EC-340 Computer Architecture

Lecture 1
Computer Arithmetic — Review
Dr Hashim Ali

Fall 2017

Department of Computer Science and Engineering
HITEC University Taxila

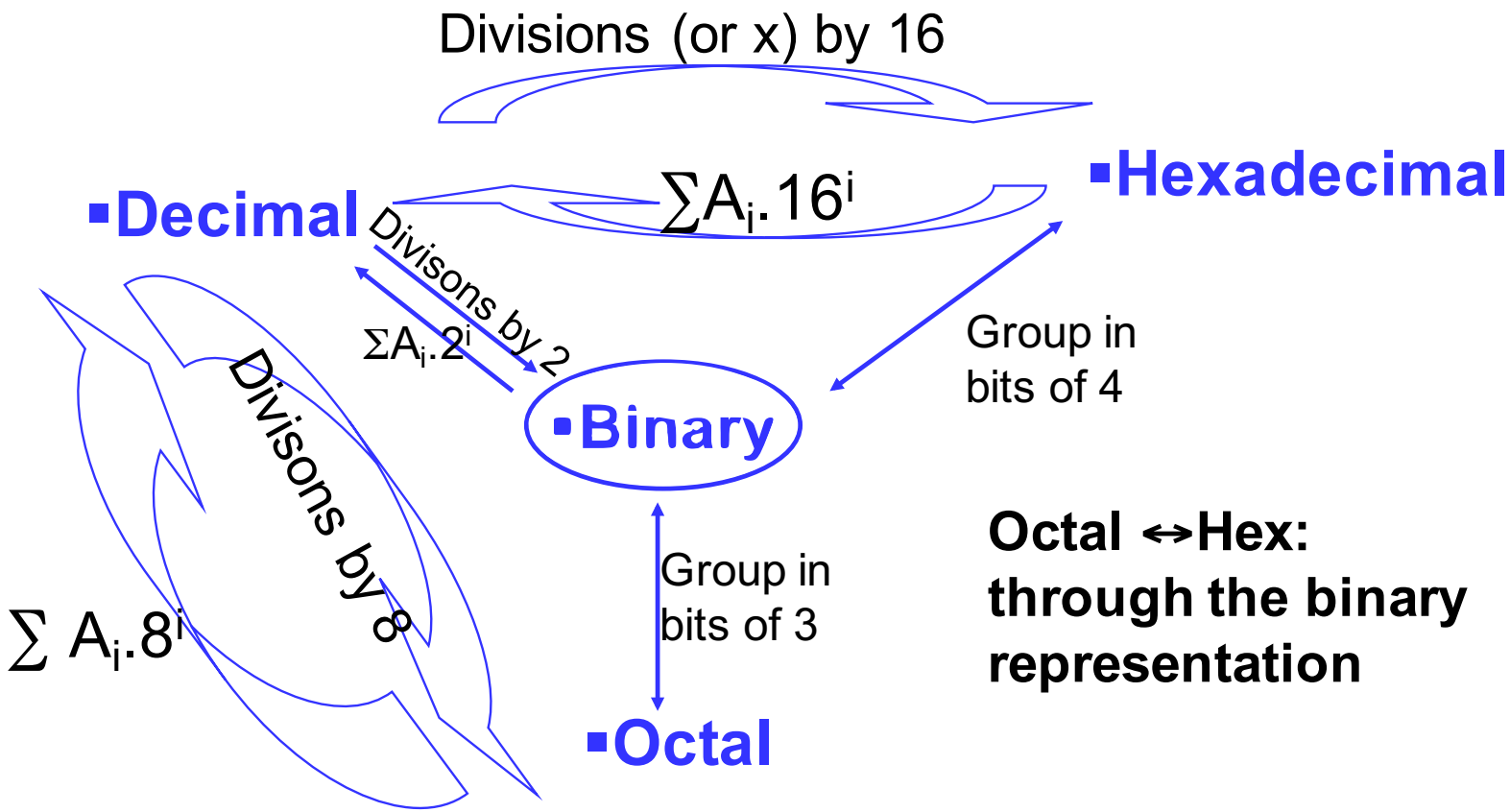
Number System

Chapter 9 — Stallings

Number Systems

- Four types of number systems:
  - Decimal — Base 10
    - Represented by: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Binary — Base 2
    - Represented by: 0, 1
  - Octal — Base 8
    - Represented by: 0, 1, 2, 3, 4, 5, 6, 7
  - Hexadecimal — Base 16
    - Represented by: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Summary



Octal ↔Hex: through the binary representation

Computer Arithmetic

Chapter 10 — Stallings
Appendix J — Hennessy

Complements of Binary Numbers

- Complements are used for simplifying the subtraction operation and for logical manipulation.

- There are two types of complements for each base-r system: the radix and the diminished radix complements.

- For binary numbers:
  - Radix Complement — 2's complement
  - Diminished Radix Complement — 1's complement

Diminished Radix Complement

[(r-1)'s Complement]

- Given a number N in base-r having n-digits, the (r-1)'s complement of N is defined as $(r^n-1)-N$.

- For example: 1's complement of 1011000

- Here;
  - N = 1011000
  - n = 7 (Total number of bits = 7)
  - r = 2 (Because N is a binary number)
  - $(2^7 - 1) - 1011000 = 1111111 - 1011000 = 0100111$

Radix Complement — [r's Complement]

- The r's complement of an n-digit number in base-r is defined as $r^n - N$, for N=0 and 0 for N=0.
  - Compare with (r – 1)'s complement, the r's complement is (r – 1)'s + 1
  - since $r^n - N = [(r^n - 1) - N] + 1$.

- For example: 2's complement of 1011000

- Here;
  - N = 1011000
  - n = 7 (Total number of bits = 7)
  - r = 2 (Because N is a binary number)
  - $[(2^7 - 1) - 1011000] + 1 = [1111111 - 1011000] + 1 = 0100111 + 1 = 0101000$

Shortcut:
Leaving all least significant 0's and the first 1 unchanged, and others have complemented

Arithmetic and Logic Unit

- Part of the computer that actually performs arithmetic and logical operations on data.
- All of the other elements of the computer system are there mainly to bring data into the ALU for it to process and then to take the results back out.
- Based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Integer Representation

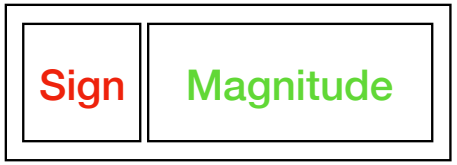
- In the binary number system arbitrary numbers can be represented with:
  - The digits zero and one.
  - The minus sign (for negative numbers).
  - The period, or radix point (for numbers with a fractional component).

- For purposes of computer storage and processing we do not have the benefit of special symbols for the minus sign and radix point.

- Only binary digits (0,1) may be used to represent numbers.

Sign-Magnitude Representation

- There are several alternative conventions used to represent negative as well as positive integers.
  - All of these alternatives involve treating the most significant (leftmost) bit in the word as a sign bit.
    - If the sign bit is 0 the number is positive.
      - Ex: +5 is 0101
    - If the sign bit is 1 the number is negative.
      - Ex: -5 is 1101



- Sign-magnitude representation is the simplest form that employs a sign bit.

- **Drawbacks:**
  - Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation.
  - There are two representation of 0, i.e. +0 — 0000 and -0 — 1000.

- Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU.

Two's Complement Representation

- Use the most significant bit as a sign bit.
- Differs from the sign-magnitude representation in the way that the other bits are interpreted.
- Bias representation is another way of integer representation.
  - Non-negative numbers don't have usual representation.
  - Negative numbers; k+bias ≥ 0
  - Typical value of bias is 2^{n-1}.

k=number, n=number of bits

Decimal	Sign-Magnitude	2's Complement
+8	—	—
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
-0	1000	—
-1	1001	1111
-2	1010	1110
-3	1011	1101
-4	1100	1100
-5	1101	1011
-6	1110	1010
-7	1111	1001
-8	—	1000

Characteristics of 2's Complement Representation

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Rpresentation of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

Negation

- Two's complement operation
  - Take the Boolean complement of each bit of the integer (including the sign bit).
  - Treating the result as an unsigned binary integer, add 1.

+18 = 00010010 (two's complement)

Bitwise complement = 11101101

+ 1

11101110 = -18

- The negative of the negative of that number is itself:

-18 = 11101110 (two's complement)

Bitwise complement = 00010001

+ 1

00010010 = +18

Range Extension

- Range of numbers that can be expressed is extended by increasing the bit length.

- In sign-magnitude notation this is accomplished by moving the sign bit to the new leftmost position and fill in with zeros.
  - Ex: +6 in 4-bit sign-magnitude representation is 0110.
  - +6 in 8-bit representation is 00000110.

- This procedure will not work for two's complement negative integers.
  - Rule is to move the sign bit to the new leftmost position and fill in with copies of the sign bit.
  - For positive numbers, fill in with zeros, and for negative numbers, fill in with ones.
    - Ex: +6 in 4-bit 2's complement representation is 0110.
    - +6 in 8-bit representation is 00000110.
    - -6 in 8-bit representation is 11111010.
  - This is called sign extension.

Addition — 4-bit numbers

a) (-7) + (+5)

1001	=	-7
+ 0101	=	5
1110	=	-2

b) (-4) + (+4)

Discard carry

1100	=	-4
+ 0100	=	4
1 0000	=	0

c) (+3) + (+4)

0011	=	3
+ 0100	=	4
0111	=	7

d) (-4) + (-1)

1100	=	-4
+ 1111	=	-1
1 1011	=	-5

Addition — with Overflow

Discard carry

e) 5 + 4

0101	=	5
+ 0100	=	4
1001	=	Overflow

f) (-7) + (-6)

1001	=	-7
+ 1010	=	-6
1 0011	=	Overflow

Result has opposite sign

- **Overflow Rule:**
  - If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Subtraction

- **Subtraction Rule:**
  - To subtract one number (**subtrahend**) from another (**minuend**), take the twos complement (negation) of the subtrahend and add it to the minuend.

Subtraction — 4-bit numbers

a) 2 - 7

Minuend (M) = 2 = 0010

Subtrahend (S) = 7 = 0111

Negation (-S) = 1001

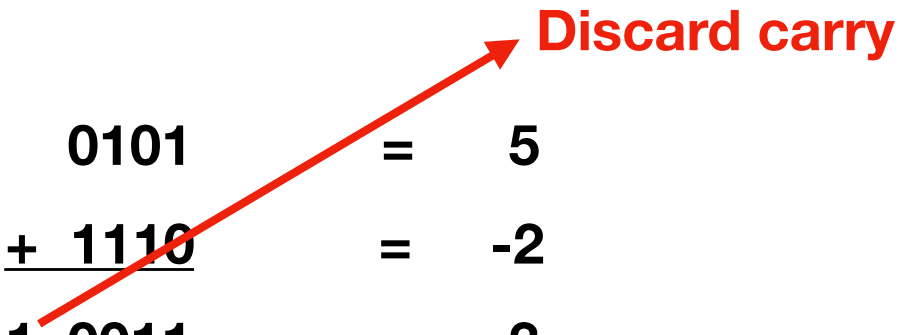
0010	=	2
+ 1001	=	-7
1011	=	-5

b) 0101 - 0010

Minuend (M) = 0101 = 5

Subtrahend (S) = 0010 = 2

Negation (-S) = 1110



0101	=	5
+ 1110	=	-2
1 0011	=	3

Subtraction — 4-bit numbers

a) - 5 - 2

Minuend (M) = -5 = 1011

Subtrahend (S) = 2 = 0010

Negation (-S) = 1110

Discard carry

1011	=	-5
+ 1110	=	-2
1 1001	=	-7

b) 0101 - 1110

Minuend (M) = 0101 = 5

Subtrahend (S) = 1110 = -2

Negation (-S) = 0010

0101	=	5
+ 0010	=	2
0111	=	7

Subtraction — with Overflow

a) 7 - (-7)

Minuend (M) = 7 = 0111

Subtrahend (S) = -7 = 1001

Negation (-S) = 0111

```
  0111      =   7
+ 0111      =   7
  1110      = Overflow
```

Result has opposite sign

b) - 6 - 4

Minuend (M) = 1010 = -6

Subtrahend (S) = 0100 = 4

Negation (-S) = 1100

Discard carry

```
  1010      =   -6
+ 1100      =   -4
1 0110      = Overflow
```

Basic Techniques of Integer Arithmetic

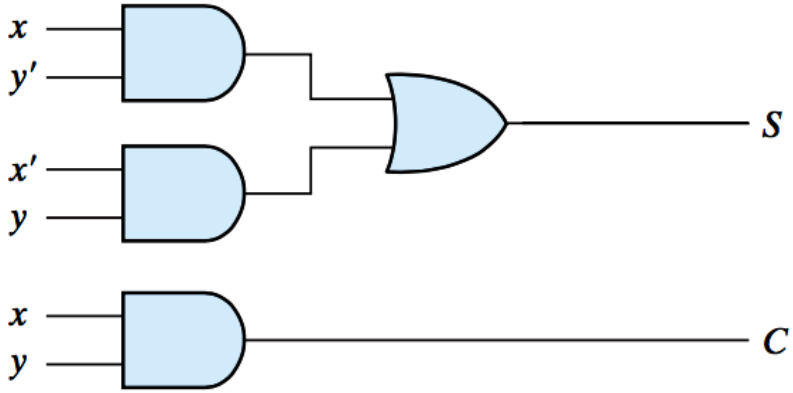
Design of a Binary Adder

- Among many fundamental tasks of any digital computer, arithmetic operations is one of them.
- The most basic arithmetic operation is the addition of two binary digits.
- A combinational circuit that performs the addition of two bits is called a Half Adder; Sum and Carry.
- One that performs addition of three bits (two bits and previous carry) is a Full Adder.
  - As name suggest, two Half Adders can be employed to implement Full Adder.

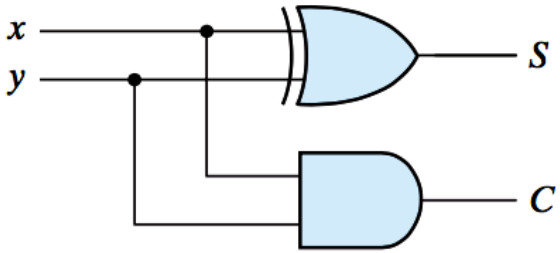
- **Half Adder**

Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

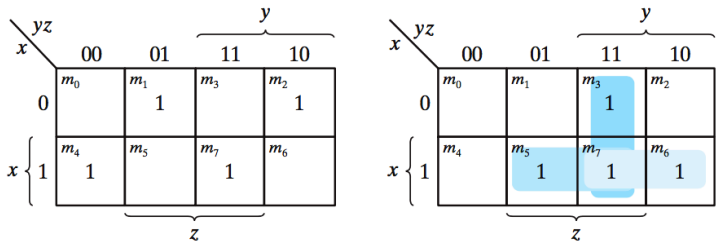


(a) $S = xy' + x'y$
$C = xy$



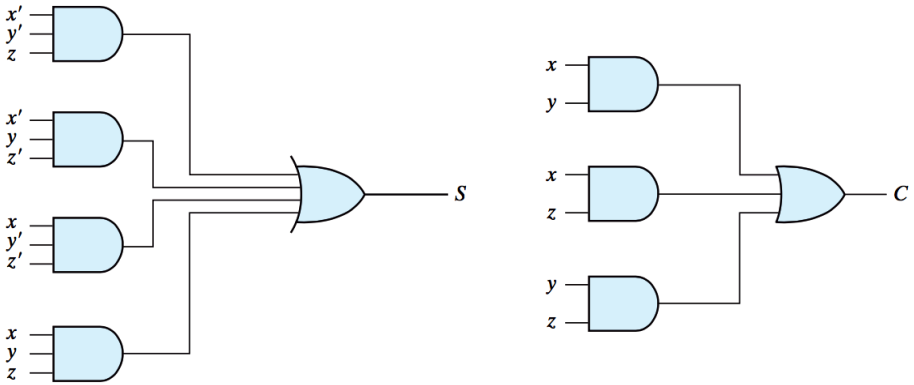
(b) $S = x \oplus y$
$C = xy$

- **Full Adder**



(a) $S = x'y'z + x'yz' + xy'z' + xyz$ (b) $C = xy + xz + yz$

K-Maps for full adder



Full Adder

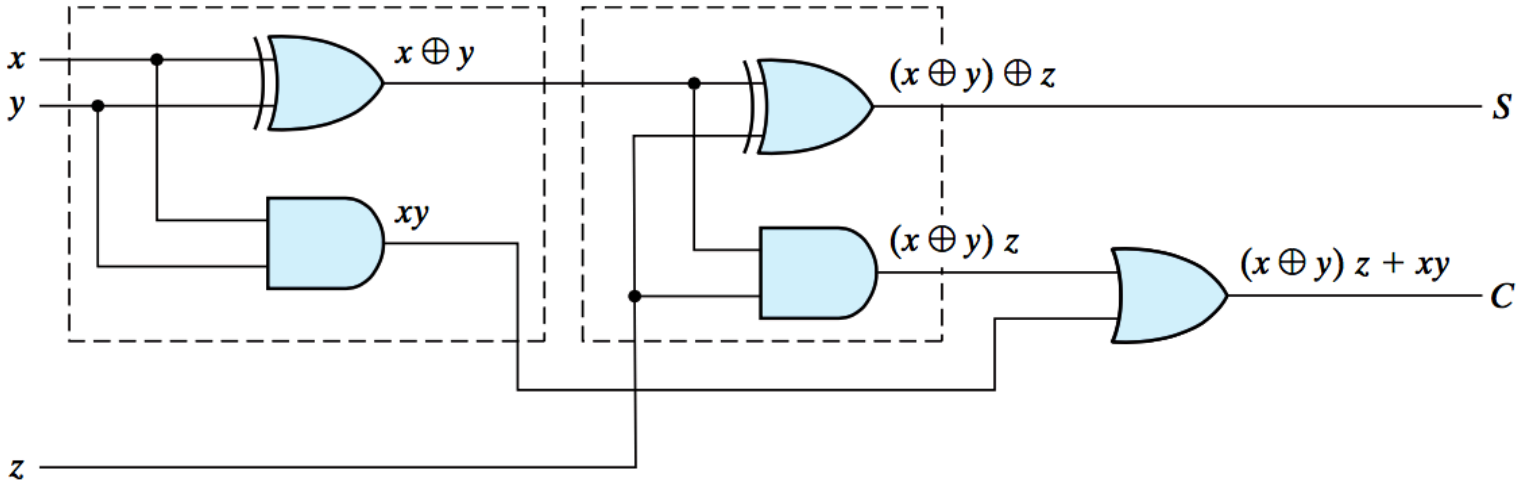
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$
\begin{aligned}
S &= z \oplus (x \oplus y) \\
&= z'(xy' + x'y) + z(xy' + x'y)' \\
&= z'(xy' + x'y) + z(xy + x'y') \\
&= xy'z' + x'yz' + xyz + x'y'z
\end{aligned}
$$

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

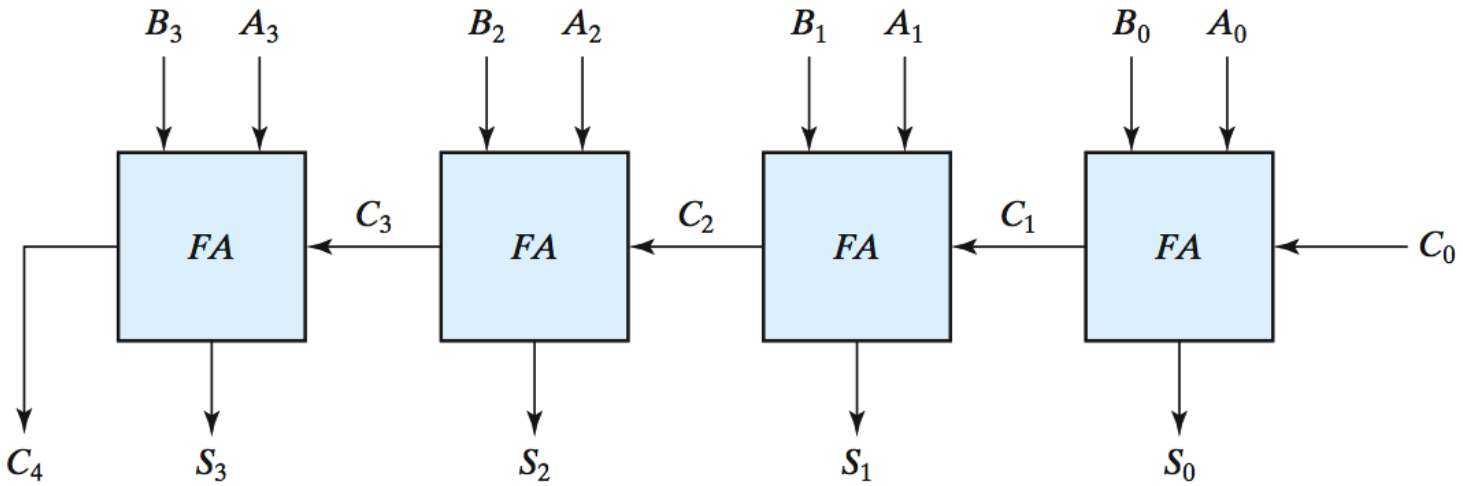
- **Full Adder using two Half Adders**

This is an abstract component.



Implementation of full adder with two half adders and an OR gate

- **4-bit Binary Adder**



Example

Subscript *i*:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

n-bit Ripple-carry Addition

[or Carry-propagation Addition]

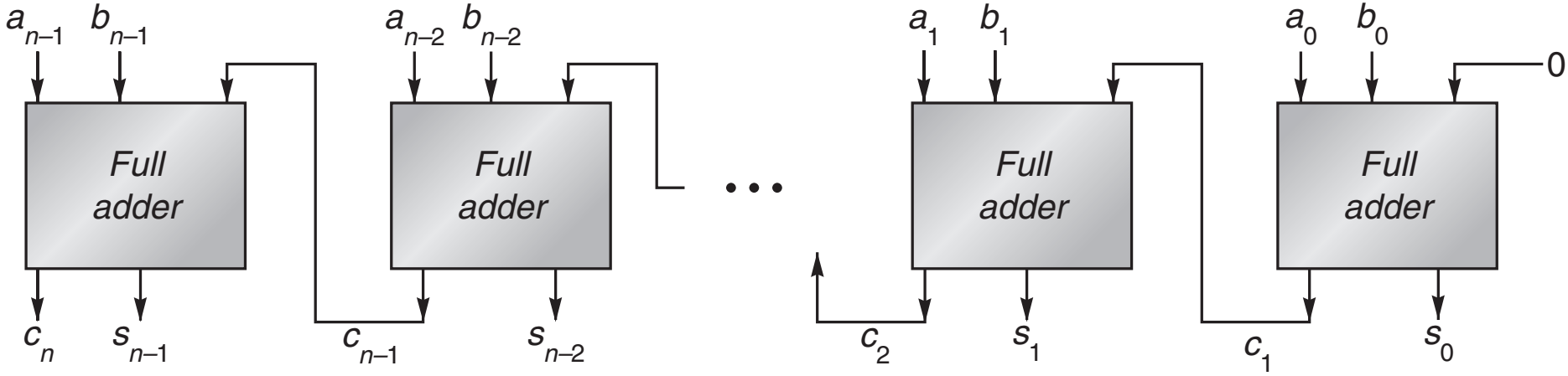


Figure J.1 Ripple-carry adder, consisting of n full adders. The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit. The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left).

- Drawbacks:
  - Carry propagation delay is an important attribute of the adder because it limits the ***speed*** with which two numbers are added.
    - Although, all arithmetic operations are implemented by successive additions, the ***time*** consumed during addition process is critical.
  - Combinational logic gate delays

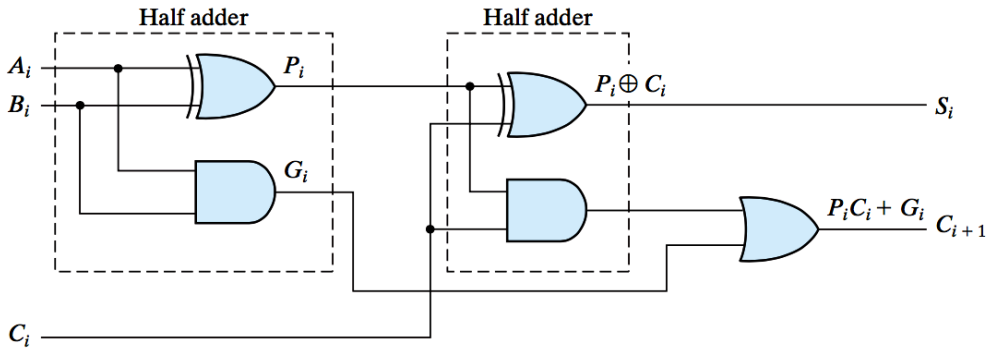
- Solution:
  - An obvious solution is to increase the complexity in such a way that the carry delay time is reduced.
  - Most widely used technique is Carry-lookahead Logic.

Carry-lookahead Logic

- Consider a circuit of full adder;
- If we define two variables,

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$



- The sum and carry can expressed as,

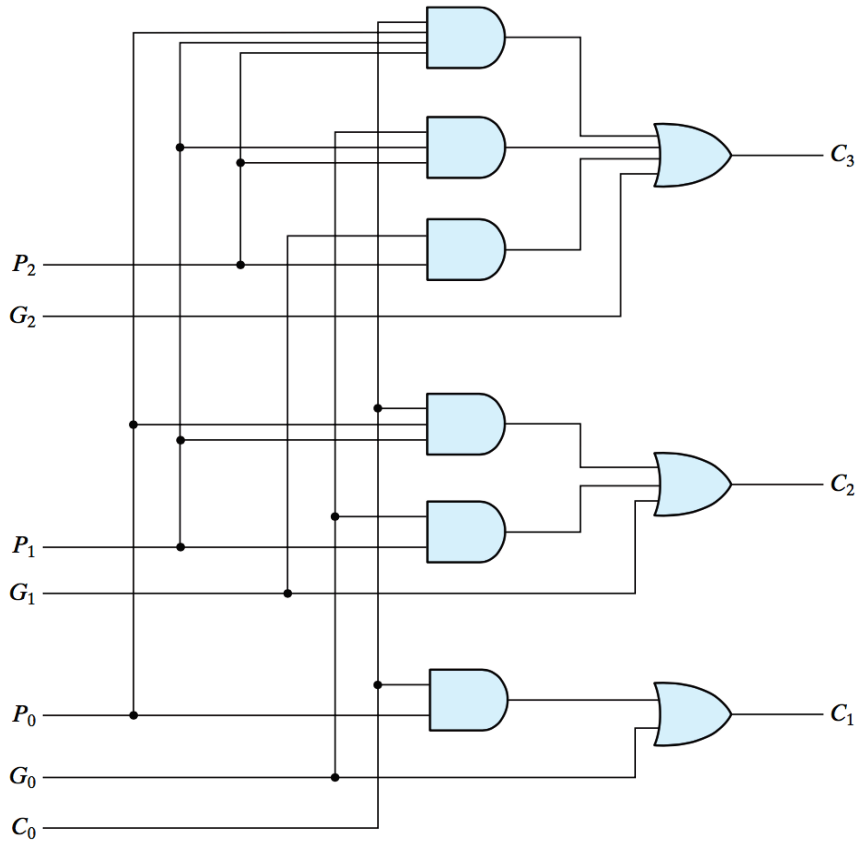
$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

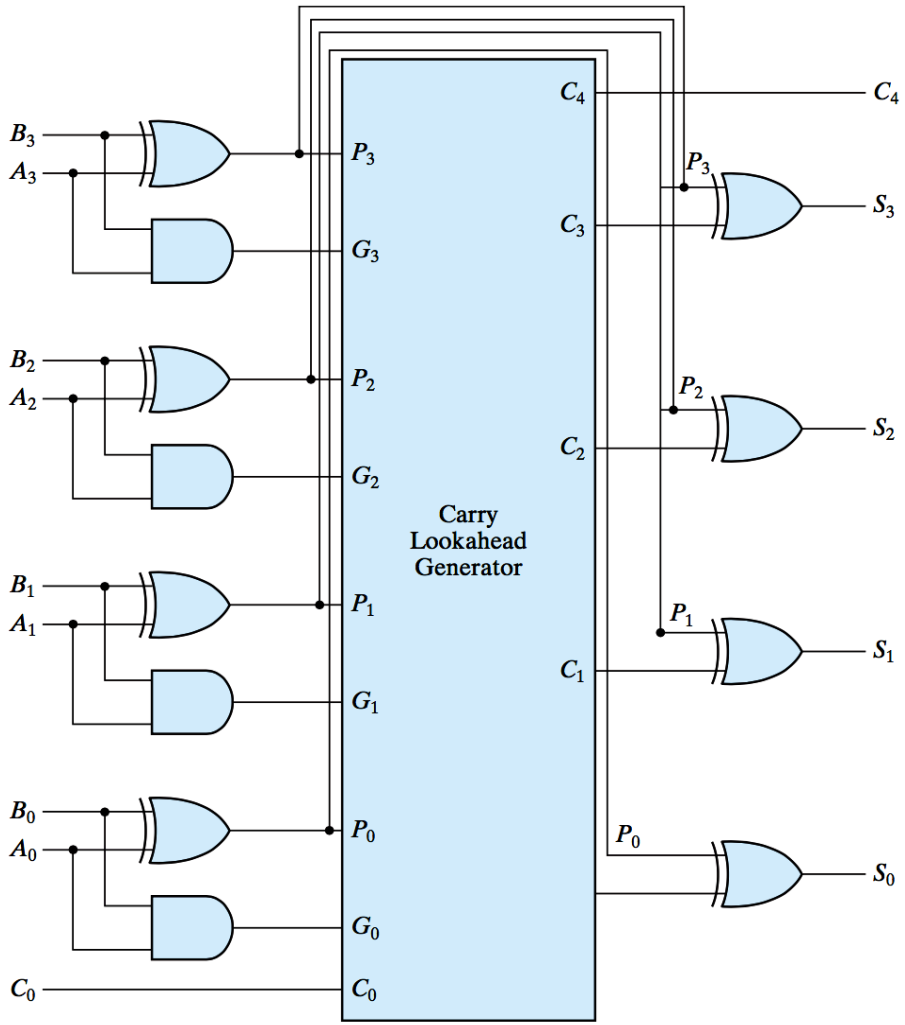
- G_i is called a carry generate, it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i.
- P_i is called a carry propagate, because it determines whether a carry into stage I will propagate into stage i+1.

- Write Boolean functions for the carry outputs of each stage and substitute the value of C_i from previous equations:-

$$C_0 = \text{input carry}$$
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 = P_2P_1P_0C_0$$

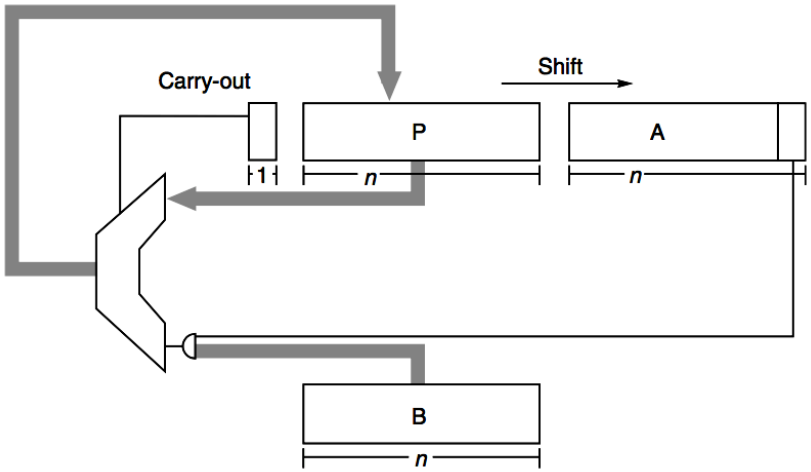


4-bit Full Adder with Carry-lookahead



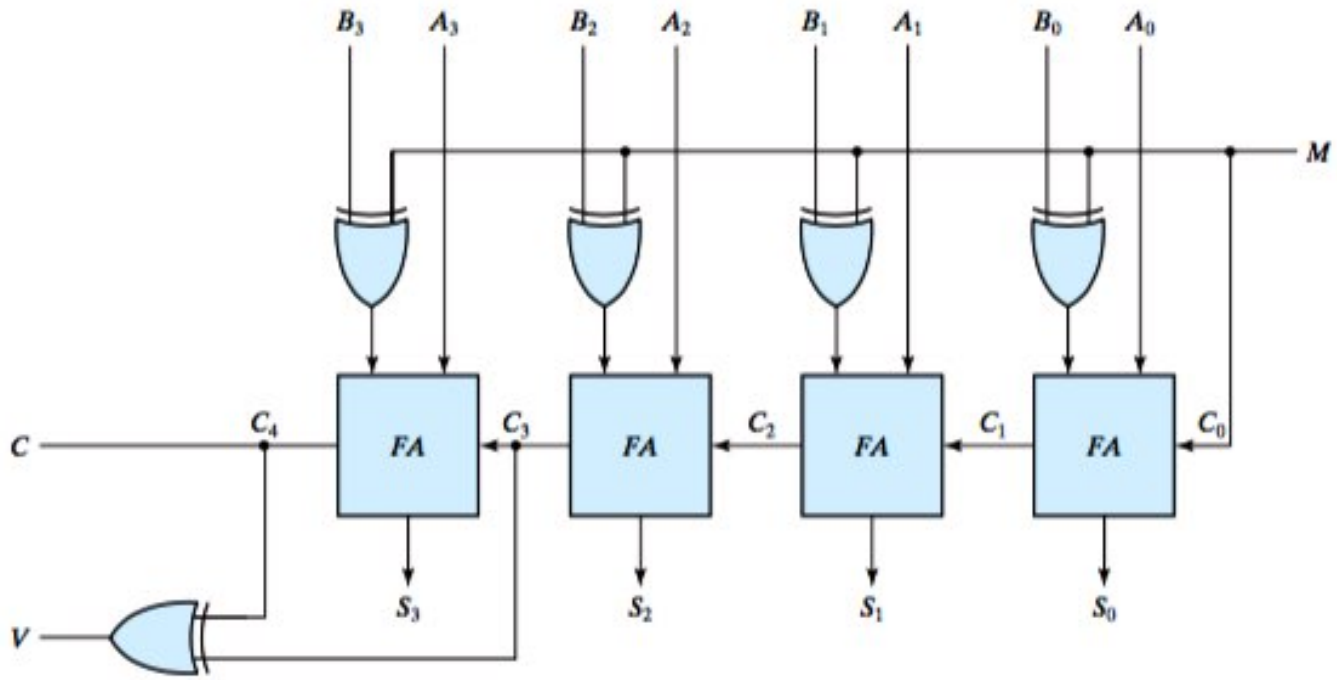
Radix-2 Multiplication

- Each multiply has two steps:-
  - If the least-significant bit of A is 1, then register B, containing $b_{n-1}b_{n-2}\ldots b_0$, is added to P; otherwise, 00…00 is added to P. The sum is placed back into P.
  - Registers P and A are shifted right, with the carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved into register A, and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.
- After n steps, the product appears in registers P and A, with A holding the lower-order bits.



Apply Booth's Algorithm for signed multiplication!!

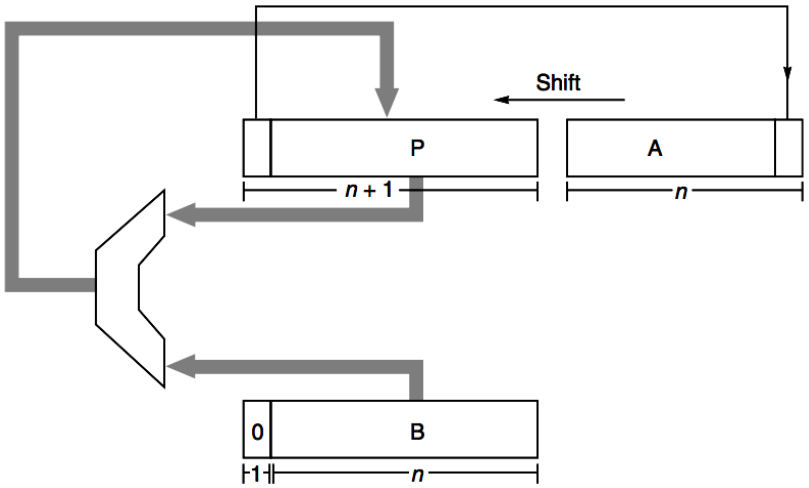
4-bit Binary Subtractor using Adder



Radix-2 Division

- Each divide step consists of four parts:
  - If P is negative,
    (a) Shift the register pair (P,A) one bit left.
    (b) Add the contents of register B to P.
  - Else
    (a) Shift the register pair (P,A) one bit left.
    (b) Subtract the contents of register B from P.
    (c) If P is negative, set the low-order bit of A to 0, otherwise set it to 1.

- After repeating this process n times, the A register will contain the quotient, and the P register will contain the remainder.



Systems Issues

- There are number of issues (related to integer arithmetic) to be resolved before designing an instruction set. Few of them are listed here:-
  1. **Overflow**
     - Signed arithmetic—There are three approaches: Set overflow bit, Trap on overflow, Do nothing (software has to check an overflow)
       - Convenient solution is to have Enable bit, if the bit is 1 then overflow causes trap otherwise overflow bit is set.
     - Unsigned arithmetic—None of the architectures trap on overflow, due to its primary use to manipulate addresses.

Machine	Trap on signed overflow?	Trap on unsigned overflow?	Set bit on signed overflow?	Set bit on unsigned overflow?
VAX	If enable is on	No	Yes. Add sets V bit.	Yes. Add sets C bit.
IBM 370	If enable is on	No	Yes. Add sets cond code.	Yes. Logical add sets cond code.
Intel 8086	No	No	Yes. Add sets V bit.	Yes. Add sets C bit.
MIPS R3000	Two add instructions; one always traps, the other never does.	No	No. Software must deduce it from sign of operands and result.	
SPARC	No	No	Addcc sets V bit. Add does not.	Addcc sets C bit. Add does not.

Figure J.5 Summary of how various machines handle integer overflow. Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

2. **Multiplication**
   - A second issue concerns multiplication of two n-bit numbers, as the result of multiplication produces 2n-bit number.
     - It is preferable to have n-bit result and singling overflow, due to the fact that same data type integers are used in high-level languages. So, compiler would not generate code that utilises double-precision result.
3. **Single instruction execution cycle**
   - A third issue concerns machines that want to execute one instruction every cycle. It is rarely impractical to perform multiplication/division in the same amount of time.
     - Three approaches to solve this problem:-
       - Single-cycle multiply step
       - Floating point unit
       - Autonomous unit in CPU to have multiplication