EECS 150 - Components and Design Techniques for Digital Systems

Lec 15 – Addition, Subtraction, and Negative Numbers

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~culler
http://inst.eecs.berkeley.edu/~cs150

Overview

- *Recall basic positional notation*
- *Binary Addition*
  Full Adder (Boolean Logic Revisited)
- Ripple Carry
- Carry-select adder
- Carry lookahead adder
- *Binary Number Representation*
  Sign & Magnitude, Ones Complement, Twos Complement

Manipulating representations of numbers

- **Example (from 2nd grade)**

	1	1	1	
	5	3	9	6
+	4	7	2	5
1	0	1	2	1

- **Sequence of decimal digits (radix 10)**
- **Position represents significance (most -> least)**
- **Carry into next position**
- **3-to-2 conversion at each step**
- **Results may be one digit longer, but assumed you could "make room" for it**

Positional Notation

- **Sequence of digits: $D_{k-1} D_{k-2} \ldots D_0$ represents the value**

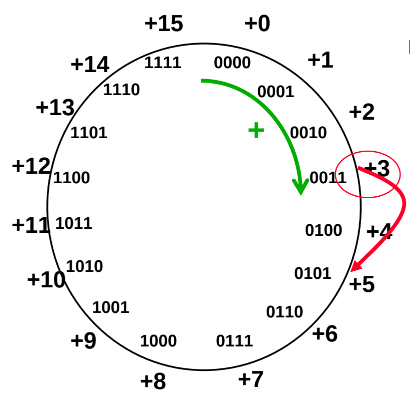
  $$D_{k-1}B^{k-1} + D_{k-2}B^{k-2} + \ldots + D_0 B^0$$

  where $D_i \in \{ 0, \ldots, B-1 \}$
- **B is the "base" or "radix" of the number system**
- **Example: 2004_{10},**
- **Can convert from any radix to any other**
  - $1101_2 = 13_{10} = 0D_{16}$
  - $1CE8_{16} = 1\cdot16^3 + 12\cdot16^2 + 14\cdot16^1 + 8\cdot16^0 = 7400_{10}$
  - $436_8 = 4\cdot8^2 + 3\cdot8^1 + 6\cdot8^0 = 286_{10}$

Computer Number Systems

- We all take positional notation for granted
  - $D_{k-1}\, D_{k-2} \ldots D_0$ represents $D_{k-1}B^{k-1} + D_{k-2}B^{k-2} + \ldots + D_0\, B^0$ where $B \in \{0, \ldots, B-1\}$
- We all understand how to compare, add, subtract these numbers
  - Add each position, write down the position bit and possibly carry to the next position
- Computers represent finite number systems
  - Generally radix 2
- How do they efficiently compare, add, sub?
  - How do we reduce it to networks of gates and FFs?
- Where does it break down?
  - Manipulation of finite representations doesn't behave like same operation on conceptual numbers

Unsigned Numbers - Addition



Example: 3 + 2 = 5

Unsigned binary addition

Is just addition, base 2

Add the bits in each position and carry

$$\begin{array}{r} {\scriptstyle 1} \\ 0\,0\,1\,1 \\ +\ \ 0\,0\,1\,0 \\ \hline 0\,1\,0\,1 \end{array}$$

How do we build a combinational logic circuit to perform addition?

=> Start with a truth table and go from there

Binary Addition: Half Adder

Ai	Bi	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

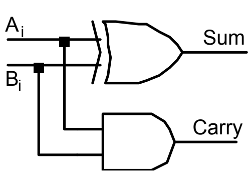
Bi \ Ai	0	1
0	0	1
1	1	0

Sum = $\overline{Ai}\, Bi + Ai\, \overline{Bi}$

= $Ai \oplus Bi$

Bi \ Ai	0	1
0	0	0
1	0	1

Carry = Ai Bi

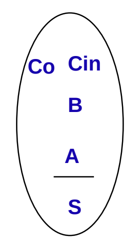


Half-adder Schematic

But each bit position may have a carry in…

Full-Adder (derivation)

$$\begin{array}{r} {\scriptstyle 1\,0} \\ 0\,0\,1\,1 \\ +\ \ 0\,0\,1\,0 \\ \hline 0\,1\,0\,1 \end{array}$$



A	B	CI	CO	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

S

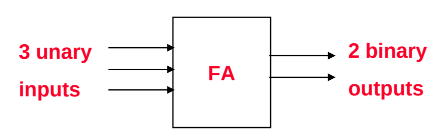
CI \ A B	00	01	11	10
0	0	1	0	1
1	1	0	1	0

CO

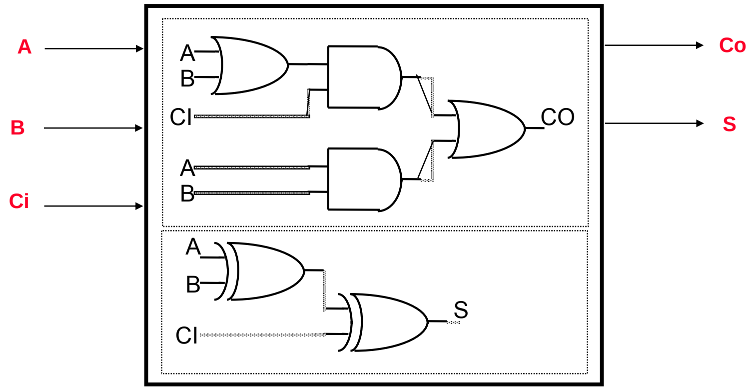
CI \ A B	00	01	11	10
0	0	0	1	0
1	0	1	1	1

S = CI xor A xor B

CO = B CI + A CI + A B = CI (A + B) + A B

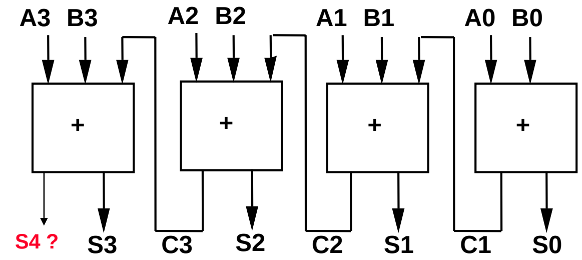


Full Adder



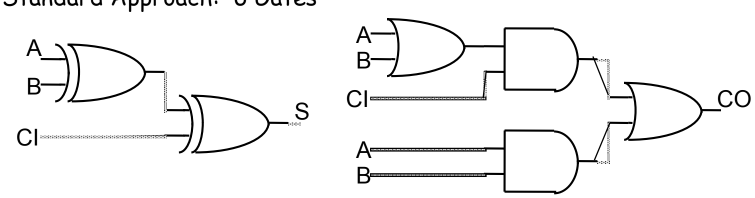
Now we can connect them up to do multiple bits…

Ripple Carry

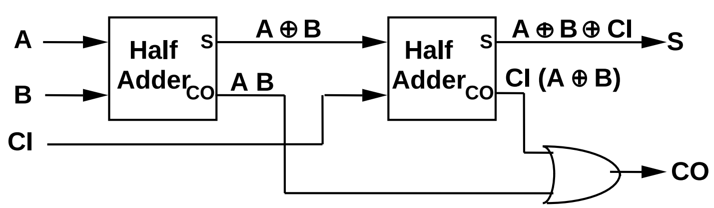


Full Adder from Half Adders (little aside)

Standard Approach: 6 Gates



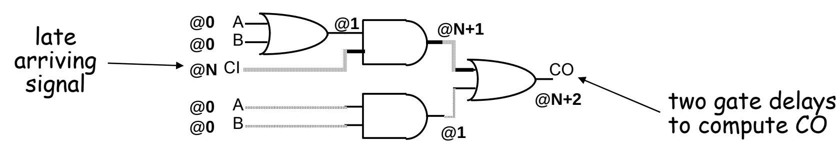
Alternative Implementation: 5 Gates

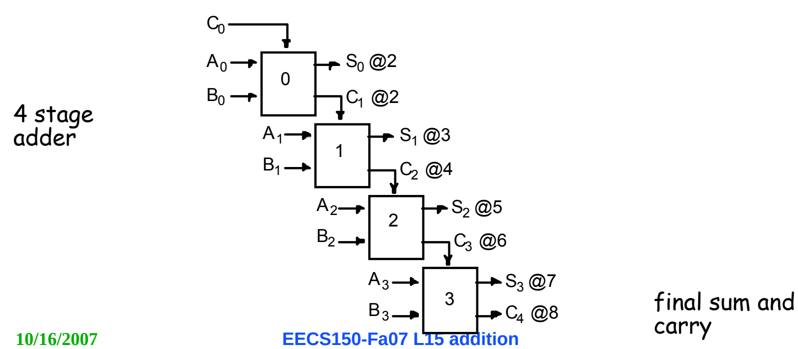


A B + CI (A xor B) = A B + B CI + A CI

Delay in the Ripple Carry Adder

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

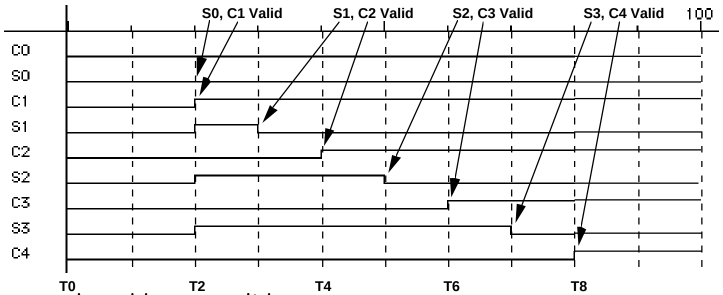




Ripple Carry Timing

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

1111 + 0001 worst case addition



T0: Inputs to the adder are valid

T2: Stage 0 carry out (C1)

T4: Stage 1 carry out (C2)

T6: Stage 2 carry out (C3)

T8: Stage 3 carry out (C4)

2 delays to compute sum

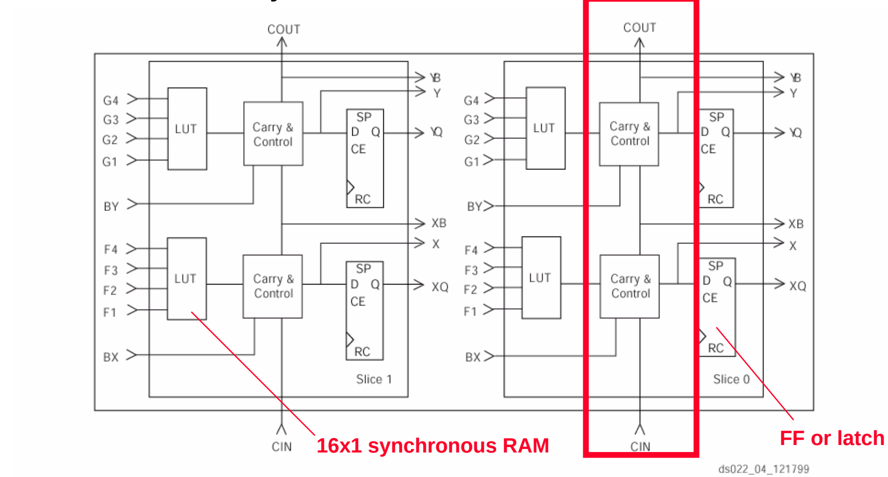
but last carry not ready until 6 delays later

Recall: Virtex-E CLB

CLB = 4 logic cells (LC) in two slices

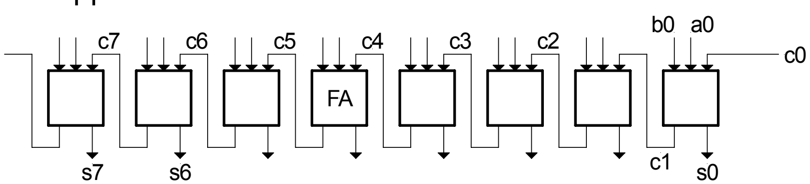
LC: 4-input function generator, carry logic, storage ele't

80 x 120 CLB array on 2000E



Adders (cont.)

Ripple Adder



Ripple adder is inherently slow because, in general s7 must wait for c7 which must wait for c6 …

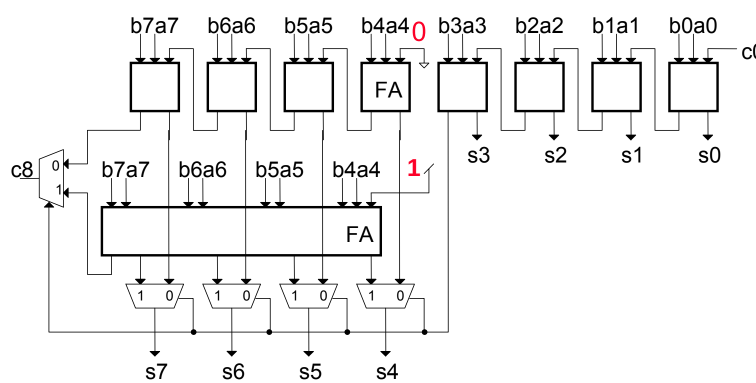
$\boldsymbol{T} \propto n, \ \boldsymbol{Cost} \propto n$

How do we make it faster, perhaps with more cost?

Classic approach: Carry Look-Ahead

Or use a MUX !!!

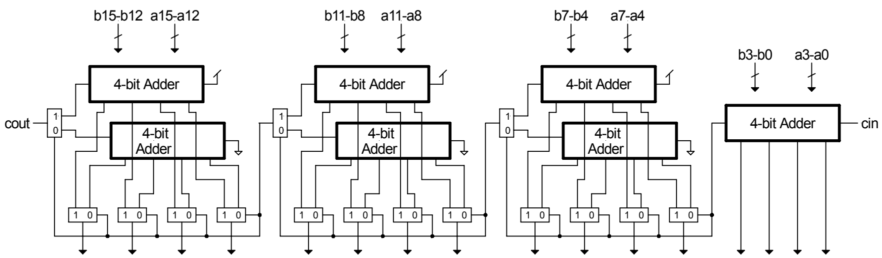
Carry Select Adder



$T = T_{ripple\_adder} / 2 + T_{MUX}$

$COST = 1.5 * COST_{ripple\_adder} + (n+1) * COST_{MUX}$

Extended Carry Select Adder

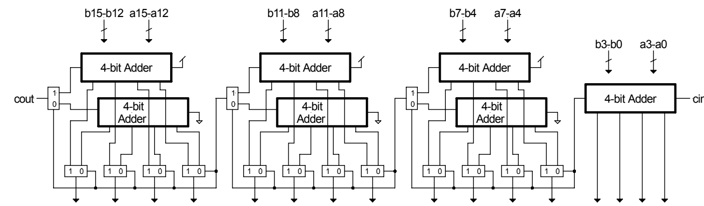


- What is the optimal # of blocks and # of bits/block?
  - If # blocks too large delay dominated by total mux delay
  - If # blocks too small delay dominated by adder delay per block

$\sqrt{N}$ stages of $\sqrt{N}$ bits

T α sqrt(N),
Cost ≈2*ripple + muxes

Carry Select Adder Performance

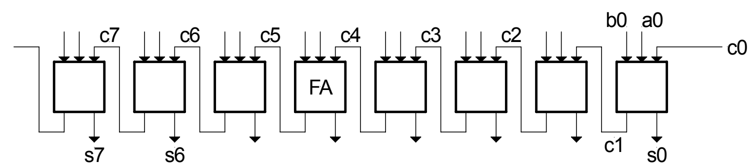


- **Compare to ripple adder delay:**
  - $T_{total} = 2\ \mathrm{sqrt}(N)\ T_{FA} - T_{FA}$, assuming $T_{FA} = T_{MUX}$
  - For ripple adder $T_{total} = N\ T_{FA}$
  - "cross-over" at N=3, Carry select faster for any value of N>3.
- **Is sqrt(N) really the optimum?**
  - From right to left increase size of each block to better match delays
  - Ex: 64-bit adder, use block sizes [12 11 10 9 8 7 7]
- **How about recursively defined carry select?**

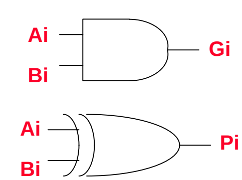
Announcements

- **Reading Katz 5.6 and Appendix A**
- **Mid III will just stay put in final slot – no more fussing with it.**
- **Project demo at Lab Lecture friday**
- **Don't hedge on lab workload reporting**
  - It matters to us and is NOT a negative in your grade
- **Lab5 | CP1 | CP2 crunch**
  - It should lighten
  - Don't hesitate to get guidance on the specifics of your approach from the TAs. They are there to help.
  - Lab5 solution walk-thru Friday after lab lecture

What really happens with the carries



A	B	Cout	S	Carry action
0	0	0	Cin	kill
0	1	Cin	~Cin	Propagate
1	0	Cin	~Cin	propagate
1	1	1	Cin	generate

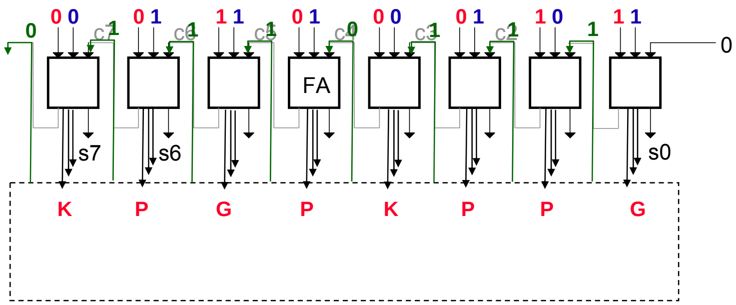


Carry Generate **Gi** = **Ai Bi** *must generate carry when A = B = 1*

Carry Propagate **Pi** = **Ai xor Bi** *carry in will equal carry out here*

All generates and propagates in parallel at first stage. No ripple.

Carry Kill / Prop / Gen example



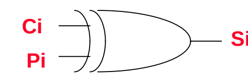
Carry Look Ahead Logic

Carry Generate Gi = Ai Bi *must generate carry when A = B = 1*

Carry Propagate Pi = Ai xor Bi *carry in will equal carry out here*

Sum and Carry can be re-expressed in terms of generate/propagate:

Si = Ai xor Bi xor Ci = **Pi xor Ci**

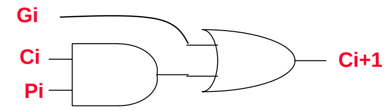


Ci+1 = Ai Bi + Ai Ci + Bi Ci

= Ai Bi + Ci (Ai + Bi)

= Ai Bi + Ci (Ai xor Bi)

= **Gi + Ci Pi**



All Carries in Parallel

Re-express the carry logic for each of the bits:

C1 = G0 + P0 C0

C2 = G1 + P1 C1 = G1 + P1 G0 + P1 P0 C0

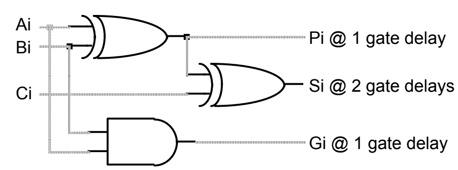
C3 = G2 + P2 C2 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 C0

C4 = G3 + P3 C3 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0 + P3 P2 P1 P0 C0

Each of the carry equations can be implemented in a two-level logic network

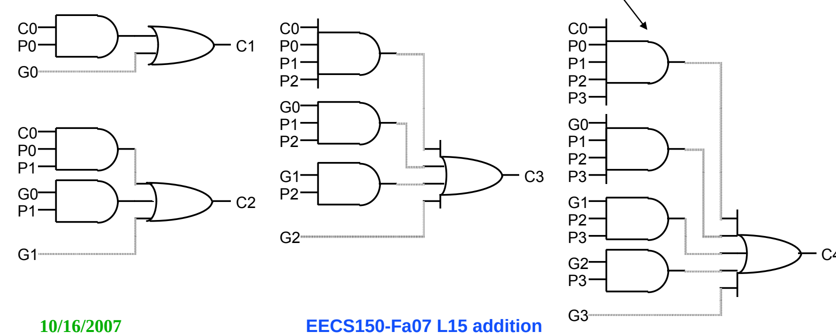
Variables are the adder inputs and carry in to stage 0!

CLA Implementation

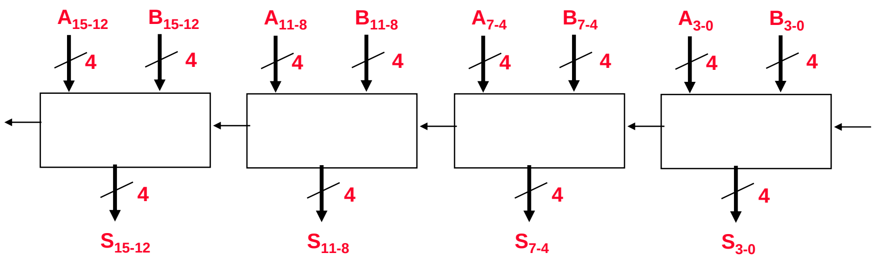


Adder with Propagate and Generate Outputs

Increasingly complex logic

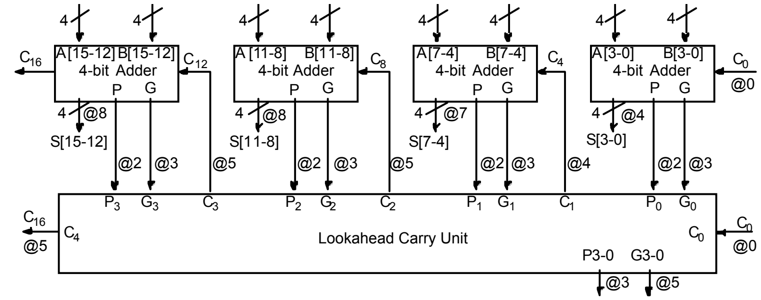


How do we extend this to larger adders?



- **Faster carry propagation**
  - **4 bits at a time**
- **But still linear**
- **Can we get to log?**
- **Compute propagate and generate for each adder BLOCK**

Cascaded Carry Lookahead



4 bit adders with internal carry lookahead

second level carry lookahead unit, extends lookahead to 16 bits

One more level to 64 bits

Trade-offs in combinational logic design

- *Time vs. Space Trade-offs*

  Doing things fast requires more logic and thus more space

  Example: carry lookahead logic

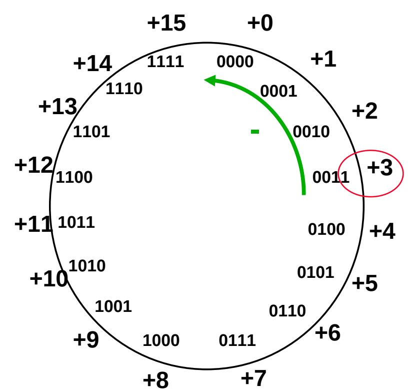
- *Simple with lots of gates vs complex with fewer*

- *Arithmetic Logic Units*

  Critical component of processor datapath

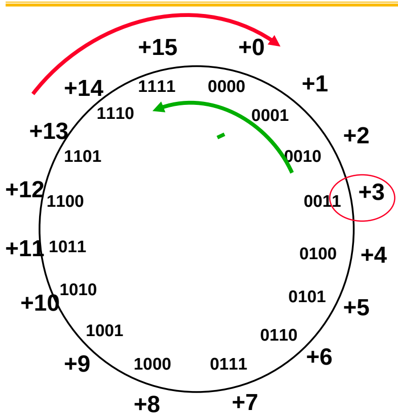
  Inner-most "loop" of most computer instructions

So what about subtraction?



- **Develop subtraction circuit using the same process**
  - **Truth table for each bit slice**
  - **Borrow in from slice of lesser significance**
  - **Borrow out to slice of greater significance**
  - **Very much like carry chain**
- **Homework exercise**

Finite representation?



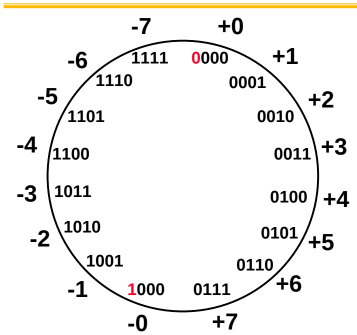
- What happens when $A + B > 2^N - 1$?
  - Overflow
  - Detect?
    - » Carry out
- What happens when $A - B < 0$?
  - Negative numbers?
  - Borrow out?

Number Systems

- Desirable properties:
  - Efficient encoding (2^n bit patterns. How many numbers?)
  - Positive and negative
    - » Closure (almost) under addition and subtraction
      - Except when overflow
    - » Representation of positive numbers same in most systems
    - » Major differences are in how negative numbers are represented
  - Efficient operations
    - » Comparison: =, <, >
    - » Addition, Subtraction
    - » Detection of overflow
  - Algebraic properties?
    - » Closure under negation?
    - » A == B iff A – B == 0
- Three Major schemes:
  - sign and magnitude
  - ones complement
  - twos complement
  - (excess notation)

Which one did you learn in 2nd grade?

Sign and Magnitude



+

0 100 = + 4

1 100 = - 4

-

Example: N = 4

High order bit is sign: 0 = positive (or zero), 1 = negative

Remaining low order bits is the magnitude: 0 (000) thru 7 (111)

Number range for n bits = +/- $2^{n-1} - 1$

Representations for 0?

Operations: =, <, >, +, - ???

Ones Complement

Bit manipulation:

simply complement each of the bits

0111 -> 1000

Algebraically ...

N is positive number, then $\overline{N}$ is its negative 1's complement

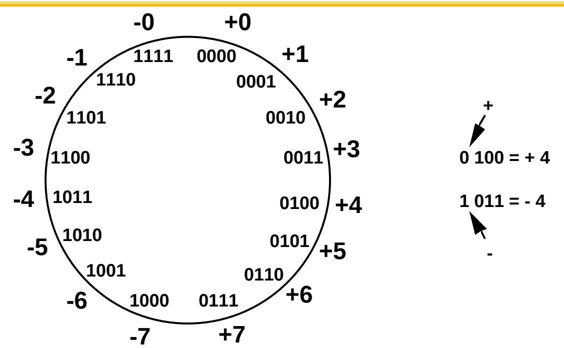
$\overline{N} = (2^n - 1) - N$

Example: 1's complement of 7

2^4	10000
-1	- 00001
	1111
-7	- 0111
	1000

-7 in 1's comp.

Ones Complement on the number wheel



Subtraction implemented by addition & 1's complement

Sign is easy to determine

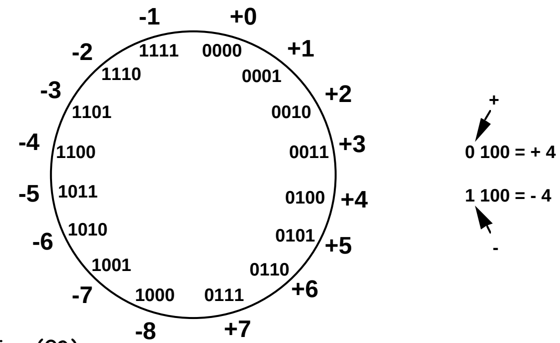
Closure under negation. If A can be represented, so can -A

Still two representations of 0!

If A = B then is A – B == 0 ?

Addition is **almost** clockwise advance, like unsigned

Twos Complement number wheel



Easy to determine sign (0?)

Only one representation for 0

Addition and subtraction just as in unsigned case

Simple comparison: A < B iff A – B < 0

One more negative number than positive number

- one number has no additive inverse

Twos Complement (algebraically)

$N^* = 2^n - N$

Example: Twos complement of 7

$$\begin{array}{rrl} & 2^4 = & 10000 \\ \text{sub } 7 & = & \underline{0111} \\ & & 1001 \text{ = repr. of -7} \end{array}$$

Example: Twos complement of -7

$$\begin{array}{rrl} & 2^4 = & 10000 \\ \text{sub } -7 & = & \underline{1001} \\ & & 0111 \text{ = repr. of 7} \end{array}$$

Bit manipulation:

Twos complement: take bitwise complement and add one

0111 -> 1000 + 1 -> 1001 (representation of -7)

1001 -> 0110 + 1 -> 0111 (representation of 7)

How is addition performed in each number system?

- **Operands may be positive or negative**

Sign Magnitude Addition

Operand have same sign: unsigned addition of magnitudes

result sign bit is the same as the operands' sign

4	0100	-4	1100
+ 3	0011	+ (-3)	1011
7	0111	-7	1111

Operands have different signs:

subtract smaller from larger and keep sign of the larger

4	0100	-4	1100
- 3	1011	+ 3	0011
1	0001	-1	1001

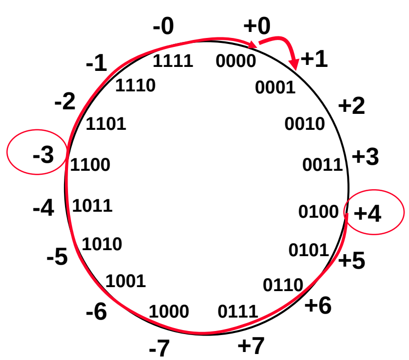
Ones complement addition

Perform unsigned addition, then add in the end-around carry

4	0100	-4	1011
+ 3	0011	+ (-3)	1100
7	0111	-7	10111
		End around carry	1
			1000

4	0100	-4	1011
- 3	1100	+ 3	0011
1	10000	-1	1110
End around carry	1		
	0001		

When carry occurs



\+

0 100 = + 4

1 011 = - 4

\-

M – N where M > N

-M - N

Why does end-around carry work?

Recall: $\overline{N} = (2^n - 1) - N$

End-around carry work is equivalent to subtracting 2^n and adding 1

$M - N = M + \overline{N} = M + (2^n - 1 - N) = (M - N) + 2^n - 1$ (when M > N)

$-M + (-N) = \overline{M} + \overline{N} = (2^n - M - 1) + (2^n - N - 1)$

$= 2^n + [2^n - 1 - (M + N)] - 1$

$M + N < 2^{n-1}$

after end around carry:

$= 2^n - 1 - (M + N)$

this is the correct form for representing -(M + N) in 1's comp!

Twos Complement Addition

Perform unsigned addition and Discard the carry out.

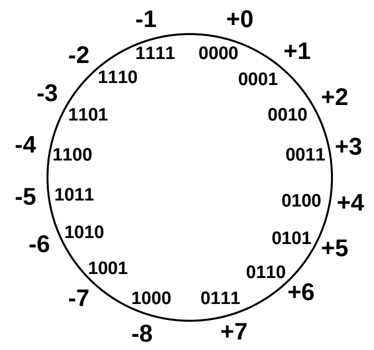
Overflow?

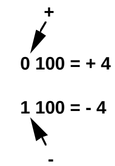
4	0100	-4	1100
+ 3	0011	+ (-3)	1101
7	0111	-7	11001

4	0100	-4	1100
- 3	1101	+ 3	0011
1	10001	-1	1111

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

Twos Complement number wheel





$-M + -N$ where $N + M \le 2n-1$

$-M + N$ when $N > M$

2s Comp: ignore the carry out

$-M + N$ when $N > M$:

$$M^* + N = (2^n - M) + N = 2^n + (N - M)$$

Ignoring carry-out is just like subtracting 2^n

$-M + -N$ where $N + M \le 2^{n-1}$

$$-M + (-N) = M^* + N^* = (2^n - M) + (2^n - N)$$
$$= 2^n - (M + N) + 2^n$$

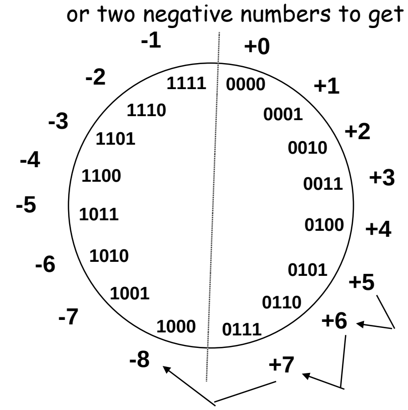
After ignoring the carry, this is just the right twos compl. representation for $-(M + N)$!

2s Complement Overflow

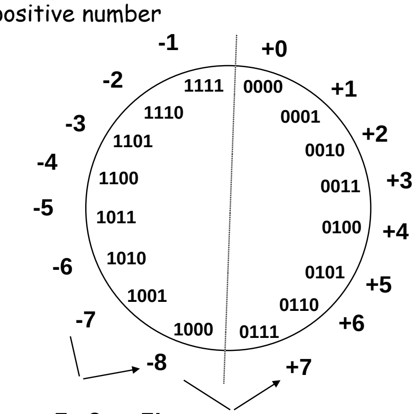
How can you tell an overflow occurred?

Add two positive numbers to get a negative number

or two negative numbers to get a positive number



5 + 3 = -8!



-7 - 2 = +7!

2s comp. Overflow Detection

	0 1 1 1
5	0 1 0 1
3	0 0 1 1
-8	1 0 0 0

Overflow

	1 0 0 0
-7	1 0 0 1
-2	1 1 0 0
7	1 0 1 1 1

Overflow

	0 0 0 0
5	0 1 0 1
2	0 0 1 0
7	0 1 1 1

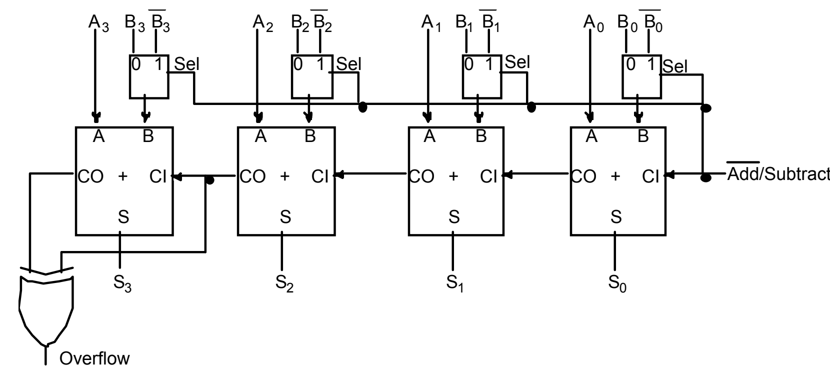
No overflow

	1 1 1 1
-3	1 1 0 1
-5	1 0 1 1
-8	1 1 0 0 0

No overflow

Overflow occurs when carry in to sign does not equal carry out

2s Complement Adder/Subtractor



$$A - B = A + (-B) = A + \overline{B} + 1$$

Summary

- Circuit design for unsigned addition
  - Full adder per bit slice
  - Delay limited by Carry Propagation
    - » Ripple is algorithmically slow, but wires are short
- Carry select
  - Simple, resource-intensive
  - Excellent layout
- Carry look-ahead
  - Excellent asymptotic behavior
  - Great at the board level, but wire length effects are significant on chip
- Digital number systems
  - How to represent negative numbers
  - Simple operations
  - Clean algorithmic properties
- 2s complement is most widely used
  - Circuit for unsigned arithmetic
  - Subtract by complement and carry in
  - Overflow when cin xor cout of sign-bit is 1