Chapter 6 Overview

- Number Systems and Radix Conversion
- Fixed point arithmetic
- Seminumeric Aspects of ALU Design
- Floating Point Arithmetic

Digital Number Systems

- Digital number systems have a base or radix b
- Using positional notation, an m digit base b number is written

$$x = x_{m-1}\, x_{m-2} \ldots x_1\, x_0$$

$$0 \le x_i \le b-1,\ 0 \le i < m$$

- The value of this unsigned integer is

$$\text{value}(x) = \sum_{i=0}^{m-1} x_i \cdot b^i$$

Eq. 6.1

C
S
D
A
2/e

Range of Unsigned m Digit Base b Numbers

- The largest number has all of its digits equal to b-1, the largest possible base b digit
- Its value can be calculated in closed form

$$x_{max} = \sum_{i=0}^{m-1} (b-1)\cdot b^i = (b-1)\cdot \sum_{i=0}^{m-1} b^i = b^m - 1 \qquad \textbf{Eq. 6.2}$$

- **An important summation—geometric series**

$$\sum_{i=0}^{m-1} b^i = \frac{b^m - 1}{b - 1} \qquad \textbf{Eq. 6.3}$$

Radix Conversion: General Matters

- Converting from one number system to another involves computation
- We call the base in which calculation is done c and the other base b
- Calculation is based on the division algorithm
  - For integers a & b, there exist integers q & r such that $a = q \cdot b + r$, with $0 \le r \le b-1$
- Notation:

$$q = \lfloor a/b \rfloor$$

$$r = a \bmod b$$

Digit Symbol Correspondence Between Bases

- Each base has b (or c) different symbols to represent the digits
- If b < c, there is a table of b+1 entries giving base c symbols for each base b symbol & b
  - If the same symbol is used for the first b base c digits as for the base b digits, the table is implicit
- If c < b, there is a table of b+1 entries giving a base c number for each base b symbol & b
  - For base b digits ≥ c, the base c numbers have more than one digit

Base 12:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**10**
Base 3:	**0**	**1**	**2**	**10**	**11**	**12**	**20**	**21**	**22**	**100**	**101**	**102**	**110**

C
S
D
A
2/e

Convert Base b Integer to Calculator's Base, c

1) Start with base b $x = x_{m-1}\ x_{m-2}\ \ldots\ x_1\ x_0$
2) Set $x = 0$ in base c
3) Left to right, get next symbol x_i
4) Lookup base c number D_i for symbol x_i
5) Calculate in base c: $x = x \cdot b + D_i$
6) If there are more digits, repeat from step 3

- Example: convert $3AF_{16}$ to base 10

$$x = 0$$
$$x = 16x + 3 = 3$$
$$x = 16 \cdot 3 + 10(=A) = 58$$
$$x = 16 \cdot 58 + 15(=F) = 943$$

Convert Calculator’s Base Integer to Base b

1) Let x be the base c integer
2) Initialize i = 0 and v = x & get digits right to left
3) Set D_i = v mod b & v = $\lfloor v/b \rfloor$. Lookup D_i to get x_i
4) i = i + 1; If v $\neq$ 0, repeat from step 3

- Example: convert 3567_{10} to base 12

  3587 ÷ 12 = 298 (rem = 11) $\Rightarrow x_0$ = B

  298 ÷ 12 = 24 (rem = 10) $\Rightarrow x_1$ = A

  24 ÷ 12 = 2 (rem = 0) $\Rightarrow x_2$ = 0

  2 ÷ 12 = 0 (rem = 2) $\Rightarrow x_3$ = 2

  Thus $3587_{10} = 20AB_{12}$

Fractions and Fixed Point Numbers

- The value of the base b fraction $.f_{-1}f_{-2}...f_{-m}$ is the value of the integer $f_{-1}f_{-2}...f_{-m}$ divided by b^m
- The value of a mixed fixed point number

$$x_{n-1}x_{n-2}...x_1x_0.x_{-1}x_{-2}...x_{-m}$$

is the value of the n+m digit integer

$$x_{n-1}x_{n-2}...x_1x_0x_{-1}x_{-2}...x_{-m}$$

divided by b^m

- Moving radix point one place left divides by b
  - For fixed radix point position in word, this is a right shift of word
- Moving radix point one place right multiplies by b
  - For fixed radix point position in word, this is a left shift of word

Converting Fraction to Calculator's Base

- Can use integer conversion & divide result by b^m
- Alternative algorithm

  1) Let base b number be $.f_{-1}f_{-2}...f_{-m}$

  2) Initialize f = 0.0 and i = -m

  3) Find base c equivalent D of f_i

  4) f = (f + D)/b; i = i + 1

  5) If i = 0, the result is f. Otherwise repeat from 3
- Example: convert 413_8 to base 10

  f = (0 + 3)/8 = .375

  f = (.375 + 1)/8 = .171875

  f = (.171875 + 4)/8 = .521484375

Non-terminating Fractions

- The division in the algorithm may give a non-terminating fraction in the calculator's base
- This is a general problem: a fraction of m digits in one base may have any number of digits in another base
- The calculator will normally keep only a fixed number of digits
  - Number should make base c accuracy about that of base b
- This problem appears in generating base b digits of a base c fraction
  - The algorithm can continue to generate digits unless terminated

Convert Fraction from Calculator's Base to Base b

1) Start with exact fraction f in base c
2) Initialize i = 1 and v = f
3) $D_{-i} = \lfloor b \cdot v \rfloor$; $v = b \cdot v - D_{-i}$; Get base b f_{-i} for D_{-i}
4) i = i + 1; repeat from 3 unless v = 0 or enough base b digits have been generated

- Example: convert $.31_{10}$ to base 8

  $.31 \times 8 = 2.48 \Rightarrow f_{-1} = 2$

  $.48 \times 8 = 3.84 \Rightarrow f_{-2} = 3$

  $.84 \times 8 = 6.72 \Rightarrow f_{-1} = 6$

- Since $8^3 > 10^2$, $.236_8$ has more accuracy than $.31_{10}$

Conversion Between Related Bases by Digit Grouping

- Let base b = c^k; for example b = c^2
- Then base b number x_1x_0 is base c number $y_3y_2y_1y_0$, where x_1 base b = y_3y_2 base c and x_0 base b = y_1y_0 base c
- Examples: $102130_4 = 10\ 21\ 30_4 = 49C_{16}$

$$49C_{16} = 0100\ 1001\ 1100_2$$

$$102130_4 = 01\ 00\ 10\ 01\ 11\ 00_2$$

$$010010011100_2 = 010\ 010\ 011\ 100_2 = 2234_8$$

C
S
D
A
2/e

Negative Numbers, Complements, & Complement Representations

We will:

- Define two <u>complement operations</u>
- Define two <u>complement number systems</u>
  - Systems represent both positive and negative numbers
- Give a relation between complement and negate in a complement number system
- Show how to compute the complements
- Explain the relation between shifting and scaling a number by a power of the base
- Lead up to the use of complement number systems in signed addition hardware

Complement Operations— for m Digit Base b Numbers

- Radix complement of m digit base b number x

$$x^c = (b^m - x) \bmod b^m$$

- Diminished radix complement of x

$$\underline{x}^c = b^m - 1 - x$$

- The complement of a number in the range $0 \le x \le b^m-1$ is in the same range
- The mod b^m in the radix complement definition makes this true for x = 0; it has no effect for any other value of x
- Specifically, the radix complement of 0 is 0

Complement Number Systems

- Complement number systems use unsigned numbers to represent both positive and negative numbers
- Recall that the range of an m digit base b unsigned number is $0 \le x \le b^m - 1$
- The first half of the range is used for positive, and the second half for negative, numbers
- Positive numbers are simply represented by the unsigned number corresponding to their absolute value

Use of Complements to Represent Negative Numbers

- The complement of a number in the range from 0 to $b^m/2$ is in the range from $b^m/2$ to b^m-1
- A negative number is represented by the complement of its absolute value
- There are an equal number (±1) of positive and negative number representations
  - The ±1 depends on whether b is odd or even and whether radix complement or diminished radix complement is used
- We will assume the most useful case of even b
  - Then radix complement system has one more negative representation
  - Diminished radix complement system has equal numbers of positive and negative representations

Reasons to Use Complement Systems for Negative Numbers

- The usual sign-magnitude system introduces extra symbols + & - in addition to the digits
- In binary, it is easy to map 0⇒+ and 1⇒-
- In base b>2, using a whole digit for the two values + & - is wasteful
- Most important, however, it is easy to do signed addition & subtraction in complement number systems

Table 6.1 Complement Representations of Negative Numbers

Radix Complement		Diminished Radix Complement	
Number	**Representation**	**Number**	**Representation**
0	0	0	0 or b^m-1
$0<x<b^m/2$	x	$0<x<b^m/2$	x
$-b^m/2 \le x<0$	$\|x\|^c = b^m - \|x\|$	$-b^m/2<x<0$	$\underline{\|x\|}^c = b^m - 1 - \|x\|$

- For even b, radix comp. system represents one more negative than positive value
- while diminished radix comp. system has 2 zeros but represents same number of pos. & neg. values

Table 6.2 Base 2 Complement Representations

8 Bit 2's Complement		8 Bit 1's Complement	
Number	**Representation**	**Number**	**Representation**
0	0	0	0 or 255
$0<x<128$	x	$0<x<128$	x
$-128\leq x<0$	$256 - \lvert x\rvert$	$-127\leq x<0$	$255 - \lvert x\rvert$

- In 1's complement, $255 = 11111111_2$ is often called -0
- In 2's complement, $-128 = 10000000_2$ is a legal value, but trying to negate it gives overflow

Negation in Complement Number Systems

- Except for $-b^m/2$ in the b's comp. system, the negative of any m digit value is also m digits
- The negative of any number x, positive or negative, in the b's or b-1's complement system is obtained by applying the b's or b-1's complement operation to x, respectively
- The 2 complement operations are related by

$$x^c = (\underline{x}^c + 1) \bmod b^m$$

- Thus an easy way to compute one of them will give an easy way to compute both

Digitwise Computation of the Diminished Radix Complement

- Using the geometric series formula, the b-1's complement of x can be written

$$x^c = b^m - 1 - x = \sum_{i=0}^{m-1} (b-1) \cdot b^i - \sum_{i=0}^{m-1} x_i \cdot b^i$$

$$= \sum_{i=0}^{m-1} (b-1-x_i) \cdot b^i \qquad \textbf{Eq. 6.9}$$

- **If $0 \le x_i \le b-1$, then $0 \le (b-1-x_i) \le b-1$, so last formula is just an m digit base b number with each digit obtained from the corresponding digit of x**

Table Driven Calculation of Complements in Base 5

Base 5 Digit	4's Comp.
0	4
1	3
2	2
3	1
4	0

- 4's complement of 201341_5 is 243103_5
- 5's complement of 201341_5 is $243103_5 + 1 = 243104_5$
- 5's complement of 444445 is $00000_5 + 1 = 00001_5$
- 5's complement of 00000_5 is
- $(44444_5 + 1) \bmod 5^5 = 00000_5$

Complement Fractions

- Since m digit fraction is same as m digit integer divided by b^m, the b^m in complement definitions corresponds to 1 for fractions
- Thus radix complement of $x = .x_{-1}x_{-2}...x_{-m}$ is

  (1-x) mod 1, where mod 1 means discard integer
- The range of fractions is roughly -1/2 to +1/2
- This can be inconvenient for a base other than 2
- The b's comp. of a mixed number

$$x = x_{m-1}x_{m-2}...x_1x_0.x_{-1}x_{-2}...x_{-n} \quad \text{is} \quad b^m - x,$$

where both integer and fraction digits are subtracted

Scaling Complement Numbers by Powers of the Base

- Roughly, multiplying by b corresponds to moving radix point one place right or shifting number one place left
- Dividing by b roughly corresponds to a right shift of the number or a radix point move to the left one place
- There are 2 new issues for complement numbers

1) What is new left digit on right shift?

2) When does a left shift overflow?

Right Shifting a Complement Number to Divide by b

- For positive $x_{m-1}x_{m-2}...x_1x_0$, dividing by b corresponds to right shift with zero fill

$$0x_{m-1}x_{m-2}...x_1$$

- For negative $x_{m-1}x_{m-2}...x_1x_0$, dividing by b corresponds to right shift with b-1 fill

$$(b-1)x_{m-1}x_{m-2}...x_1$$

- This holds for both b's and b-1's comp. systems
- For even b, the rule is: fill with 0 if $x_{m-1} < b/2$ and fill with (b-1) if $x_{m-1} \geq b/2$

Complement Number Overflow on Left Shift to Multiply by b

- For positive numbers, overflow occurs if any digit other than 0 shifts off left end
- Positive numbers also overflow if the digit shifted into left position makes number look negative, i.e. digit ≥ b/2 for even b
- For negative numbers, overflow occurs if any digit other than b-1 shifts off left end
- Negative numbers also overflow if new left digit makes number look positive, i.e. digit<b/2 for even b

Left Shift Examples With Radix Complement Numbers

- Non-overflow cases:

  Left shift of $762_8 = 620_8$, -14_{10} becomes -112_{10}

  Left shift of $031_8 = 310_8$, 25_{10} becomes 200_{10}

- Overflow cases:

  Left shift of $241_8 = 410_8$ shifts $2 \neq 0$ off left

  Left shift of $041_8 = 410_8$ changes from + to -

  Left shift of $713_8 = 130_8$ changes from - to +

  Left shift of $662_8 = 620_8$ shifts $6 \neq 7$ off left

Fixed Point Addition and Subtraction

- If the radix point is in the same position in both operands, addition or subtraction act as if the numbers were integers
- Addition of signed numbers in radix complement system needs only an unsigned adder
- So we only need to concentrate on the structure of an m digit, base b unsigned adder
- To see this let x be a signed integer and rep(x) be its 2's complement representation
- The following theorem summarizes the result

Theorem on Signed Addition in a Radix Complement System

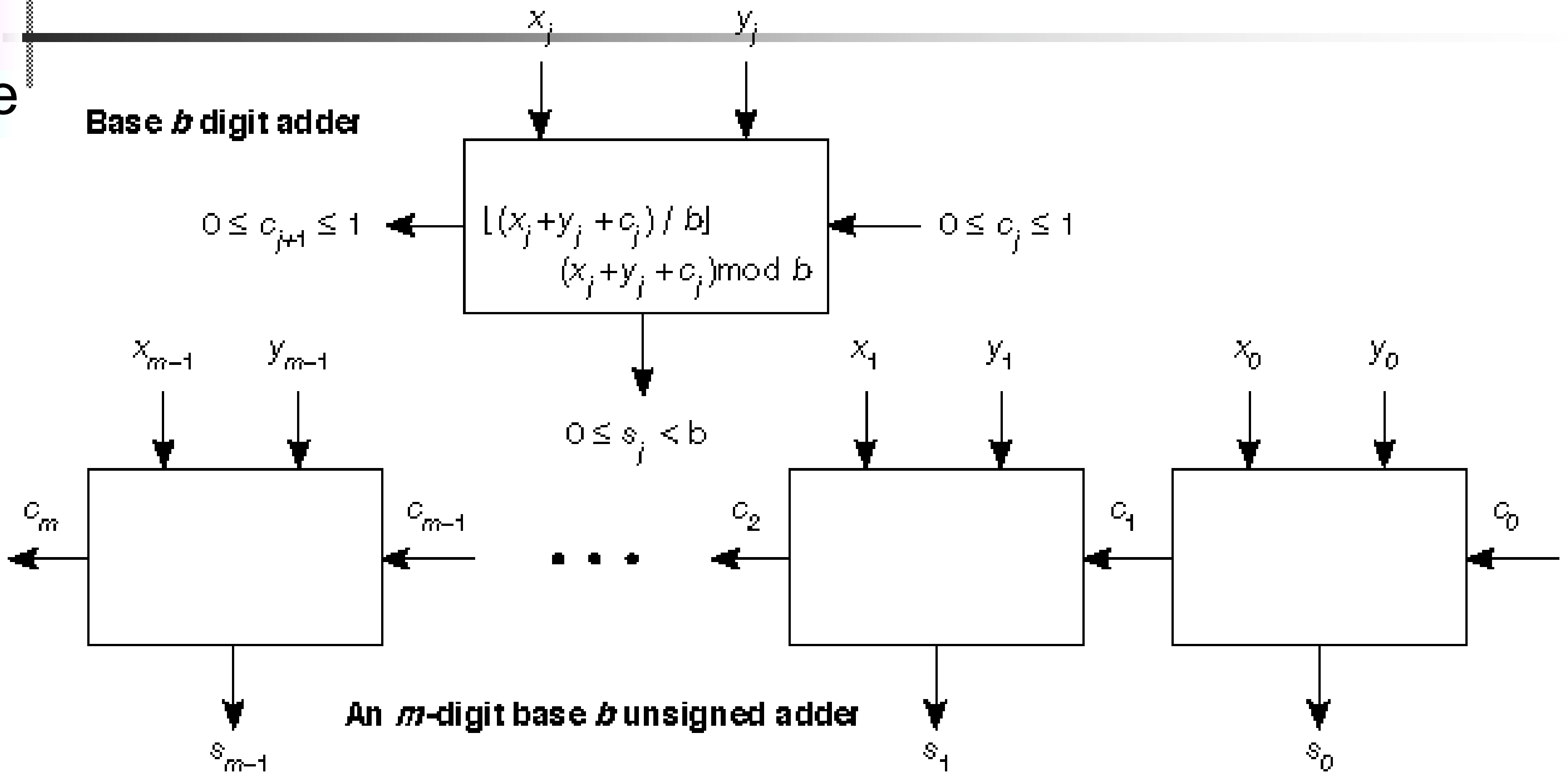
- Theorem: Let s be unsigned sum of rep(x) & rep(y). Then s = rep(x+y), except for overflow
- Proof sketch: Case 1, signs differ, $x \geq 0$, $y < 0$. Then $x+y = x-|y|$ and $s = (x+b^m-|y|) \bmod b^m$.

  If $x-|y| \geq 0$, mod discards b^m, giving result, if

  $x-|y| < 0$, then $\text{rep}(x+y) = (b-|\,x-|y|\,|) \bmod b^m$.

  Case 3, $x<0$, $y<0$. $s = (2b^m - |x| - |y|) \bmod b^m$, which reduces to $s = (b^m - |x+y|) \bmod b^m$. This is rep(x+y) provided the result is in range of an m digit b's comp. representation. If it is not, the unsigned $s<b^m/2$ appears positive.

Fig. 6.1 An m-digit base B unsigned adder



- Typical cell produces $s_j = (x_j + y_j + c_j) \bmod b$ and $c_{j+1} = \lfloor (x_j + y_j + c_j)/b \rfloor$
- Since $x_j, y_j \le b-1$, $c_j \le 1$ implies $c_{j+1} \le 1$, and since $c_0 \le 1$, all carries are ≤ 1, regardless of b

Unsigned Addition Examples

	12.03_4	$= 6.1875_{10}$	$.9A2C_{16}$	
	13.21_4	$= 7.5625_{10}$	$.7BE2_{16}$	Overflow
Carry	01 01		1 11 0	for 16 bit
Sum	31.30_4	$= 13.75_{10}$	$1.160E_{16}$	word

- If result can only have a fixed number of bits, overflow occurs on carry from leftmost digit
- Carries are either 0 or 1 in all cases
- A table of sum and carry for each of the b^2 digit pairs, and one for carry in = 1, define the addition

Base 4

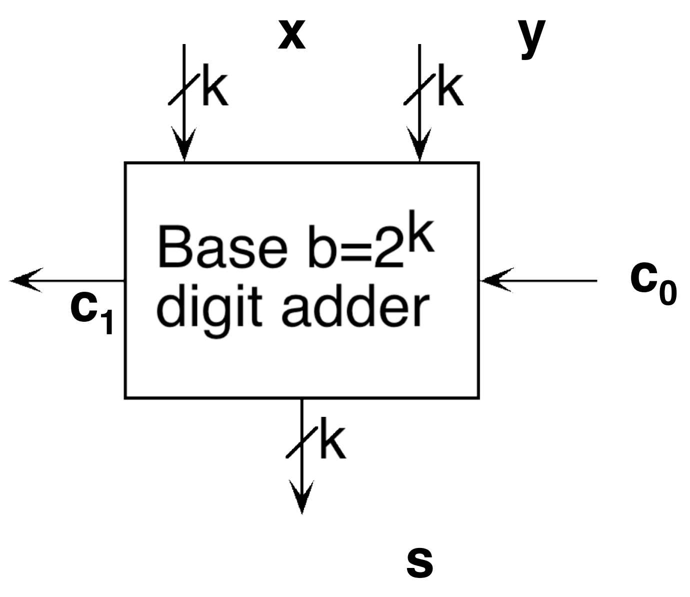
+	0	1	2	3
0	00	01	02	03
1	01	02	03	10
2	02	03	10	11
3	03	10	11	12

Implementation Alternatives for Unsigned Adders

- If $b = 2^k$, then each base b digit is equivalent to k bits
- A base b digit adder can be viewed as a logic circuit with 2k+1 inputs and k+1 outputs

Fig 6.1a

- **This combinational logic circuit can be designed with as few as 2 levels of logic**
- **PLA, ROM, and multi-level logic are also alternatives**
- **If 2 level logic is used, max. gate delays for m digit base b unsigned adder is 2m**



Two Level Logic Design of a Base 4 Digit Adder

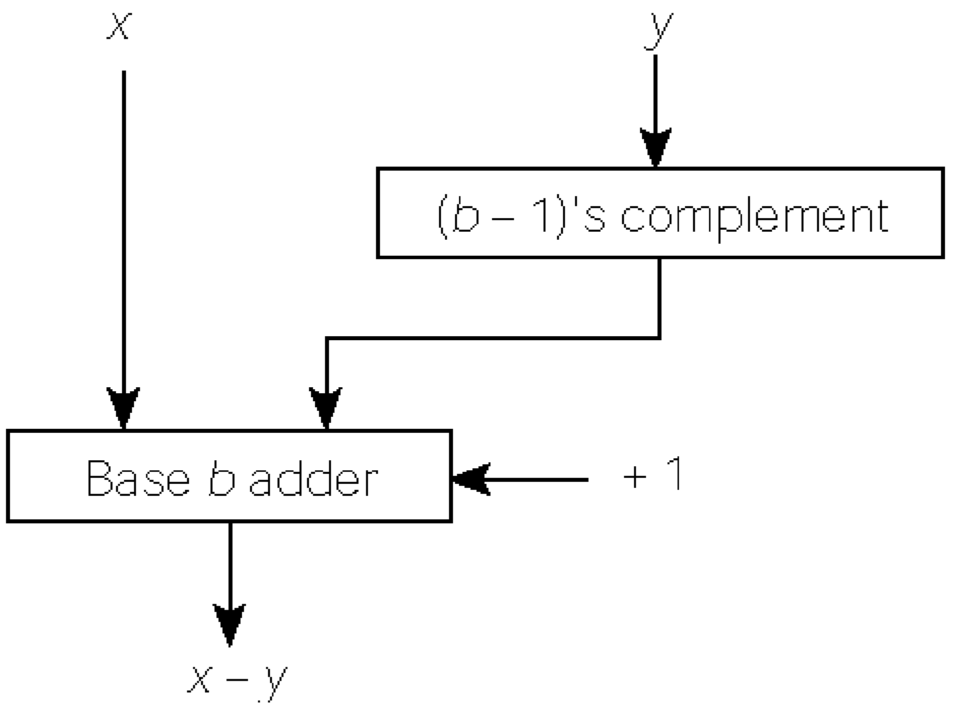
x_b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
x_a	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
y_b	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
y_a	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
c_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
c_1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1
s_b	0	0	0	1	1	1	1	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	0	0	0	0	1	1	1
s_a	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1

- The base 4 digit x is represented by the 2 bits x_b x_a, y by y_b y_a, and s by s_b s_a
- s_a is independent of x_b and y_b, c_1 is given by $y_by_ac_0+x_ay_bc_0+x_bx_ac_0+x_by_ac_0+x_bx_ay_a+x_ay_by_a+x_by_b$,

while s_b is a 12 input OR of 4 input ANDs

Fig. 6.2 base b Complement Subtracter

- To do subtraction in the radix complement system, it is only necessary to negate (radix complement) the 2nd operand
- It is easy to take the diminished radix complement, and the adder has a carry in for the +1



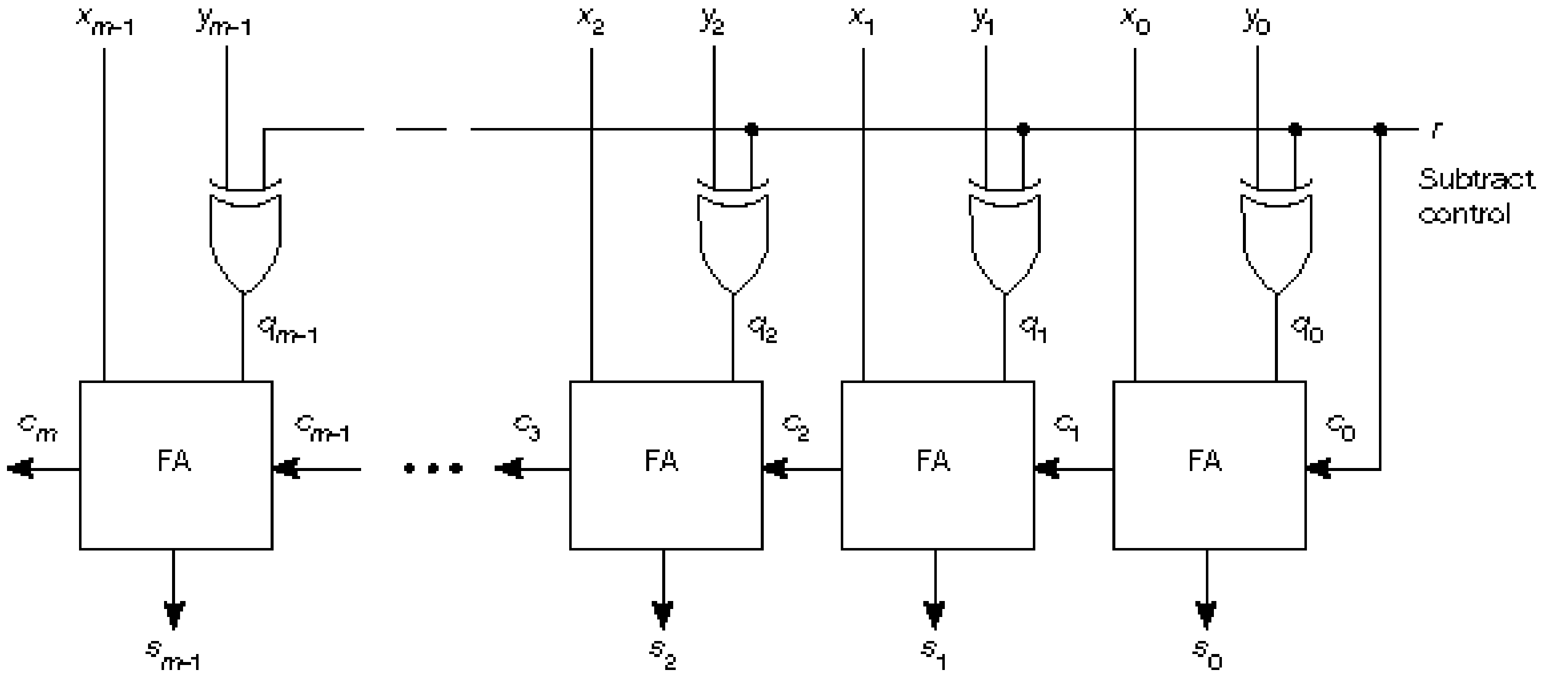
Overflow Detection in Complement Add & Subtract

- We saw that all cases of overflow in complement addition came when adding numbers of like signs, and the result seemed to have the opposite sign
- For even b, the sign can be determined from the left digit of the representation
- Thus an overflow detector only needs x_{m-1}, y_{m-1}, s_{m-1}, and an add/subtract control
- It is particularly simple in base 2

CSDA 2/e

Fig. 6.3 2’s Complement Adder/Subtracter

- A multiplexer to select y or its complement becomes an exclusive OR gate



Speeding Up Addition With Carry Lookahead

- Speed of digital addition depends on carries
- A base $b = 2^k$ divides length of carry chain by k
- Two level logic for base b digit becomes complex quickly as k increases
- If we could compute the carries quickly, the full adders compute result with 2 more gate delays
- Carry lookahead computes carries quickly
- It is based on two ideas:

—a digit position generates a carry

—a position propagates a carry in to the carry out

Binary Propagate and Generate Signals

- In binary, the generate for digit j is $G_j = x_j \cdot y_j$
- Propagate for digit j is $P_j = x_j + y_j$
  - Of course x_j+y_j covers $x_j \cdot y_j$ but it still corresponds to a carry out for a carry in
- Carries can then be written: $c_1 = G_0 + P_0 \cdot c_0$
- $c_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0$
- $c_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0$
- $c_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0$
- In words, the c_2 logic is: c_2 is one if digit 1 generates a carry, or if digit 0 generates one and digit 1 propagates it, or if digits 0&1 both propagate a carry in

Speed Gains With Carry Lookahead

- It takes one gate to produce a G or P, two levels of gates for any carry, & 2 more for full adders
- The number of OR gate inputs (terms) and AND gate inputs (literals in a term) grows as the number of carries generated by lookahead
- The real power of this technique comes from applying it recursively
- For a group of, say 4, digits an overall generate is $G^1_0 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0$
- An overall propagate is $P^1_0 = P_3 \cdot P_2 \cdot P_1 \cdot P_0$

Recursive Carry Lookahead Scheme

- If level 1 generates G^1_j and propagates P^1_j are defined for all groups j, then we can also define level 2 signals G^2_j and P^2_j over groups of groups
- If k things are grouped together at each level, there will be $\log_k m$ levels, where m is the number of bits in the original addition
- Each extra level introduces 2 more gate delays into the worst case carry calculation
- k is chosen to trade-off reduced delay against the complexity of the G and P logic
- It is typically 4 or more, but the structure is easier to see for k=2

Fig. 6.4 Carry Lookahead Adder for Group Size k = 2

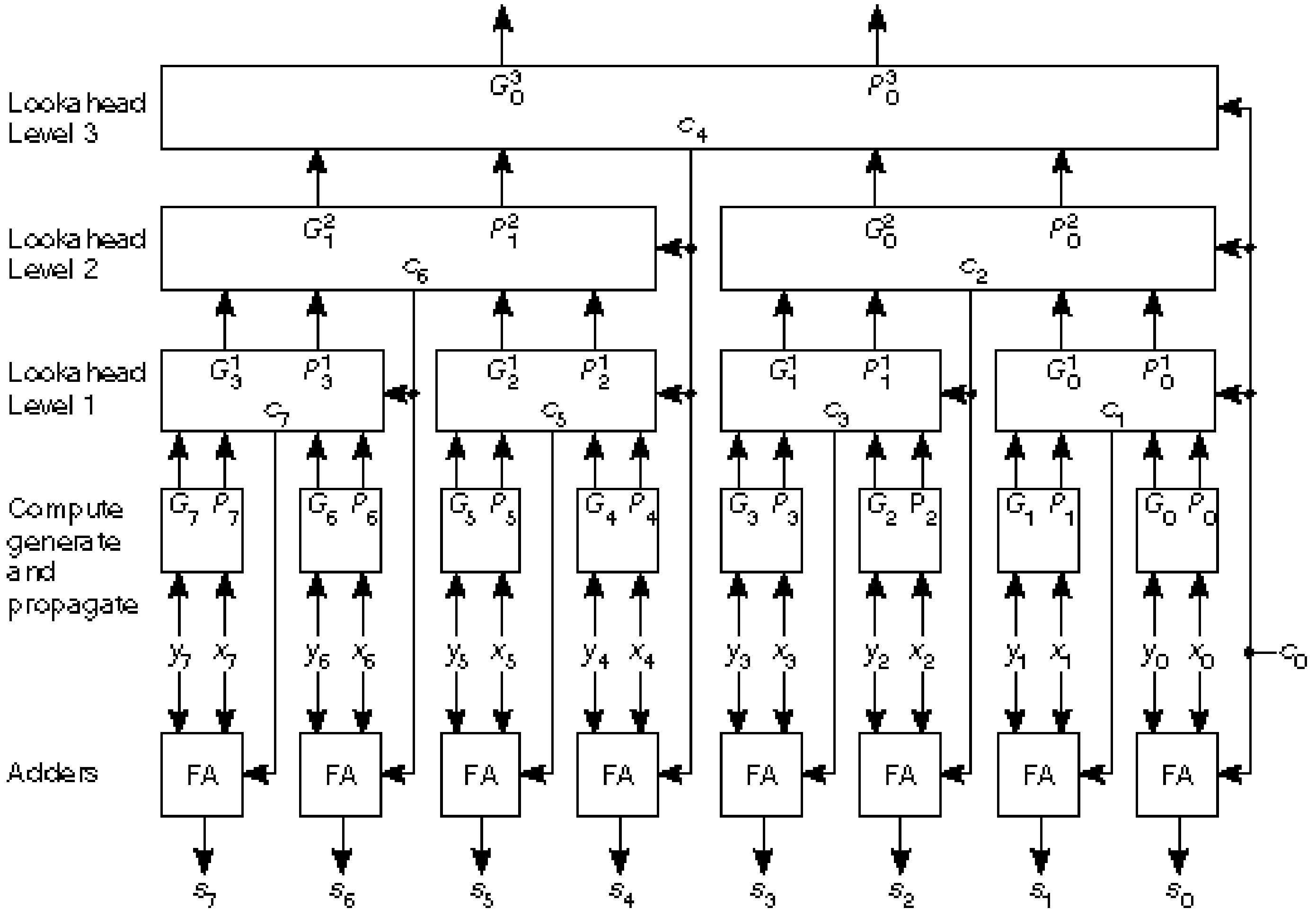


Fig. 6.5 Digital Multiplication Schema

$$
\begin{array}{cccccccccl}
 & & & & x_3 & x_2 & x_1 & x_0 & & \text{multiplicand} \\
 & & & & y_3 & y_2 & y_1 & y_0 & & \text{multiplier} \\
\hline
 & & & (xy_0)_4 & (xy_0)_3 & (xy_0)_2 & (xy_0)_1 & (xy_0)_0 & & pp_0 \\
 & & (xy_1)_4 & (xy_1)_3 & (xy_1)_2 & (xy_1)_1 & (xy_1)_0 & & & pp_1 \\
 & (xy_2)_4 & (xy_2)_3 & (xy_2)_2 & (xy_2)_1 & (xy_2)_0 & & & & pp_2 \\
(xy_3)_4 & (xy_3)_3 & (xy_3)_2 & (xy_3)_1 & (xy_3)_0 & & & & & pp_3 \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 & &
\end{array}
$$

p: product

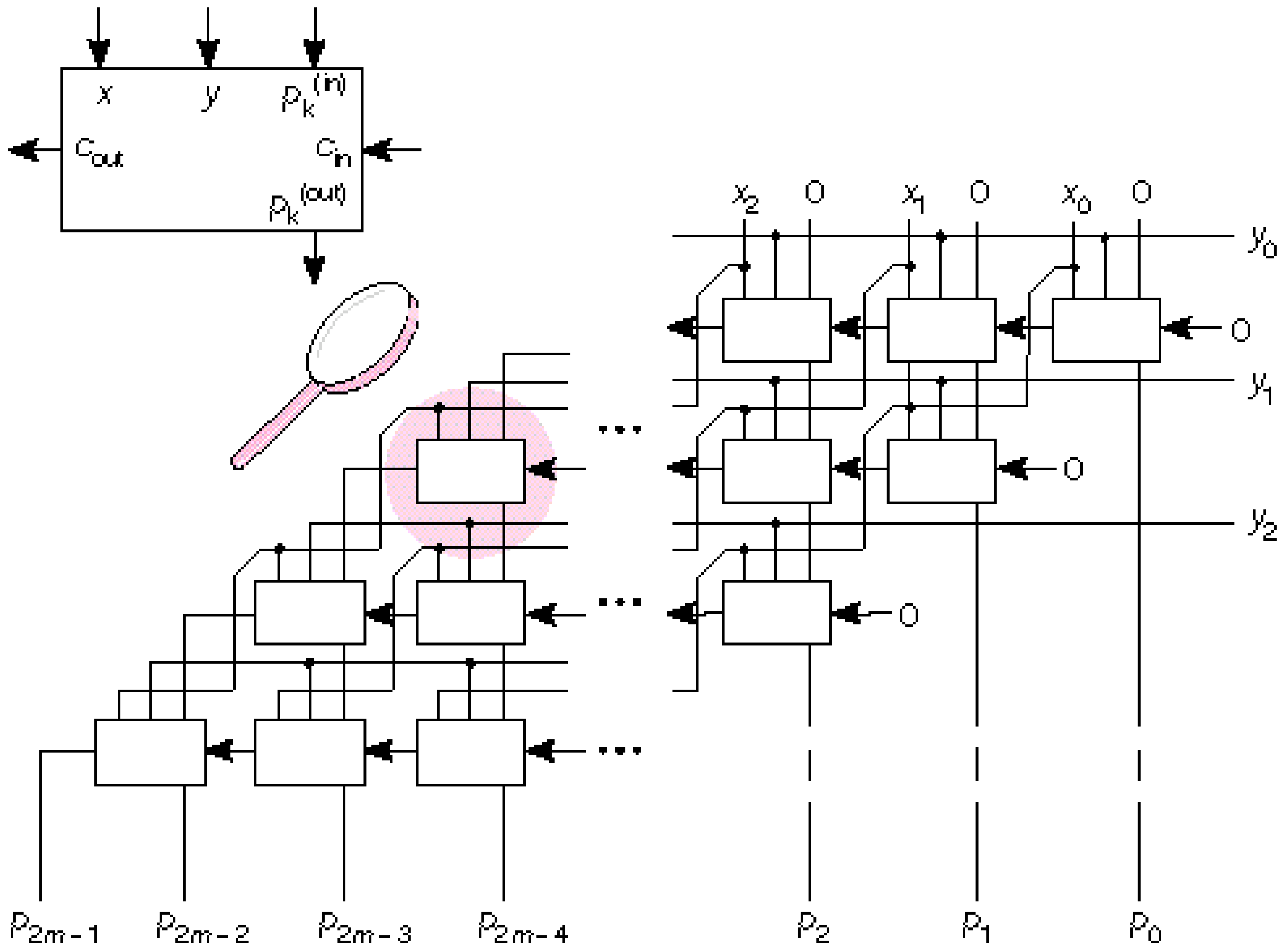
pp: partial product

Serial By Digit of Multiplier, Then By Digit of Multiplicand

```
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 := 1 step 1 until m-1
7.                  begin
8.                      $p_{j+i}$ := ($p_{j+i}$ + $x_i$ $y_j$ + c) mod b;
9.                      c := $\lfloor(p_{j+i} + x_i\ y_j + c)/b\rfloor$;
10.                 end;
11.             $p_{j+m}$ := c;
12.         end;
```

- If $c \leq b-1$ on the RHS of 9, then $c \leq b-1$ on the LHS of 9 because $0 \leq p_{j+i}, x_i, y_j \leq b-1$

Fig. 6.6 Parallel Array Multiplier for Unsigned Base b Numbers



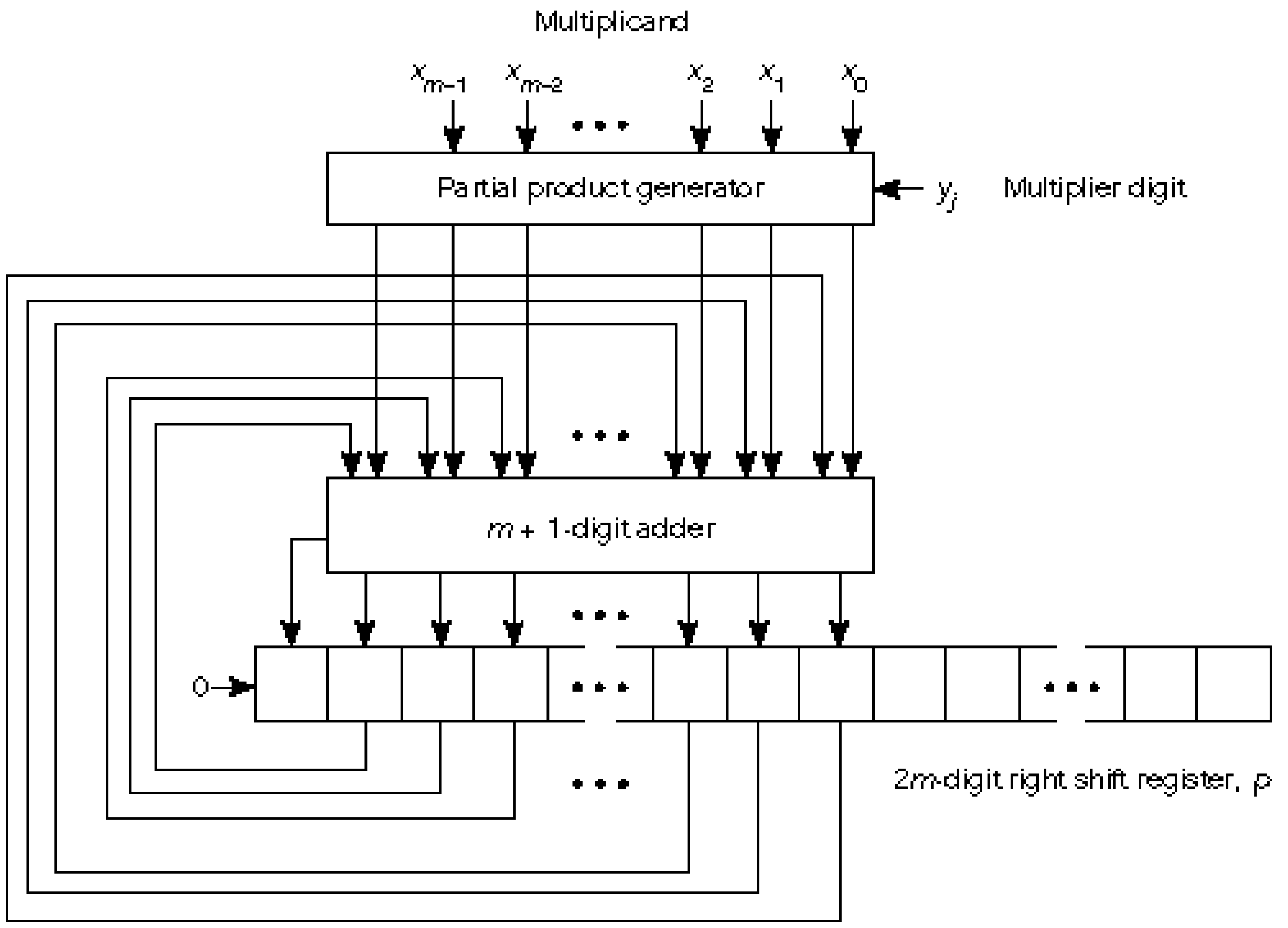
Operation of the Parallel Multiplier Array

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

Series Parallel Multiplication Algorithm

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

Fig. 6.7 Unsigned Series-Parallel Multiplication Hardware



Steps for Using the Unsigned Series-Parallel Multiplier

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.

Multiply with Fixed Length Words: Integer and Fraction Multiply

- If words can store only m digits, and the radix point is in a fixed position in the word, 2 positions make sense

  integer: right end, and fraction: left end
- In integer multiply, overflow occurs if any of the upper m digits of the 2m digit product ≠0
- In fraction multiply, the upper m digits are the most significant, and the lower m digits are discarded or rounded to give an m digit fraction

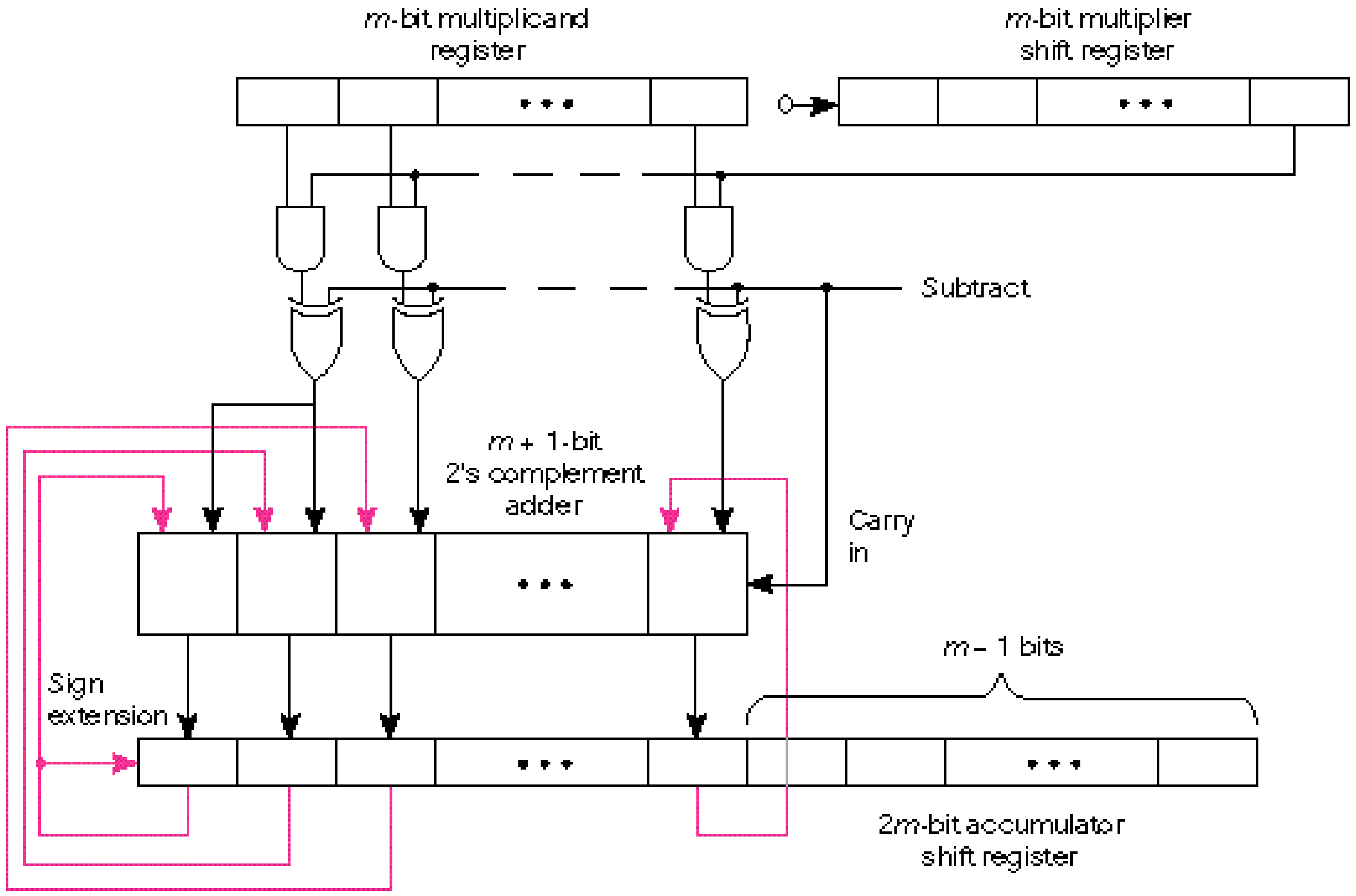
Signed Multiplication

- The sign of the product can be computed immediately from the signs of the operands
- For complement numbers, negative operands can be complemented, their magnitudes multiplied, and the product recomplemented if necessary
- A complement representation multiplicand can be handled by a b's complement adder for partial products and sign extension for the shifts
- A 2's complement multiplier is handled by the formula for a 2's complement value: add all PP's except last, subtract it.

$$\text{value}(x) = -x_{m-1}2^{m-1} + \sum_{i=0}^{m-2} x_i 2^i \qquad \text{Eq. 6.25}$$

CSDA 2/e

Fig. 6.8 2's Complement Signed Multiplier Hardware



m-bit multiplicand register
m-bit multiplier shift register
0
Subtract
m + 1-bit 2's complement adder
Carry in
m – 1 bits
Sign extension
2*m*-bit accumulator shift register

Steps for Using the 2's Complement Multiplier Hardware

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

C
S
D
A
2/e

Examples of 2’s Complement Multiplication

-5/8=	1. 0 1 1	6/8=	0. 1 1 0
× 6/8=	× 0. 1 1 0	×-5/8=	× 1. 0 1 1
pp_0	0 0. 0 0 0	pp_0	0 0. 1 1 0
acc.	0 0. 0 0 0 0	acc.	0 0. 0 1 1 0
pp_1	1 1. 0 1 1	pp_1	0 0. 1 1 0
acc.	1 1. 1 0 1 1 0	acc.	0 0. 1 0 0 1 0
pp_2	1 1. 0 1 1	pp_2	0 0. 0 0 0
acc.	1 1. 1 0 0 0 1 0	acc.	0 0. 0 1 0 0 1 0
pp_3	0 0. 0 0 0	pp_3	1 1. 0 1 0
res.	1 1. 1 0 0 0 1 0	res.	1 1. 1 0 0 0 1 0

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 1. This is 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

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

$$\qquad\quad \downarrow \qquad\qquad \downarrow$$

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

CSDA 2/e

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.

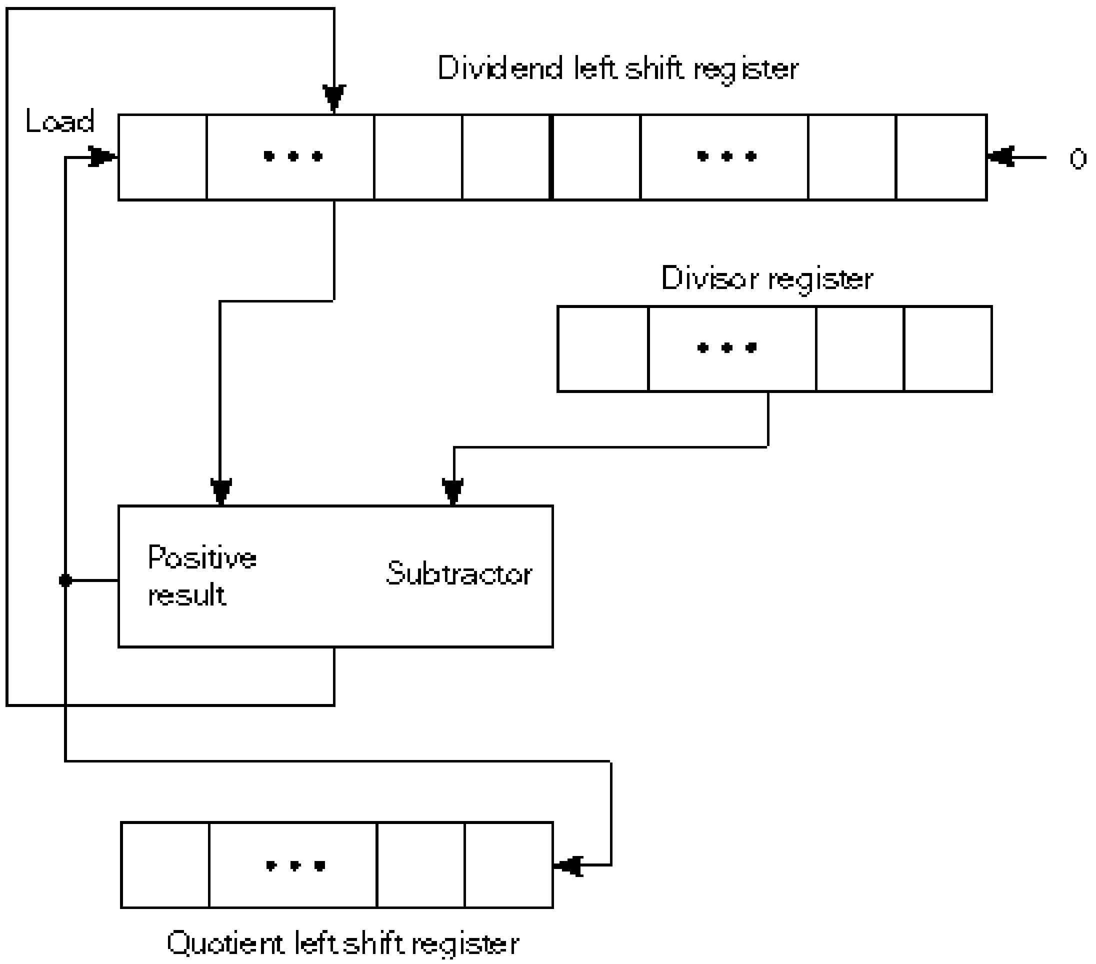
Table 6.5 Radix-4 Booth Encoding (Bit-Pair Encoding)

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	1		–2	Begin string of 1s
1	0	1		1	–1	Single 0
1	1	0		1	–1	Begin string of 1s
1	1	1		0	0	String of 1s

Digital Division: Terminology and Number Sizes

- A <u>dividend</u> is divided by a <u>divisor</u> to get a <u>quotient</u> and a <u>remainder</u>
- A 2m digit dividend divided by an m digit divisor does <u>not</u> necessarily give an m digit quotient and remainder
- If the divisor is 1, for example, an integer quotient is the same size as the dividend
- If a fraction D is divided by a fraction d, the quotient is only a fraction if $D<d$
- If $D \geq d$, a condition called <u>divide overflow</u> occurs in fraction division

Fig 6.9 Unsigned Binary Division Hardware



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

Use of Division Hardware for Integer Division

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.

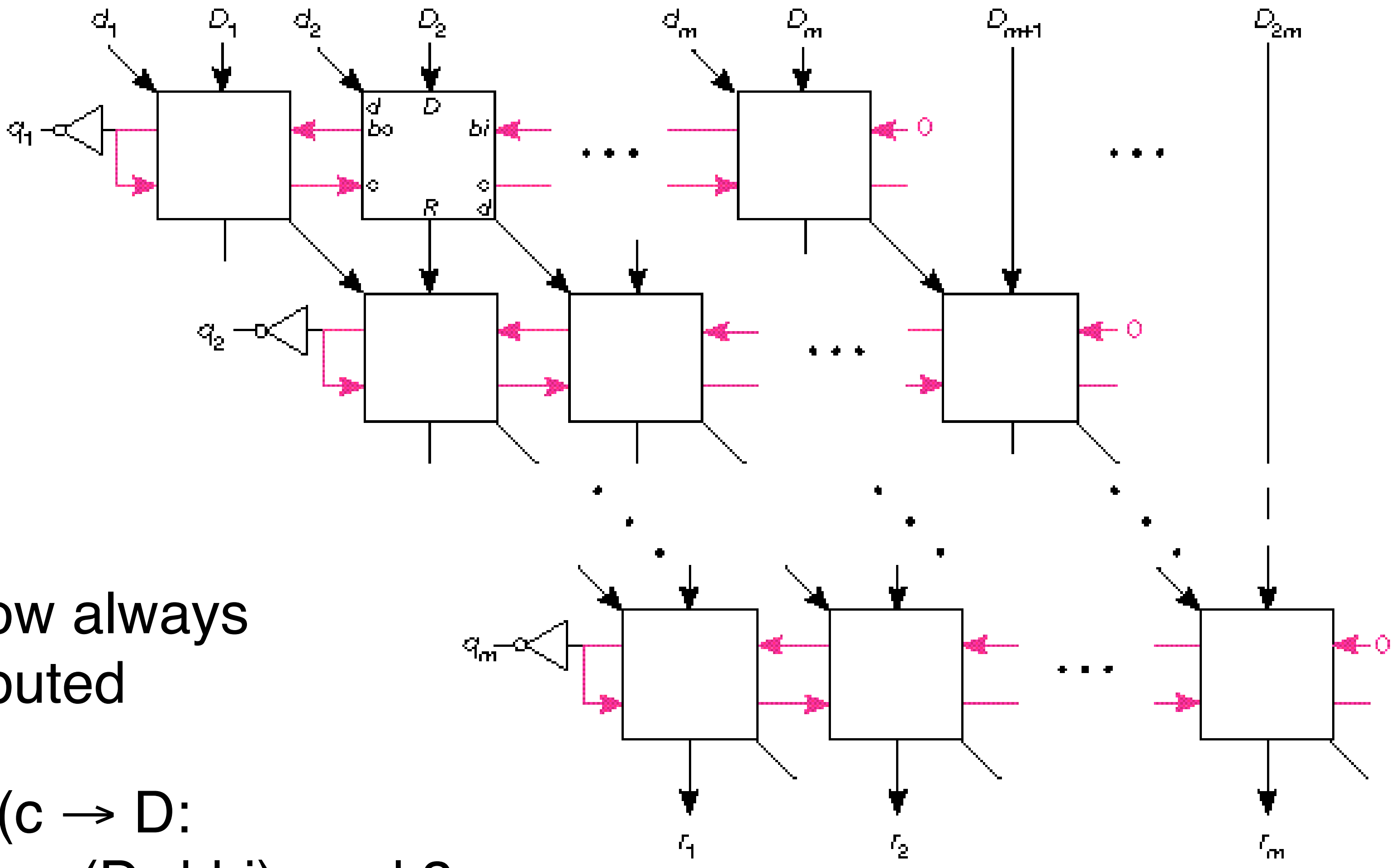
Use of Division Hardware for Fraction Division

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.

Integer Binary Division Example: D=45, d=6, q=7, r=3

	D	0 0 0 0 0 0 1 0 1 1 0 1		
	d	0 0 0 1 1 0		
Init.	D	0 0 0 0 0 1 0 1 1 0 1 -		
	d	0 0 0 1 1 0		
diff(-)	D	0 0 0 0 1 0 1 1 0 1 - -	q	0
	d	0 0 0 1 1 0		
diff(-)	D	0 0 0 1 0 1 1 0 1 - - -	q	0 0
	d	0 0 0 1 1 0		
diff(-)	D	0 0 1 0 1 1 0 1 - - - -	q	0 0 0
	d	0 0 0 1 1 0		
diff(+)	D	0 0 1 0 1 0 1 - - - - -	q	0 0 0 1
	d	0 0 0 1 1 0		
diff(+)	D	0 0 1 0 0 1 - - - - - -	q	0 0 0 1 1
	d	0 0 0 1 1 0		
diff(+)	rem.	0 0 0 0 1 1	q	0 0 0 1 1 1

Fig 6.10 Parallel Array Divider



Borrow always computed

R := (c → D:
¬c → (D-d-bi) mod 2

Branching on Arithmetic Conditions

- An ALU with two m bit operands produces more than just an m bit result
- The carry from the left bit and the true/false value of 2’s complement overflow are useful
- There are 3 common ways of using outcome of compare (subtract) for a branch condition

1) Do the compare in the branch instruction

2) Set special condition code bits and test them in the branch

3)Set a general register to a comparison outcome and branch on this logical value

Drawbacks of Condition Codes

- Condition codes are extra processor state; set, and overwritten, by many instructions
- Setting and use of CCs also introduces hazards in a pipelined design
- CCs are a scarce resource, they must be used before being set again
  - The PowerPC has 8 sets of CC bits
- CCs are processor state that must be saved and restored during exception handling

C S D A 2/e

Drawbacks of Comparison in Branch and Set General Register

- Branch instruction length: it must specify 2 operands to be compared, branch target, and branch condition (possibly place for link)
- Amount of work before branch decision: it must use the ALU and test its output—this means more branch delay slots in pipeline
- Setting a general register to a particular outcome of a compare, say ≤ unsigned, uses a register of 32 or more bits for a true/false value

CSDA 2/e

Use of Condition Codes: Motorola 68000

- The HLL statement:

  if (A > B) then C = D

  translates to the MC68000 code:

For 2's comp. A & B	For unsigned A & B
`MOVE.W A, D0`	`MOVE.W A, D0`
`CMP.W B, D0`	`CMP.W B, D0`
`BLE Over`	`BLS Over`
`MOVE.W D, C`	`MOVE.W D, C`
`Over: . . .`	`Over: . . .`

CSDA 2/e

Standard Condition Codes: NZVC

- Assume compare does the subtraction s = x-y
- N: negative result, $s_{m-1} = 1$ if $x_{m-1} y_{m-1} \overline{s}_{m-1} + \overline{x}_{m-1} \overline{y}_{m-1} s_{m-1}$
- Z: zero result, s = 0
- V: 2's comp. overflow, C: carry from leftm~~os~~t bit~~ positio~~n, $s_m = 1$
- Information in N, Z, V, and C determines several signed & unsigned relations of x & y

Correspondence of Conditions and NZVC Bits

Condition	Unsigned Integers	Signed Integers
carry out	C	C
overflow	C	V
negative	n.a.	N
$>$	$\overline{C}\cdot\overline{Z}$	$(N\cdot V+\overline{N}\cdot\overline{V})\cdot\overline{Z}$
$\geq$	$\overline{C}$	$N\cdot V+\overline{N}\cdot\overline{V}$
$=$	Z	Z
$\neq$	$\overline{Z}$	$\overline{Z}$
$\leq$	$C+Z$	$(N\cdot\overline{V}+\overline{N}\cdot V)+Z$
$<$	C	$N\cdot\overline{V}+\overline{N}\cdot V$

C S D A 2/e

Branches That Do Not Use Condition Codes

- SRC compares a single number to zero
- The simple comparison can be completed in pipeline stage 2
- The MIPS R2000 compares 2 numbers using a branch of the form: `bgtu R1, R2, Lbl`
- Different branch instructions are needed for each signed or unsigned condition
- The MIPS R2000 also allows setting a general register to 1 or 0 on a compare outcome

```
sgtu R3, R1, R2
```

ALU Logical, Shift and Rotate Instructions

- Shifts are often combined with logic to extract bit fields from, or insert them into, full words
- A MC68000 example extracts bits 30..23 of a 32 bit word (exponent of a floating point #)

```
MOVE.L D0, D1    ;Get # into D1
ROL.L  #9, D1    ;exponent to bits 7..0
ANDI.L #FFH, D1  ;clear bits 31..8
```

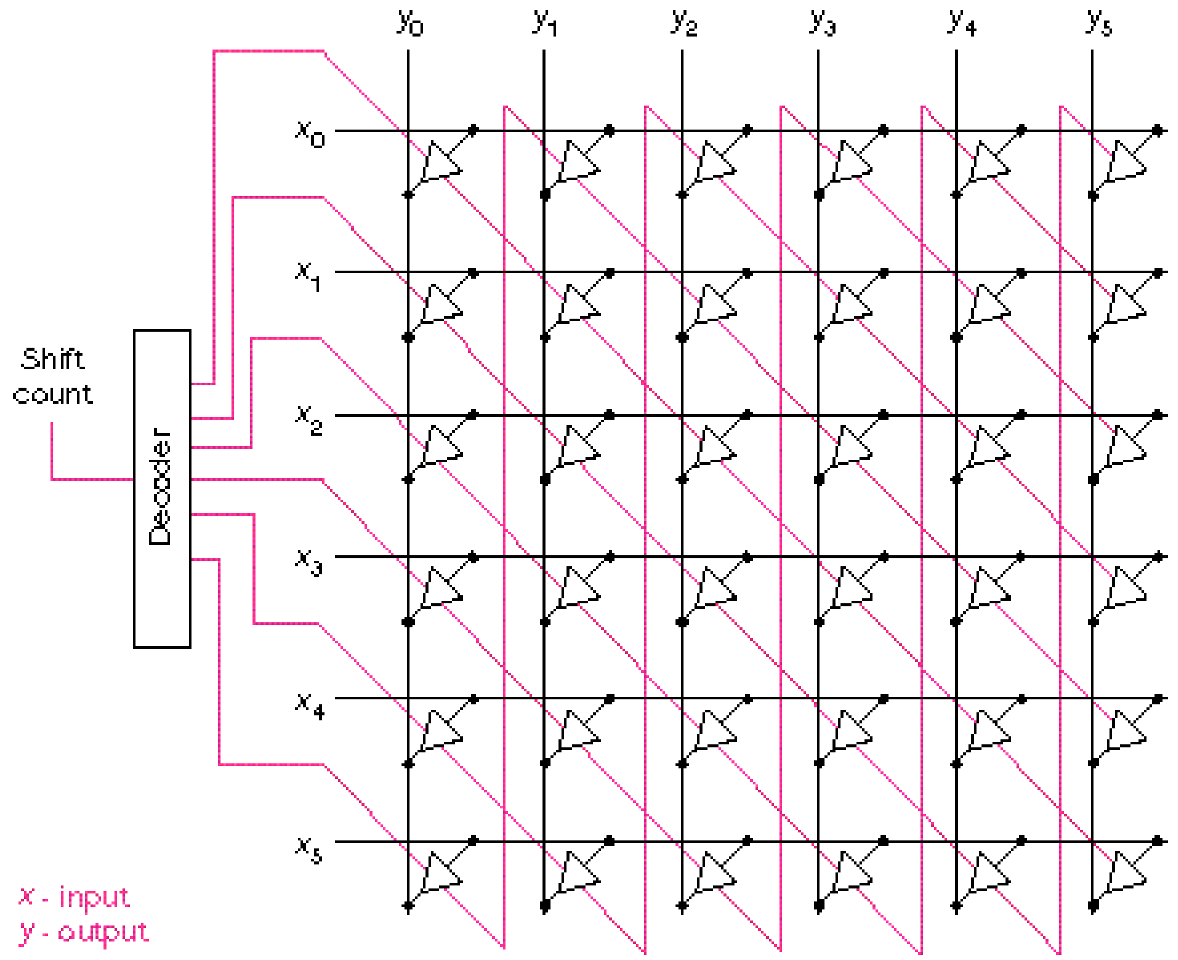
- MC68000 shifts take 8+2n clocks, where n = shift count, so `ROL.L #9` is better then `SHR.L #23 in` the above example

Types and Speed of Shift Instructions

- Rotate right is equivalent to rotate left with a different shift count
- Rotates can include the carry or not
- Two right shifts, one with sign extend, are needed to scale unsigned and signed numbers
- Only a zero fill left shift is needed for scaling
- Shifts whose execution time depends on the shift count use a single bit ALU shift repeatedly, as we did for SRC in Chap. 4
- Fast shifts, important for pipelined designs, can be done with a barrel shifter

C
S
D
A
2/e

Fig 6.11 A 6 Bit Crossbar Barrel Rotator for Fast Shifting

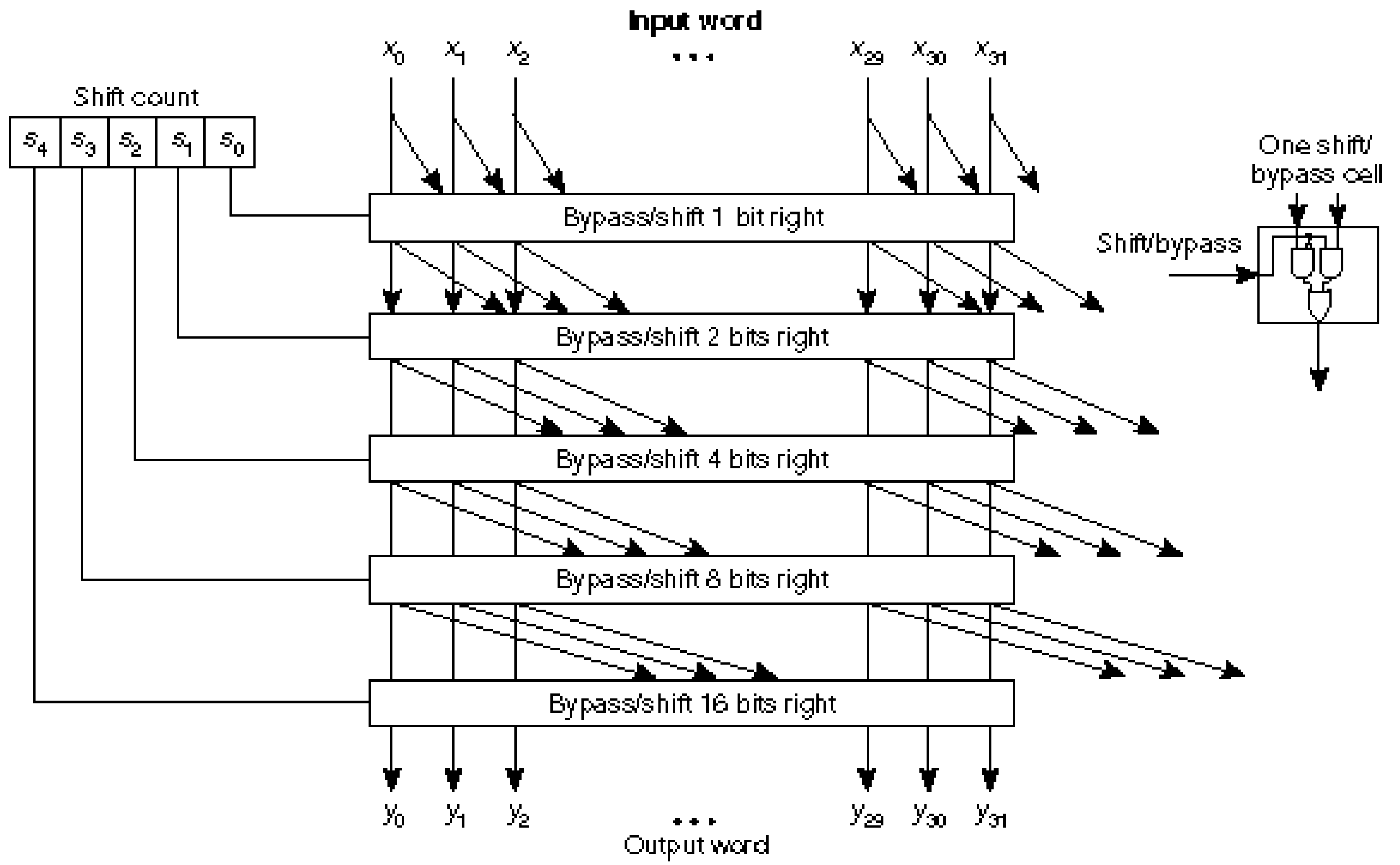


Properties of the Crossbar Barrel Shifter

- There is a 2 gate delay for any length shift
- Each output line is effectively an n way multiplexer for shifts of up to n bits
- There are n^2 3-state drivers for an n bit shifter
  - For n = 32, this means 1024 3-state drivers
- For 32 bits, the decoder is 5 bits to 1 out of 32
- The minimum delay but large number of gates in the crossbar prompts a compromise:

  the logarithmic barrel shifter

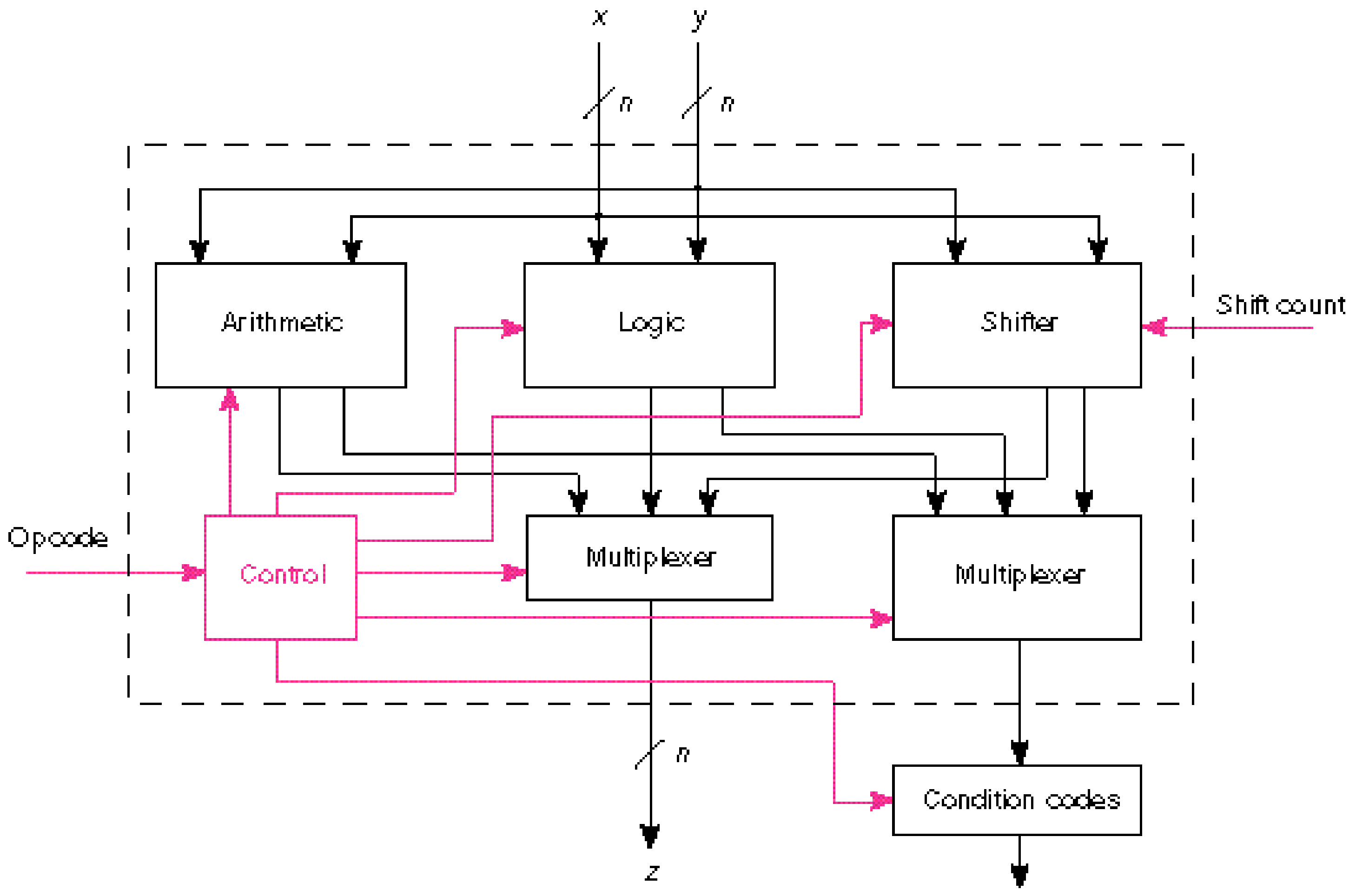
Fig 6.12 Barrel Shifter with a Logarithmic Number of Stages


Input word
x_0 x_1 x_2 . . . x_{29} x_{30} x_{31}
Shift count
s_4 s_3 s_2 s_1 s_0
Bypass/shift 1 bit right
Bypass/shift 2 bits right
Bypass/shift 4 bits right
Bypass/shift 8 bits right
Bypass/shift 16 bits right
One shift/ bypass cell
Shift/bypass
y_0 y_1 y_2 . . . y_{29} y_{30} y_{31}
Output word

Elements of a Complete ALU

- In addition to the arithmetic hardware, there must be a controller for multi-step operations, such as series parallel multiply
- The shifter is usually a separate unit, and may have lots of gates if it is to be fast
- Logic operations are usually simple
- The arithmetic unit may need to produce condition codes as well as a result number
- Multiplexers select the result and condition codes from the correct sub-unit

Fig 6.13 Complete Arithmetic Logic Unit Block Diagram



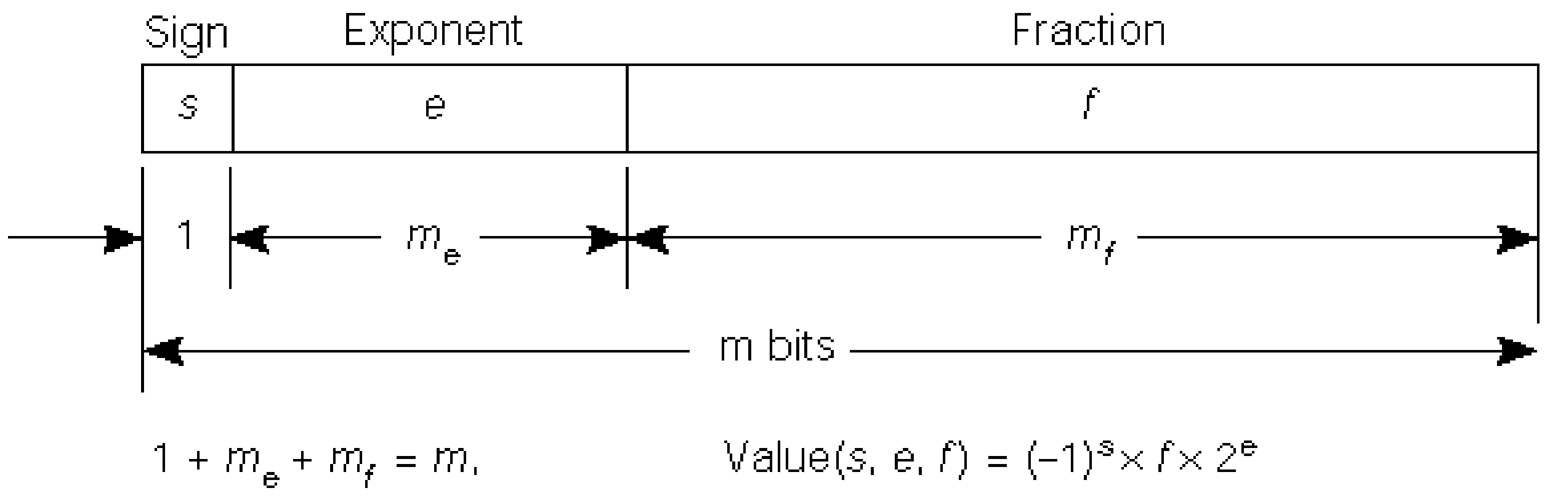
Floating Point Preliminaries: Scaled Arithmetic

- Software can use arithmetic with a fixed binary point position, say left end, and keep a separate scale factor e for a number $f\times 2^e$
- Add or subtract on numbers with same scale is simple, since $f\times 2^e + g\times 2^e = (f+g)\times 2^e$
- Even with same scale for operands, scale of result is different for multiply and divide

  $(f\times 2^e)\cdot(g\times 2^e) = (f\cdot g)\times 2^{2e}$; $(f\times 2^e)\div(g\times 2^e) = f\div g$
- Since scale factors change, general expressions lead to a different scale factor for each number—floating point representation

CSDA 2/e

Fig 6.14 Floating Point Numbers Include Scale & Number in One Word



- All floating-point formats follow a scheme similar to the one above
- s is sign, e is exponent, and f is significand
- We will assume a fraction significand, but some representations have used integers

Signs in Floating Point Numbers

- Both significand and exponent have signs
- A complement representation could be used for f, but sign-magnitude is most common now
- The sign is placed at the left instead of with f so test for negative always looks at left bit
- The exponent could be 2's complement, but it is better to use a <u>biased</u> exponent
- If $-e_{min} \leq e \leq e_{max}$, where e_{min}, $e_{max} > 0$, then

  $e = e_{min} + e$ is always positive, so e replaced by e
- ^We will see that a sign at the left & a positive exponent left of ^ the significand, helps compare

Exponent Base and Floating Point Number Range

- In a floating point format using 24 out of 32 bits for significand, 7 would be left for exponent
- A number x would have a magnitude $2^{-64} \le x \le 2^{63}$, or about $10^{-19} \le x \le 10^{19}$
- For more exponent range, bits of significand would have to be given up with loss of accuracy
- An alternative is an exponent base >2
- IBM used exponent base 16 in the 360/370 series for a magnitude range about $10^{-75} \le x \le 10^{75}$
- Then 1 unit change in e corresponds to a binary point shift of 4 bits

Normalized Floating Point Numbers

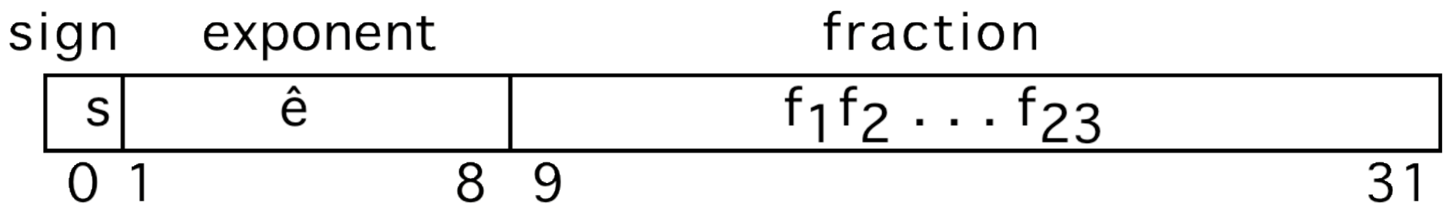
- There are multiple representations for a FP #
- If f_1 and $f_2 = 2^d f_1$ are both fractions & $e_2 = e_1 - d$, then (s, f_1, e_1) & (s, f_2, e_2) have same value
- Scientific notation example: $.819 \times 10^3 = .0819 \times 10^4$
- A normalized floating point number has a leftmost digit non-zero (exponent small as possible)
- With exponent base b, this is a base b digit: for the IBM format the leftmost 4 bits (base 16) are $\neq 0$
- Zero cannot fit this rule; usually written as all 0s
- In norm. base 2 left bit =1, so it can be left out
    - So called hidden bit

C S D A 2/e

Comparison of Normalized Floating Point Numbers

- If normalized numbers are viewed as integers, a biased exponent field to the left means an exponent unit is more than a significand unit
- The largest magnitude number with a given exponent is followed by the smallest one with the next higher exponent
- Thus normalized FP numbers can be compared for <,≤,>,≥,=,≠ as if they were integers
- This is the reason for the s,e,f ordering of the fields and the use of a biased exponent, and one reason for normalized numbers

Fig 6.15 IEEE Single-Precision Floating Point Format



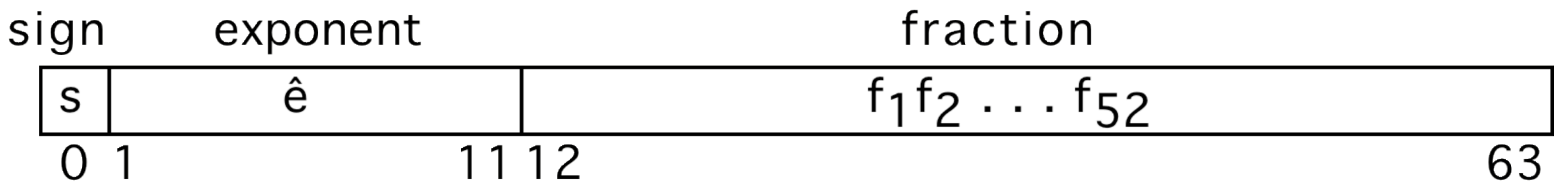
ê	e	Value	Type
255	none	none	Infinity or NaN
254	127	$(-1)^s \times (1.f_1f_2...) \times 2^{127}$	Normalized
...	...	...	...
2	-125	$(-1)^s \times (1.f_1f_2...) \times 2^{-125}$	Normalized
1	-126	$(-1)^s \times (1.f_1f_2...) \times 2^{-126}$	Normalized
0	-126	$(-1)^s \times (0.f_1f_2...) \times 2^{-126}$	Denormalized

- Exponent bias is 127 for normalized #s

Special Numbers in IEEE Floating Point

- An all zero number is a normalized 0
- Other numbers with biased exponent e = 0 are called denormalized
- Denorm numbers have a hidden bit of 0 and an exponent of -126; they may have leading 0s
- Numbers with biased exponent of 255 are used for ±∞ and other special values, called NaN (not a number)
- For example, one NaN represents 0/0

Fig 6.16 IEEE Standard Double Precision Floating Point



- Exponent bias for normalized #s is 1023
- The denorm biased exponent of 0 corresponds to an unbiased exponent of -1022
- Infinity and NaNs have a biased exponent of 2047
- Range increases from about $10^{-38} \le |x| \le 10^{38}$ to about $10^{-308} \le |x| \le 10^{308}$

Decimal Floating Point Add and Subtract Examples

Operands	Alignment	Normalize & round
6.144×10^{2}	0.06144×10^{4}	1.003644×10^{5}
$+9.975 \times 10^{4}$	$+9.975 \times 10^{4}$	$+ .0005 \times 10^{5}$
	10.03644×10^{4}	1.004×10^{5}

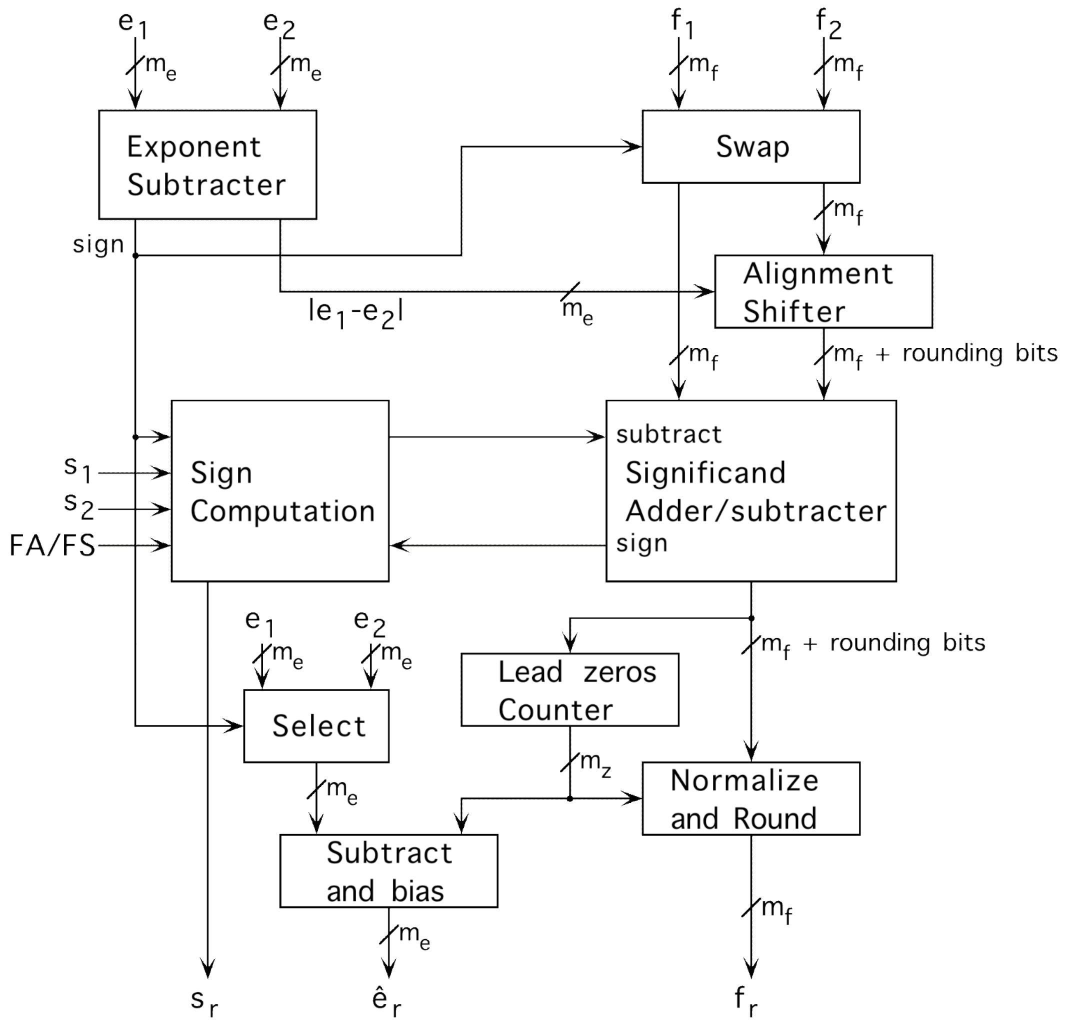
Operands	Alignment	Normalize & round
1.076×10^{-7}	1.076×10^{-7}	7.7300×10^{-9}
-9.987×10^{-8}	-0.9987×10^{-7}	$+ .0005 \times 10^{-9}$
	0.0773×10^{-7}	7.730×10^{-9}

Floating Add, FA, and Floating Subtract, FS, Procedure

Add or subtract (s_1, e_1, f_1) and (s_2, e_2, f_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) $e_r \leftarrow e_r - z$; check over- or underflow; bias & pack

Fig 6.17 Floating Point Add & Subtract Hardware



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

Decimal Floating Point Examples for Multiply & Divide

- Multiply fractions and add exponents
- These examples assume normalzed result is 0.xxx

Sign, fraction & exponent	Normalize & round
$(-0.1403 \quad \times 10^{-3})$	-0.4238463×10^{2}
$\underline{\times(+0.3021 \quad \times 10^{6})}$	$\underline{-0.00005 \quad \times 10^{2}}$
$-0.04238463 \times 10^{-3+6}$	$-0.4238 \quad \times 10^{2}$

- Divide fractions and subtract exponents

Sign, fraction & exponent	Normalize & round
$(-0.9325 \quad \times 10^{2})$	$+0.9306387 \times 10^{9}$
$\underline{\div(-0.1002 \quad \times 10^{-6})}$	$\underline{+0.00005 \quad \times 10^{9}}$
$+9.306387 \quad \times 10^{2-(-6)}$	$+0.9306 \quad \times 10^{9}$

Floating Point Multiply of Normalized Numbers

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

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 OV
4) Handle overflow for exponent too positive and underflow for exponent too negative
5) Pack result, encoding or reporting exceptions

Floating Point Divide of Normalized Numbers

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

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

C S D A 2/e

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