

## Combinational logic design case studies

- Arithmetic circuits
  - integer representations
  - addition/subtraction
  - how redundant logic can make circuits faster
- General design procedure
- Case studies
  - BCD to 7-segment display controller
  - calendar subsystem
  - arithmetic/logic units

## Arithmetic circuits

- Excellent examples of combinational logic design
- Time vs. space trade-offs
  - doing things fast may require more logic and thus more space
  - example: carry look-ahead logic
- Arithmetic and logic units
  - general-purpose building blocks
  - critical components of processor data-paths
  - used within most computer instructions

## Number systems

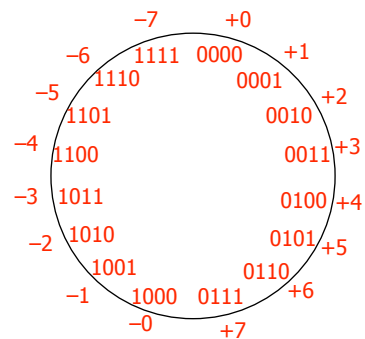
- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
  - sign and magnitude
  - 1s complement
  - 2s complement
- Assumptions
  - we'll assume a 4 bit machine word
  - 16 different values can be represented
  - roughly half are positive, half are negative

## Sign and magnitude

- One bit dedicate to sign (positive or negative)
  - sign: 0 = positive (or zero), 1 = negative
- Rest represent the absolute value or magnitude
  - three low order bits: 0 (000) thru 7 (111)
- Range for n bits
  - $\pm 2^{n-1} - 1$  (two representations for 0)
- Cumbersome addition/subtraction
  - must compare magnitudes to determine sign of result

$$0\ 100 = +4$$

$$1\ 100 = -4$$

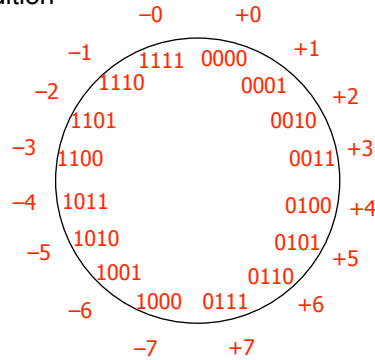


## 1s complement

- Subtraction: first form 1s complement and then add
- Two representations of 0
  - causes some complexities in addition
- High-order bit can act as sign bit

$$0\ 100 = +4$$

$$1\ 011 = -4$$

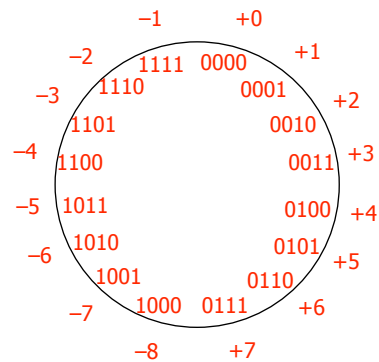


## 2s complement

- Same as 1s complement but with negative numbers shifted one position towards 0 (merge two 0s into a single representations)
  - only one representation for 0
  - one more negative number than positive numbers
  - high-order bit can act as sign bit

$$0\ 100 = +4$$

$$1\ 100 = -4$$



## 2s complement (cont'd)

- If N is a positive number, then the negative of N (its 2s complement or  $N^*$ ) is  $N^* = 2^n - N$

- example: 2s complement of 7

$$\begin{array}{r} 2^4 = 10000 \\ \text{subtract } 7 = \underline{0111} \end{array}$$

1001 = repr. of -7

- example: 2s complement of -7

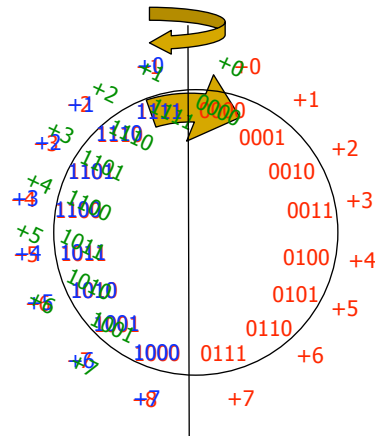
$$\begin{array}{r} 2^4 = 10000 \\ \text{subtract } -7 = \underline{1001} \end{array}$$

0111 = repr. of 7

- shortcut: 2s complement = bit-wise complement + 1
  - 0111 -> 1000 + 1 -> 1001 (representation of -7)
  - 1001 -> 0110 + 1 -> 0111 (representation of 7)

## 2s complement (cont'd)

- Why does bit-wise complement + 1 work?



## 2s complement addition and subtraction

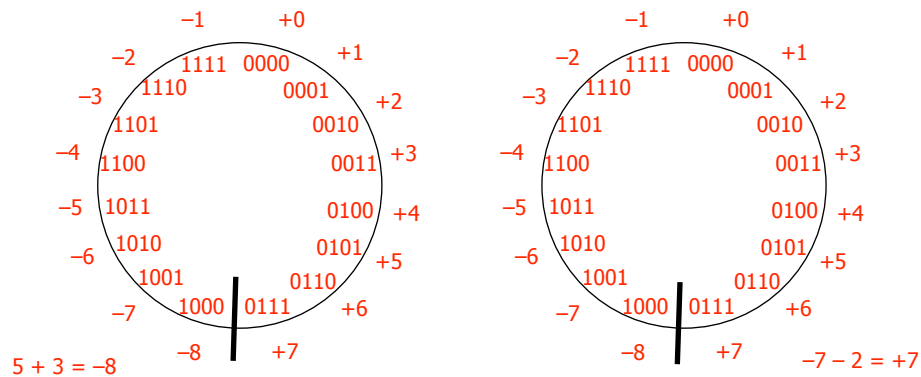
- Simple addition and subtraction
  - simple scheme makes 2s complement the unanimous choice for integer number systems in computers

$$\begin{array}{r}
 4 \quad 0100 \\
 + 3 \quad 0011 \\
 \hline
 7 \quad 0111
 \end{array}
 \qquad
 \begin{array}{r}
 -4 \quad 1100 \\
 + (-3) \quad 1101 \\
 \hline
 -7 \quad 11001
 \end{array}$$

$$\begin{array}{r}
 4 \quad 0100 \\
 - 3 \quad 1101 \\
 \hline
 1 \quad 10001
 \end{array}
 \qquad
 \begin{array}{r}
 -4 \quad 1100 \\
 + 3 \quad 0011 \\
 \hline
 -1 \quad 1111
 \end{array}$$

## Overflow in 2s complement addition/ subtraction

- Overflow conditions
  - add two positive numbers and end up with a negative number
  - add two negative numbers and end up with a positive number



## Overflow conditions

- Overflow

- $A_3' B_3' S_3 + A_3 B_3 S_3'$

- Another way to say same thing

- When carry into sign bit position is not equal to carry-out

$\begin{array}{r} 0111 \\ 5 \ 0101 \\ \underline{-3 \ 0011} \\ -8 \ 1000 \end{array}$	$\begin{array}{r} 0000 \\ 5 \ 0101 \\ \underline{-2 \ 0010} \\ 7 \ 0111 \end{array}$	$\begin{array}{r} 1000 \\ -7 \ 1001 \\ \underline{-2 \ 1110} \\ 7 \ 10111 \end{array}$	$\begin{array}{r} 1111 \\ -3 \ 1101 \\ \underline{-5 \ 1011} \\ -8 \ 11000 \end{array}$
overflow	no overflow	overflow	no overflow

## Circuits for binary addition

- Half adder (add 2 1-bit numbers)

- $\text{Sum} = A_i' B_i + A_i B_i' = A_i \text{ xor } B_i$
- $\text{Cout} = A_i B_i$

- Full adder (carry-in to cascade for multi-bit adders)

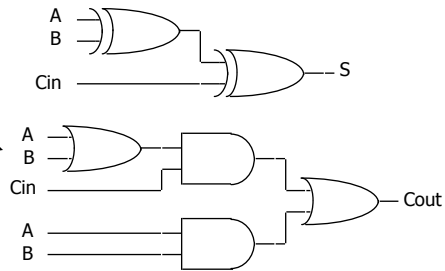
- $\text{Sum} = \text{Cin} \text{ xor } A \text{ xor } B$
- $\text{Cout} = B \text{ Cin} + A \text{ Cin} + A B = \text{Cin} (A + B) + A B$

A <sub>i</sub>	B <sub>i</sub>	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

A <sub>i</sub>	B <sub>i</sub>	C <sub>in</sub>	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

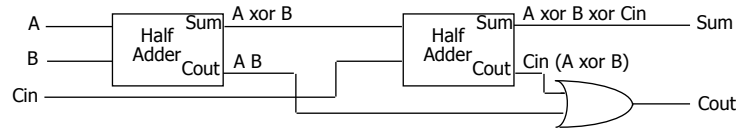
## Full adder implementations

- Standard approach
  - 6 gates
  - 2 XORs, 2 ANDs, 2 ORs



- Alternative implementation
  - 5 gates
  - half adder is an XOR gate and AND gate
  - 2 XORs, 2 ANDs, 1 OR

$$Cout = A B + Cin (A \text{ xor } B) = A B + B Cin + A Cin$$



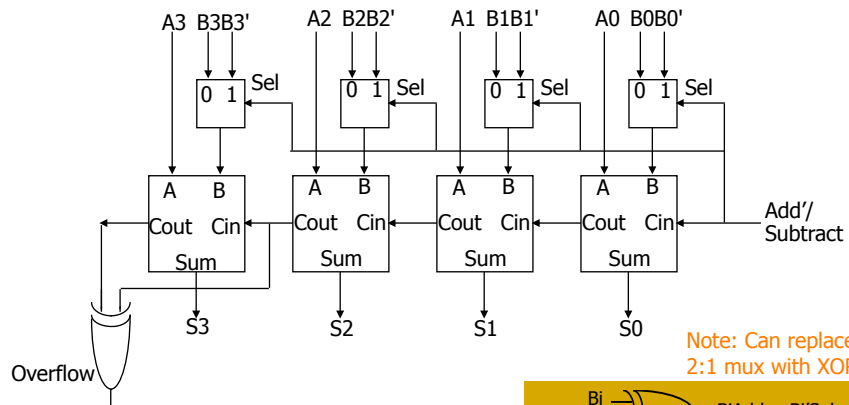
Winter 2010

CSE370 - X - Adders

13

## Adder/subtractor

- Use an adder to do subtraction thanks to 2s complement representation
  - $A - B = A + (-B) = A + B' + 1$
  - control signal selects B or 2s complement of B



Note: Can replace 2:1 mux with XOR gate

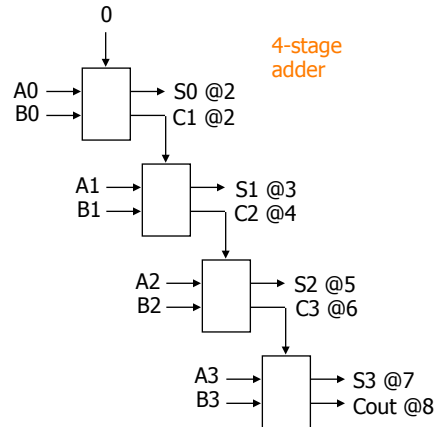
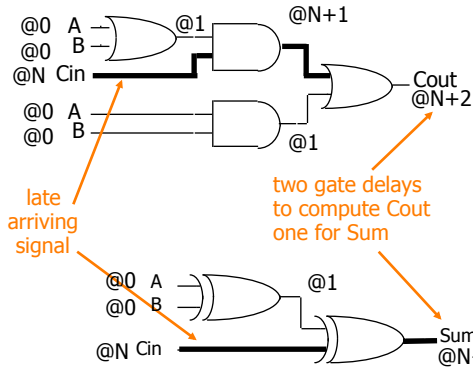
Winter 2010

CSE370 - X - Adders

14

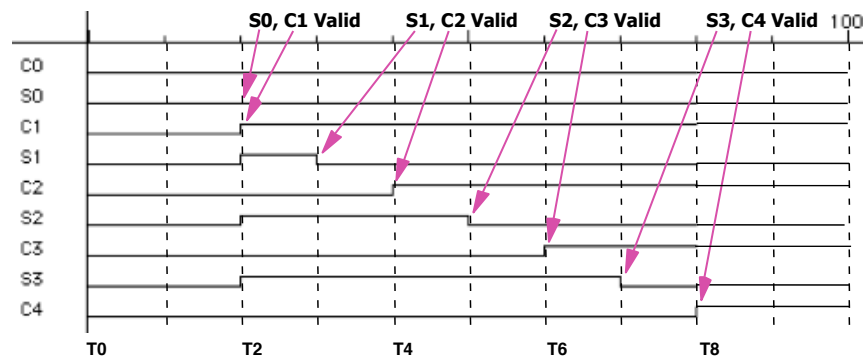
## Ripple-carry adders

- Critical delay
  - the propagation of carry from low to high order stages



## Ripple-carry adders (cont'd)

- Critical delay
  - the propagation of carry from low to high order stages
  - 1111 + 0001 is the worst case addition
  - carry must propagate through all bits





## Carry-lookahead logic

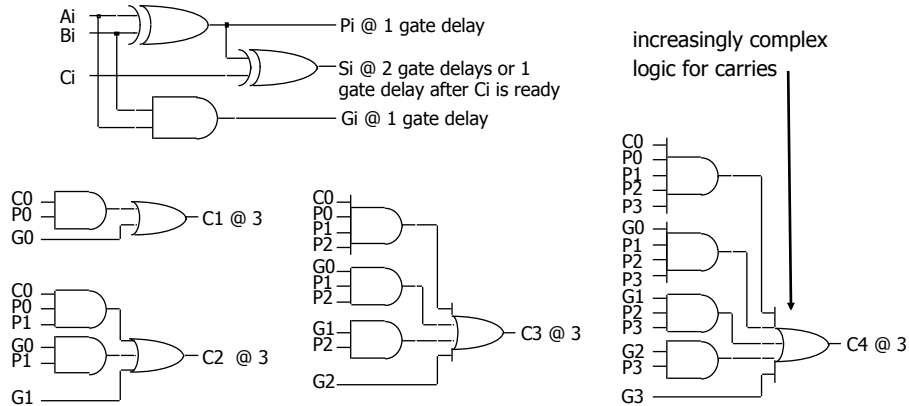
- Carry generate:  $G_i = A_i B_i$ 
  - must generate carry when  $A = B = 1$
- Carry propagate:  $P_i = A_i \text{ xor } B_i$ 
  - carry-in will equal carry-out in these two cases:  $A = 0, B = 1$  or  $A = 1, B = 0$
- Carry kill:  $K_i = A_i' B_i'$ 
  - carry-out will be zero no matter what carry-in when  $A = B = 0$
- $G_i + P_i + K_i = 1$
- Sum and Cout can be re-expressed in terms of generate/propagate (or in terms of generate/kill):
  - $S_i = A_i \text{ xor } B_i \text{ xor } C_i$   
 $= P_i \text{ xor } C_i$
  - $C_{i+1} = A_i B_i + A_i C_i + B_i C_i$   
 $= A_i B_i + C_i (A_i + B_i)$   
 $= A_i B_i + C_i (A_i \text{ xor } B_i)$   
 $= G_i + P_i C_i$
  - $C_{i+1}' = K_i + P_i C_i'$   
 $C_{i+1} = K_i' (P_i' + C_i)$

## Carry-lookahead logic (cont'd)

- Re-express the carry logic as follows:
  - $C_1 = G_0 + P_0 C_0$
  - $C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$
  - $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$
  - $C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$   
 $+ P_3 P_2 P_1 P_0 C_0$
- Each of the carry equations can be implemented with two-level logic
  - all inputs are now directly derived from data inputs and NOT from intermediate carries
  - this allows computation of all sum outputs to proceed in PARALLEL

## Carry-lookahead implementation

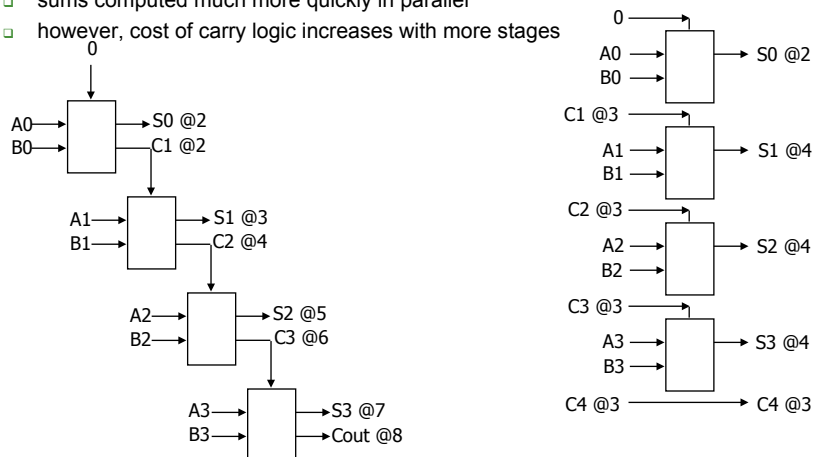
- Adder with propagate and generate outputs



## Carry-lookahead implementation (cont'd)

- Carry-lookahead logic generates individual carries

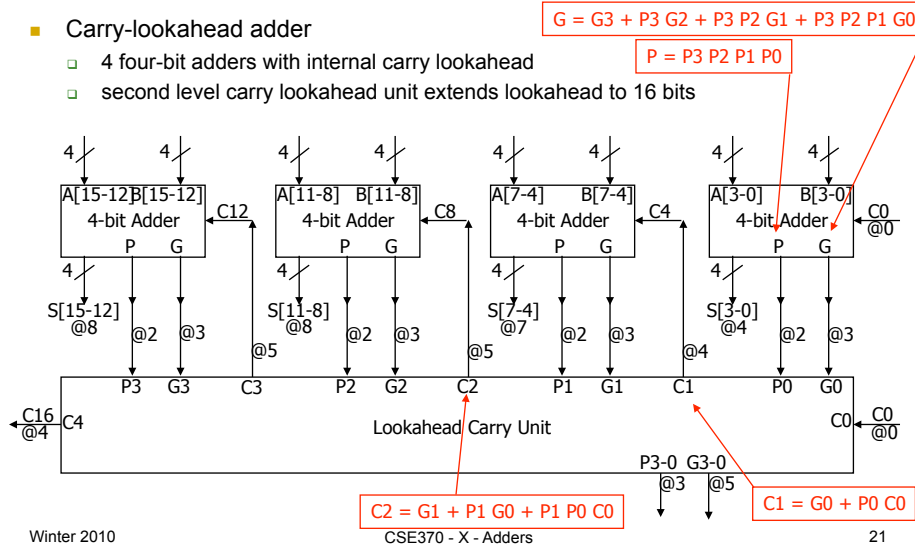
- sums computed much more quickly in parallel
- however, cost of carry logic increases with more stages



## 16-bit carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder

- 4 four-bit adders with internal carry lookahead
- second level carry lookahead unit extends lookahead to 16 bits



## Carry-select adder

- Redundant hardware to make carry calculation go faster

- compute two high-order sums in parallel while waiting for carry-in
- one assuming carry-in is 0 and another assuming carry-in is 1
- select correct result once carry-in is finally computed

