



UNIVERSITY OF MASSACHUSETTS
Dept. of Electrical & Computer Engineering

Digital Computer Arithmetic

ECE 666

Part 5a
Fast Addition

Israel Koren
Spring 2008

ECE666/Koren Part.5a.1

Copyright 2008 Koren

Ripple-Carry Adders

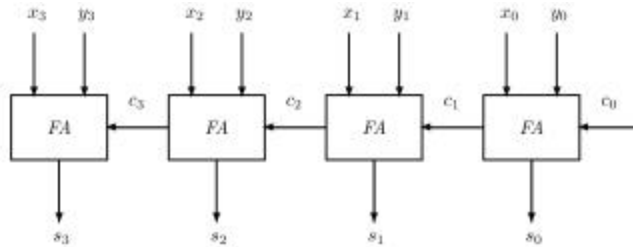
- ◆ **Addition** - most frequent operation - used also for multiplication and division - fast two-operand adder essential
- ◆ Simple parallel adder for adding $x_{n-1}, x_{n-2}, \dots, x_0$ and $y_{n-1}, y_{n-2}, \dots, y_0$ - using n full adders
- ◆ **Full adder** - combinational digital circuit with input bits x_i, y_i and incoming carry bit c_i and output sum bit s_i and outgoing carry bit c_{i+1} - incoming carry for next FA with input bits x_{i+1}, y_{i+1}
- ◆ $s_i = x_i \oplus y_i \oplus c_i$
- ◆ $c_{i+1} = x_i \times y_i + c_i \times (x_i + y_i)$
- ◆ \oplus - exclusive-or ; \times - AND ; $+$ - OR

ECE666/Koren Part.5a.2

Copyright 2008 Koren

Parallel Adder with 4 FAs

◆ Ripple-Carry Adder



◆ In a parallel arithmetic unit

- * All $2n$ input bits available at the same time
- * Carries propagate from the FA in position 0 (with inputs x_0 and y_0) to position i before that position produces correct sum and carry-out bits
- * Carries ripple through all n FAs before we can claim that the sum outputs are correct and may be used in further calculations

Ripple-Carry Adder

- ◆ FA in position i - combinatorial circuit - sees incoming $c_i=0$ at start of operation - accordingly produces s_i
- ◆ c_i may change later on - resulting in change in s_i
- ◆ Ripple effect observed at sum outputs of adder until carry propagation is complete
- ◆ In add operation $c_0=0$ - FA can be simpler - adding only two bits - half adder (HA) - Boolean equations obtained by setting $c_i=0$
- ◆ FA used for adding 1 in least-significant position (ulp) - to implement subtract operation in two's complement
- ◆ One's complement of subtrahend taken and a forced carry added to FA in position 0 by setting $c_0=1$

Example

* $x_0=1111$; $y_3, y_2, y_1, y_0=0001$

* D_{FA} - operation time - delay

* Assuming equal delays for sum and carry-out

* Longest carry propagation chain when adding two 4-bit numbers

$T = 0$		1111
	+	0001
$T = \Delta_{FA}$	Carry	0001
	Sum	1110
$T = 2\Delta_{FA}$	Carry	0011
	Sum	1100
$T = 3\Delta_{FA}$	Carry	0111
	Sum	1000
$T = 4\Delta_{FA}$	Carry	1111
	Sum	0000

◆ In synchronous arithmetic units - time allowed for adder's operation is worst-case delay - nD_{FA}

◆ Adder assumed to produce correct sum after this fixed delay even for very short carry propagation time as in $0101+0010$

◆ Subtract $0101-0010$

* adding two's complement of subtrahend to minuend

* one's complement of 0010 is 1101

* forced carry $C_0=1$ * result 0011

ECE666/Koren Part.5a.5

Copyright 2008 Koren

Reducing Carry Propagation Time

◆ (1) Shorten carry propagation delay

◆ (2) Detect completion of carry propagation - no waiting for fixed delay nD_{FA}

◆ **Second approach** - variable addition time - inconvenient in a synchronous design

◆ Concentrate on first approach - several schemes for accelerating carry propagation exist

◆ **Exercise** - Find a technique for detection of carry completion

ECE666/Koren Part.5a.6

Copyright 2008 Koren

Carry-Look-Ahead Adders

- ◆ Objective - generate all incoming carries in parallel
- ◆ Feasible - carries depend only on $x_{n-1}, x_{n-2}, \dots, x_0$ and $y_{n-1}, y_{n-2}, \dots, y_0$ - information available to all stages for calculating incoming carry and sum bit
- ◆ Requires large number of inputs to each stage of adder - impractical
- ◆ Number of inputs at each stage can be reduced - find out from inputs whether new carries will be generated and whether they will be propagated

Carry Propagation

- ◆ If $x_i=y_i=1$ - carry-out generated regardless of incoming carry - no additional information needed
- ◆ If $x_i, y_i=10$ or $x_i, y_i=01$ - incoming carry propagated
- ◆ If $x_i=y_i=0$ - no carry propagation
- ◆ $G_i=x_i y_i$ - generated carry ; $P_i=x_i+y_i$ - propagated carry
- ◆ $C_{i+1} = x_i y_i + C_i (x_i + y_i) = G_i + C_i P_i$
- ◆ Substituting $C_i=G_{i-1}+C_{i-1}P_{i-1}$ ® $C_{i+1}=G_i+G_{i-1}P_i+C_{i-1}P_{i-1}P_i$
- ◆ Further substitutions -

$$c_{i+1} = G_i + G_{i-1}P_i + G_{i-2}P_{i-1}P_i + c_{i-2}P_{i-2}P_{i-1}P_i = \dots$$

$$= G_i + G_{i-1}P_i + G_{i-2}P_{i-1}P_i + \dots + c_0P_0P_1 \dots P_i.$$
- ◆ All carries can be calculated **in parallel** from $x_{n-1}, x_{n-2}, \dots, x_0$, $y_{n-1}, y_{n-2}, \dots, y_0$, and forced carry C_0

Example - 4-bit Adder

$$c_1 = G_0 + c_0P_0,$$

$$c_2 = G_1 + G_0P_1 + c_0P_0P_1,$$

$$c_3 = G_2 + G_1P_2 + G_0P_1P_2 + c_0P_0P_1P_2,$$

$$c_4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + c_0P_0P_1P_2P_3$$

Delay of Carry-Look-Ahead Adders

- ◆ D_G - delay of a single gate
- ◆ At each stage
 - * Delay of D_G for generating all P_i and G_i
 - * Delay of $2D_G$ for generating all C_i (two-level gate implementation)
 - * Delay of $2D_G$ for generating sum digits S_i in parallel (two-level gate implementation)
 - * Total delay of $5D_G$ regardless of n - number of bits in each operand
- ◆ Large n ($=32$) - large number of gates with large fan-in
 - * Fan-in - number of gate inputs, $n+1$ here
- ◆ Span of look-ahead must be reduced at expense of speed

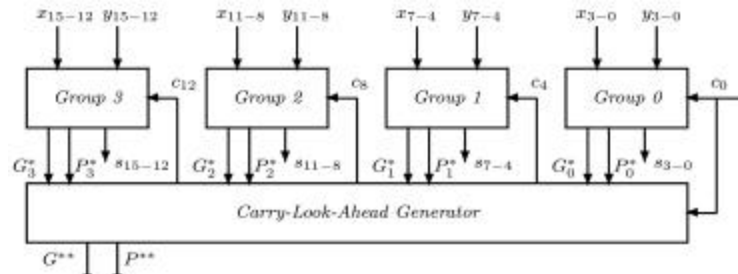
Reducing Span

- ◆ **n** stages divided into groups - separate carry-look-ahead in each group
- ◆ Groups interconnected by ripple-carry method
 - * Equal-sized groups - modularity - one circuit designed
 - * Commonly - group size **4** selected - $n/4$ groups
 - * **4** is factor of most word sizes
 - * Technology-dependent constraints (number of input/output pins)
 - * ICs adding two **4** digits sequences with carry-look-ahead exist
 - » **DG** needed to generate all **P_i** and **G_i**
 - » **2DG** needed to propagate a carry through a group once the **P_i, G_i, C_o** are available
 - » $(n/4)2DG$ needed to propagate carry through all groups
 - » **2DG** needed to generate sum outputs
 - * Total - $(2(n/4)+3)DG = ((n/2)+3)DG$ - a reduction of almost **75%** compared to $2nDG$ in a ripple-carry adder

Further Addition Speed-Up

- ◆ Carry-look-ahead over groups
- ◆ **Group-generated carry** - $G^*=1$ if a carry-out (of group) is generated internally
- ◆ **Group-propagated carry** - $P^*=1$ if a carry-in (to group) is propagated internally to produce a carry-out (of group)
- ◆ For a group of size **4**:
$$G^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$
$$P^* = P_0P_1P_2P_3.$$
- ◆ Group-carries for several groups used to generate group carry-ins similarly to single-bit carry-ins
- ◆ A combinatorial circuit implementing these equations available - carry-look-ahead generator

Example - 16-bit 2-level Carry-look-ahead Adder



- ◆ $n=16$ - 4 groups
- ◆ Outputs $G^*0, G^*1, G^*2, G^*3, P^*0, P^*1, P^*2, P^*3$
- ◆ Inputs to a carry-look-ahead generator with outputs $C4, C8, C12$

$$c_4 = G_0^* + c_0 P_0^*$$

$$c_8 = G_1^* + G_0^* P_1^* + c_0 P_0^* P_1^*$$

$$c_{12} = G_2^* + G_1^* P_2^* + G_0^* P_1^* P_2^* + c_0 P_0^* P_1^* P_2^*$$

ECE666/Koren Part.5a.13

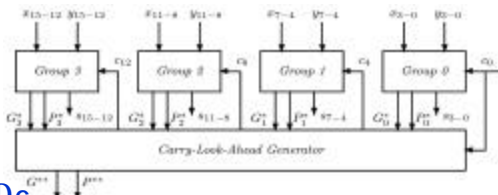
Copyright 2008 Koren

Example - Cont.

◆ Operation - 4 steps:

- * 1. All groups generate in parallel G_i and P_i - delay D_G
- * 2. All groups generate in parallel group-carry-generate - G^*i and group-carry-propagate - P^*i - delay $2D_G$
- * 3. Carry-look-ahead generator produces carries $C4, C8, C12$ into the groups - delay $2D_G$
- * 4. Groups calculate in parallel individual sum bits with internal carry-look-ahead - delay $4D_G$

- ◆ Total time - $9D_G$
- ◆ If external carry-look-ahead generator not used and carry ripples among the groups - $11D_G$
- ◆ Theoretical estimates only - delay may be different in practice



ECE666/Koren Part.5a.14

Copyright 2008 Koren

Additional Levels of Carry-look-ahead

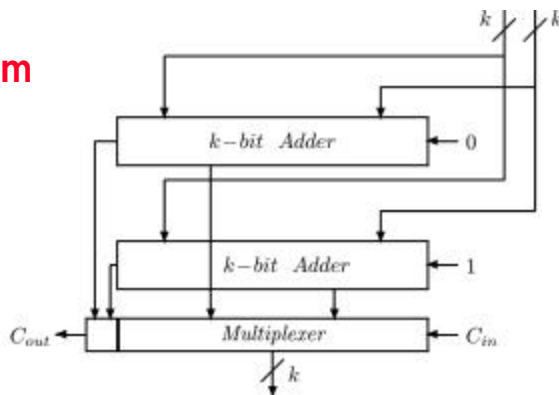
- ◆ Carry-look-ahead generator produces G^{**} and P^{**} , - section-carry generate and propagate
 - * section is a set of 4 groups and consists of 16 bits
- ◆ For 64 bits, either use 4 circuits with a ripple-carry between adjacent sections, or use another level of carry-look-ahead for faster execution
- ◆ This circuit accepts 4 pairs of section-carry-generate and section-carry-propagate, and produces carries C_{16} , C_{32} , and C_{48}
- ◆ As n increases, more levels of carry-look-ahead generators can be added
- ◆ Number of levels (for max speed up) $\approx \log_b n$
 - * b is the blocking factor - number of bits in a group, number of groups in a section, and so on
- ◆ Overall addition time is proportional to $\log_b n$

ECE666/Koren Part.5a.15

Copyright 2008 Koren

Conditional Sum Adders

- ◆ Logarithmic speed-up of addition



- ◆ For given k operand bits - generate two outputs - each with k sum bits and an outgoing carry - one for incoming carry 0 and one for 1
- ◆ When incoming carry known - select correct output out of two - no waiting for carry to propagate
- ◆ Not to all n bits at once

ECE666/Koren Part.5a.16

Copyright 2008 Koren

Dividing into Groups

- ◆ Divide n bits into smaller groups - apply above to each
- ◆ Serial carry-propagation inside groups done in parallel
- ◆ Groups can be further divided into subgroups
- ◆ Outputs of subgroups combined to generate output of groups
- ◆ Natural division of n - two groups of $n/2$ bits each
- ◆ Each can be divided into two groups of $n/4$, and so on
- ◆ If n power of 2 - last subgroup is of size 1 and $\log_2 n$ steps are needed
- ◆ Division not necessarily into equal-sized subgroups - scheme can be applied even if n not a power of 2

ECE666/Koren Part.5a.17

Copyright 2008 Koren

Example - Combining Single Bits into Pairs

- ◆ s_i^0 / s_i^1 - sum bit at position i under the assumption that incoming carry into currently considered group is 0 / 1
- ◆ Similarly - outgoing carries (from group)
 c_{i+1}^0 / c_{i+1}^1
- ◆ **Step 1** - each bit constitutes a separate group:

i	7	6	
x_i	1	0	
y_i	0	0	
s_i^0	1	0	Assuming incoming carry = 0
c_{i+1}^0	0	0	
s_i^1	0	1	Assuming incoming carry = 1
c_{i+1}^1	1	0	

ECE666/Koren Part.5a.18

Copyright 2008 Koren

Example - Step 2

- ◆ **Step 2** - two bit positions combined (using data selectors) into one group of size 2
- ◆ Carry-out from position 6 becomes internal (to group) carry and appropriate set of outputs for position 7 selected

i	7	6	$i, i-1$	7, 6	
x_i	1	0	x_i, x_{i-1}	10	
y_i	0	0	y_i, y_{i-1}	00	
s_i^0	1	0	s_i^0, s_{i-1}^0	10	Assuming incoming carry = 0
c_{i+1}^0	0	0	c_{i+1}^0	0	
s_i^1	0	1	s_i^1, s_{i-1}^1	11	Assuming incoming carry = 1
c_{i+1}^1	1	0	c_{i+1}^1	0	

ECE666/Koren Part.5a.19

Copyright 2008 Koren

Example - Addition of Two 8-bit Operands

- ◆ $\text{Log}_2 8=3$ steps

	i	7	6	5	4	3	2	1	0
	x_i	1	0	1	1	0	1	1	0
	y_i	0	0	1	0	1	1	0	1
Step 1	s_i^0	1	0	0	1	1	0	1	1
	c_{i+1}^0	0	0	1	0	0	1	0	0
Step 2	s_i^1	0	1	1	0	0	1	0	
	c_{i+1}^1	1	0	1	1	1	1	1	
Step 3	s_i^0	1	1	0	1	0	0	1	1
	c_{i+1}^0	0				1			
Result	s_i^1	1	1	1	0				
	c_{i+1}^1	0							
		1	1	1	0	0	0	1	1

- ◆ Forced carry (=0 here) available at start
- ◆ Only one set of outputs generated for rightmost group at each step

ECE666/Koren Part.5a.20

Copyright 2008 Koren

Carry-Select Adder

- ◆ Variation of conditional sum adder
- ◆ n bits divided into groups - not necessarily equal
- ◆ Each group generates two sets of sum bits and an outgoing carry bit - incoming carry selects one
- ◆ Each group is not further divided into subgroups

- ◆ Comparing Conditional-sum and Carry-look-ahead
 - * Both methods have same speed
 - * Design of conditional sum adder less modular
 - * Carry-look-ahead adder more popular

Optimality of Algorithms and Their Implementations

- ◆ Numerous algorithms for fast addition proposed - technology keeps changing making new algorithms more suitable
- ◆ Performance of algorithm affected by its unique features and number system used to represent operands and results
- ◆ Many studies performed to compare performance of different algorithms - preferably independently of implementation technology
- ◆ Some studies find the limit (bound) on the performance of any algorithm in executing a given arithmetic operation

Optimal Addition Algorithms

- ◆ Execution time reduced by avoiding (or limiting) carry-propagation
- ◆ Number systems such as the **residue** number system and the **SD** number system have almost carry-free addition - provide fast addition algorithms
- ◆ These number systems not frequently used - conversions between number systems needed - may be more complex than addition - not always practical

Lower Bound on Addition Speed

- ◆ **Theoretical model** - derives a bound independent of implementation technology
- ◆ **Assumptions:**
 - * Circuit for addition realized using only one type of gate - (f,r) gate - r is radix of number system used and f is **fan-in** of gate (maximum number of inputs)
 - * All (f,r) gates are capable of computing any r -valued function of f (or less) arguments in exactly the same time
 - * This fixed time period is the unit delay - computation time of adder circuit measured in these units
- ◆ (f,r) gate can compute any function of f arguments - all we need to find out is how many such gates are required and how many circuit levels are needed in order to properly connect the gates

Lower Bound - Cont.

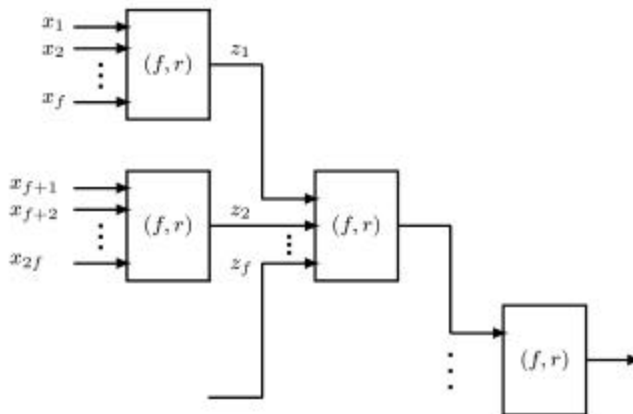
- ◆ A circuit for adding two radix- r operands with n digits each - $2n$ inputs and $n+1$ outputs
- ◆ Consider output requiring all $2n$ inputs - can be reduced to a smaller number of arguments by using $\lceil 2n/f \rceil$ such (f,r) gates operating in parallel
- ◆ Number of intermediate arguments - $\lceil 2n/f \rceil$ - can be further reduced by a second level of (f,r) gates
- ◆ Number of levels in tree - at least $\lceil \log_f 2n \rceil$
- ◆ Lower bound - assumes that no argument is needed as input to more than one (f,r) gate
- ◆ Lower bound on addition time - measured in units of (f,r) gate delay -

$$T_{add} \geq \lceil \log_f 2n \rceil$$

ECE666/Koren Part.5a.25

Copyright 2008 Koren

Circuit Implemented with (f,r) Gates



ECE666/Koren Part.5a.26

Copyright 2008 Koren

Limitations of Model

- ◆ Only fan-in limitation considered - fan-out ignored
- ◆ Fan-out of gate - ability of its output to drive a number of inputs to similar gates in the next level
- ◆ In practice fan-out is constrained
- ◆ More important - model assumes that any r -valued function of f arguments can be calculated by a single (f,r) gate in one unit delay - not true in practice
- ◆ Many functions require either a more complex gate (longer delay) or are implemented using several simple gates organized in two or more levels

Improved Bound

- ◆ Previous bound assumes at least one output digit that depends on all $2n$ input digits
- ◆ If not - a better (lower) value for the bound exists - smaller trees (with fewer inputs) can be used
- ◆ This occurs if carry cannot propagate from least-significant to most-significant position
- ◆ Example - only $X_i, Y_i, X_{i-1}, Y_{i-1}$ needed to determine sum digit S_i -
$$T_{add} \geq \lceil \log_f 4 \rceil$$
- ◆ In the binary system - carry can propagate through all n positions -
$$T_{add} \geq \lceil \log_f 2n \rceil$$
- ◆ In the two addition algorithms - carry-look-ahead and conditional sum - execution time proportional to $\log n$ - previous bound approached

Implementation Cost

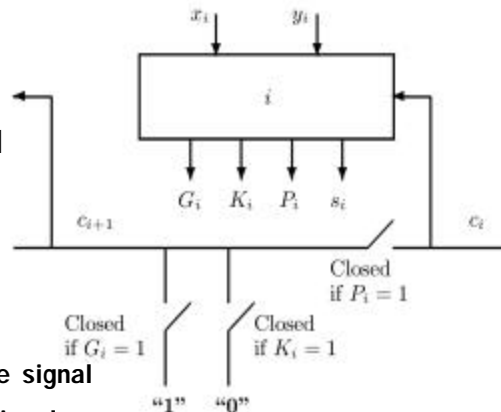
- ◆ Implementation cost must be considered in addition to execution time
- ◆ Implementation cost measure depends on technology
- ◆ **Example - discrete gates**
 - * Number of gates measures implementation cost
 - * Number of gates along the critical (longest) path (number of circuit levels) determines execution time
- ◆ **Example - full custom VLSI technology**
 - * Number of gates - limited effect on implementation cost
 - * Regularity of design and length of interconnections more important - affect both silicon area and design time
- ◆ **Trade-off between implementation cost and addition speed exists**

Performance - Cost Trade-off

- ◆ If performance more important - carry-look-ahead adder preferable
- ◆ Implementation cost can be reduced - determined by regularity of design and size of required area
- ◆ Taking advantage of the available degree of freedom in design - the blocking factor - bounded by fan-in constraint
- ◆ Additional constraints exist - e.g., number of pins
- ◆ Highest blocking factor - not necessarily best
- ◆ **Example** - blocking factor of **2** results in a very regular layout of binary trees with up to $\log_2 n$ levels - total area approximately $n \times \log_2 n$

Manchester Adder

- ◆ If lower implementation cost required - ripple-carry method with speed-up techniques is best
- ◆ **Manchester adder** uses switches that can be realized using pass transistors



- * $P_i = x_i \hat{\Delta} y_i$ carry-propagate signal
- * $G_i = x_i y_i$ carry-generate signal
- * $K_i = \bar{x}_i \bar{y}_i$ carry-kill signal

- ◆ Only one of the switches is closed at any time
- ◆ $P_i = x_i \hat{\Delta} y_i$ used instead of $P_i = x_i + y_i$
- ◆ If $G_i = 1$ - an outgoing carry is generated always
- ◆ If $K_i = 1$ - incoming carry not propagated
- ◆ If $P_i = 1$ - incoming carry propagated

ECE666/Koren Part.5a.31

Copyright 2008 Koren

Manchester Adder - Cont.

- ◆ Switches in units 0 through $n-1$ set simultaneously - propagating carry experiences only a single switch delay per stage
- ◆ Number of carry-propagate switches that can be cascaded is limited - depends on technology
- ◆ n units partitioned into groups with separating devices (buffers) between them
- ◆ **In theory** - execution time linearly proportional to n
- ◆ **In practice** - ratio between execution time and that of another adder (e.g., carry-look-ahead) depends on particular technology
- ◆ Implementation cost - measured in area and/or design regularity - lower than carry-look-ahead adder

ECE666/Koren Part.5a.32

Copyright 2008 Koren