Introduction to Computer Engineering – EECS 203

http://ziyang.eecs.northwestern.edu/~dickrp/eecs203/

Instructor:	Robert Dick	TA:	Neal Oza
Office:	L477 Tech	Office:	Tech. Inst. L375
Email:	dickrp@northwestern.edu	Phone:	847-467-0033
Phone:	847–467–2298	Email:	nealoza@u.northwestern.edu
		TT:	David Bild
		Office:	Tech. Inst. L470
		Phone:	847-491-2083
		Email:	d-bild@northwestern.edu

NORTHWESTERN UNIVERSITY

Other number/encoding systems

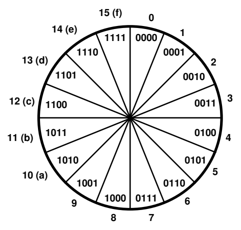
- Binary-coded decimal
  - Wastes fractional bit for alignment
- ASCII: 7-bit characters
  - Parity
  - Best to see a chart
  - 0x61 = 97 = 'a'
  - 0x47 = 71 = 'G'

Other number/encoding systems

Unicode: 16-bit

- Similar to ASCII
- International
- Allows about $2^{16} = 65,536$ characters
- Enough for symbolic writing systems

Standard unsigned binary numbers



Given an n-bit number in which d_i is the ith digit, the number is

$$\sum_{i=1}^{n} 2^{i-1} d_i$$

Unsigned addition

Consider adding 9 (1001) and 3 (0011)

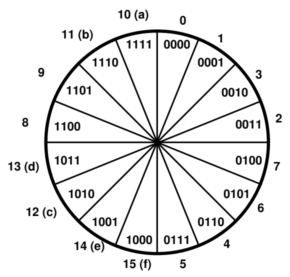
$$\begin{array}{rcccc} & & 1 & 1 & \\ & 1 & 0 & 0 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Why an extra column?

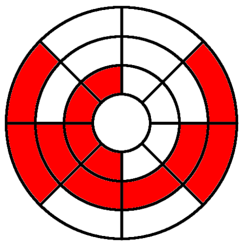
Overflow

- If the result of an operation can't be represented in the available number of bits, an *overflow* occurs
- E.g., 0110 + 1011 = 10001
- Need to detect overflow

Gray code



Useful for shaft encoding



Sequence?

Gray code

- To convert from a standard binary number to a Gray code number XOR the number by it's half (right-shift it)
- To convert from a Gray code number to a standard binary number, XOR each binary digit with the parity of the higher digits

Given that a number contains n digits and each digit, d_i, contributes 2^{i-1} to the number

$$\mathcal{P}_j^k = d_j \oplus d_{j+1} \cdots \oplus d_{k-1} \oplus d_k$$
$$d_i = d_i \oplus \mathcal{P}_{i+1}^n$$

Gray code

- Converting from Gray code to standard binary is difficult
  - Take time approximately proportional to n
- Doing standard arithmetic operations using Gray coded numbers is difficult
- Generally slower than using standard binary representation
- E.g., addition requires two carries
- Why use Gray coded numbers?
  - Analog to digital conversion
  - Reduced bus switching activity

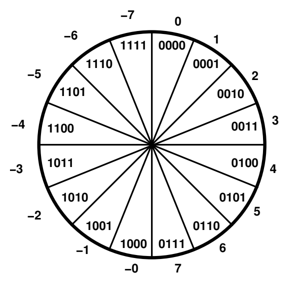
Signed number systems

- Three major schemes
  - Sign and magnitude
  - One's complement
  - Two's complement

Number system assumptions

- Four-bit machine word
- 16 values can be represented
- Approximately half are positive
- Approximately half are negative

Sign and magnitude



Sign and magnitude

- d_n represents sign
  - 0 is positive, 1 is negative
- Two representations for zero
- What is the range for such numbers?
  - Range: $[-2^{n-1}+1, 2^{n-1}-1]$

Sign and magnitude

- How is addition done?
- If both numbers have the same sign, add them like unsigned numbers and preserve sign
- If numbers have differing signs, subtract smaller magnitude from larger magnitude and use sign of large magnitude number

Sign and magnitude

- Consider $5 + -6$
- Note that signs differ
- Use magnitude comparison to determine large magnitude: $6 - 5$
- Subtract smaller magnitude from larger magnitude: 1
- Use sign of large magnitude number: -1

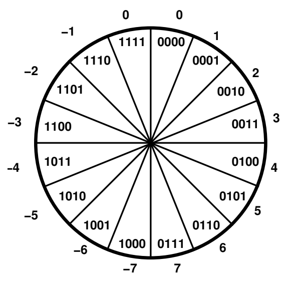
Direct subtraction

Consider subtracting 5 (0101) from 6 (0110)

$$\begin{array}{ccccc} & & & b & \\ & 0 & 1 & 1 & 0 \\ - & 0 & 1 & 0 & 1 \\ \hline & 0 & 0 & 0 & 1 \end{array}$$

- Note that this operation is different from addition
- Sign and magnitude addition is complicated

One's compliment



One's compliment

- If negative, complement all bits
- Addition somewhat simplified
- Do standard addition except wrap around carry back to the 0th bit
- Potentially requires two additions of the whole width
  - Slow

One's complement addition

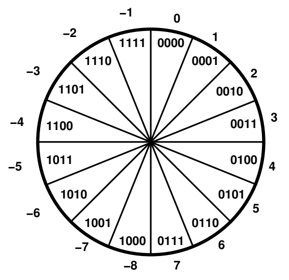
Consider adding -5 (1010) and 7 (0111)

$$\begin{array}{ccccc} & 1 & 1 & 1 & 1 \\ & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 \\ \hline & 0 & 0 & 01 & 10 \end{array}$$

Two's complement

- To negate a number, invert all its bits and add 1
- Like one's complement, however, rotated by one bit
- Counter-intuitive
  - However, has some excellent properties

Two's complement



Two's complement

- Only one zero
  - Leads to more natural comparisons
- One more negative than positive number
  - This difference is irrelevant as *n* increases
- Substantial advantage – Addition is easy!

Two's complement addition

Consider adding -4 (1100) and 6 (0110)

$$\begin{array}{ccccc} 1 & 1 & & & \\ & 1 & 1 & 0 & 0 \\ + & 0 & 1 & 1 & 0 \\ \hline & 0 & 0 & 1 & 0 \end{array}$$

Two's complement

- No looped carry – Only one addition necessary
- If carry-in to most-significant bit $\neq$ carry-out to most-significant bit, overflow occurs
- What does this represent?
- Both operands positive and have carry-in to sign bit
- Both operands negative and don't have carry-in to sign bit

Two's complement overflow

a	b	cin	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

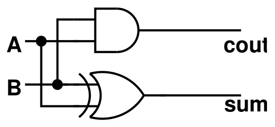
Half adder

For two's complement, don't need subtracter

A	B	cout	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$cout = AB$$
$$sum = A \oplus B$$

Half adder



Full adder

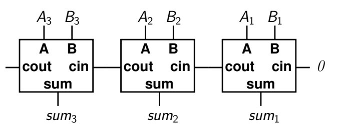
Need to deal with carry-in

A	B	cin	cout	sum
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

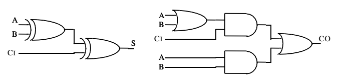
Full adder

$$sum = A \oplus B \oplus cin$$
$$cout = AB + A\ ci + B\ ci$$

Cascaded full-adders

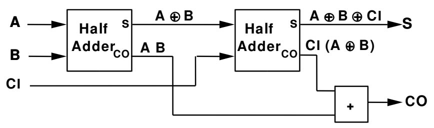


Full adder standard implementation



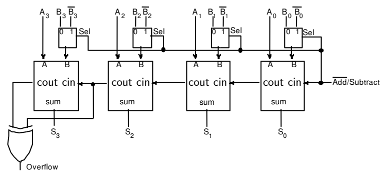
Six logic gates

Full adder composed of half-adders



$$AB + ci(A \oplus B) = AB + B\ ci + A\ ci$$

Adder/subtracter



Consider input to cin

Ripple-carry delay analysis

- The critical path (to *cout*) is two gate delays per stage
- Consider adding two 32-bit numbers
- 64 gate delays
  - Too slow!
- Consider faster alternatives

Carry lookahead adder

- Lecture notes give detail for completeness
- However, primarily important to understand that carry lookahead compresses many levels of a carry chain into fewer levels for speed
- Will need to understand carry select adders in detail
- Carry generate: $G = AB$
- Carry propagate: $P = A \oplus B$
- Represent *sum* and *cout* in terms of G and P

Overview: Carry lookahead adder

$$\begin{aligned} sum &= A \oplus B \oplus cin \\ &= P \oplus cin \end{aligned}$$

$$\begin{aligned} cout &= AB + A\ cin + B\ cin \\ &= AB + cin(A + B) \\ &= AB + cin(A \oplus B) \\ &= G + cin\ P \end{aligned}$$

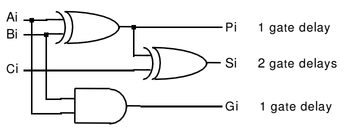
Overview: Carry lookahead adder

Flatten carry equations

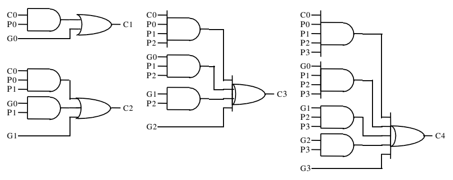
$$\begin{aligned} cin_1 &= G_0 + P_0\ cout_0 \\ cin_2 &= G_1 + P_1\ cout_1 = G_1 + P_1 G_0 + P_1 P_0\ cout_0 \\ cin_3 &= G_2 + P_2\ cout_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0\ cout_0 \\ cin_4 &= G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + \\ &\quad P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0\ cout_0 \end{aligned}$$

Each *cin* can be implemented in three-level logic

Carry lookahead building block



Carry lookahead adder



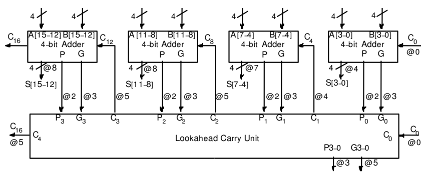
Carry lookahead delay analysis

- Assume a 4-stage adder with CLA
- Propagate and generate signals available after 1 gate delays
- Carry signals for slices 1 to 4 available after 3 gate delays
- Sum signal for slices 1 to 4 after 4 gate delays

Carry lookahead

- No carry chain slowing down computation of most-significant bit
  - Computation in parallel
- More area required
- Each bit has more complicated logic than the last
- Therefore, limited bit width for this type of adder
- Can chain multiple carry lookahead adders to do wide additions
- Note that even this chain can be accelerated with lookahead
  - Use internal and external carry lookahead units

Cascaded carry lookahead adder



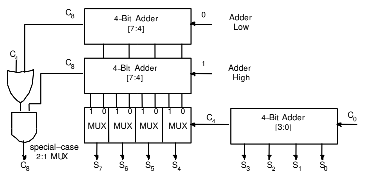
Delay analysis for cascaded carry lookahead

- Four-stage 16-bit adder
- *cin* for MSB available after five gate delays
- *sum* for MSB available after eight gate delays
- 16-bit ripple-carry adder takes 32 gate delays
- Note that not all gate delays are equivalent
- Depends on wiring, driven load
- However, carry lookahead is usually much faster than ripple-carry

Carry select adders

- Trade even more hardware for faster carry propagation
- Break a ripple carry adder into two chunks, *low* and *high*
- Implement two *high* versions
  - $high_0$ computes the result if the carry-out from *low* is 0
  - $high_1$ computes the result if the carry-out from *low* is 1
- Use a MUX to select a result once the carry-out of *low* is known
  - $high_0$'s *cout* is never greater than $high_1$'s *cout* so special-case MUX can be used

Carry select adder



Delay analysis of carry select adder

- Consider 8-bit adder divided into 4-bit stages
- Each 4-bit stage uses carry lookahead
- The 2:1 MUX adds two gate delays
- 8-bit sum computed after 6 gate delays
- 7 gate delays for carry lookahead
- 16 gate delays for ripple carry

Summary

- Number systems
- Adders and subtracters

Reading assignment

- M. Morris Mano and Charles R. Kime. *Logic and Computer Design Fundamentals*. Prentice-Hall, NJ, fourth edition, 2008
- Finish Sections 5.1–5.6