Quiz Review

Only 11 classes to go!

Quiz 2 is next class Tuesday 27 March

Quantifying Information
(Claude Shannon, 1948)

Suppose you're faced with N equally probable choices, and I give you a fact that narrows it down to M choices. Then I've given you

$\log_2(N/M)$ bits of information

Information is measured in bits (binary digits) = number of 0/1's required to encode choice(s)

Examples:

- information in one coin flip: $\log_2(2/1) = 1$ bit
- roll of a single die: $\log_2(6/1) = \sim 2.6$ bits
- outcome of a Football game: 1 bit

  (well, actually, "they won" may convey more information than "they lost"…)

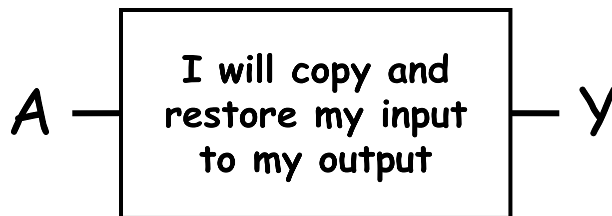
Information Summary

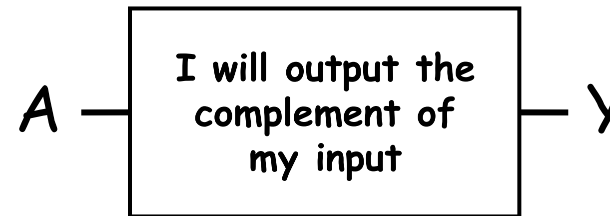
Information resolves uncertainty

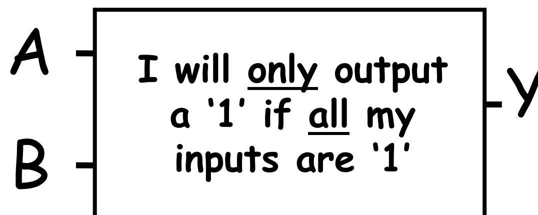
- Choices equally probable:
  - N choices down to M $\rightarrow$
    $\log_2(N/M)$ bits of information
- Choices not equally probable:
  - choice_i with probability $p_i \rightarrow$
    $\log_2(1/p_i)$ bits of information
  - average number of bits = $\Sigma p_i \log_2(1/p_i)$
  - use variable-length encodings

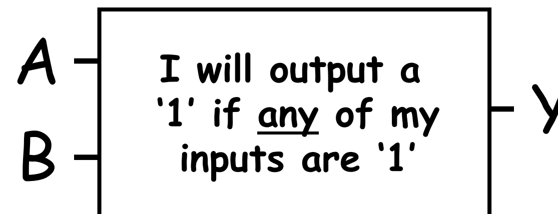
Digital Processing Elements

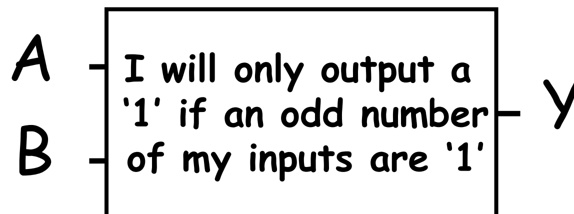
Some digital processing elements occur so frequently that we give them special names and symbols





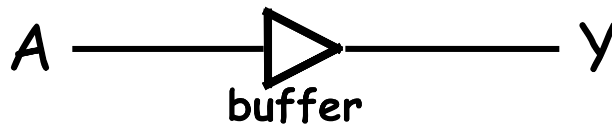


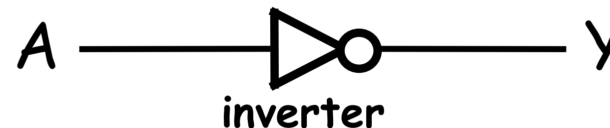


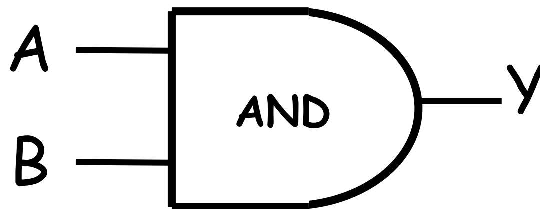


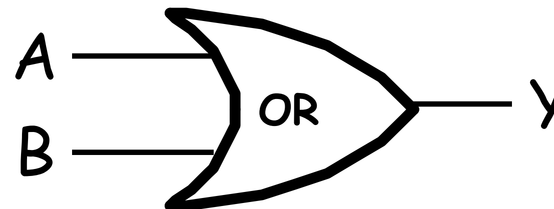
Digital Processing Elements

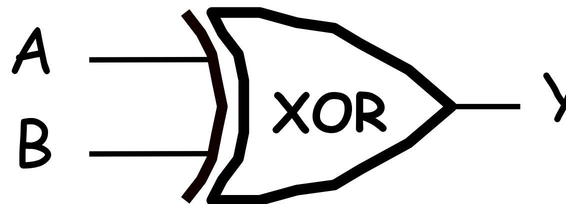
Some digital processing elements occur so frequently that we give them special names and symbols





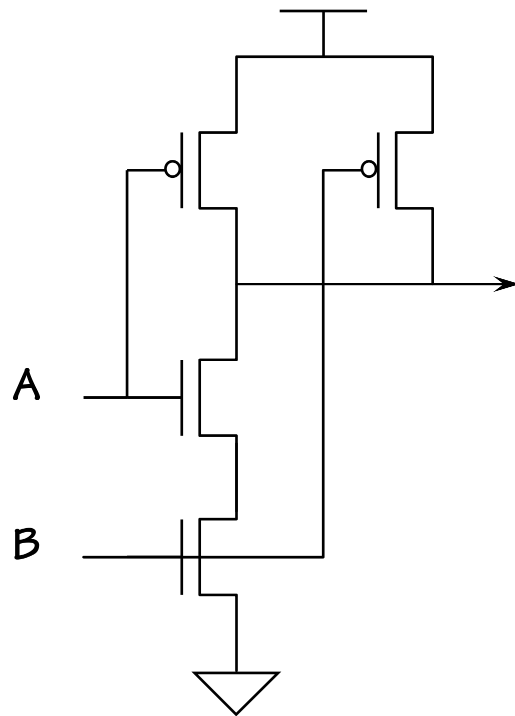






In honor of the richest man in the world we will henceforth refer to digital processing elements as "GATES"

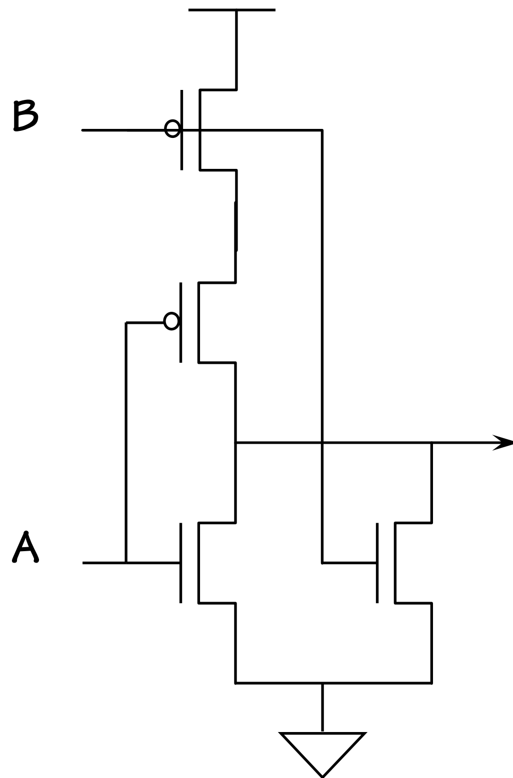
A Two Input Logic Gate



What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

Here's Another…



What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

Here's a Design Approach

Truth Table

C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

1) Write out our functional spec as a truth table
2) Write down a Boolean expression for every '1' in the output

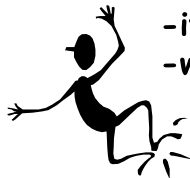
$$Y = \overline{C}\,\overline{B}A + \overline{C}BA + CB\overline{A} + CBA$$

3) Wire up the gates, call it a day, and go home!

This approach will always give us logic expressions in a particular form:

SUM-OF-PRODUCTS

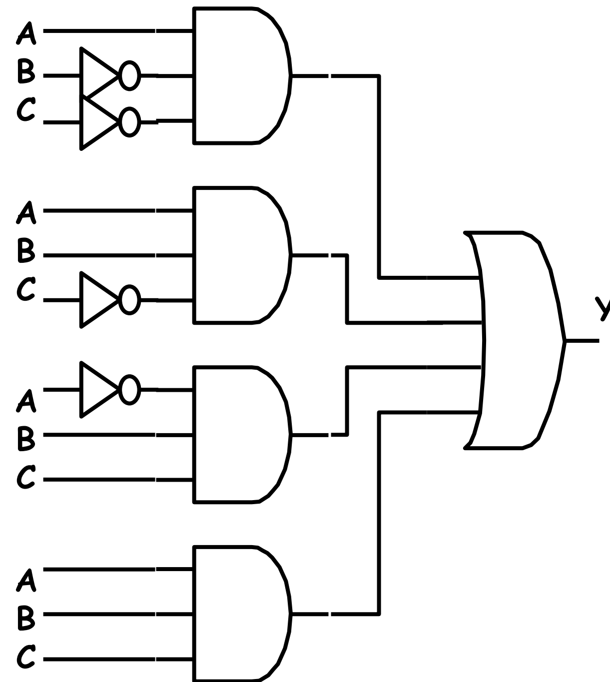
-it's systematic!
-it works!
-it's easy!
-we get to go home!



Straightforward Synthesis

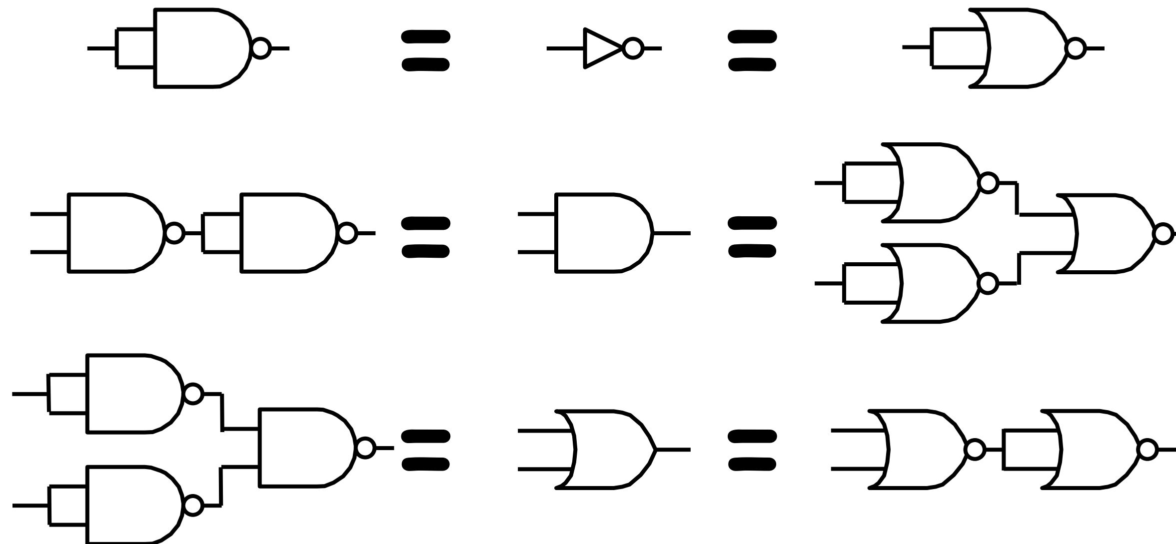
We can implement
SUM-OF-PRODUCTS
with just three levels of logic.

INVERTERS/AND/OR



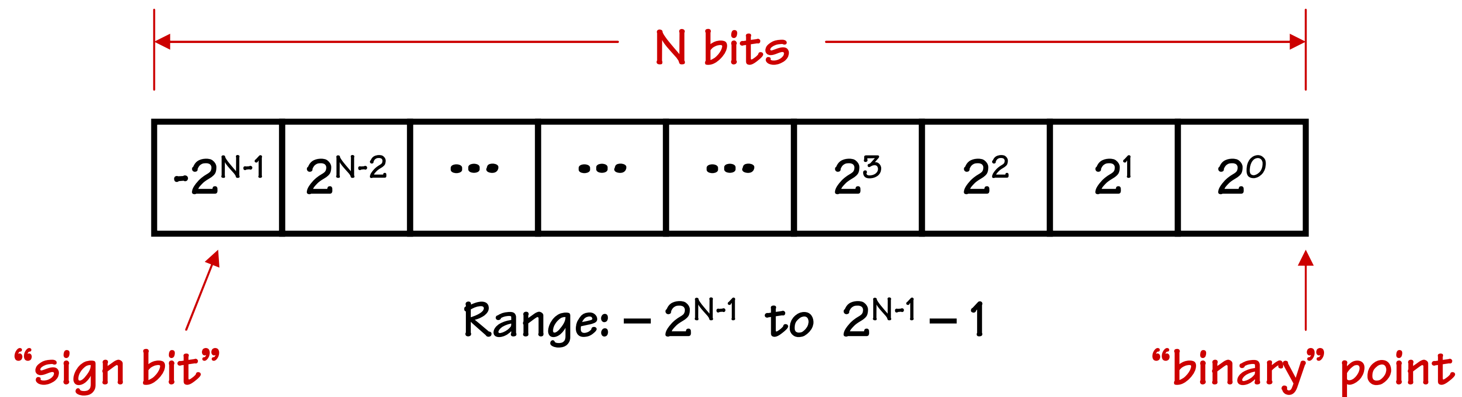
One Will Do!

NANDs and NORs are universal



Ah!, but what if we want more than 2-inputs

Review: 2's Complement



8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's-complement representation for signed integers, the same binary addition procedure will work for adding both signed and unsigned numbers.

By moving the implicit "binary" point, we can represent fractions too:

$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

Binary Addition

Here's an example of binary addition as one might do it by "hand":

```
       1 1 0 1      <- Carries from previous column
A:      1101
B:  +   0101
       10010
```

Adding two N-bit numbers produces an (N+1)-bit result

Let's start by building a block that adds one column:

A B CO FA CI S

Then we can cascade them to add two numbers of any size…

A3 B3 A2 B2 A1 B1 A0 B0

S4 S3 S0 S1 S0

Designing a Full Adder: From Last Time

1) Start with a truth table:

C_i	A	B	C_o	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

2) Write down eqns for the "1" outputs

$$C_o = \overline{C_i}AB + C_i\overline{A}B + C_iA\overline{B} + C_iAB$$
$$S = \overline{C_i}\,\overline{A}B + \overline{C_i}A\overline{B} + C_i\overline{A}\,\overline{B} + C_iAB$$

3) Simplifing a bit

$$C_o = C_i(A + B) + AB$$
$$S = C_i \oplus A \oplus B$$

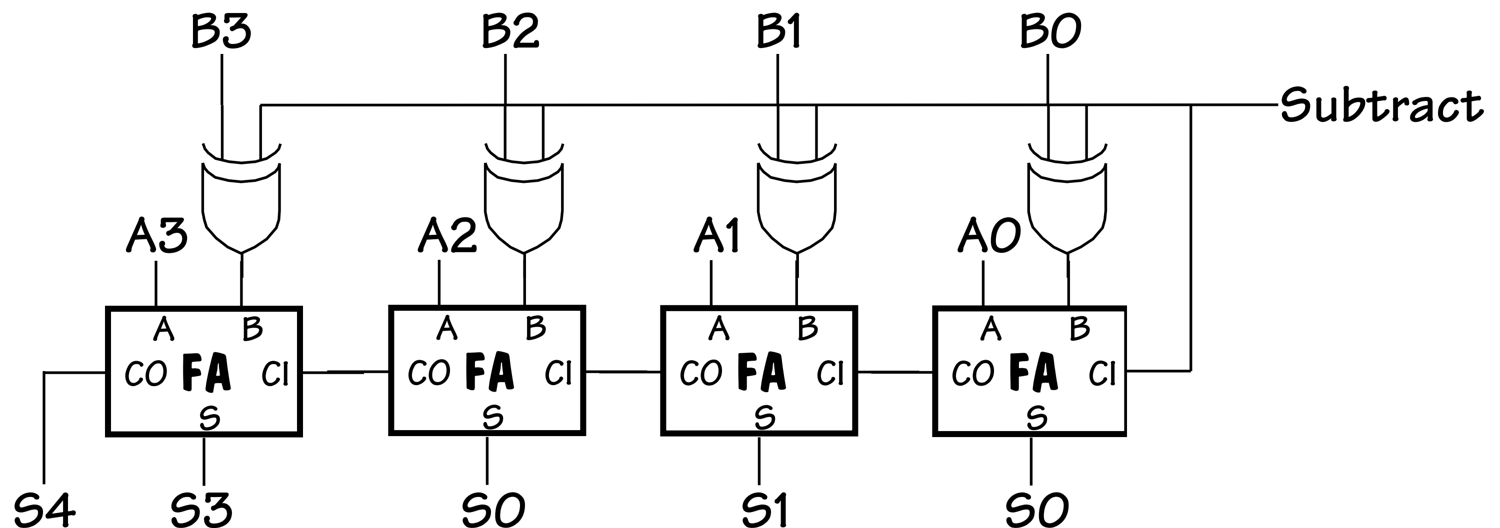
$$C_o = C_i(A \oplus B) + AB$$
$$S = C_i \oplus (A \oplus B)$$

Subtraction: A-B = A + (-B)

Using 2's complement representation: $-B = \sim B + 1$

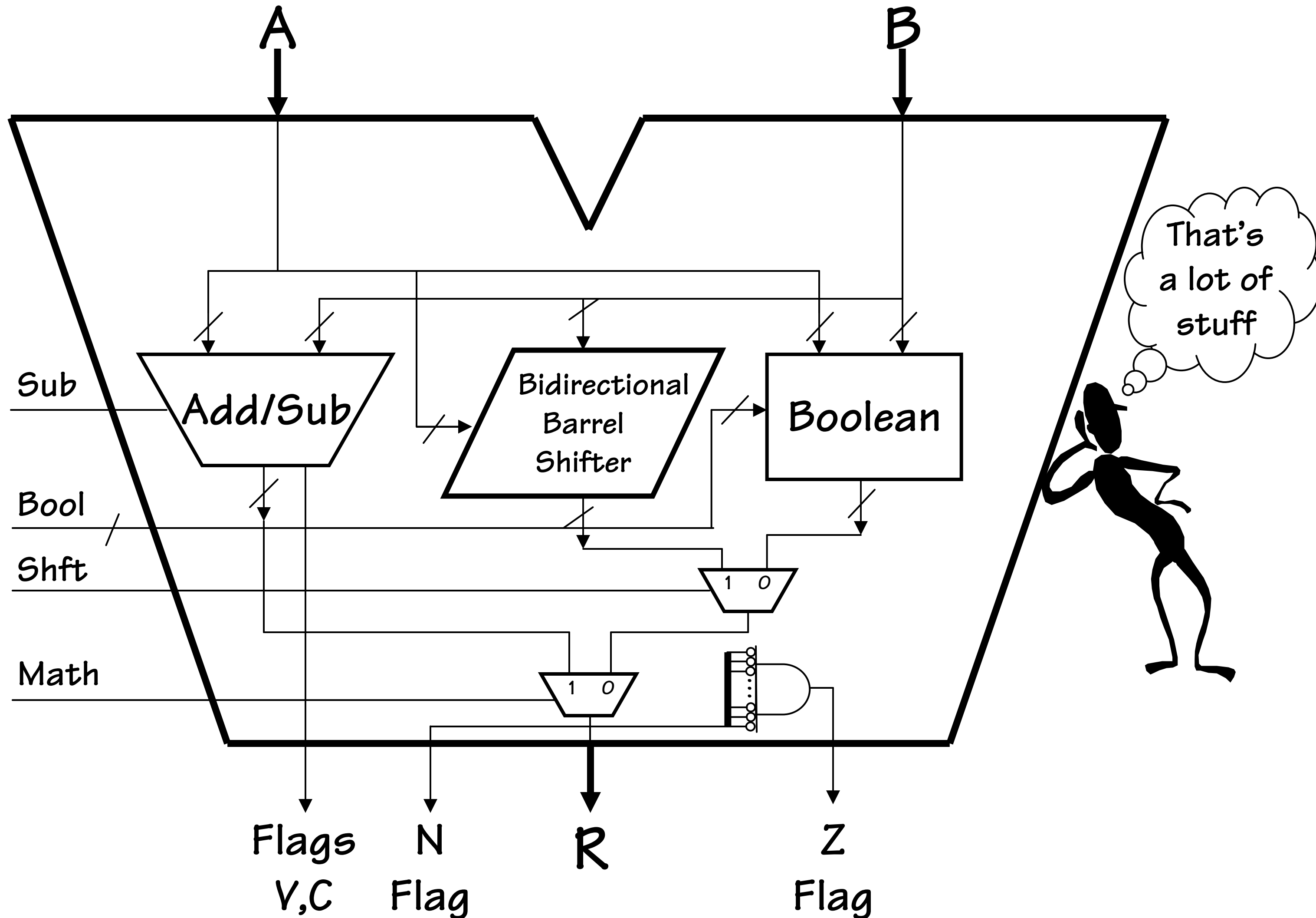
~ = bit-wise complement

So let's build an arithmetic unit that does both addition and subtraction. Operation selected by control input:



An ALU, at Last

Now we're ready for a big one! An Arithmetic Logic Unit.

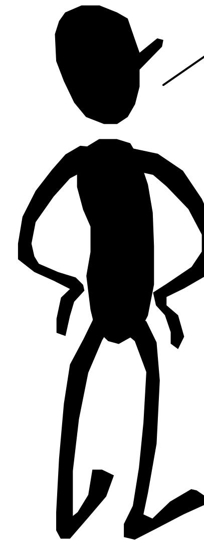


Binary Multipliers

The key trick of multiplication is memorizing a digit-to-digit table…
Everything else was just adding

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

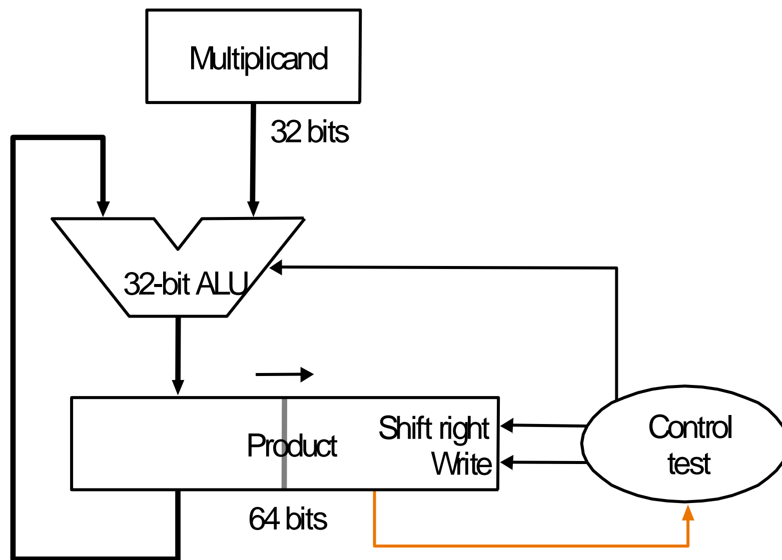
×	0	1
0	0	0
1	0	1

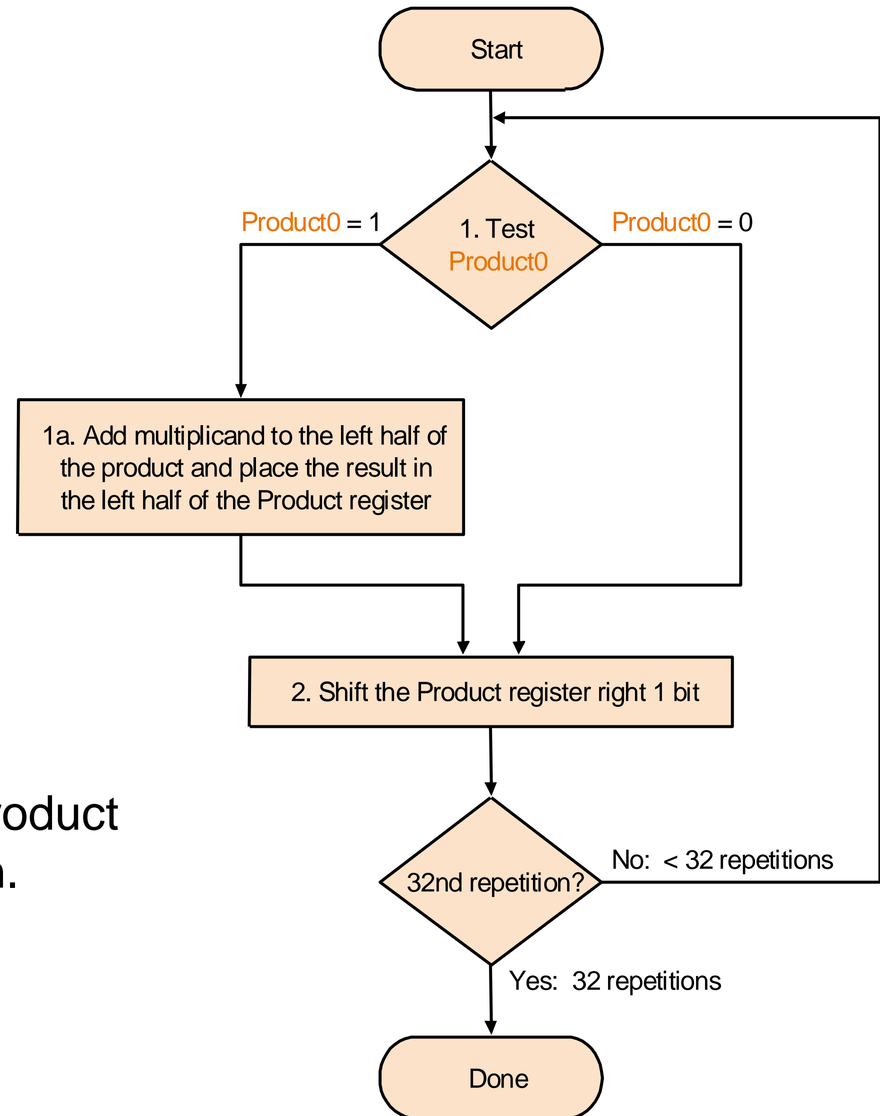


You've got to be kidding… It can't be that easy

Reading: Study Chapter 3.

Final Version





The trick is to use the lower half of the product to hold the multiplier during the operation.

FP == "Binary" Scientific Notation

- IEEE single precision floating-point format

0	1	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

"S" Sign Bit — "E" Exponent + 127 — "F" Significand (Mantissa) - 1

0x42280000 in hexadecimal

- Exponent: Unsigned "Bias 127" 8-bit integer

  E = Exponent + 127

  Exponent = 10000100 (132) – 127 = 5

- Significand: Unsigned fixed binary point with "hidden-one"

  Significand = "1"+ 0.01010000000000000000000 = 1.3125

- Putting it all together

  $N = -1^{S} (1 + F) \times 2^{E-127} = -1^{0} (1.3125) \times 2^{5} = 42$

Zeros

- How do you represent 0?
- Zero   Sign = ?, Exponent = ?, Significand = ?
  - Here's where the hidden "1" comes back to bite you
  - Hint: Zero is small. What's the smallest number you can generate?
  - E = Exponent + 127, Exponent = -127, Signficand = 1.0
    $1^0 (1.0) \times 2^{-127} = 5.87747 \times 10^{-39}$
- IEEE Convention
  - When E = 0 (Exponent = -127), we'll interpret numbers differently…

0 00000000 00000000000000000000000 = 0 not 1.0×2^{-127}

1 00000000 00000000000000000000000 = -0 not -1.0×2^{-127}

Yes, there are "2" zeros. Setting E=0 is also used to represent a few other small numbers besides 0. In all of these numbers there is no "hidden" one assumed in F, and they are called the "unnormalized numbers". WARNING: If you rely these values you are skating on thin ice!

Floating point AIN'T NATURAL

It is CRUCIAL for computer scientists to know that Floating Point arithmetic is NOT the arithmetic you learned since childhood

1.0 is NOT EQUAL to 10*0.1 (Why?)

- 1.0 * 10.0 == 10.0
- 0.1 * 10.0 != 1.0
- 0.1 decimal == 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + … ==
  - 0.0 0011 0011 0011 0011 0011 …
- In decimal 1/3 is a repeating fraction 0.333333…
- If you quit at some fixed number of digits, then 3 * 1/3 != 1

Floating Point arithmetic IS NOT associative

- x + (y + z) is not necessarily equal to (x + y) + z

Addition may not even result in a change

- (x + 1) MAY == x

How to Improve Performance?

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

$$\text{MIPS} = \frac{\text{Freq}}{\text{CPI}}$$

So, to improve performance (everything else being equal) you can either

Decrease the # of required cycles for a program, or (improve ISA/Compiler)

Decrease the clock cycle time or, said another way,

Increase the clock rate. (reduce propagation delays or use pipelining)

Decrease the CPI (average clocks per instruction) (new H/W)

Now that We Understand Cycles

- A given program will require
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (average clocks per instruction)
    - *a floating point intensive application might have a higher CPI*
  - MIPS (millions of instructions per second)
    - *this would be higher for a program using simple instructions*

Compiler's Performance Impact

Two different compilers are being tested for a 500 MHz machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software. The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 2 million Class C instructions. The second compiler's code uses 7 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

Which program uses the fewest instructions?

$$\text{Instructions}_1 = (5+1+2) \times 10^6 = 8 \times 10^6$$

$$\text{Instructions}_2 = (7+1+1) \times 10^6 = 9 \times 10^6$$

Which sequence uses the fewest clock cycles?

$$\text{Cycles}_1 = (5(1)+1(2)+2(3)) \times 10^6 = 13 \times 10^6$$

$$\text{Cycles}_2 = (7(1)+1(2)+1(3)) \times 10^6 = 12 \times 10^6$$

Amdahl's Law

$$t_{improved} = \frac{t_{affected}}{r_{speedup}} + t_{unaffected}$$

Example:

"Suppose a program runs in 100 seconds on a machine, where multiplies are executed 80% of the time. How much do we need to improve the speed of multiplication if we want the program to run 4 times faster?"

$25 = 80/r + 20$ $\quad r = 16x$

How about making it 5 times faster?

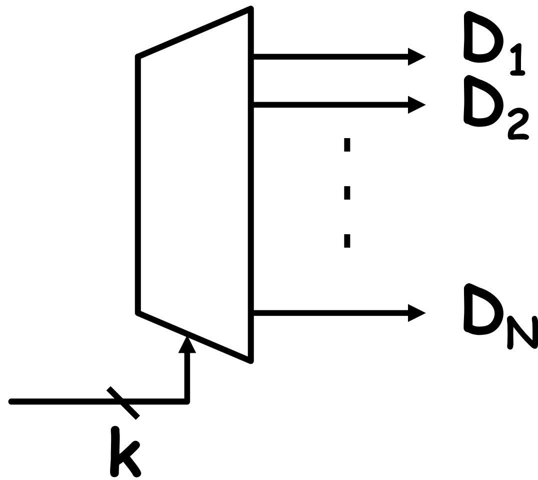
$20 = 80/r + 20$ $\quad r = ?$

Principle: Make the common case fast

Remember

- Performance is specific to a particular programs
  - Total execution time is a consistent summary of performance
- For a given architecture performance comes from:
  1) increases in clock rate (without adverse CPI affects)
  2) improvements in processor organization that lower CPI
  3) compiler enhancements that lower CPI and/or instruction count
- Pitfall: Expecting improvements in one aspect of a machine's performance to affect the total performance
- You should not always believe everything you read! Read carefully!

A New Combinational Device



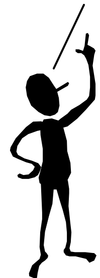
DECODER:

k SELECT inputs,

$N = 2^k$ DATA OUTPUTs.

Selected D_j HIGH;
all others LOW.

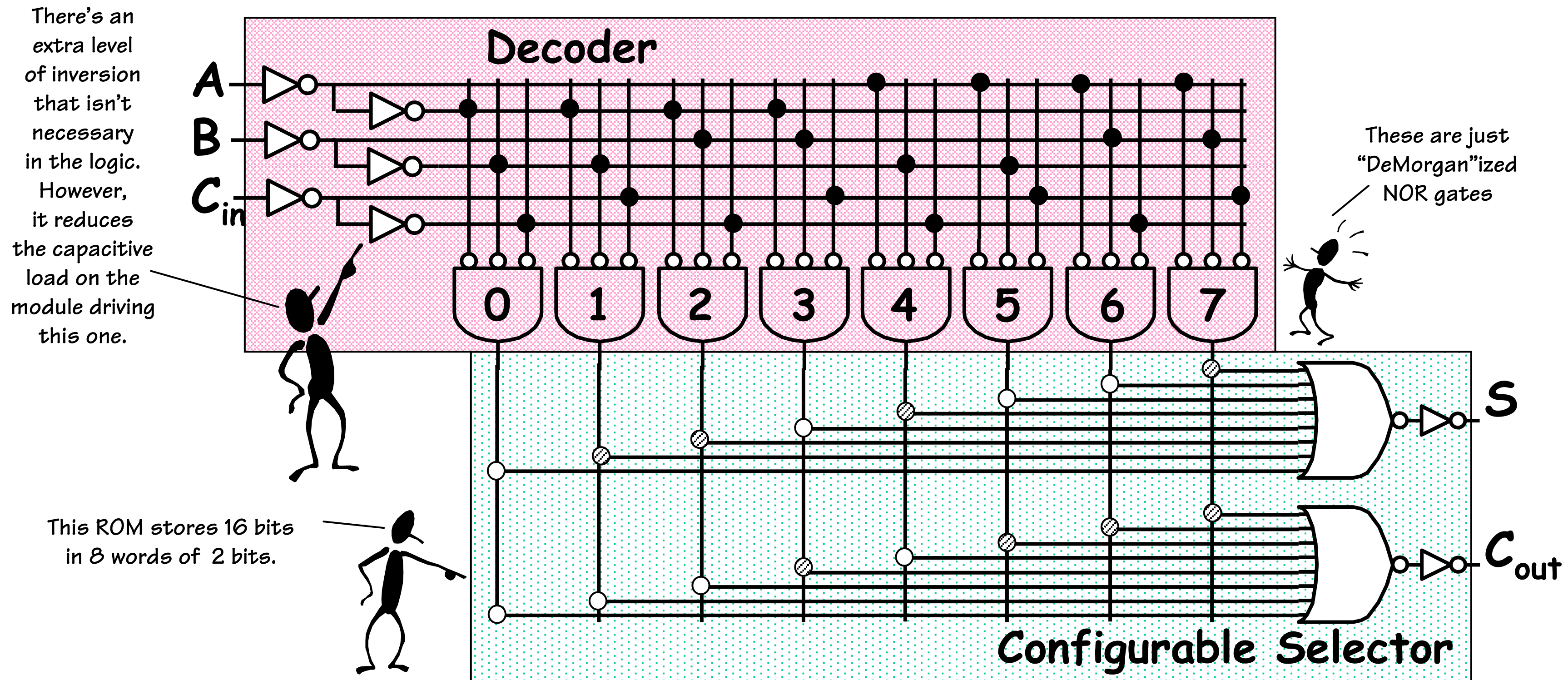
Have I mentioned that HIGH is a synonym for '1' and LOW means the same as '0'



NOW, we are well on our way to building a general purpose table-lookup device.

We can build a 2-dimensional ARRAY of decoders and selectors as follows ...

Shared Decoding Logic



We can build a general purpose "table-lookup" device called a Read-Only Memory (ROM), from which we can implement any truth table and, thus, any combinational device

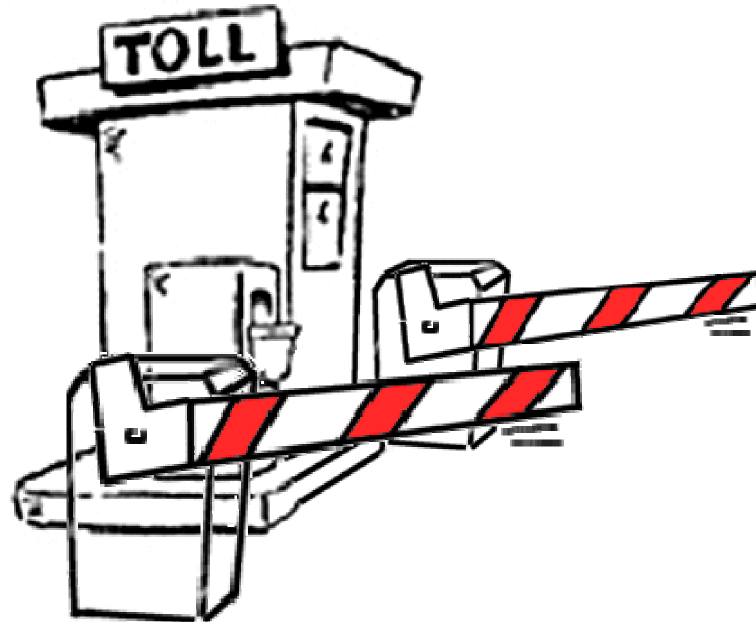
Made from PREWIRED connections ●, and CONFIGURABLE connections that can be either connected ◍ or not connected ○

Escapement Strategy

The Solution:
  Add two gates
  and only open
  one at a time.

Escapement Strategy

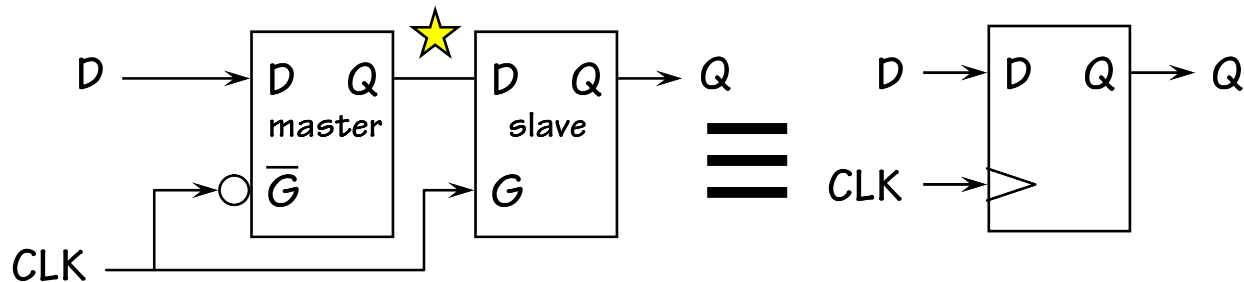
The Solution:
Add two gates and only open one at a time. (Psst… Don't tell the toll folks)



KEY: At no time is there an open path through both gates…

Edge-triggered Flip Flop

logical "escapement"

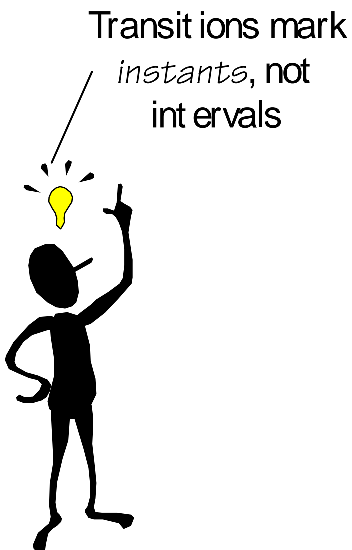


Observations:

- only one latch "transparent" at any time:
  - master closed when slave is open (CLK is high)
  - slave closed when master is open (CLK is low)

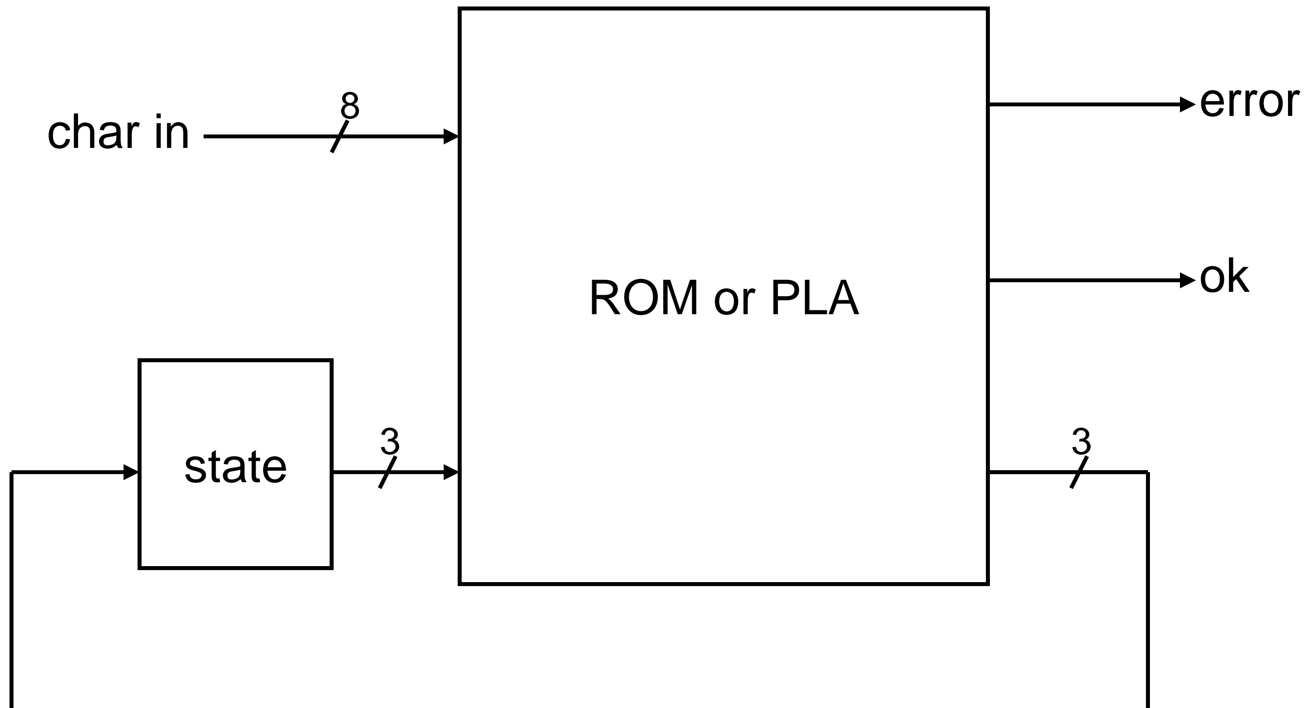
  → no combinational path through flip flop

- Q only changes shortly after 0 →1 transition of CLK, so flip flop appears to be "triggered" by rising edge of CLK



Transitions mark *instants*, not intervals

FSM Implementation



Our PLA has:

- 11 inputs
- 5 outputs