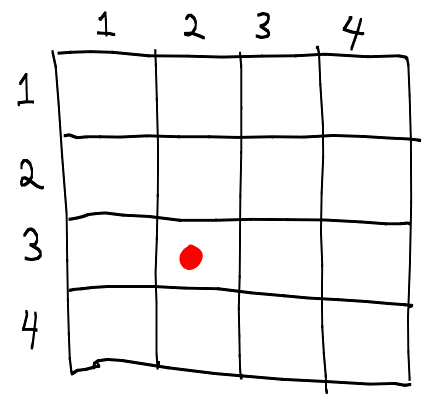
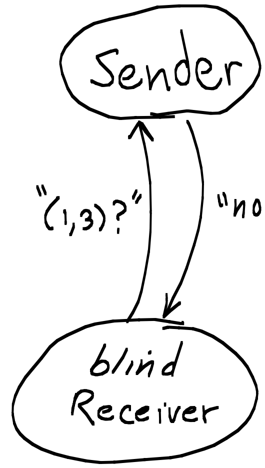
CODING and INFORMATION
We need encodings for data.

where's the ball?





- Ask yes/no questions.
- Are all questions-answers equally informative?
- Min number of questions?

  How many questions must be asked to be certain where the ball is. (cases: avg, worst, best)

- How much do you learn from each yes/no?

- Is that a Good series of questions, "In (1,3)"?

Avg number of questions needed (assume equally likely boxes)?

P(Hit 1st) = 1/16

$\Rightarrow$ P(Hit 2nd) = P(Hit 2nd | Miss 1st)P(Miss 1st) = (1/15)(15/16) = 1/16

P(Hit 3rd) = (1/14) * P(Miss 2nd and 1st) = (1/14)(14/15)(15/16) = 1/16

E(n) = 1*(1/16) + 2*(1/16) + ... + 15*(1/16) + 15(1/16) = (1+2+3+...+15+15) / 16 = 8 1/2 - 1/16

- Different set of question?

  "in (1,*) or (2,*)", "in (1,*)", "in (2,1) or (2,2)" ...

  $\Rightarrow$ Each Q reduces space by ½ $\Rightarrow$ Exactly 4 questions

$\Rightarrow$ Most information if each Q-A splits possibilities 50-50.

Measure info content of answer?

prob(yes) = prob(no) $\Rightarrow$ $\log_2(\text{Prob(yes)})$

$= \log_2(2^{-1})$

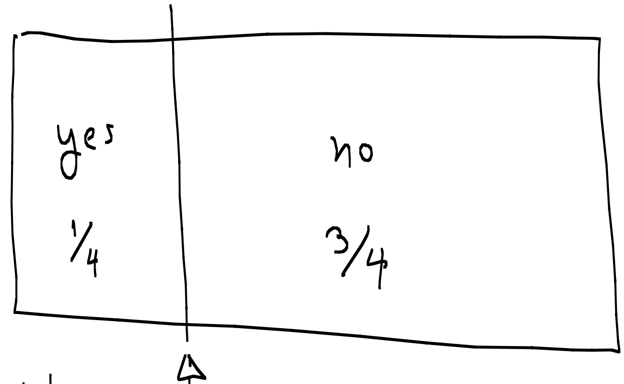
$= -1$ (hmm, make +?)

$\Rightarrow$ 1 bit

$\Downarrow$

$-\log_2(\text{Prob})$ = **info measure**

$\text{Prob(yes)} = 1/4$
$\text{Prob(no)} = 3/4$ } assume



Q = "Left of here?"

yes: $\log(2^{-2}) = 2$ bits

no: $\log(3/4) \approx -\log(.7) \approx -\log(1/\sqrt{2}) = 1/2$ bit

Avg info of answer?

$= \text{prob(yes)} \cdot (2 \text{ bits}) + \text{prob(no)} \cdot (1/2 \text{ bit})$

$= 1/4\,(2) + 3/4\,(1/2) = \frac{1}{2} + 3/8 = 7/8$ bit

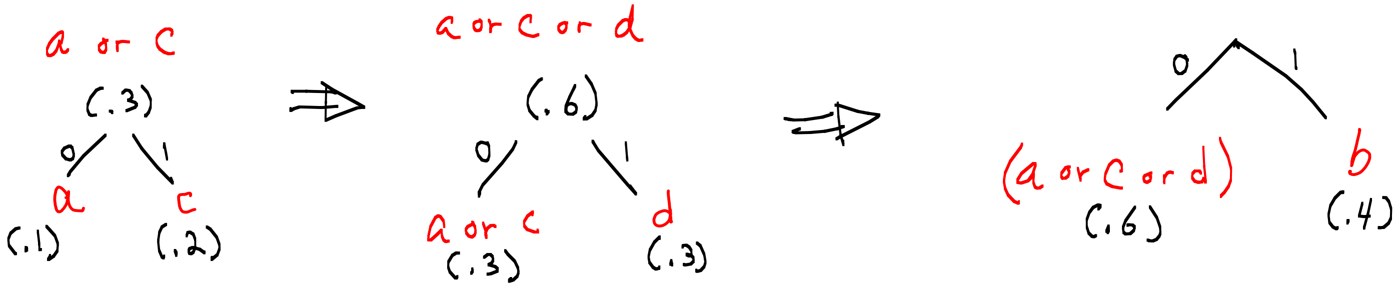
Extreme: $\text{Prob(yes)} = 1/2^{10}$ $\quad \text{Prob(no)} = (2^{10}-1)/2^{10} \approx 1$

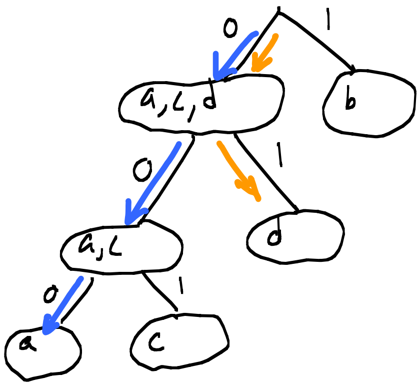
$\text{avg} = \frac{1}{2^{10}}(10 \text{ bits}) + (1)\underbrace{\log(1)}_{\to\, 0 \text{ bits}} \Rightarrow 1/100$ bit

Thm max avg. if $P_i = \frac{1}{n}$ for n possibilities.
$(\sum p_i = 1)$

How to encode to get 50-50? $\Rightarrow$ (Pair least Probable) ~ equally likely.

(a, b, c, d) w/ prob. (.1, .4, .2, .3)





Huffman Code	msg
0 0 0	a
0 0 1	c
0 1	d
1	b

Shannon Info Theory = Expected # bits = $E[(-\log(P_r))]$

Avg. content (Entropy, H) $= -\left[\underset{a}{0.1\log(0.1)} + \underset{b}{0.4\log(0.4)} + \underset{c}{0.2\log(0.2)} + \underset{d}{0.3\log(0.3)}\right]$

$= 0.33 + 0.53 + 0.46 + 0.52 = 1.84$ bits per message.

How'd we do?

Avg. # bits $= \underset{a}{(0.1)\,3} + \underset{b}{(0.4)\,1} + \underset{c}{(0.2)\,3} + \underset{d}{(0.3)\,2}$

$= 0.3 + 0.4 + 0.6 + 0.6 = 1.9$ bits

We are sending more bits than information content, but we are very close.

MIN-Length code ==> MAX compression ==> most info bits in least number of communicated bits.

Suppose n different "messages" to send, n = 2^k.
Maximum entropy => equally likely: Prob(message-i) = (1/n) for any message-i.

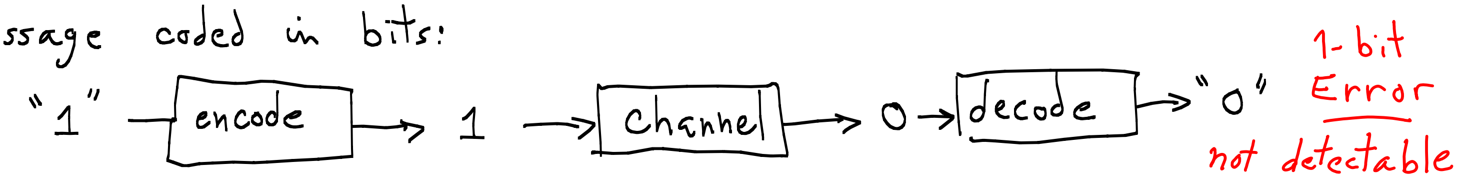
Expected information per message is,

Sum[- (1/n) log[1/n]] = - n (1/n log[1/n]) = -1 log[2^-k] = -1 (-k) = k bits per message. If we use a k-bit code for our messages, we will be 100% compressed. (k-bit integers? Are they equally likely?)

Error Detection / Correction

"message" could be a bit, a string of bits, a character, a page of characters, ...

message coded in bits:

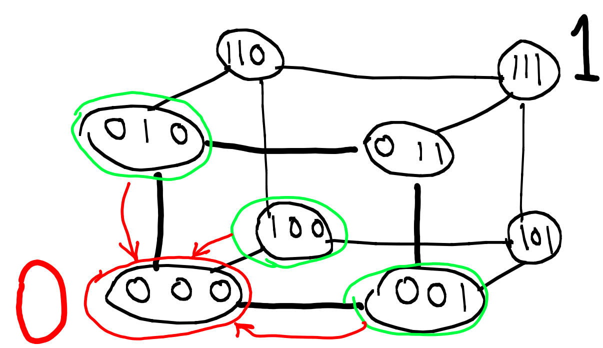


2-bit encoding

Code words: 00 and 11 --- "0" and "1"
Code words: 10 and 01 --- 1-bit errors: odd parity codeword indicates error.
Works for k-bit messages w/ 1 parity bit, but only if 2-bit errors very unlikely (never occur?).

1-bit Error Correction w/ 3-bit code words:
"0" ==> 000
"1" ==> 111

001 ==> "0"	011 ==> "1"
010 ==> "0"	101 ==> "1"
100 ==> "0"	110 ==> "1"



1-bit Correction, 2-bit Detection

-- odd parity: 1-bit error corrected

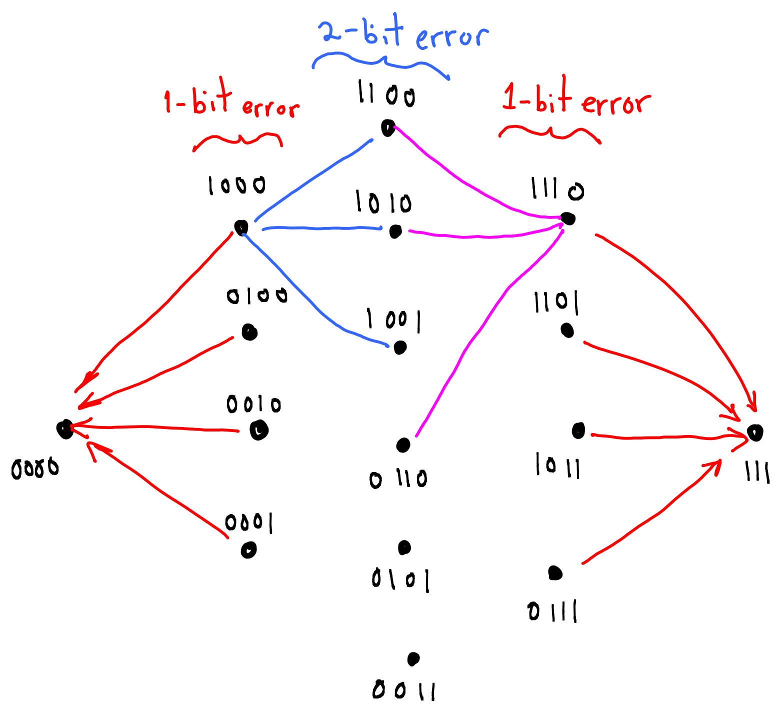
-- exactly two 1's: 2-bit error detected

-- otherwise: no error

How many extra bits are needed at minimum? Depends on noise in channel: Shannon Noisey Coding Theorem.

Can you think of a scheme like the parity-bit scheme that uses as few bits as possible? (See Reed-Solomon codes, for instance.)

More bits, higher error probability.



ALU, numbers

Positional notation for numbers

d_i is a "digit", a symbol for a value: value("d_i")

b is a value, the "base" of the number notation.

There is a rule to find the value, given the symbols.

$$\text{value}(\text{"}d_n d_{n-1} \cdots d_1 d_0\text{"}) = d_n \cdot b^n + d_{n-1} \cdot b^{n-1} + \cdots d_1 \cdot b^1 + d_0 \cdot b^0$$

unsigned 3-bit binary
binary:
--- digits = { "0", "1" }
--- base = 2

$$000 \rightarrow \text{value} \triangleq 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 0$$

$$001 \rightarrow \text{value} \triangleq 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1$$

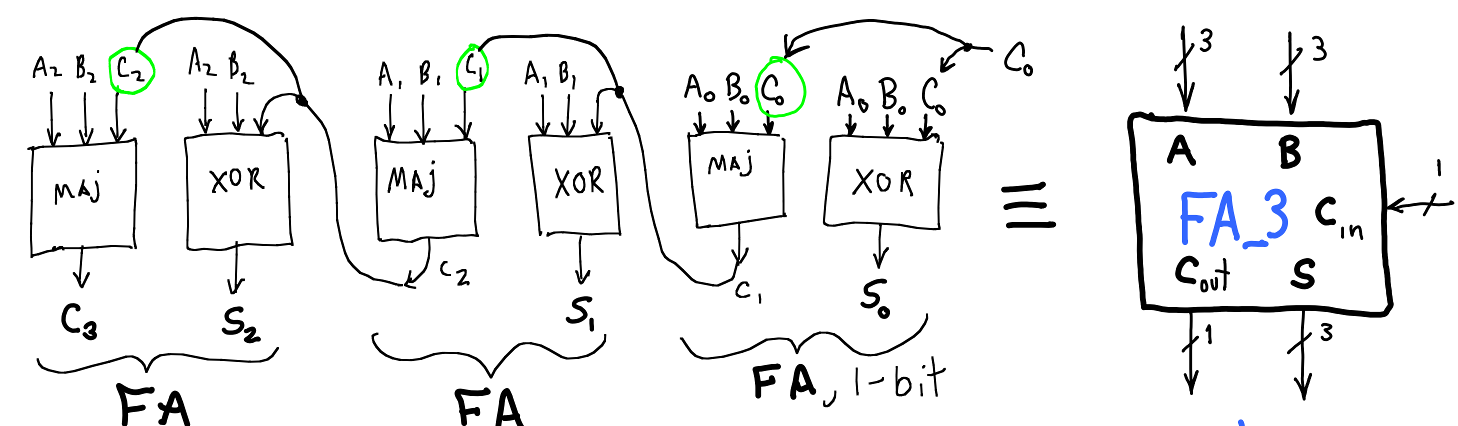
$$\cdots \qquad \cdots \qquad \cdots$$

$$111 \rightarrow \text{value} \triangleq 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 7$$

oops, more symbols?

Let's do some 3-bit arithmetic.
ADD:

$$A_2 A_1 A_0 + B_2 B_1 B_0 = C_3 S_2 S_1 S_0$$



3-bit Full Adder

Let's try SUBTRACTION

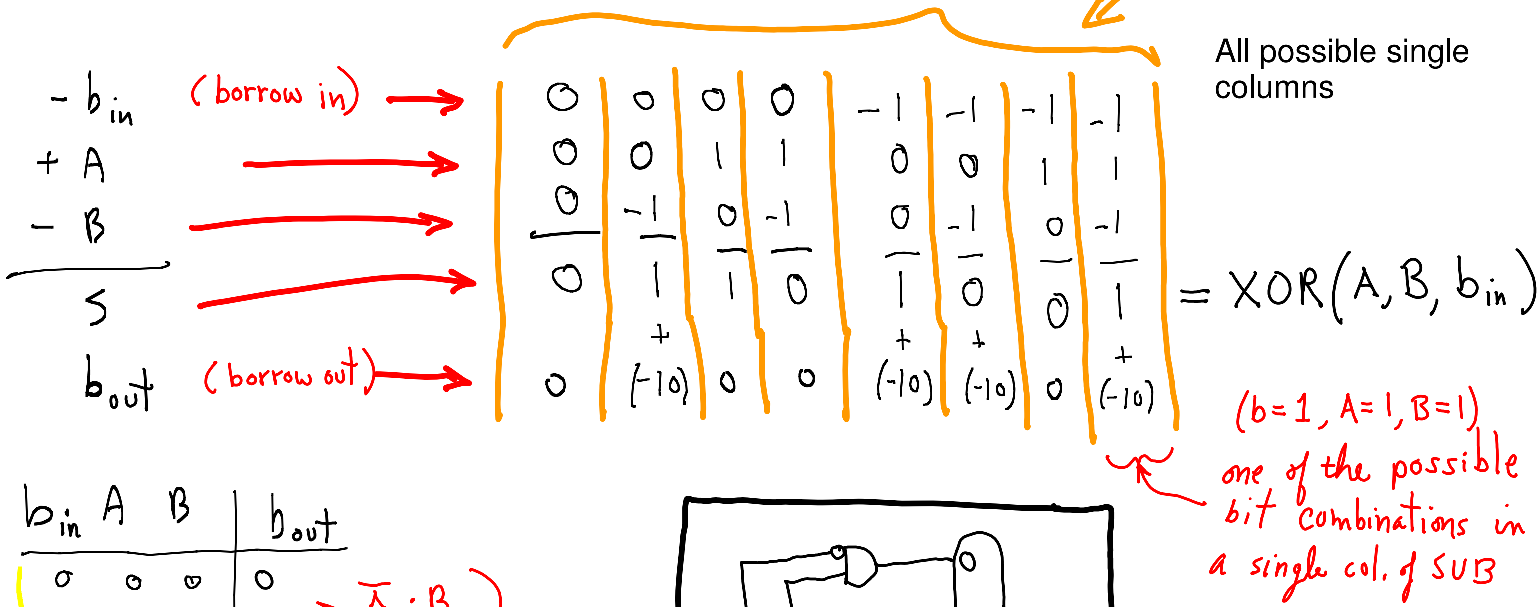
$$A_2 A_1 A_0 - B_2 B_1 B_0 = b_3 \; S_2 S_1 S_0$$

possible borrow

But first, let's look at a single column of subtraction

All possible 1-bit subtractions

All possible single columns

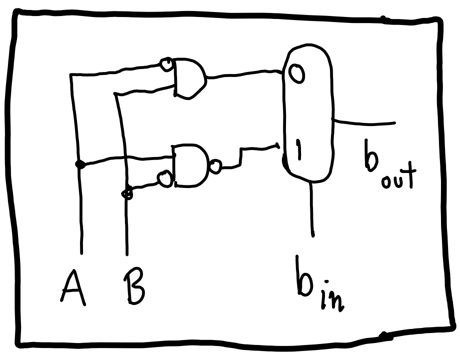


b_{in}	A	B	b_{out}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

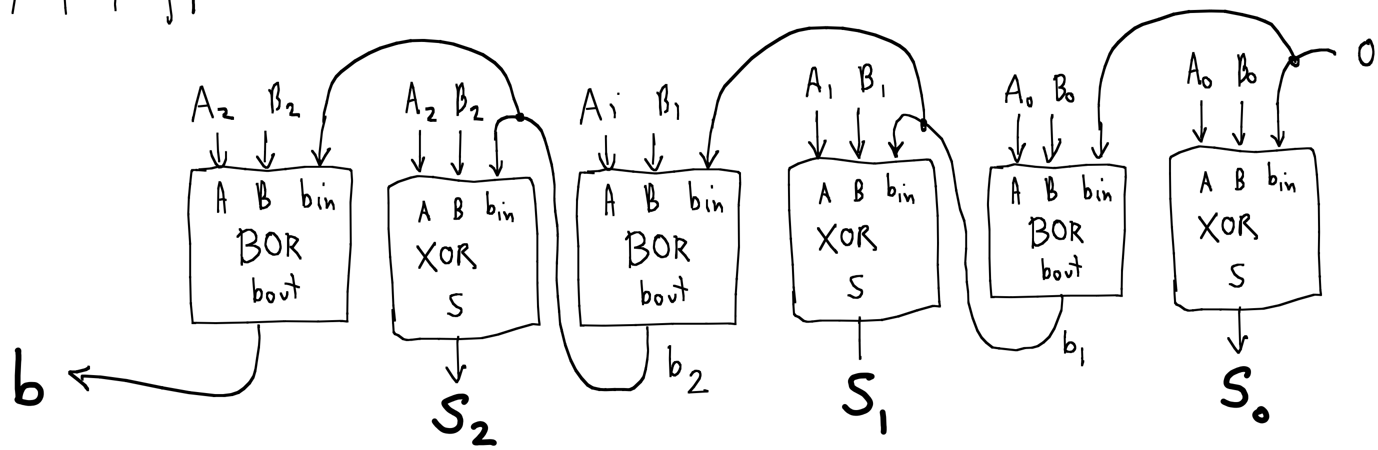
$\rightarrow \overline{A_i}\cdot B_i$

$\rightarrow \overline{A_i\cdot\overline{B_i}}$

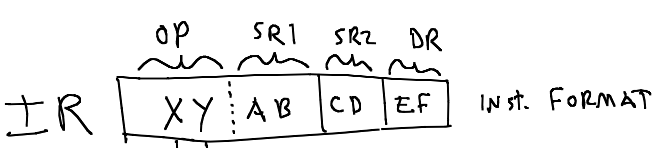
$\Rightarrow$



BOR



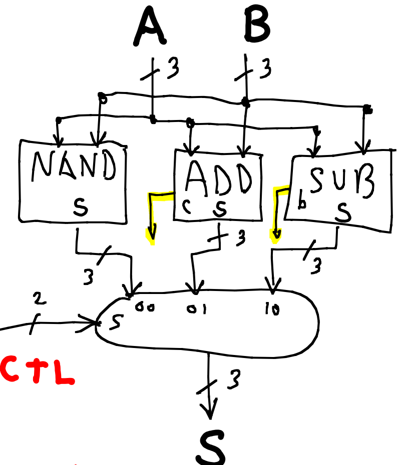
ALU



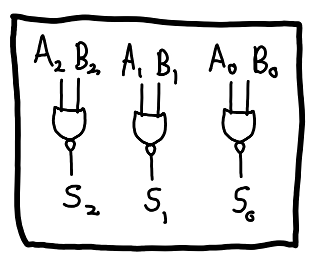
OP CODES

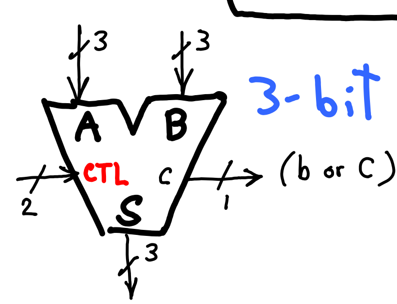
00 NAND
01 ADD
10 SUB

(simplified instruction, only shows ALU ops)



3-bit, bit-wise NAND



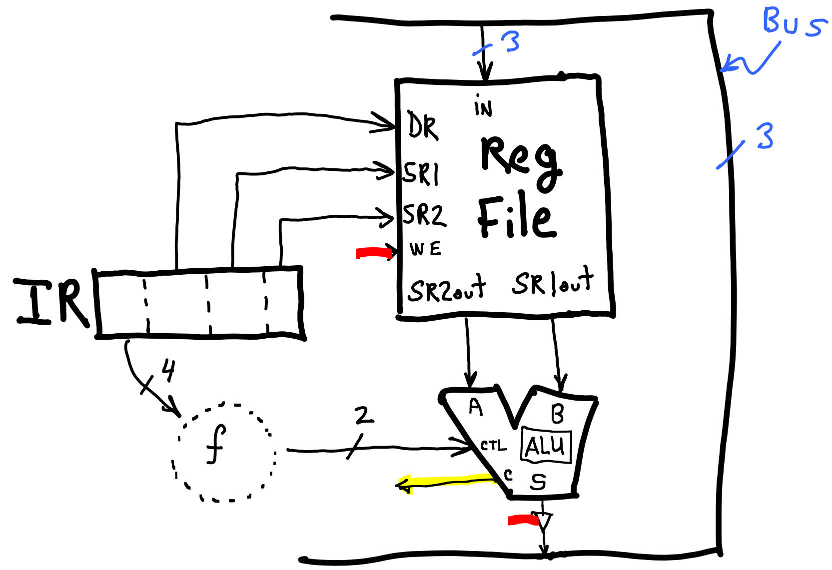


3-bit ALU

It's almost this simple in the LC3.

This is a 3-bit version of LC3 (sort of).

Some sort of function *f* converts 4-bit opcode to 2-bit ALU.ctl. In uCoded control, this function is implemented as 0/1 control bits in ROM.



Unsigned errors

c = carry/borrow output

$$A + B > 7 \Rightarrow c = 1 , \quad S = (A + B) \bmod 2^3$$
$$A - B < 0 \Rightarrow c = 1 , \quad S = (A - B) \bmod 2^3$$

$$C_3 S_2 S_1 S_0 \Rightarrow \left(\underline{C_3 \cdot 2^3} + S_2 \cdot 2^2 + S_1 \cdot 2^1 + S_0 \cdot 2^0 \right) \bmod 2^3 = S_2 \cdot 2^2 + S_1 \cdot 2^1 + S_0 \cdot 2^0$$

$$b_3 S_2 S_1 S_0 \Rightarrow \left(\underline{b_3 (-1) \cdot 2^3} + S_2 \cdot 2^2 + S_1 \cdot 2^1 + S_0 \cdot 2^0 \right) \bmod 2^3 = S_2 \cdot 2^2 + S_1 \cdot 2^1 + S_0 \cdot 2^0$$

$c = 1 \Rightarrow$ Overflow

We have 8 possible 3-bit patterns (or symbols).

How else might we assign an interpretation to them?

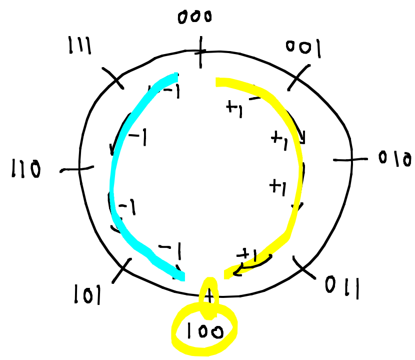
What else might we want as number values?

3-bit Code	interpretation as value
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

(NB – We are representing the values with an alternate representation: base 10)
Is there no end to this madness!?!

Two's-Complement Encoding:
We can represent POSTIVE and NEGATIVE

CODE	Value
0 1 1	+3
0 1 0	+2
0 0 1	+1
0 0 0	0
1 1 1	−1
1 1 0	−2
1 0 1	−3
1 0 0	−4

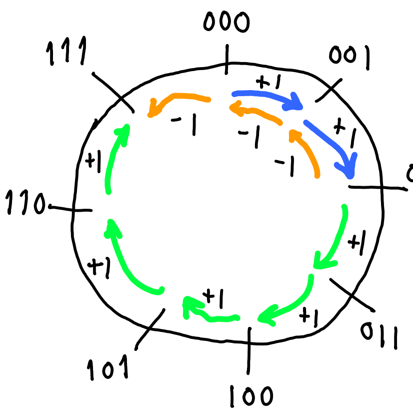


moving +4 ≡ moving −4
Which value makes sense?

sanity check: 0 - 1?

$$\begin{array}{r} 0\ 0\ 0 \\ -\ 0\ 0\ 1 \\ \hline (-1)\ 1\ 1\ 1 \end{array}$$

hmm, kind of makes sense.



$2 - (+3)$

$2 + (-3)$

move +2
move +5
(−1)

$-k$ can be moving:

k steps

OR

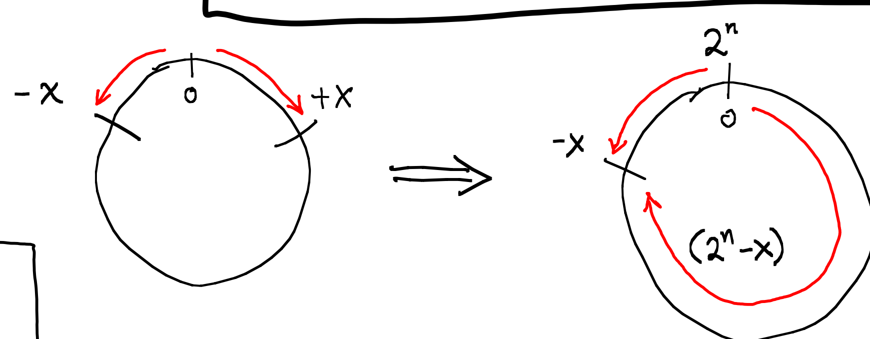
$2^n - k$ steps:

$-3 \Rightarrow$ 3 steps

$\Rightarrow 2^3 - 3 = 5$ steps

n-bit Two's Comp (X)

$-X \Rightarrow 2^n - X$



Sanity check $-(-x)$?

$-x \Rightarrow (2^n - x)$

$-(-x) \Rightarrow 2^n - (2^n - x)$

$= x$

$-(-X) = X$ in 2's comp.

Try $n=3$ 2's comp : $2^n = 2^3 = 8$ $(-(-3))$

$2^3 - (2^3 - 3)$

$2^3 - (8 - 3)$

$2^3 - (5)$

$8 - 5$

3

$\Rightarrow 011$ +3 (in 3-bit 2's comp.)

Converting To 2's Comp?

n-bit

How do we do this simply, in general?

$$\begin{array}{rcl}
 & & \overset{\text{borrow}}{} \quad 10 \quad 10 \quad \cdots \quad 10 \quad 10 \\
 & & -1 \quad -1 \quad -1 \quad \cdots \quad -1 \quad -1 \\
2^n &=& 1\ 0\ 0\ \cdots\ 0\ 0\ 0\ 0\ \cdots\ 0 \\
-x &=& -\ x_{n-1}\ x_{n-2}\ \cdots\ x_{n-j}\ 1\ 0\ 0\ \cdots\ 0 \\
\hline
S &=& S_{n-1}\ S_{n-2}\ \cdots\ S_{n-j}\ 1\ 0\ 0\ \cdots\ 0
\end{array}$$

Notice: The 1st non-zero bit of x gets copied to sum S:

borrow = 10
subtract bit = - 1
----
sum bit = 1

Note: columns with borrows give a bit flip.

$$\begin{array}{r} 1 \\ -\,x_k \\ \hline \bar{x}_k \end{array}$$

$$\begin{array}{rcl}
 & & \qquad\qquad\qquad\qquad 10 \\
2^n &=& 0\ 1\ 1\ \cdots\ 1\ 0\ 0\ 0\ \cdots\ 0 \\
-x &=& -\ x_{n-1}\ x_{n-2}\ \cdots\ x_{n-j}\ 1\ 0\ 0\ \cdots\ 0 \\
\hline
(2^n - x) &=& \bar{x}_{n-1}\ \bar{x}_{n-2}\ \cdots\ \bar{x}_{n-j}\ 1\ 0\ 0\ \cdots\ 0 \\
(2^n - x) - 1 &=& \underbrace{\bar{x}_{n-1}\ \bar{x}_{n-2}\ \cdots\ \bar{x}_{n-j}}\ 0\ 1\ 1\ \cdots\ 1
\end{array}$$

-1 $\quad$ $+1$ $\quad$ negate

Notice: These are the negated bits of x.

$$x = x_{n-1}\ x_{n-2}\ \cdots\ x_{n-j}\ \cdots\ x_2\ x_1\ x_0$$

To Get 2sComp(x):

Negate bits,
add 1.

Produce - x in 2's Complement (regardless of whether x is + or -):

Negate bits (aka 1's Complement), then add 1.

Simple logic: inverter on each bit, carry in to lowest FA set to 1.

==> We can use adder for signed subtraction

Let's try

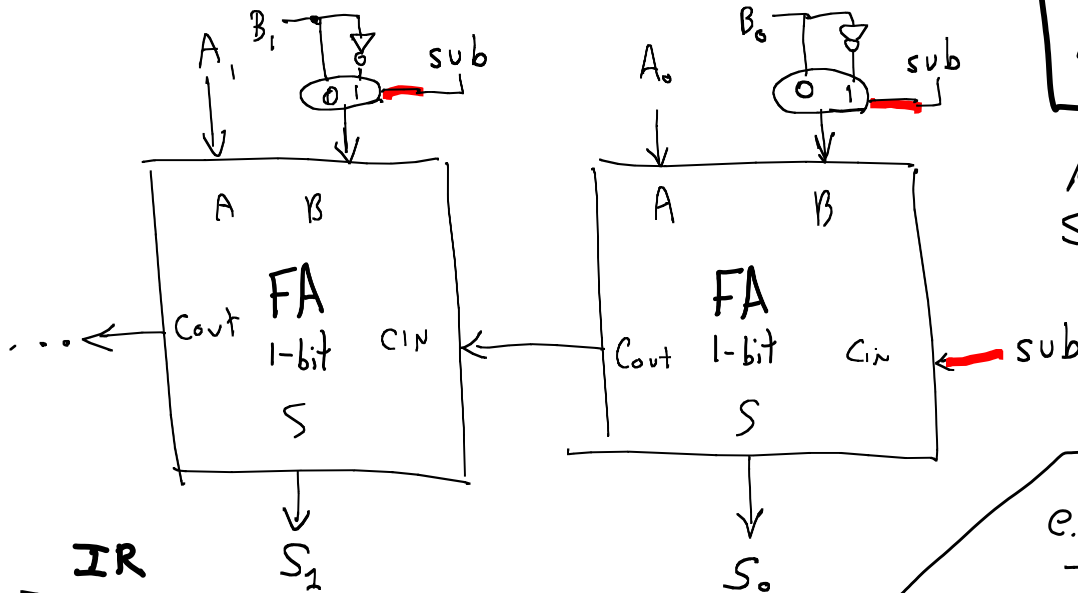
2's Comp of neg. number (expressed in 2's comp).

$$2\text{'s Comp}(\underbrace{1\,x_3\,x_2\,x_1\,x_0})$$ — a neg. number in 2's comp.

$$0\ \bar{x}_3\ \bar{x}_2\ \bar{x}_1\ \bar{x}_0 + 1$$

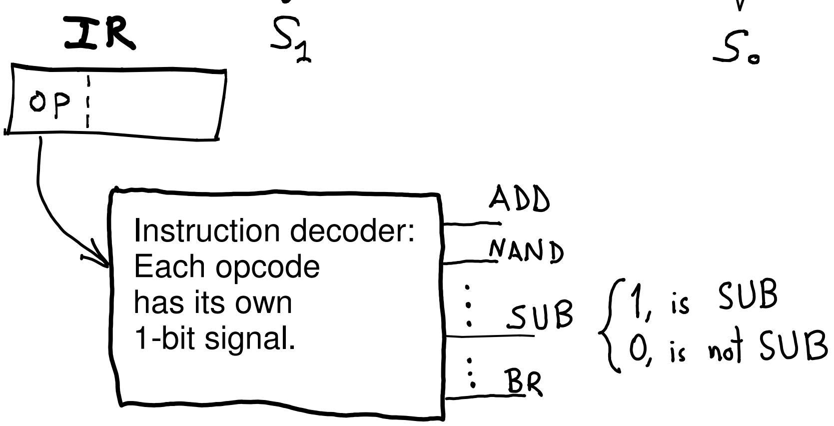
flip bits, add 1

	Least neg.	between	most neg.	
	1111	1abc	1000	
flip ⇒	0000	$0\bar{a}\bar{b}\bar{c}$	0111	⇐ flip
+1 ⇒	0001	0xyz	1000	⇐ +1
			?	
	Extreme case ✓ok	in-between case ✓ok	Extreme case ※ oops! what's wrong?	



A + 2sComp(B)

ADD : sub = 0
SUB : sub = 1



e.g., 3-bit (3 − 1)

A = 3 ⇒ 011
B = 1 ⇒ 001
2sComp(B) ⇒ 110 + 1 = 111

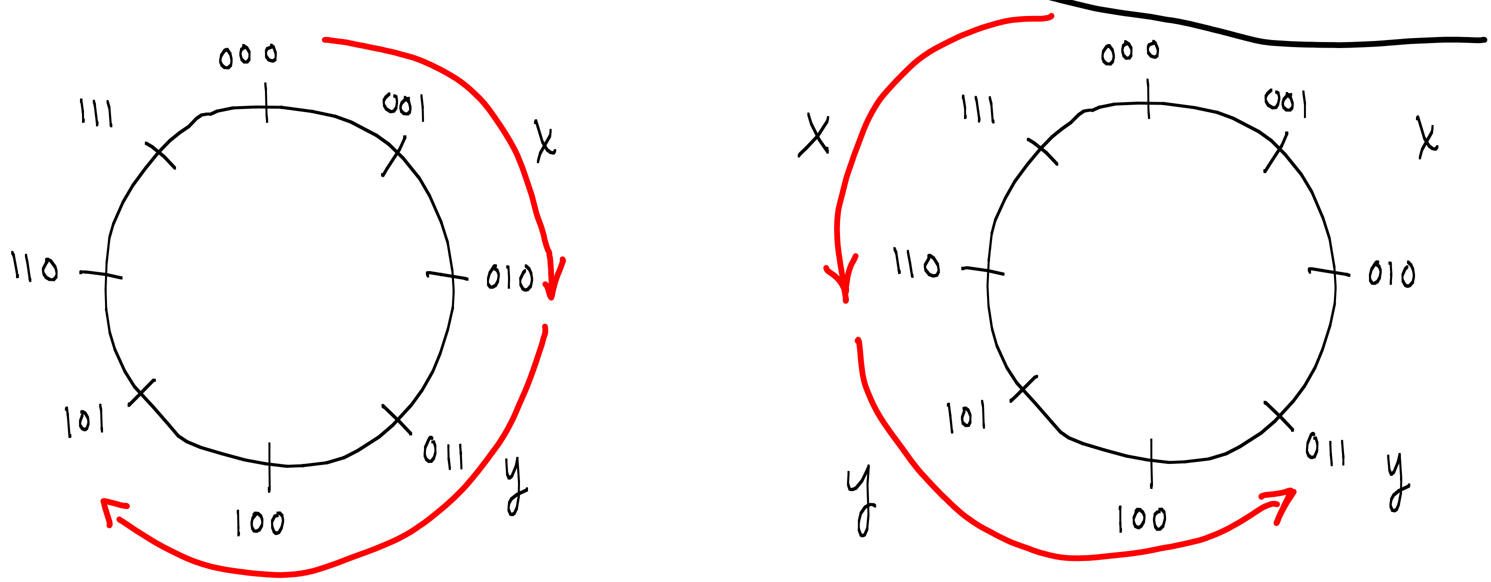
carries: 1 1 1

A	⇒	1 0 1 1
+ (−B)		+ 1 1 1
		(1) 0 1 0

carry ignored

Pos 2

2's complement Arith., Overflow



x > 0, y > 0
and x+y < 0

0xxx
0yyy

1sss

x > 0, y < 0
and x-y < 0

2's comp

0xxx
- 1yyy => 0zzz*

1sss

x < 0, y <0
and x+y > 0

1xxx
1yyy

0sss

x < 0, y > 0
and x-y > 0

2's comp

1xxx
- 0yyy => 1zzz

0sss

* could be 1000

ERROR = Same signs in, diff. out

Hex, oct

binary : base=2

digits = {0,1}

"$b_i b_{i-1} \cdots b_0$" $\xrightarrow{\text{means}}$ $b_i 2^i + b_{i-1} 2^{i-1} + \cdots + b_0 2^0 = n$

$100_2 \xrightarrow{\text{means}} (1)\cdot 2^2 + (0)\cdot 2^1 + (0)\cdot 2^2 \Rightarrow 4_{10}$

octal : base = 8 $(=2^3)$

"$d_i d_{i-1} \cdots d_0$" $\xrightarrow{\text{means}}$ $d_i (2^3)^i + d_{i-1} (2^3)^{i-1} + \cdots d_0 \cdot 1$

digits = {0,1,2,3,4,5,6,7}

$301_8 \xrightarrow{\text{means}} (3)\cdot(2^3)^2 + (0)\cdot(2^3)^1 + (1)\cdot 1$

$3 \cdot 64 + 0 + 1 \Rightarrow 193_{10}$

octal ⟷ binary

$501_8 \longrightarrow (101_2) \cdot (2^3)^2 + 0 \cdot (2^3)^1 + (001_2)(2^3)^0$

$\rightarrow 1\cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \; (2^3)^2 + 0 \cdot (2^3)^1 + 1 \cdot 2^0 \; (2^3)^0$

$\rightarrow 1\cdot 2^{2+6} + 0 \cdot 2^{1+6} + 1 \cdot 2^6 + 0 \cdot 2^3 + 1 \cdot 2^0$

$\rightarrow 1 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

$\rightarrow (\underline{1\;0\;1}\;\underline{0\;0\;0}\;\underline{0\;0\;1})_2$

$\rightarrow (5 \quad 0 \quad 1)_8$

$\Rightarrow$ Octal digit ⟷ 3-bit binary representation of digit

Hex

hexadecimal : base = 16 = (2^4)

hex digits	bin rep.	hex digits	bin rep.
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

$\Rightarrow$ Hex digit ⟷ 4-bit binary representation of digit

Multiply

$$\begin{array}{r} 1 \\ \times 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ \times 10 \\ \hline 10 \end{array} \qquad \begin{array}{r} 10 \\ \times 10 \\ \hline 100 \end{array} \qquad \begin{array}{r} 100 \\ \times 10 \\ \hline 1000 \end{array}$$

(does $2x$ = Left shift?)

1st 1

$$2x = x + x \Rightarrow \begin{array}{rccccccccc} & x_n & x_{n-1} & \ldots & x_k & 1 & 0 & 0 & \ldots & 0 \\ + & x_n & x_{n-1} & \ldots & x_k & 1 & 0 & 0 & \ldots & 0 \\ \hline c_{n+1} & S_n & S_{n-1} & \ldots & S_k & 0 & 0 & 0 & \ldots & 0 \end{array}$$

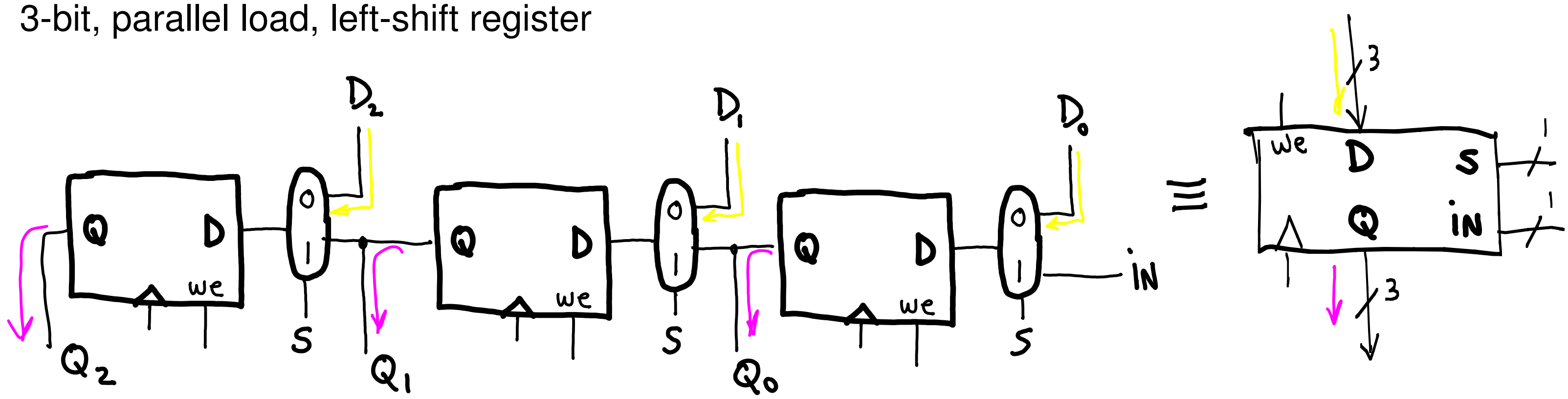
Per Col.

$$\begin{bmatrix} c_j \\ x_j \\ + x_j \\ \hline S_j \\ c_{j+1} \end{bmatrix} \Rightarrow \begin{bmatrix} c_j \\ 0 \\ + 0 \\ \hline S_j \\ c_{j+1} \end{bmatrix} \text{ OR } \begin{bmatrix} c_j \\ 1 \\ + 1 \\ \hline S_j \\ c_{j+1} \end{bmatrix} \begin{array}{l} \to S_j = c_j \\ \to c_{j+1} = x_j \end{array} \Rightarrow \begin{bmatrix} S_{j+1} = c_{j+1} & \\ & c_{j+1} = x_j \\ S_{j+1} & = x_j \end{bmatrix} \checkmark$$

Left shift

In each column, if x = 0 then S is equal to the carry in.
If x = 1, there is a carry out and S is equal to the carry in.

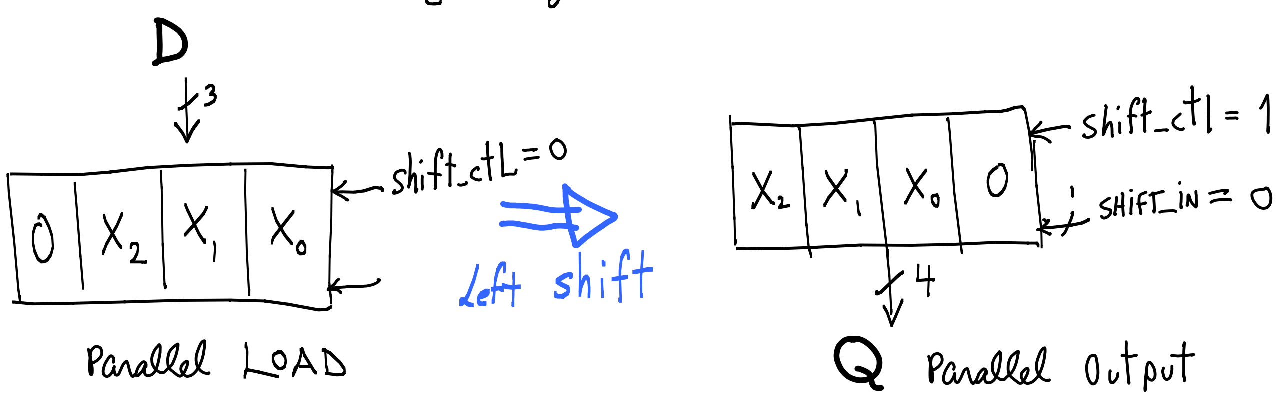
3-bit, parallel load, left-shift register



Parallel write/load: we=1 and S=0: Q[2:0] <== D[2:0] after next clock tick.

Shift Left: we=1 and S=1: Q[2:0] <== { Q[1:0], IN } after next tick.

3-bit Doubler [using 4-bit LSR] (unsigned)



What about signed numbers?

Convert to unsigned.

Multiply.

Convert back.

$$x \cdot 7 = x(4+2+1) = \underbrace{4x}_{\substack{\text{2 shifts} \\ C}} + \underbrace{2x}_{\substack{\text{1 shift} \\ B}} + \underbrace{x}_{\substack{\text{0 shifts} \\ A}}$$

$y = 00\ldots0111$

MULTIPLY:
LSR: partial products, initially x.
S: partial sum, initially 0.
RSR: initially y.
Z: all 0s

$$\begin{array}{rcccccccl}
 & & 0 & 0 & \cdots & 0 & 0 & & = S^0 \\
+ & & x_n & x_{n-1} & \cdots & x_1 & x_0 & & + (A) \\
\hline
 & & S'_n & S'_{n-1} & \cdots & S'_1 & S'_0 & & S'
\end{array}$$

shift ←

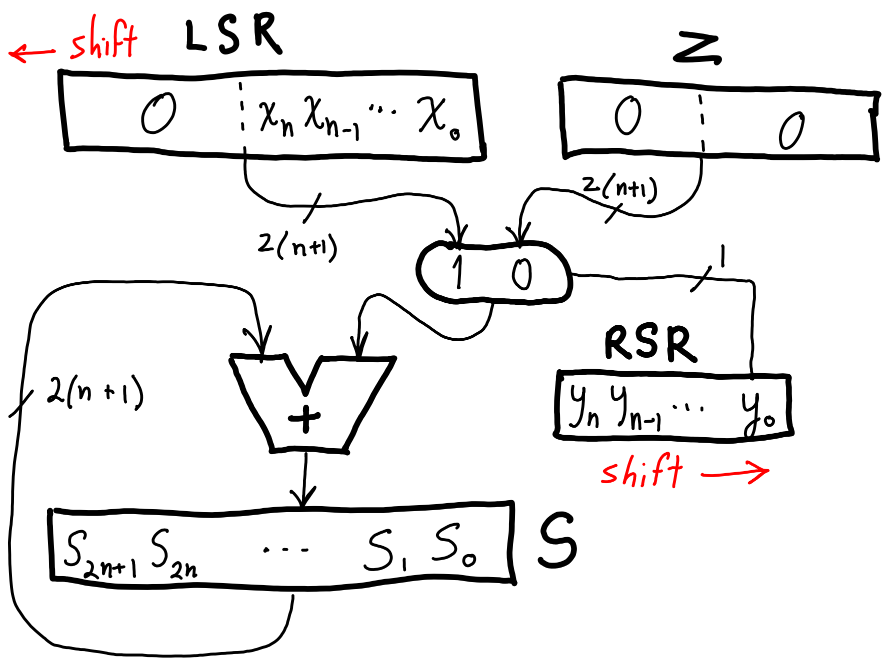
$$\begin{array}{rcccccccl}
+ & x_n & x_{n-1} & \cdots & x_1 & x_0 & 0 & & + (B) \\
\hline
 & S''_{n+1} & S''_n & \cdots & S''_2 & S''_1 & S''_0 & & S''
\end{array}$$

shift ←

$$\begin{array}{rccccccccl}
+ & x_n & x_{n-1} & \cdots & x_1 & x_0 & 0 & 0 & & + (C) \\
\hline
 S'''_{n+2} & S'''_{n+1} & \cdots & S'''_3 & S'''_2 & S'''_1 & S'''_0 & & & S'''
\end{array}$$

What if y has a 0 bit? Then add 0 instead of shifted x: e.g., y = 0...101 add 0, not **B**.

RSR's low-bit MUXes Z or LSR to adder.



e.g.

$$\begin{array}{r}
1011 \\
\times\ 0101 \\
\hline
\ldots 1011 \\
+\ .1011.. \\
\hline
0\ 0110111
\end{array}$$

(possible carry)

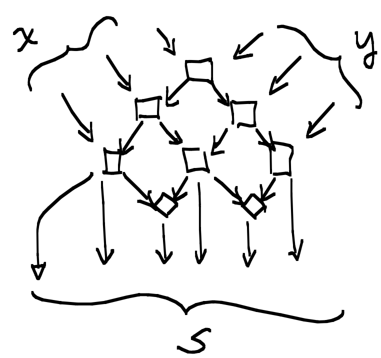
rewrite ⟹

											multiplier bit
					1	0	1	1			
				×	0	1	0	1			
	0	0	0	0	0	0	0	0	←	start	
+	0	0	0	0	1	0	1	1	←	0-shift (1011)	← $y_0 = 1$
+	0	0	0	0	0	0	0	0	←	1-shift ( )	← $y_1 = 0$
+	0	0	1	0	1	1	0	0	←	2-shift ( )	← $y_2 = 1$
+	0	0	0	0	0	0	0	0	←	3-shift ( )	← $y_3 = 0$
=	0	0	1	1	0	1	1	1			

⟹ we shift left (1011) every time, but add either the shifted (1011) or all zeroes, depending on whether y_i is 1 or 0.

Can we simply multiplier?
-- Get rid of zero register and mux.
-- Use y_i to write enable write-enable S register.

Can we speed up multiply? We currently iterate n times to multiply n-bit numbers. Add more hardware? How?



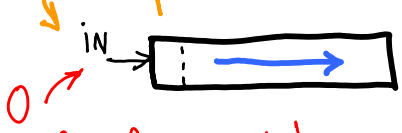
DIVIDE: left-shift(y) == y X 2 ===> right-shift(y) == y / 2

Ok, for division by a power of 2.

Div by 2

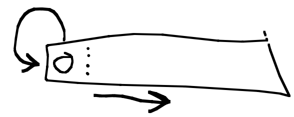
arithmetic, sign extended

R-Shift:



logical, $\emptyset$ padded

(let's assume pos.



)

Integer Division = drop remainder

011 R-shift → 001

3 ÷ 2 = 1

0111 $(\text{R-shift})^2$ → 0001

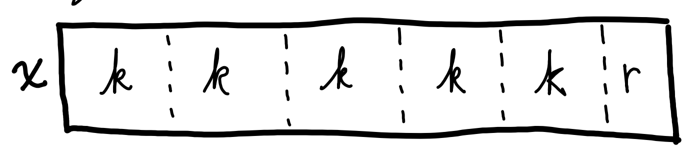
7 ÷ 4 = 1

R-shift(n) = divide-by-2^n

But, if divisor is not power of 2?

$x = k \cdot q + r$ { k is divisor, q is quotient }

$q = \#k\text{s in } x$




```
divBySubtraction( x, k )
   q = 0;
   while( x >= k )
      q++
      x = x - k
   endWhile
```

```
divByAddition( x, k )
   q = 0;    y = 0
   while ( x - y >= k )
      q++
      y = y + k
   endWhile
```

time = $O(q)$

We'd like $O(\log(q))$ = # bits of q

⟹ long division

$$k \overline{\big)\, x}^{\,d_n 00\cdots 0}$$
$$-k \cdot d_n 00\ldots 0$$
$$x'$$

$$3\overline{)176}^{\,50}$$
$$-150$$
$$26$$

$$3\overline{)26}^{\,8}$$
$$-24$$
$$2$$

50 + 8 → 58

1. Try largest power of 10, subtract from x.
2. If partial sum is non-negative, save digit.
3. Try next smaller power of 10, subtract from x or from remainder depending on prior result.
etc.

INTEGER (unsigned) DIVISON

x = kq +r k = divisor, q = quotient, r = remainder (let's ignore r for now). FIND q.

$$x = kq = kq_n2^n + kq_{n-1}2^{n-1} \cdots + kq_02^0$$

try $q_n = 1$

$$(x - \underbrace{k\cdot 1\cdot 2^n}_{k \text{ L-shifted } n}) \geq 0 \quad \text{then } q_n = 1$$

$$\boxed{0\,|\,0\,|\,0\,|\,0\,|\,0\,|\cdots|\,0\,|\,0}\ q$$

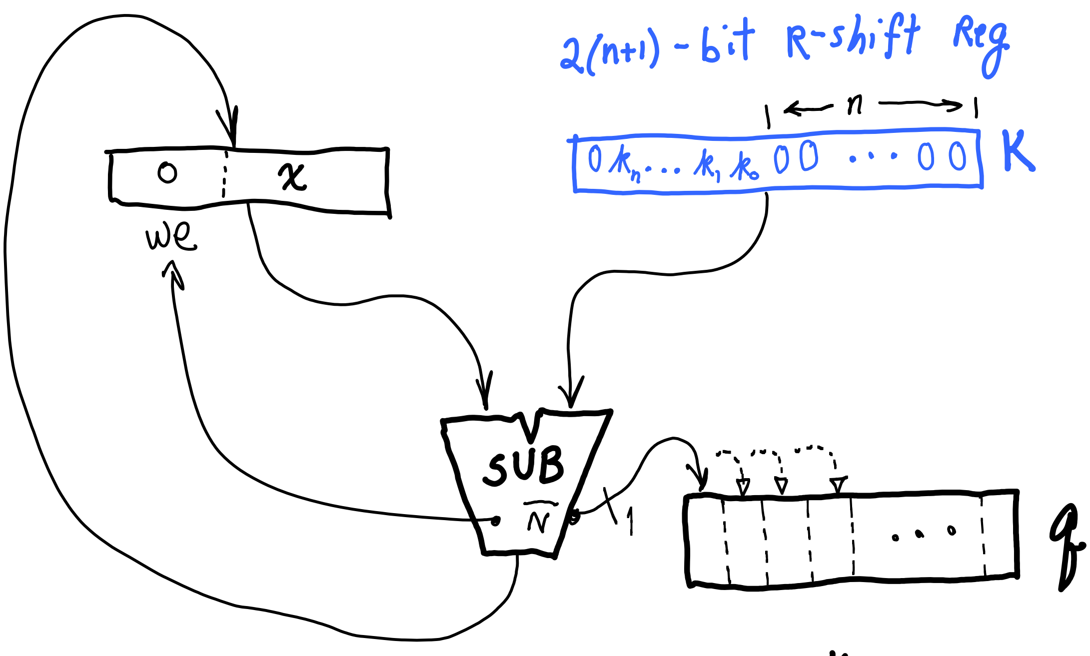
0 or 1

$$x \leftarrow kq_{n-1}2^{n-1} + \cdots + kq_02^0 \qquad x \leftarrow (x - kq_n2^n)$$

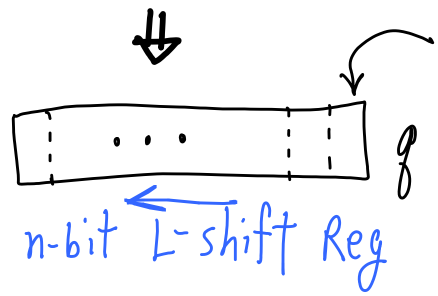
Try $q_{n-1} = 1$

$$(x - \underbrace{k\cdot 1\cdot 2^{n-1}}_{k \text{ L-shifted } (n-1)}) \geq 0 \quad \text{then } q_{n-1} = 1$$

$$\boxed{q_n\,|\,0\,|\,0\,|\,0\,|\cdots|\,0\,|\,0\,|\,0}\ q$$



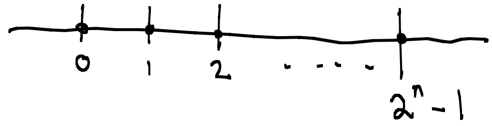
Move *notNegative* RIGHT one bit position after each SUB.



Move ***q*** LEFT one bit position after each SUB.

(write is to lowest bit position)

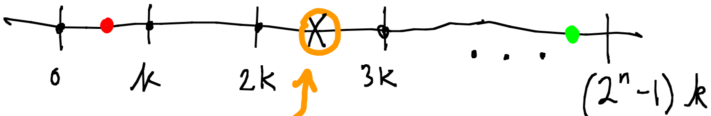
Floating Point



n-bit integers, range $= 2^n$

no discretization error

K-scaled integers: n-bit integer **x** represents k***x**, range = k*2^n.



this can't be represented, error $\approx k/2$

discretization error $\approx k/2$

Near • k, % error $\Rightarrow \frac{k/2}{k} = 50\%$

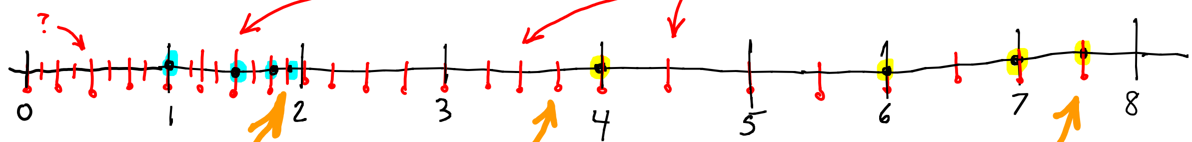
Near • $k(2^n-1)$ $\Rightarrow \frac{k/2}{k(2^n-1)} \cong 1/2^{n+1}$

FP, exponential scaling

$2^m(1.XYZ)$

Can we get more consistent errors?

values we can represent



$2^0(1.111)$

$2^0(1 + 1/2 + 1/4 + 1/8)$

$\Rightarrow \underbrace{1 + 1/2 + 1/4 + 1/8}_{\approx 2} \pm (\frac{1}{16})$

$\Rightarrow$ error $\approx \frac{(1/16)}{2} = 1/32$

$2^1(1.111) = (2+1+1/2+1/4)$

$2^2(1.111)$

$\underbrace{4+2+1+1/2}_{\approx 8} \pm (1/4)$

error $\approx \frac{(1/4)}{8} = \frac{1}{32}$

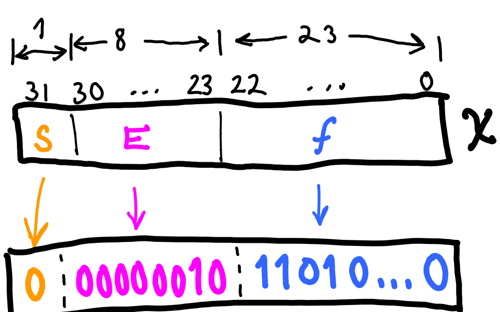
We can't represent every number.
We choose what type of errors to live with.

The part inside "(...)" is essentially integer.
The exponent determines the scaling.

===> geometrical-progression scaled integers

FP Format, single Float

32-bit:



$+ \; 2^2 \times 1.11010\ldots0$

not STORED

$\Rightarrow \; +2^2 \times (1 + 1/2 + 1/4 + 1/16)$

value$(x) = S \; 2^E \times 1.f$

$S=0$: +

$S=1$: −

E pos.?
E neg.?

Use 2's comp for E?

Represent 0?

0	0 … 0	0 … 0

$$+2^{0} \times 1.0 \cdots 0 \Rightarrow 1\ ?$$

If we used 2s-Comp for E, what's the smallest: 8-bit, 1000 0000 = −128

$$2^{-128} \times 1.00\ldots0$$ that's small. Do we need it?

S	10000000	00 … 00

means 0

Well, maybe we can live with that?
We have to stop somewhere.

How many bits do we need for 4 decimal digits of precision?

2 digits

6.023×10^{23} ← a nice number

4 digits

3 bits per digit $(2^3 \to 0..7)$
4 bits per digit $(2^4 \to 0..15)$

≈ 3.5 bits per digit

$$4\text{ dig.} \times \frac{3\text{ bit}}{\text{digit}} = 12\text{ bits}$$

$$4\text{ dig.} \times \frac{4\text{ bit}}{\text{digit}} = 16\text{ bits}$$

$$\Rightarrow 12 \le \left(\begin{matrix}\text{Precision}\\ \text{of}\\ f\end{matrix}\right) \le 16$$

(23 bits are enough)

---

range of E?
(2 dec. digits)

$$2\text{ dig.} \times \frac{3\text{ bit}}{\text{dig}} = 6\text{ bits}$$

$$2\text{ dig.} \times \frac{4\text{ bit}}{\text{dig.}} = 8\text{ bits}$$

$$6 \le \left(\begin{matrix}\text{bits}\\ \text{of}\\ E\end{matrix}\right) \le 8$$

Let's check

$$10^{23} \approx 8^{23} = (2^3)^{23} = 2^{69}$$

E
how many bits do we need for E?

E = 69, how many bits needed?

$$\log(69) \approx 1 + \log(64) = 7$$

Sorting is most common operation for numerical data

Checking x > y seems hard for floats.

Checking n > m for ints: do (n - m) and check sign bit, if 0 then True.

Can we check x > y using integer hardware?

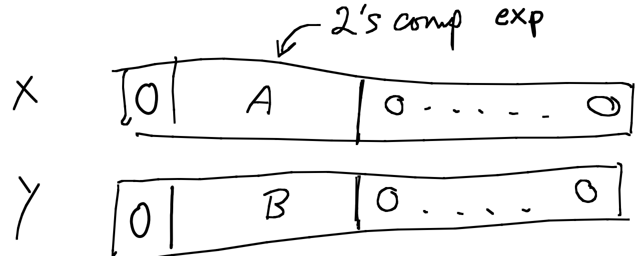
That is, can we treat x and y as if they were integers, and do integer subtraction?

$\Rightarrow$ Looks like + 2's-Comp

$\Rightarrow$ Looks like − 2's comp

$\Rightarrow$ ALL (+) floats are greater than all (−) floats, as 2's-comp.

How about the exponent part? x − y as 2's-comp



A = 011
B = 111

3-bit, 2s-comp exp

+ exp	011
	010
	001
	000
− exp	111
	110
	101
	100

these are larger as unsigned, darn.

$$x = +2^{3} \cdot (1.00 \cdots 0)$$
$$y = +2^{-1} \cdot (1.0 \ \cdots 0)$$

$\Rightarrow$

$(0 \ 011 \ 00 \cdots 0) \ x$

$(0 \ 111 \ 00 \cdots 0) \ y$

y looks like a bigger 2's comp. number than x.

OOPS!

$\to$ Let's see if we can patch this up.

Recall, our only problem is if both x and y have the same sign.

suppose $\text{sign}(x) = \text{sign}(y)$

(mag. comarison)

X: | 0 | E_2 | f_2 |

Y: | 0 | E_1 | f_1 |

or

X: | 1 | E_2 | f_2 |

Y: | 1 | E_1 | f_1 |

(Reverse result for neg.)

(for signs $\neq$ result is obvious)

let e_i = value(E_i)

Note: $e_1 > e_2 \Rightarrow 2^{e_1}(1.f_1) > 2^{e_2}(1.f_2)$

regardless of the fractional parts.

Let's check

Suppose $e_2 = e_1 + 1$

1.111....1

$\underbrace{2^{e_2}\cdot(1.0)}_{\text{Smallest possible } x}$ $\qquad$ $\underbrace{2^{e_1}(2-\varepsilon)}_{\text{Largest possible } y} = 2^{e_1+1}(1-\varepsilon/2)$

$= 2^{e_2}(1-\varepsilon/2)$

Y is less than X

So, make E_1 look bigger than E_2

---

what we have so far

8-bit exponent (single float)

2's complement →

0111 1111 $\quad$ ($2^7-1 = 127$)

⋮

0000 0000 $\quad$ (0)

1111 1111

1111 1110

⋮

1000 0001 $\quad$ (−127, Not used, reserved for signal)

1000 0000 $\quad$ (−128, Not used, signals 0?)

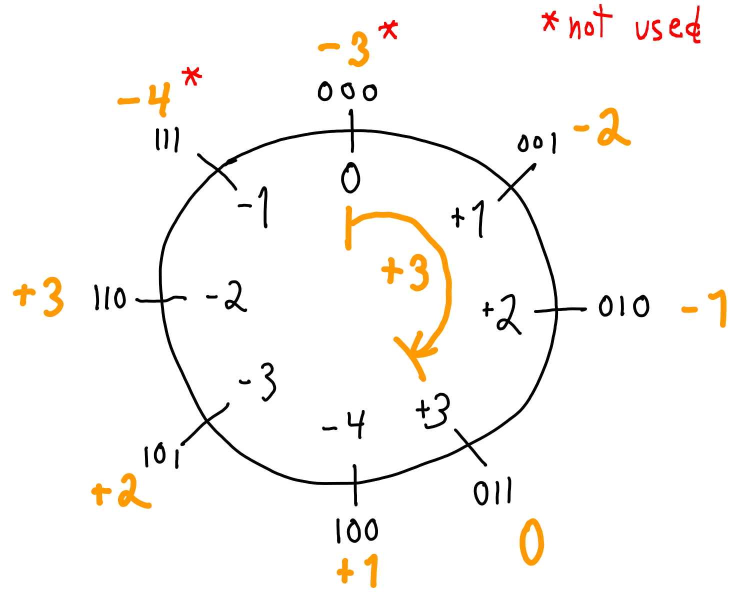
Let's fix Exp

E.G., 3-bit exponents in 2s-complement

goal: make all negative exponents look smaller than all positive exponents AS unsigned ints.

S	E	f

$E = e + 011$



value	e in 2s-comp		E in excess-3	
+3	011	+011 ==>	110	
+2	010	+011 ==>	101	
+1	001	+011 ==>	100	
0	000	+011 ==>	011	
-1	111	+011 ==>	010	
-2	110	+011 ==>	001	
-3	101	+011 ==>	000	*
-4	100	+011 ==>	111	*

* These codes are reserved for special uses.
The exponent values -3 and -4 are not allowed.

Rotate 2'comp so that +3 becomes largest number available.

How to represent 0?

	e	
	111	0

$\rightarrow \pm\infty$

	e	
	111	$\neq 0$

$\rightarrow$ NaN

	e	
	000	0

$\rightarrow 0 \quad (\pm 0)$

	e	
	000	$\neq 0$

$\rightarrow$ not normalized

$\rightarrow 2^{-3} \times 0.f$

8-bit FP, ADD

	S	E	f
X	0	001	0110
y	0	110	0010

$001 - 011 = 110 \rightarrow -2$

Convert from excess-3 to 2's comp.

$110 - 011 = 011 \rightarrow +3$

0	110	0110
0	011	0010

$+2^{-2} \times 1.0110$

$+2^{3} \times 1.0010$

shift/align exponents

$2^3 \times 0.00000\,10110$

$2^3 \times 1.0010$

TRUNCation?

add f part

$2^3 \times 1.001010$

round to nearest

rounding

$2^3 \times 1.0010$

normalize

no shift needed in this example

$2^3 \times 1.0010$

encode

0 | 011 | 0010

To excess-3

$\underbrace{\text{0 | 110 | 0010}}_{y}$

$x + y = y$!?! ⇒ discretization, truncation, rounding

→ ERRORS be careful!

8-bit FP, MULT

0 | 001 | 1010 → 0 | 110 | 1010 → $+2^{-2} \times 1.1010$

0 | 110 | 0000 → 0 | 011 | 0110 → $+2^{3} \times 1.0110$

Convert E_s from excess-3 To 2s Comp

add exponents

check for overflow

2^1

Shift and MULT as unsigned INTs. Keep exp. To normalize

```
    11010  (x 2^-4)
  x 10110  (x 2^-4)
-----------
 + 110100
 + 11010
+ 110100
-----------
1000111100  (x 2^-8)
```

normalize

$2^1 \times 1.000111100$

add exponents

$2^2 \times 1.00011110$

round To nearest

$2^2 \times 1.0010$

encode

0 | 010 | 0010

excess 3 code

0 | 101 | 0010

Convert to 32-bit FP | 28

1. Convert to binary

$$\begin{array}{rlcr}
28 & & & \\
-16 & \rightarrow 1\cdot 2^4 & = & 10000 \\
12 & \quad + & & \\
-8 & \rightarrow 1\cdot 2^3 & = & 1000 \\
4 & \quad + & & \\
 & \rightarrow 1\cdot 2^2 & = & 100 \\
\hline
 & & & 11100
\end{array}$$

2. Normalize

$11100. \Rightarrow 2^4 \times 1.1100$

$e = 4$

3. Encode

	8-bits	23-bits
0	0000 0100	1100...0
+	$e=4$	1.1100

4. Convert e to excess $(2^{n-1}-1)$

excess $(2^{8-1}-1 = 127)$

$$\begin{array}{rl}
00000100 & = e \\
+\ 01111111 & = 127 \\
\hline
10000011 & = E
\end{array}$$

5. Replace e with E

0	10000011	1100 ... 0

Convert back

1. decode: $+\ 2^{10000011} \times 1.1100\ldots0$

2. convert E

$$\begin{array}{rl}
 & \text{-1 1 1 1 1 1} \leftarrow \text{borrows} \\
 & 10000011 \\
- & 01111111 \\
\hline
 & 0000100 = 4 = e
\end{array}$$

$\Rightarrow 2^4 \times 1.11$ (convert f mult. by 2^e) $\Rightarrow 1.1100 = 11100$

$\times 2^4$

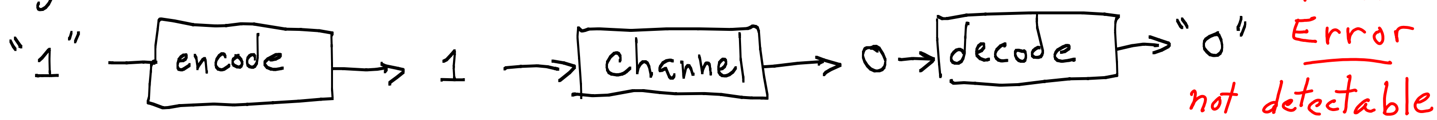
(convert to dec.) $\Rightarrow 16 + 8 + 4 = 28$

So much for encoding data. We could go on to audio, video, ... But, back to noise and errors.

Error Detection / Correction

"message" could be a bit, a string of bits, a character, a page of characters, ...

message coded in bits:

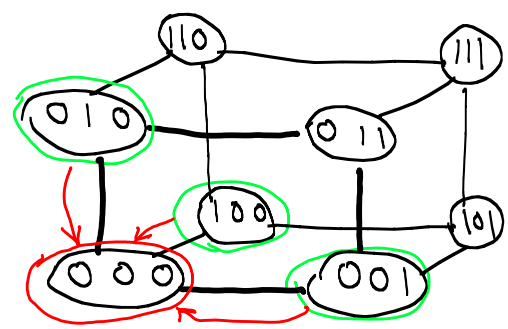


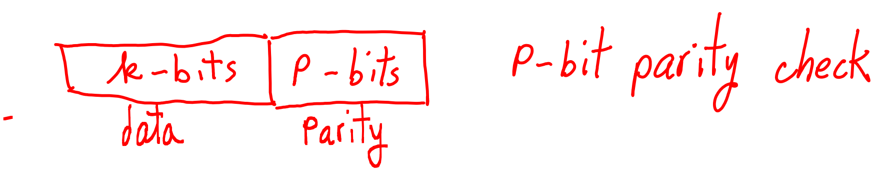
2-bit encoding

Code words 00 and 11 are good data, 10 and 01 indicate 1-bit errors. Last bit is "parity" bit, odd parity codeword indicates error. Works for k-bit messages w/ 1 parity bit (if 2-bit errors very unlikely).

1-bit Error Correction w/ 3-bit code words:
"0" ==> 000
"1" ==> 111

001 ==> "0"	011 ==> "1"
010 ==> "0"	101 ==> "1"
100 ==> "0"	110 ==> "1"





1-bit Correction, 2-bit Detection

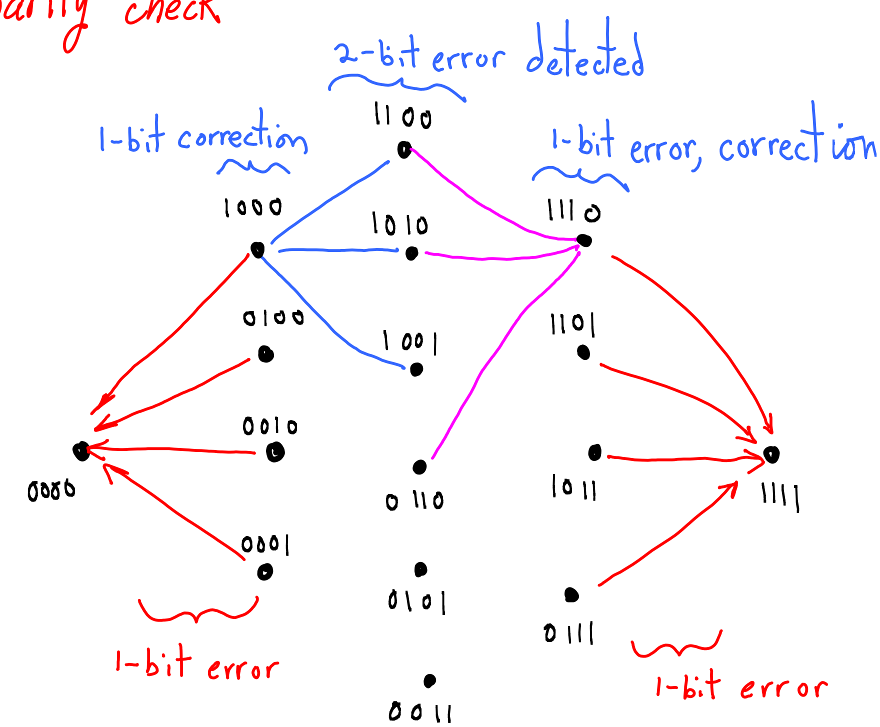
-- odd parity: 1-bit error corrected

-- exactly two 1's: 2-bit error detected

-- otherwise: no error

How many extra bits, at minimum?
Depends on noise in channel:
Shannon Noisey Coding Theorem.

We use 4 bits, 1-bit data.

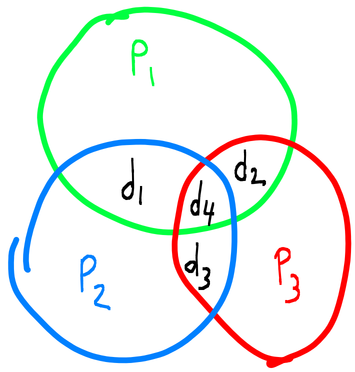


Hamming (7,4) code (Single Error Detection, Single Error Correction)

7 bits per code word

4 data bits

3 parity bits



$d_1 \, d_2 \, d_3 \, d_4 \;\; p_1 \, p_2 \, p_3$

$p_1 = \text{parity}(d_1 \, d_2 \, d_4)$

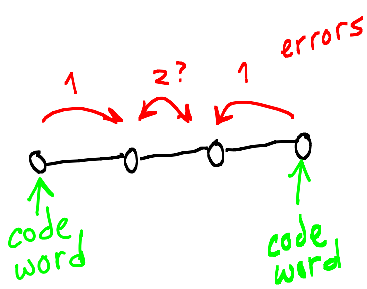
$p_2 = \text{parity}(d_1 \, d_3 \, d_4)$

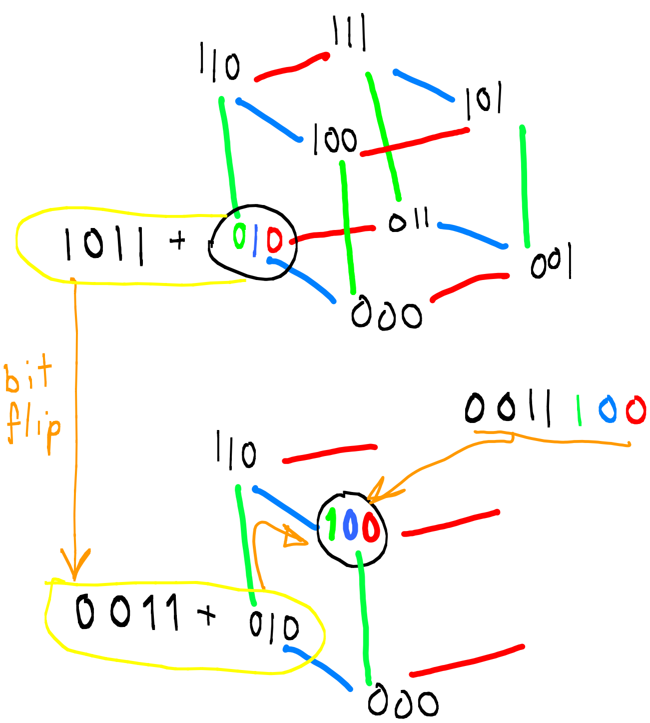
$p_3 = \text{Parity}(d_2 \, d_3 \, d_4)$

3 steps to next codeword

1-bit error: can detect and correct

2-bit error: cannot detect





other neighbor code words

0001 111

0010 011

0111 001

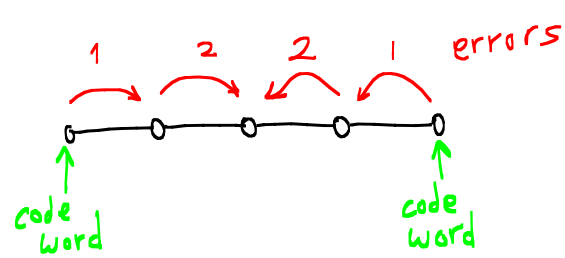
What can we do about 2-bit errors?
Add another parity bit.

$$p_4 = \text{parity}(d_1 \, d_2 \, d_3 \, d_4 \, p_1 \, p_2 \, p_3)$$

$$\text{Code word} = d_1 \, d_2 \, d_3 \, d_4 \, p_1 \, p_2 \, p_3 \, p_4$$

$\Longrightarrow$ 4 steps min to next code word

1-bit error: detect + correct
2-bit error: detect



0 0 0 0 0 0 0

Hamming 7,4 code:
Find distances to all other code words.
GREEN-PARITY: Bits[3, 2, 0]
BLUE-PARYT: Bits[3, 1, 0]
RED-PARITY: Bits[2, 1, 0]

0 0 0 1 1 1 1

(1-bit flip causes 3 other flips
Hamming distance = 4)

0 0 1 0 0 1 1

Hamming distance = 3

0 1 0 0 1 0 1

Hamming distance = 3

1 0 0 0 1 1 0

Hamming distance = 3

0 0 1 1 1 0 0

HD = 3

0 1 0 1 0 1 0

HD = 3

1 0 0 1 0 0 1

HD = 3

0 1 1 0 1 1 0

HD = 4

1 0 1 0 1 0 1

HD = 4

1 1 0 0 0 1 1

HD = 4

0 1 1 1 0 0 1

HD = 4

1 0 1 1 0 1 0

HD = 4

1 1 0 1 1 0 0

HD = 4

1 1 1 0 0 0 0

HD = 3

1 1 1 1 1 1 1

HD = 7

ASCII (See back cover of PP)

HEX CODE	MEANING	Printable?
00	NUL	no
01	SOH	no
...	...	
20	space	yes
...	...	
30	"0"	yes
31	"1"	yes
...	...	
41	"A"	yes
42	"B"	yes
...	...	
61	"a"	yes
62	"b"	yes
...	...	
7A	"z"	yes
...	...	

(other stuff, non-standard)

} data communications control signals

who's on first?

6D	41	2F	32
↓	↓	↓	↓
'm'	'A'	'/'	'2'

Byte Addressable

addr	Memory bits		print order
0	0011 0010 (low bit)	→ 32 ("2")	↓
1	0010 1111	→ 2F ("/")	
2	0100 0001	→ 41 ("A")	
3	0110 1101	→ 6D ("m")	

high bit

What to Print	Starting Memory Address	What is displayed (left-to-right)
4-byte number (in hex notation)	0	6D412F32
two 2-byte numbers (in hex)	0	2F32 6D41
four 1-byte numbers (in hex)	0	32 2F 41 6D
one 4-byte string	0	2 / A m

(see "od" in unix)