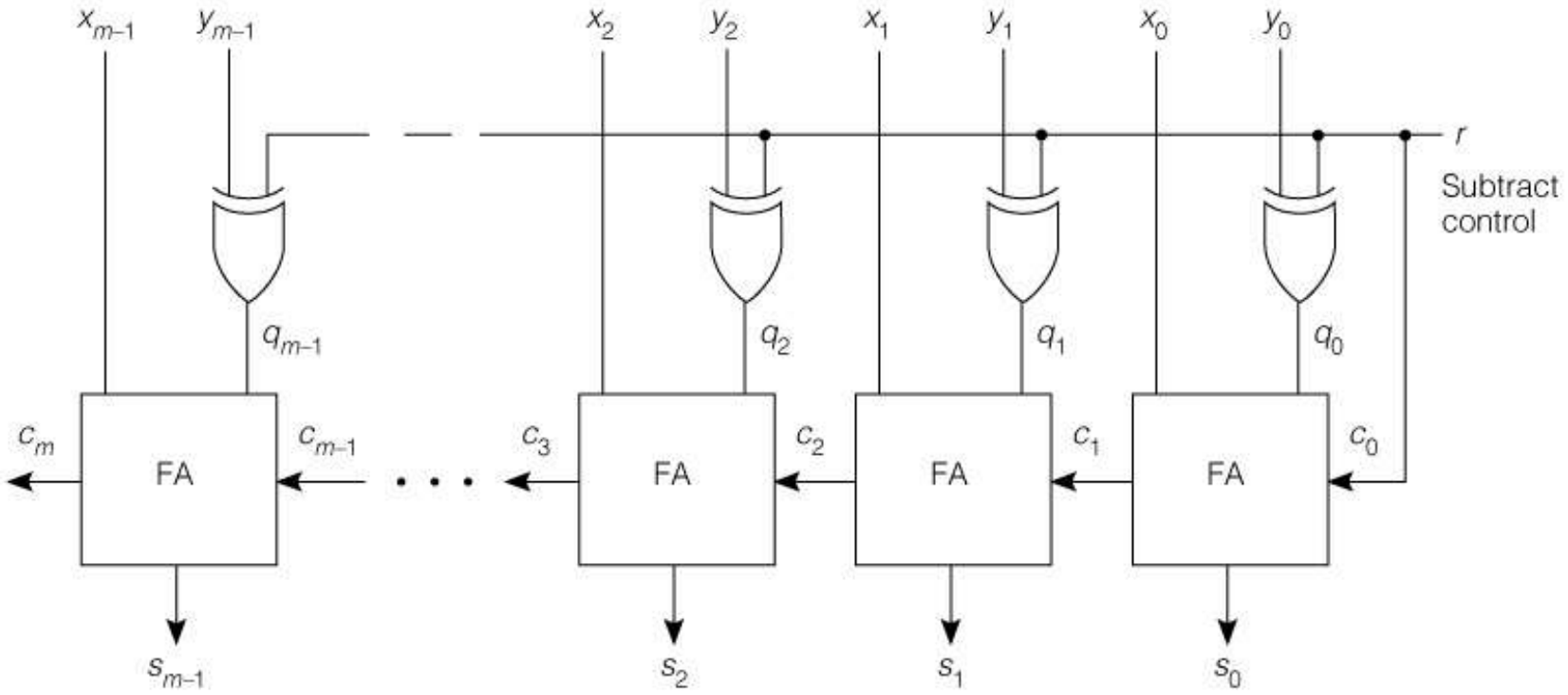
Complement Subtracter

- Subtraction in radix complement is addition with negated (complemented) second input
  - x – y = x + (-y)
  - Must supply overflow detection
- Radix complement is addition of 1 to diminished radix complement
  $(x^c = (\hat{x}^c + 1) \bmod bm)$
  - Easy to take diminished radix complement and use carry in of adder to supply +1 for radix complement



2's Complement Ripple-Carry

- Binary ripple-carry adder/subtracter

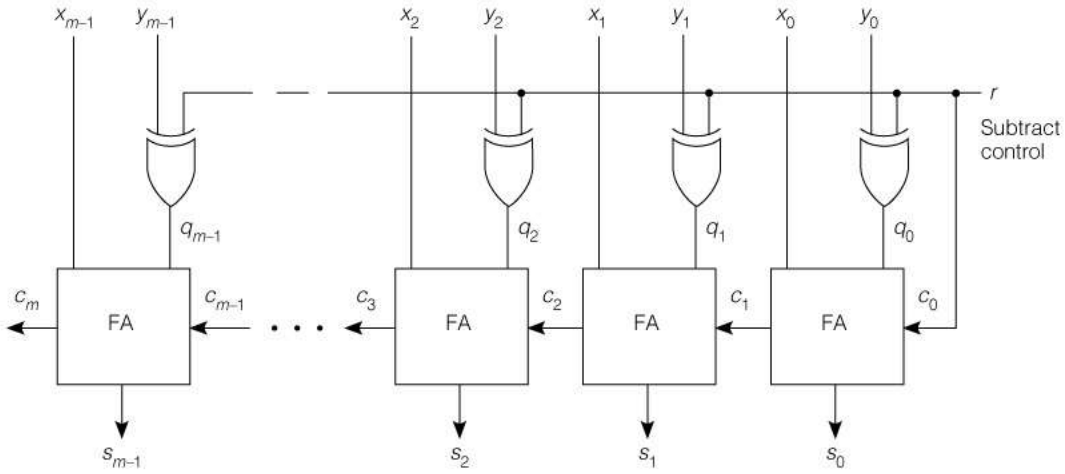


- XOR gates select y for addition or complement of y for subtraction in base 2

Overflow Detection

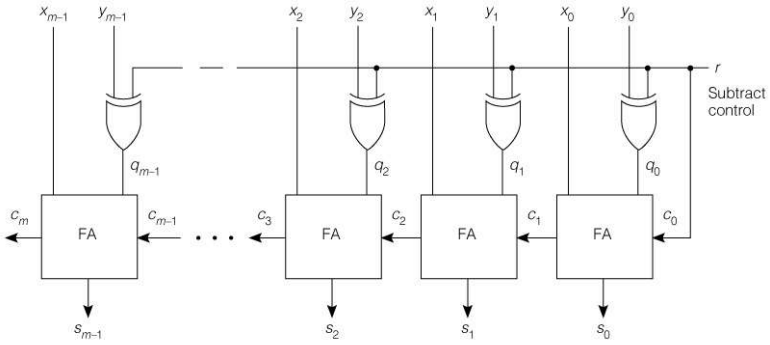
- Occurs when adding number of like sign and the result seems to have opposite sign
- For even b: sign determined by the leftmost digit
  - Overflow detector only requires
    - x_{m-1}, y_{m-1}, $s_{m-1,}$ add/subtract control (r)

Binary ripple-carry adder/subtracter



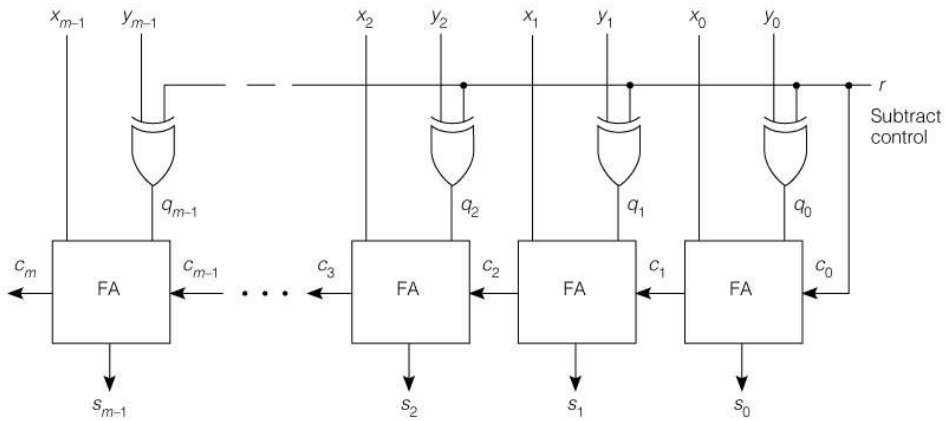
Carry Lookahead

- Speed of addition depends on carries
  - Carries need to propagate from lsb to msb
- Two level logic for base b digit becomes complex quickly for increasing k ($b=2^k$)
  - Length of carry chain divided by k
- Need to compute carries quickly
  1. Determine if addition in position j generates a carry
  2. Determine if carry is propagated from input to output of digit j



Binary Generate and Propagate Signals

- Generate: digit at position j will have a carry
  - $G_j = xjy_j$
- Propagate: carry in passes through to carry out
  - $P_j = xj + yj$
- Carry is defined as 1 if the sum generates a carry or if a carry is propagated
  - $c_{j+1} = Gj + Pjc_j$



Carry Lookahead Speed

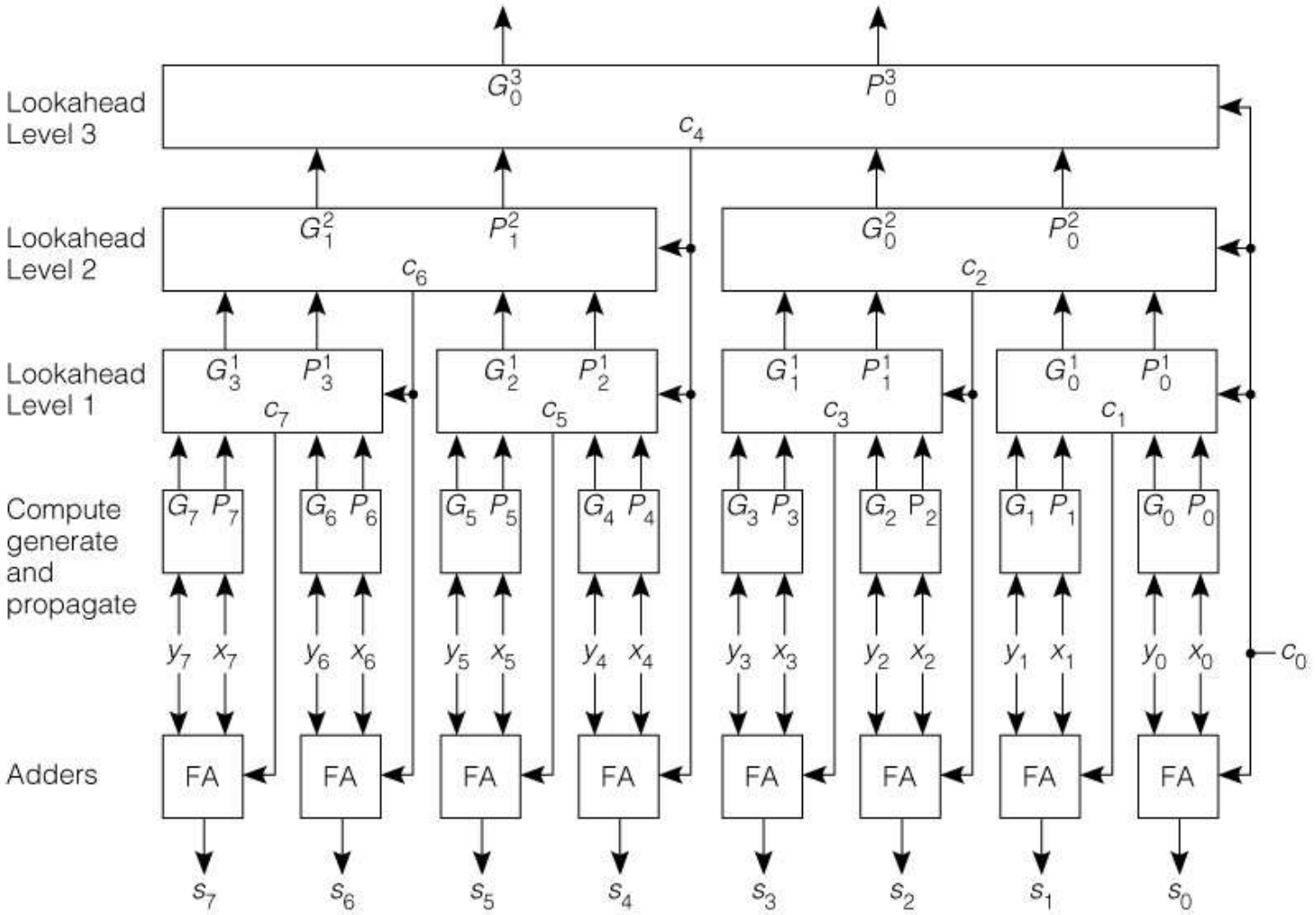
- 4 bit carry equations
  - $c_1 = G_0 + P_0c_0$
  - $c_2 = G_1 + P_1G_0 + P_1P_0c_0$
  - $c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$
  - $c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$
- Carry lookahead delay
  - One gate delay for to calculate G or P
  - 2 levels of gates for a carry
  - 2 gate delays for full adder (s_j)
- The number of OR gate inputs (terms) and AND gate inputs (literals in a term) grows as the number of carries generated by lookahead

Recursive Carry Lookahead

- Apply lookahead logic to groups of digits
- Group of 4 digits (level 1)
  - Group generate:
    - $G^1{}_0 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$
  - Group propagate:
    - $P^1{}_0 = P_3P_2P_1P_0$
  - Can further define level 2 signals which are groups of level 1 groups
- Group k terms at each level → $\log_k m$ levels for m bit addition
  - Each level introduces 2 more gate delays
  - k chosen to trade-off reduced delay and complexity of G and P logic
    - Typically $k \geq 4$ however structure easier to see for k=2

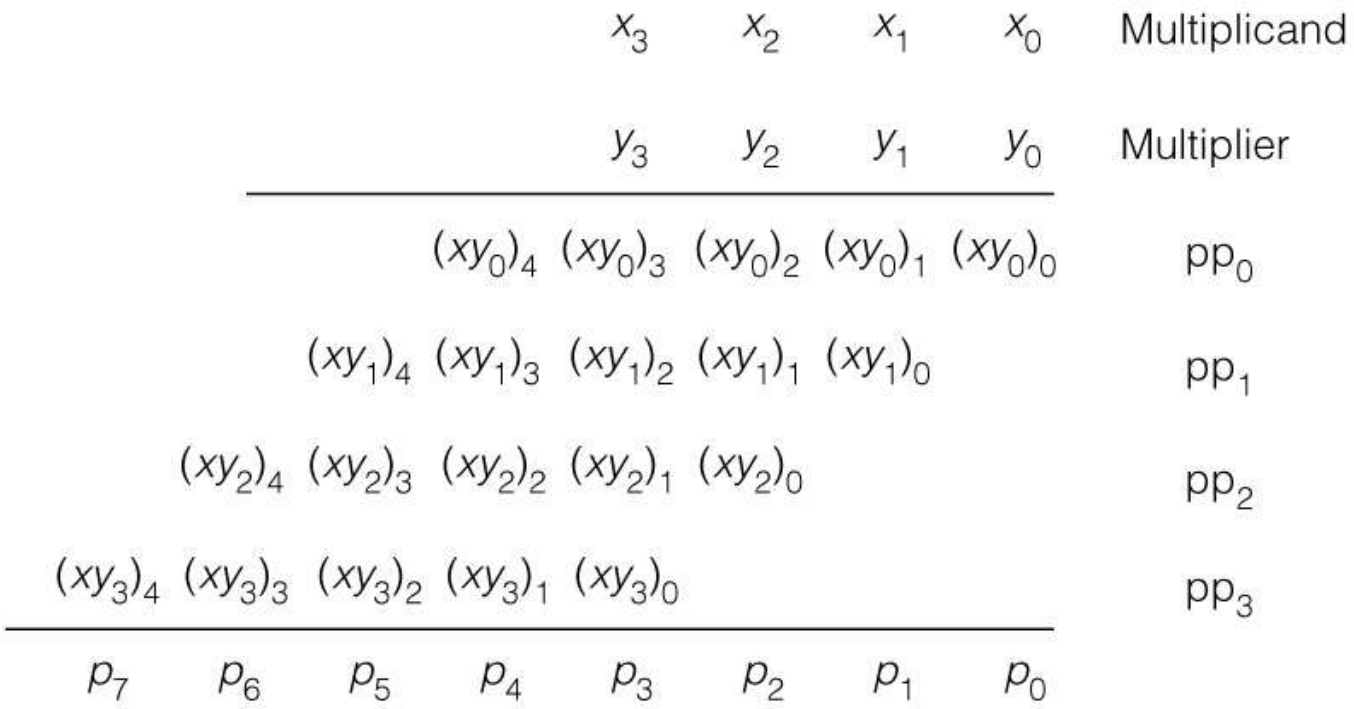
Carry Lookahead Adder Diagram

- Group size k=2

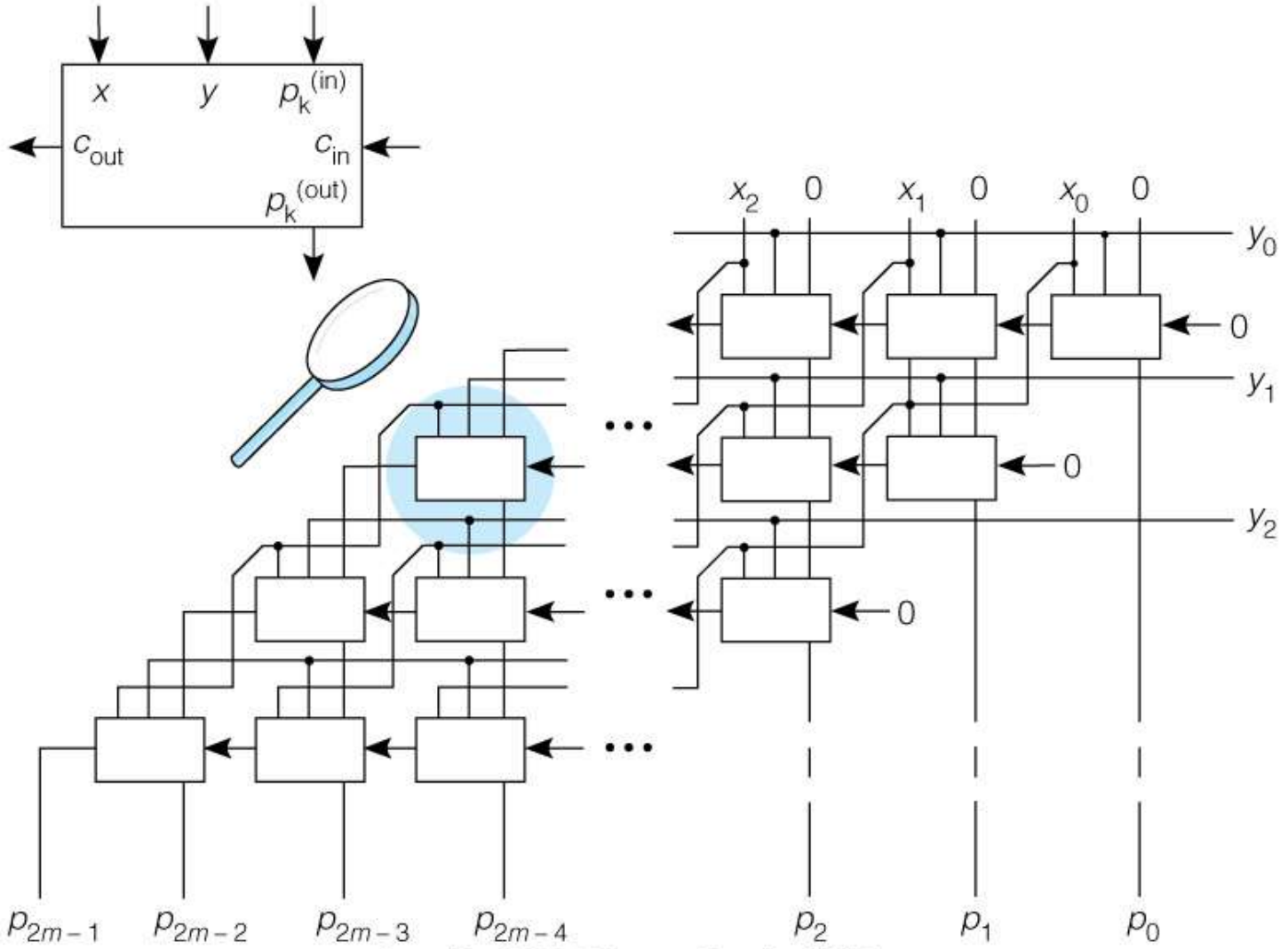


Digital Multiplication

- Based on digital addition
  - Generate partial products (from each digit) and sum for the complete product
  - "Pencil and paper addition"



Parallel Array Multiplier



Parallel Array Multiplier Operation

- Each box in array does the base b digit calculations
  - $p_k(\text{out}) := (p_k(\text{in}) + xy + c(\text{in})) \bmod b$
  - $c\ (\text{out}) := \lfloor (pk(\text{in}) + xy + c)/b \rfloor$
- Inputs and outputs of boxes are single base b digits (including carries)
- Worst case path from input to output is about 6m gates if each box is a 2 level circuit
  - In binary, each box is a full adder with an extra AND gate to compute xy

Accumulated Partial Product

- Partial products accumulated rather than collected and added in the end

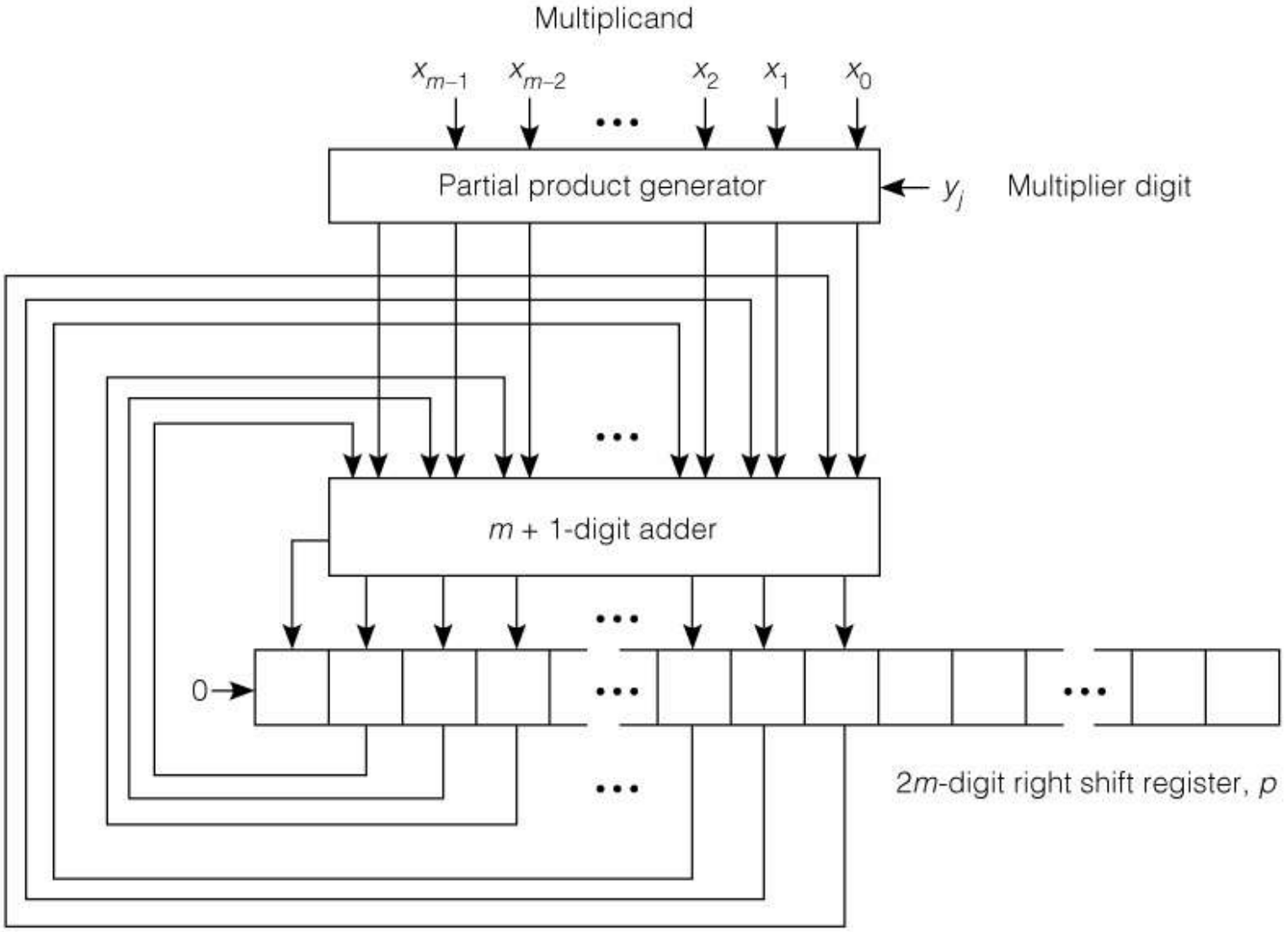
```
1.      for i := 0 step 1 until 2m-1
2.              p_i := 0;
3.      for j := 0 step 1 until m-1
4.              begin
5.                      c := 0;
6.                      for i := 0 step 1 until m-1
7.                              begin
8.                                      p_{j+i} := (p_{j+i} + x_i y_j + c) mod b;
9.                                      c := ⌊(p_{j+i} + x_i y_j + c)/b⌋;
10.                             end;
11.                     p_{j+m} := c;
12.             end;
```

c is a single base b digit
(no longer 0, 1 as in addition)

Series Parallel Multiplication

- Hardware multiplies the full multiplicand by one multiplier digit and adds it to a running product
  - $p := p + xy_j b^j$
- Multiplication by b^j is done by scaling
  - xy_j shifted left, or
  - p shifted right by j digits
- Generation of the partial product xy_j is more difficult than the shifted add
  - Exception: base 2
    - partial product is either x or 0

Unsigned Series-Parallel Multiplication Hardware



Steps for Unsigned Multiplication Hardware Use

1. Clear product shift register p.
2. Initialize multiplier digit number $j = 0$.
3. Form the partial product xy_j.
4. Add partial product to upper half of p.
5. Increment $j = j + 1$, and if $j = m$ go to step 8.
6. Shift p right one digit.
7. Repeat from step 3.
8. The 2m digit product is in the p register

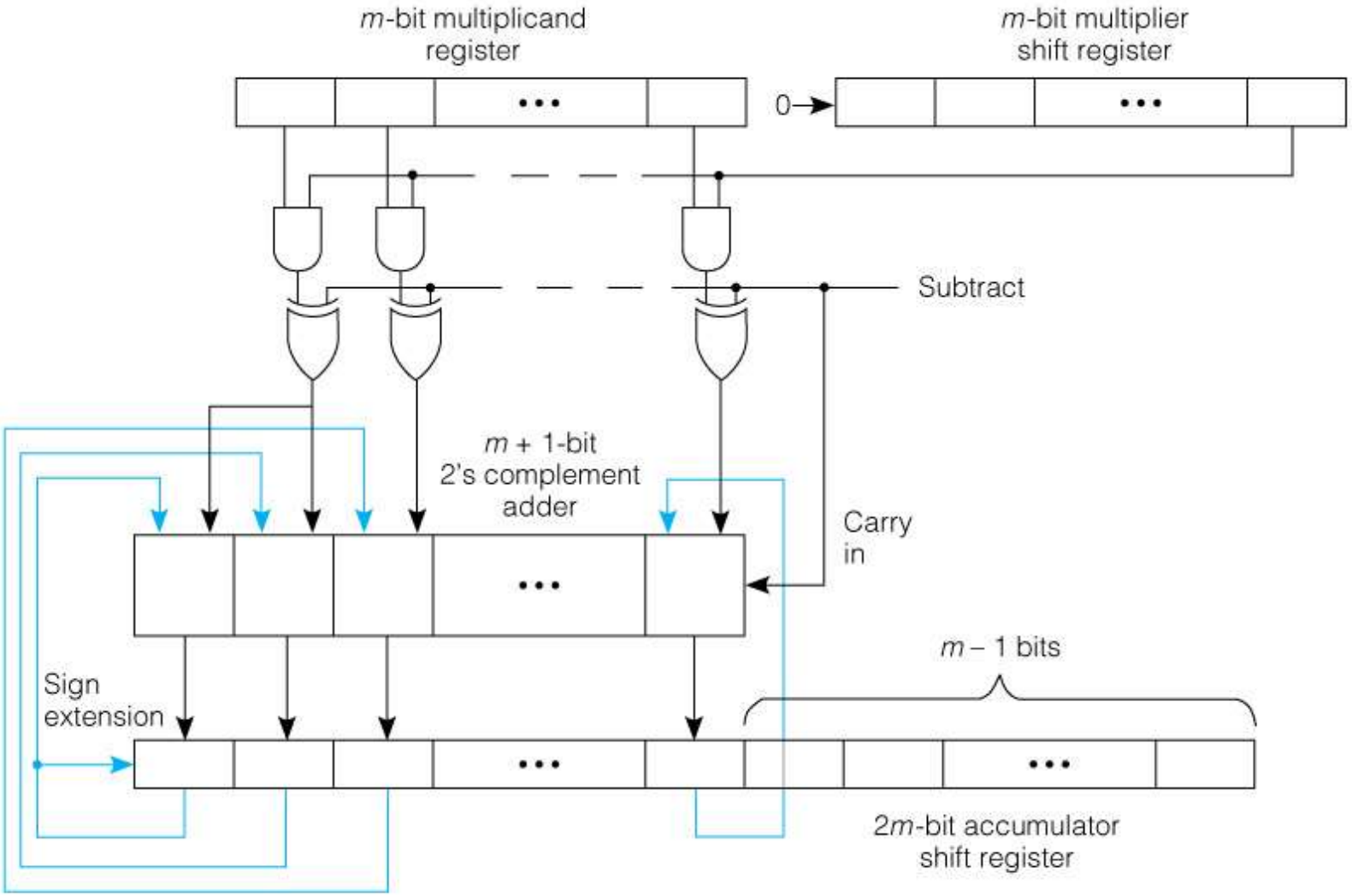
Fixed-Point Multiplication

- Radix point is in fixed position of word
  - Right end for integer
  - Left end for a fraction
- Overflow (integer)
  - Result is lower m digits
  - Upper m digits of p contain non-zero entries
- Accuracy loss (fraction)
  - Result is upper m digits
  - Lower m digits discarded (or rounded)

Signed Multiplication

- Unsigned method
  - Negative operands can be complemented and magnitudes multiplied, product complemented if necessary
    - Sign of product easily computed by signs of operands
- Direct method
  - Use b's complement adder (discard carry out) and use sign extension for shifts
  - Add all partial products but subtract last partial product
  - $\text{Value}(x) = -x_{m-1}x^{m-1} + \sum_{i=0}^{m-1} x_i b^i$

2's Complement Signed Multiplier Hardware



Steps for Signed Multiplication Hardware Use

1. Clear the bit counter and partial product accumulator register.
2. Add the product (AND) of the multiplicand and rightmost multiplier bit.
3. Shift accumulator and multiplier registers right one bit.
4. Count the multiplier bit and repeat from 2 if count less than m-1.
5. Subtract the product of the multiplicand and bit m-1 of the multiplier.

Note: bits of multiplier used at rate product bits produced

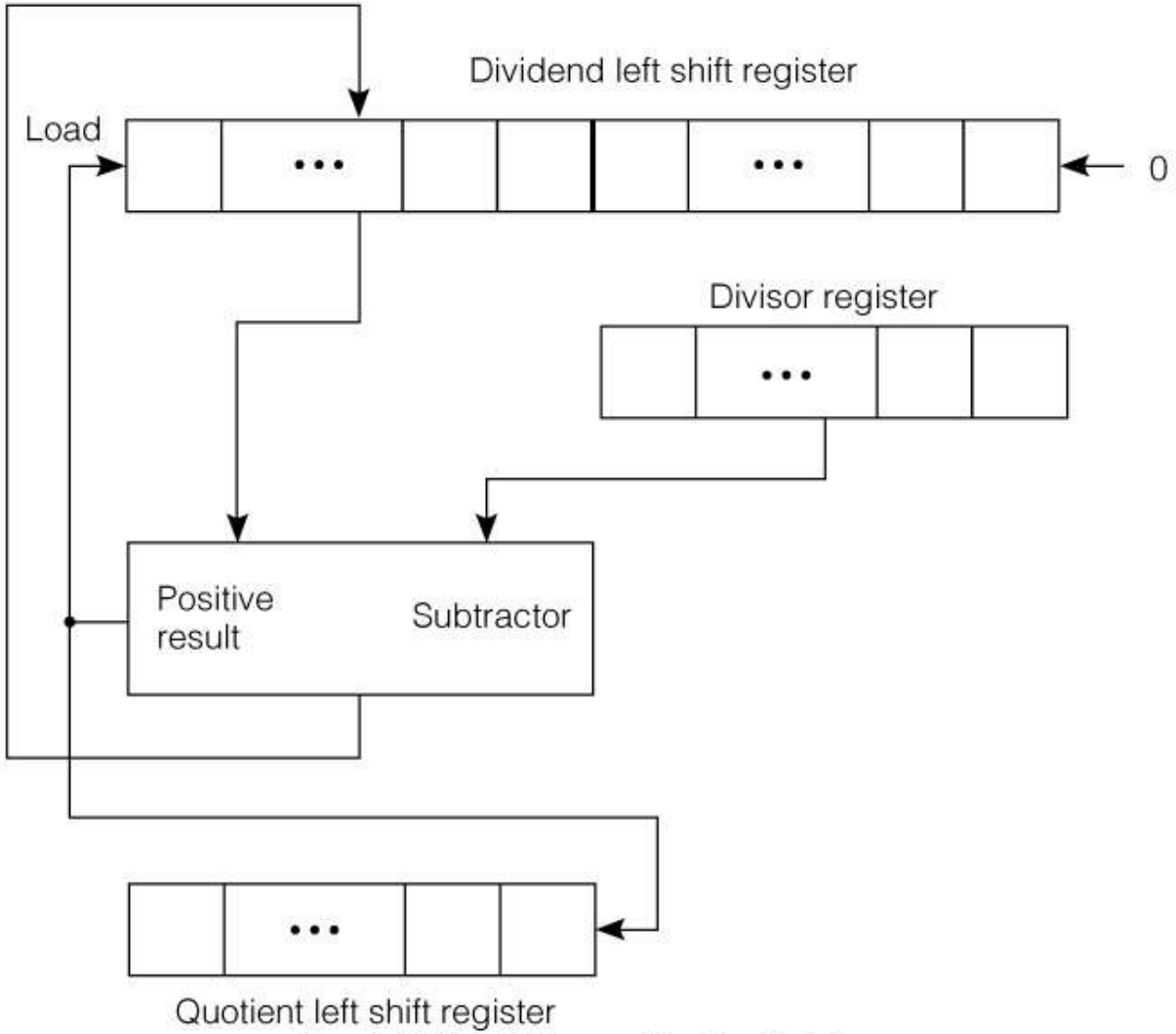
2's Complement Multiplication Examples

-5/8 =	1.011	6/8 =	0.110
× 6/8 =	× 0.110	× -5/8 =	× 1.011
pp0	00.000	pp0	00.110
acc	00.0000	acc	00.0110
pp1	11.011	pp1	00.110
acc	11.10110	acc	00.10010
pp2	11.011	pp2	00.000
acc	11.100010	acc	00.010010
pp3	00.000	pp3	11.010
result	11.100010	result	11.100010

Digital Division

- Dividend is dived by a divisor to get a quotient and a remainder
- A $2m$ digit dividend divided by an m digit divisor does not necessarily give an m digit quotient and remainder
  - Divisor of 1 gives same size as dividend
- Fraction division (D/d both fractions)
  - D/d quotient is only a fraction if $D < d$
  - Divide overflow occurs when $D \geq d$
    - Why?

Binary Division Hardware



- 2m bit dividend register
- m bit divisor
- m bit quotient
- Divisor can be subtracted from dividend, or not

Integer Division Hardware Steps

1. Put dividend in lower half of register and clear upper half. Put divisor in divisor register. Initialize quotient bit counter to zero.
2. Shift dividend register left one bit.
3. If difference positive, shift 1 into quotient and replace upper half of dividend by difference. If negative, shift 0 into quotient.
4. If fewer than m quotient bits, repeat from 2.
5. m bit quotient is an integer, and an m bit integer remainder is in upper half of dividend register

Fraction Division Hardware Steps

1. Put dividend in upper half of dividend register and clear lower half. Put divisor in divisor register. Initialize quotient bit counter to zero.
2. If difference positive, report divide overflow.
3. Shift dividend register left one bit.
4. If difference positive, shift 1 into quotient and replace upper part of dividend by difference. If negative, shift 0 into the quotient.
5. If fewer than m quotient bits, repeat from 3.
6. m bit quotient has binary point at the left, and remainder is in upper part of dividend register.

Binary Division Example

Divide D=45 by d=6			
D	000000 101101		
-d	111010	q	0
D	000001 01101-		
-d	111010		
D-d (diff -)	111011 01101-	q	0
D	000010 1101--		
-d	111010		
D-d (diff -)	111101 1101--	q	00
D	000101 101---		
-d	111010		
D-d (diff -)	111111 101---	q	000
D	001011 01----		
-d	111010		
D-d (diff +)	1 000101 01----	q	0001
D	001010 1-----		
-d	111010		
D-d (diff +)	1 000100 1-----	q	00011
D	001001 ------		
-d	111010		
D-d (diff +)	1 000011 ------	q	000111
remainder	000011		

Fixed-Point Arithmetic Highlights

- Digitwise addition with single-bit carry between digits is basis for add/sub
- Carry lookahead is the principal technique for fast add/sub
  - Carry propagation is slow in ripple-adder
- Fixed-point mult/sub is built on repeated add/sub
  - Replicated hardware for parallel computation
  - Repetitive calculations done with single adder/subtracter with a register for immediate result

Branching on Arithmetic Conditions

- ALU operations produce result as well as carry from left bit and overflow indicator
- 3 common methods for using outcome of compare (subtraction) for a branch condition
  1. Do compare in branch instruction
  2. Set special condition code bits and test them at in branch instruction
  3. Set general register to comparison outcome and branch on this logical value

Considerations for Branch Alternatives

- Comparison in branch
  - Increased branch instruction length
    - Must specify 2 operands to be compared, branch target, and branch condition (possibly place for link)
  - Computations before branch decision
    - ALU must be used and output tested
      - Longer instruction time
      - Need for more branch delay slots in pipeline
- Condition codes
  - Extra processor state set and overwritten by many instructions
    - Must be used before being set again
    - Introduces hazards in pipelined design
  - Must be saved/restored during exception handling
- General register
  - Large register used for single true/false comparison value

Motorola 68000 CC Use

- HLL statement
  - if(A > B) then C = D
- MC68000 assembly

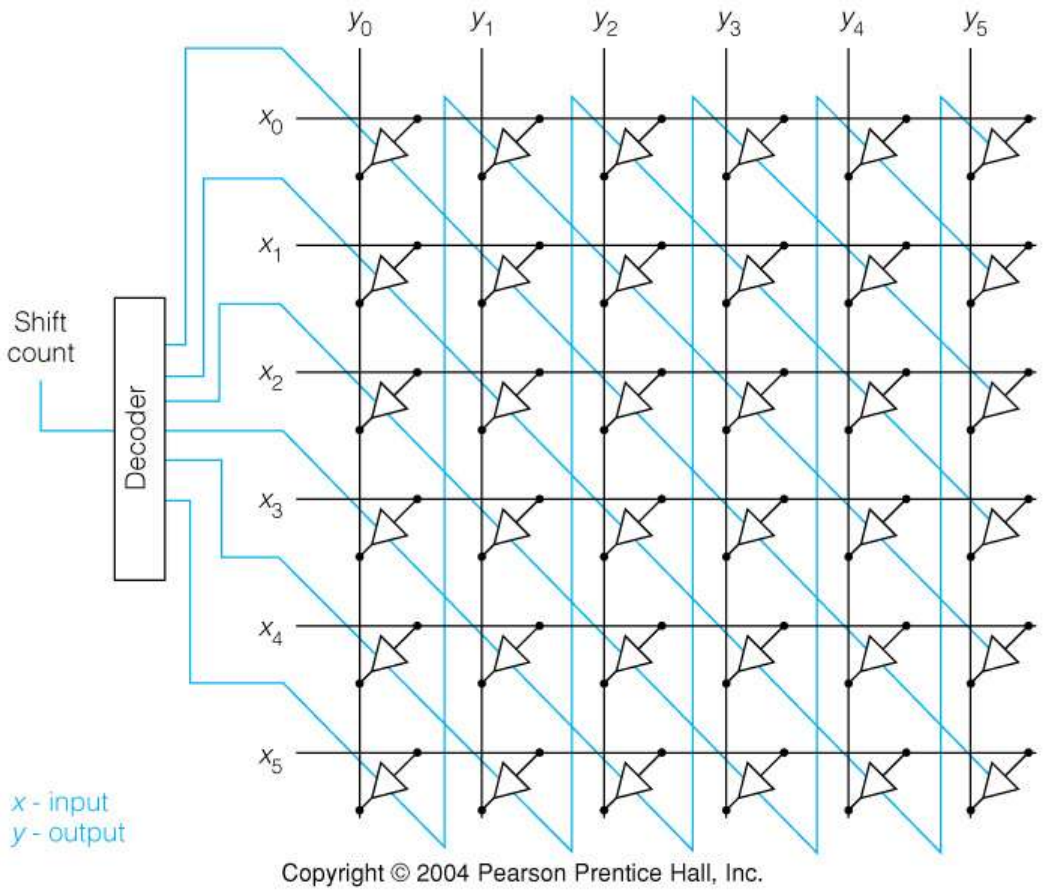
```
            MOVE.W   A, D0
            CMP.W    B D0
            BLE      Over
            MOVE.W   D, C
Over:       …
```

Actually test less than equal rather than greater than for fall through on A > B

ALU Logical Shift/Rotate

- Useful for isolating bit fields from words
  - Masking operation (e.g. extracting opcode bits from an instruction)
- Rotate right is same as rotate left but with differing shift counts
- Right shifts must handle both signed and unsigned
- Left shifts only need zero fill
- Shift timing
  - Execution depends on shift count – repeated single bit shift
  - Fast (constant time) shifts – done with barrel shifter (important for pipelining)

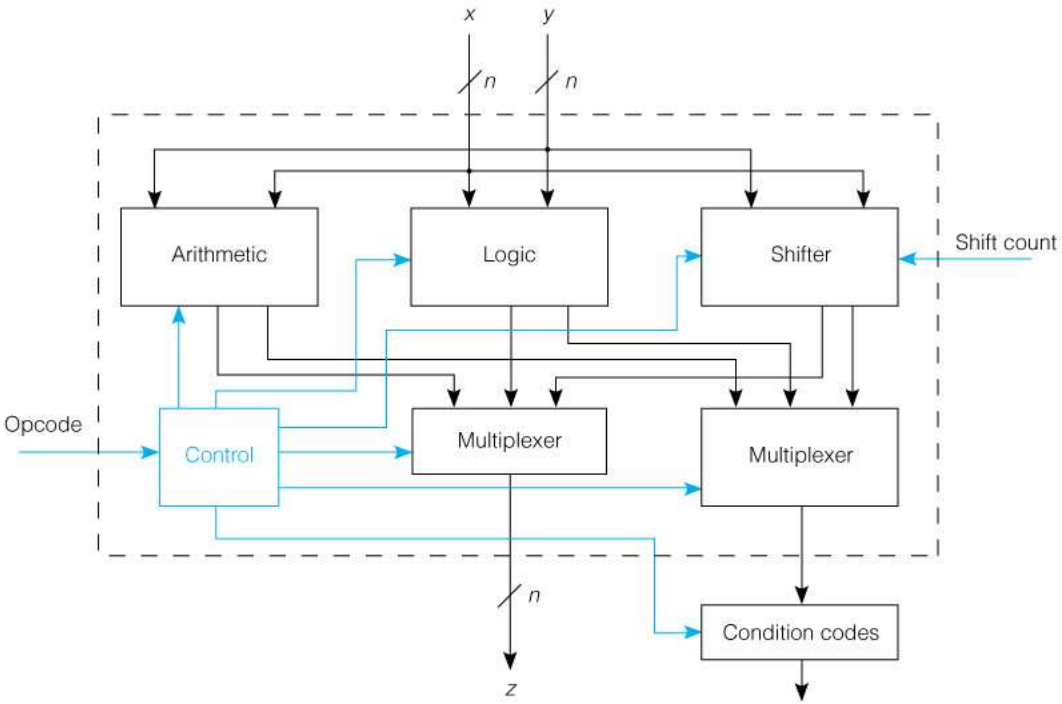
Crossbar Barrel Rotator



- 2 gate delay for any shift
- Each output line is an effective n-way multiplexer for shifts up to n bits
- n^2 tri-states needed
- How large is the decoder for 32 bit word?

Elements of an ALU

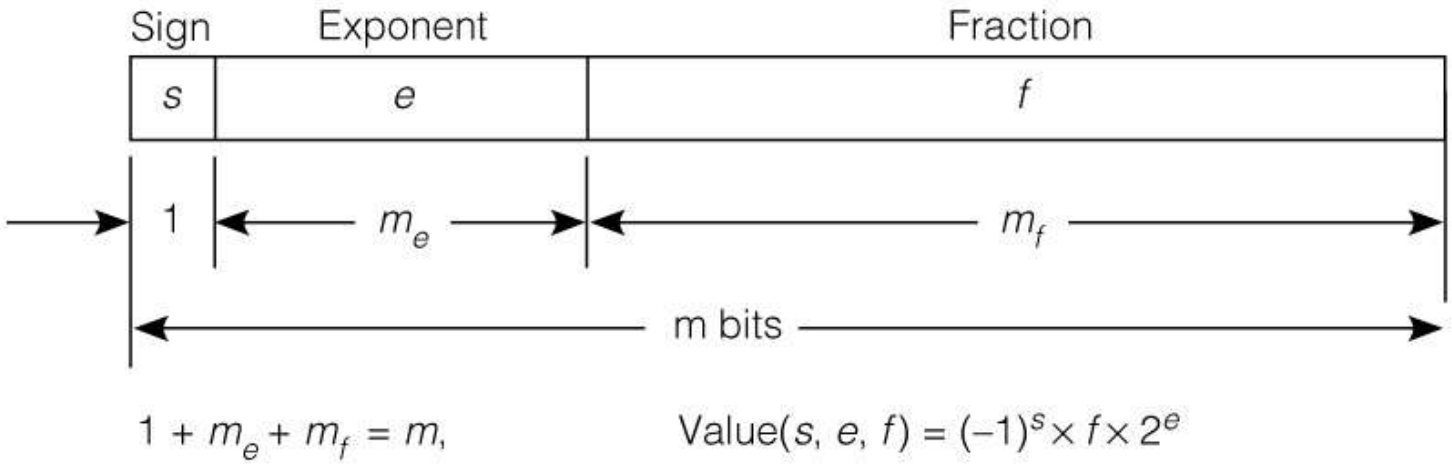
- Arithmetic hardware
  - Conditions codes may be produced
- Controller for multi-step operations (e.g. series parallel multiply)
- Shifter usually separate unit (many gates for speed)
- Logic operations typically simple
- Multiplexors select appropriate result and conditions codes



Floating Point (FP) Representations

- Decimal scientific notation (e.g. -2.72×10^{-2})
  - Sign (+/-)
  - Significand (2.72)
  - Exponent (-2)
- With fixed radix position, a separate scale factor e is assumed for number $f \times 2^e$
  - Addition/Subtraction simple (same scale)
    - $f \times 2^e + \mathrm{g} \times 2^e = (f + g) \times 2^e$
  - Multiplication/Divide more complex
    - $(f \times 2^e)(f \times 2e) = fg \times 2^{2e}$
    - $(f \times 2^e) \div (f \times 2e) = f \div g$

Floating Point Word



- s = sign
- e = exponent
- f = significand
  - Typically will be a fraction

Signs in FP Numbers

- Both significand and exponent have sign
- Significand is commonly represented by sign-magnitude
- Sign placed at left instead of with f so left most bit is always the sign bit
- Exponent is represented by bias (excess)
  - $-e_{min} \leq e \leq e_{max}, \quad e_{min}, e_{max} > 0$
  - $\hat{e} = e_{min} + e$
  - Positive exponent is helpful for comparison

Exponent Base and FP Range

- FP format with 24 of 32 bits for significand has 7 bits for exponent
  - Magnitude of a number x is in range $2^{-64} \leq x \leq 2^{64}$ (assuming base 2)
  - In order to have increased exponent range, bits taken from significand resulting in loss of accuracy
- IBM used base 16 for exponent for increased range in 360/370 series

Normalized FP

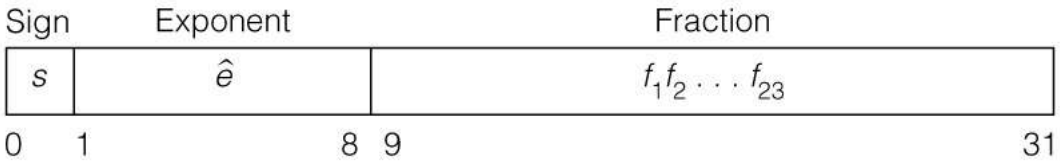
- Multiple representations of a FP exist
  - $f_2 = 2^d f_1$ define two fractions and $e_2 = e_1 - d$
  - $(\text{s}, f_1, e_1) = (\text{s}, f_2, e_2)$
  - E.g. $.819 \times 10^3 = .0819 \times 10^4$
- Normalized FP has leftmost signifcand digit non-zero (exponent as small as possible)
  - Zero handled separately as all 0 because it does not fit rule
- Base 2 has a hidden bit
  - Left most bit always = 1 therefore not necessary to waste space

Comparison of Normalized FP

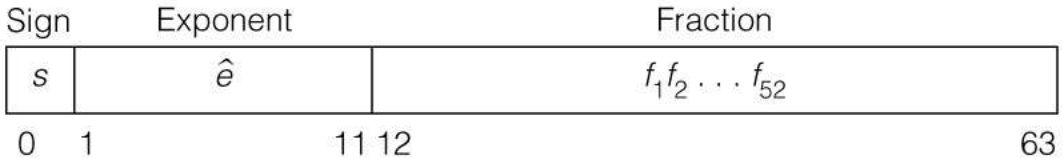
- Examining normalized FP word like an integer
  - Exponent field to left of significand field means an exponent unit is greater than a significand unit
  - Larger exponent field is greater than smaller exponent field
  - Significand ordering same as integer for fixed exponent
- FP numbers can be compared as if they were integers

IEEE FP Standards

- Single Precision (32-bits)



- Double precision (64-bits)



- Exponent bias (127 for SP, 1023 DP)
- Special numbers
  - All zero number is normalized 0
  - SP: 255 biased exponent indicate infinity or NaN
    - Non a number (NaN) e.g. 0/0

Decimal FP Add/Sub Examples

Operands	Alignment	Normalize & Round
6.144×10^{2}	0.06144×10^{4}	1.003644×10^{5}
$+ 9.975 \times 10^{4}$	$+ 9.97500 \times 10^{4}$	$+ 0.000500 \times 10^{5}$
	10.03644×10^{4}	1.004×10^{5}

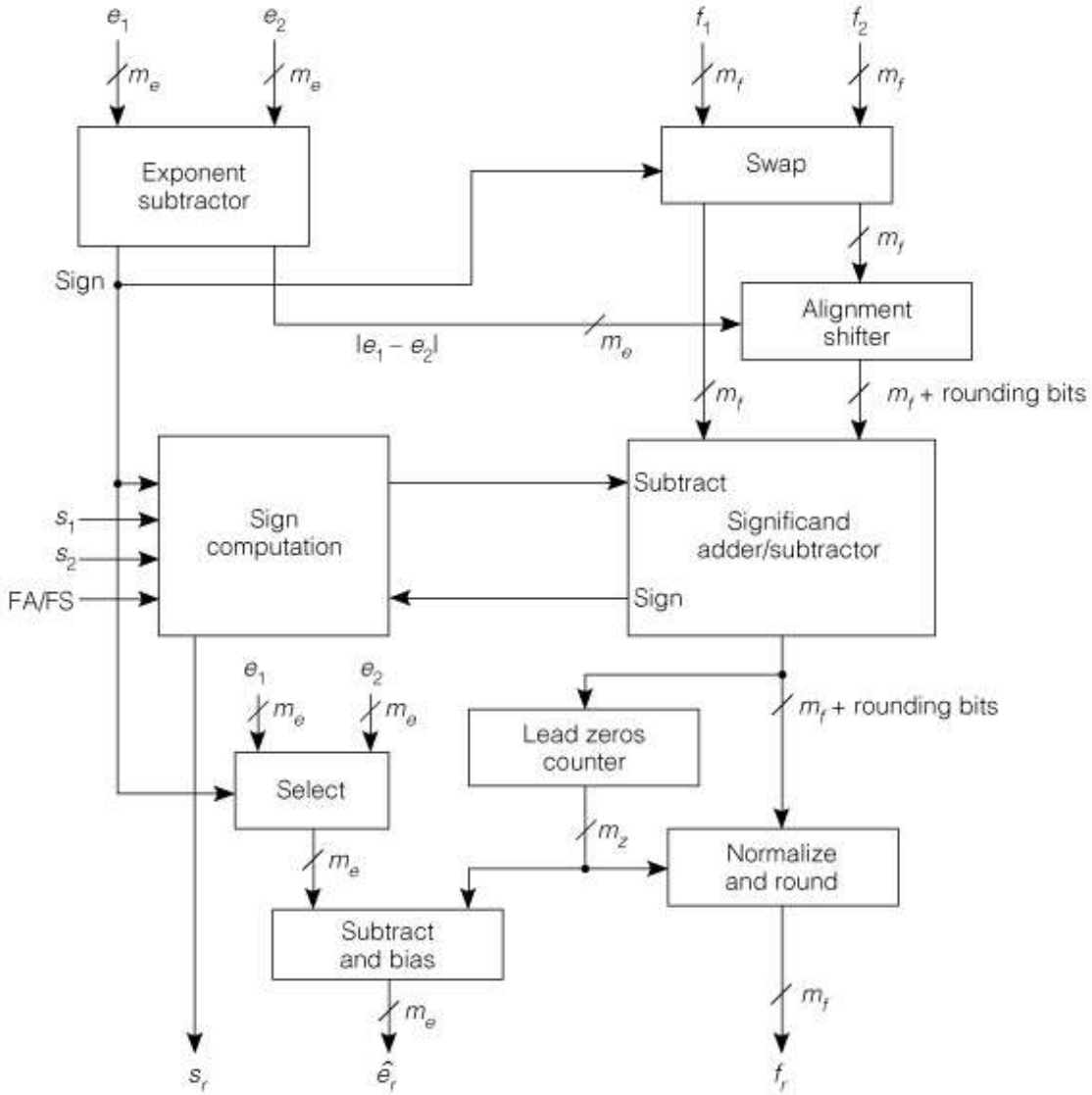
Operands	Alignment	Normalize & Round
1.076×10^{-7}	1.0760×10^{-7}	7.7300×10^{-9}
$- 9.987 \times 10^{-8}$	$- 0.9987 \times 10^{-7}$	$+ 0.0005 \times 10^{-9}$
	0.0773×10^{-7}	7.730×10^{-9}

Floating Add (FA) Floating Sub (FS)

- Add/subtract (s_1, f_1, e_1) and (s_2, f_2, e_2)

1. Unpack (s, e, f); handle special operands
2. Shift fraction of # with smaller exponent right by $|e_1-e_2|$ bits
3. Set result exponent $e_r = \max(e_1, e_2)$
4. For FA & $s_1=s_2$ or FS & $s_1 \neq s_2$, add significands, otherwise subtract them
5. Count lead zeros, z; carry can make z=-1; shift left z bits or right 1 bit if z=-1
6. Round result, shift right & adjust z if round OV
7. $e_r = e_r - z$; check over- or underflow; bias & pack

FP Add/Sub Hardware



- Adders for exponents and significands
- Shifters for alignment and normalize
- Multiplexers for exponent and swap of significands
- Lead zeros counter

Decimal FP Mult/Div Examples

- Multiply fractions and add exponents

Operands	Normalize & Round
-0.1403×10^{-3}	-0.4238463×10^{2}
$\times +0.3031 \times 10^{+6}$	-0.00005×10^{2}
$-0.04238463 \times 10^{-3+6}$	-0.4238×10^{2}

- Divide fractions and subtract exponents

Operands	Normalize & Round
$-0.9325 \times 10+^{2}$	$+0.9306387 \times 10^{9}$
$\div +0.1002 \times 10^{-6}$	$+0.00005 \times 10^{9}$
$+9.306387 \times 10^{2-(-6)}$	$+0.9306 \times 10^{9}$

Floating Multiply Steps

- Multiply $(s_r\ f_r, e_r) = (s_1, f_1, e_1) \times (s_2, f_2, e_2)$

1. Unpack (s, e, f); handle special operands
2. Compute $s_r = s_1 \oplus s_2$; $e_r = e_1 + e_2$; $f_r = f_1 \times f_2$
3. If necessary, normalize by 1 left shift & subtract 1 from e_r; round & shift right if round overflow (OV)
4. Handle overflow for exponent too positive and underflow for exponent too negative
5. Pack result, encoding or reporting exceptions

Floating Divide Steps

- Multiply $(s_r\ f_r, e_r) = (s_1, f_1, e_1) \div (s_2, f_2, e_2)$

1. Unpack (s, e, f); handle special operands
2. Compute $s_r = s_1 \oplus s_2$; $e_r = e_1 - e_2$; $f_r = f_1 \div f_2$
3. If necessary, normalize by 1 right shift & add 1 to e_r; round & shift right if round OV
4. Handle overflow for exponent too positive and underflow for exponent too negative
5. Pack result, encoding or reporting exceptions

FP Highlights

- FP generally have sign-magnitude significand and biased exponent representation
  - Exponent base is implicit
- FP Standards must specify the base, representation, and bit widths
- Normalization eliminates multiple representations of the same value for simple comparisons and arithmetic
- Arithmetic is composed of multiple fixed point operations on exponents and significands
- FA/FS are more difficult than mult/div because of the required exponent comparison and shifting to line up radix points
  - Mult/div requires a max of 1-bit shift of significand to normalize

Chapter 6 Summary

- Digital number representations and algebraic tools for the study of arithmetic
- Complement representation for addition of signed numbers
- Fast addition by large base & carry lookahead
- Fixed point multiply and divide overview
- Non-numeric aspects of ALU design
- Floating point number representations
- Procedures and hardware for float add & sub
- Floating multiply and divide procedures

Booth Recoding and Similar Methods

- Forms the basis for a number of signed multiplication algorithms
- Based upon recoding the multiplier, y, to a recoded value, z.
- The multiplicand remains unchanged.
- Uses signed digit (SD) encoding:
- Each digit can assume three values instead of just 2:
  - +1, 0, and -1,
  - encoded as 1, 0, and $\overline{1}$.
  - Known as signed digit (SD) notation.

A 2's Complement Integer's Value can be Represented as:

$$value(y) = -y_{m-1}2^{m-1} + \sum_{i=0}^{m-2} Y_i 2^i \qquad \text{(Eq 6.26}$$

This means that the value can be computed by *adding* the weighted values of all the digits except the most significant, and *subtracting* that digit.

Example: Represent -5 in SD Notation

$-5 = 1011$ in 2's Complement Notation

$1011 = \bar{1}011 = -8 + 0 + 2 + 1 = -5$ in SD Notation

The Booth Algorithm (Sometimes Known as "Skipping Over 1's.)

Consider $-1 = 1111$. In SD Notation this can be represented as $000\bar{1}$

The Booth method is:
1. Working from lsb to msb, replace each 0 digit of the original number with 0 in the recoded number until a 1 is encountered.
2. When a 1 is encountered, insert a $\bar{1}$ in that position in the recoded number, and skip over any succeeding 1's until a 0 is encountered.
3. Replace that 0 with a 1. If you encounter the msb without encountering a 0, stop and do nothing.

Example of Booth Recoding

$$0011 \quad 1101 \quad 1001 = 512 + 256 + 128 + 64 + 16 + 8 + 1 = 985$$

$$\downarrow \qquad \downarrow$$

$$0100 \quad 0\bar{1}10 \quad \bar{1}01\bar{1} = 1024 - 64 + 32 - 8 + 2 - 1 = 985$$

Tbl 6.4 Booth Recoding Table

Consider pairs of numbers, y_i, y_{i-1}. Recoded value is z_i.

y_i	y_{i-1}	z_i	*Value*	*Situation*
0	0	0	0	String of 0's
0	1	1	+1	End of string of 1's
1	0	$\bar{1}$	–1	Begin string of 1's
1	1	0	0	String of 1's

Algorithm can be done in parallel.
Examine the example of multiplication 6.11 in text.

Recoding using Bit Pair Recoding

- Booth method may actually increase number of multiplies.
- Consider pairs of digits, and recode each pair into 1 digit.
- Derive Table 6.5, pg. 279 on the blackboard to show how bit pair recoding works.
- Demonstrate Example 6.13 on the blackboard as an example of multiplication using bit pair recoding.
- There are many variants on this approach.

Table 6.5 Radix-4 Booth Encoding (Bit-Pair Encoding)

Original Bit Pair		Digit to Right	Recoded Bit Pair		Multiplier Value	Situation
y_i	y_{i-1}	y_{i-2}	z_i	z_{i-1}		
0	0	0		0	0	String of 0s
0	0	1		1	+1	End string of 1s
0	1	0		1	+1	Single 1
0	1	1	1		+2	End string of 1s
1	0	0	$\overline{1}$		–2	Begin string of 1s
1	0	1		$\overline{1}$	–1	Single 0
1	1	0		$\overline{1}$	–1	Begin string of 1s
1	1	1		0	0	String of 1s